RAIRO-Oper. Res. 50 (2016) 241–267
DOI: 10.1051/ro/2015017

RAIRO Operations Research
www.rairo-ro.org

GENERALIZED HYPERTREE DECOMPOSITION FOR SOLVING NON BINARY CSP WITH COMPRESSED TABLE CONSTRAINTS*

Zineb Habbas[1], Kamal Amroun[2] and Daniel Singer[1]

Abstract. Many real-world problems can be modelled as Constraint Satisfaction Problems (CSPs). Although CSP is NP-complete, it has been proved that non binary CSP instances may be efficiently solved if they are representable as Generalized Hypertree Decomposition (GHD) with small width. Algorithms for solving CSPs based on a GHD consider an extensional representation of constraints together with join and semi-join operations giving rise to new large constraint tables (or relations) needed to be temporarily saved. Extensional representation of constraints is quite natural and adapted to the specification of real problems but unfortunately limits significantly the practical performance of these algorithms. The present work tackles this problem using a compact representation of constraint tables. Consequently, to make algorithms compatible with this compact representation, new "compressed join" and "compressed semi-join" operations have to be defined. This last step constitutes our main contribution which, as far as we know, has never been presented. The correctness of this approach is proved and validated on multiple benchmarks. Experimental results show that using compressed relations and compressed operations improves significantly the practical performance of the basic algorithm proposed by Gottlob *et al.* for solving non binary CSPs with a Generalized Hypertree Decomposition.

Mathematics Subject Classification. 68T20.

Received May 18, 2015. Accepted June 1st, 2015.

1. Introduction

Many real-world problems can be modelled as Constraint Satisfaction Problems (CSPs) but solving CSPs is combinatorial by nature, making an efficient algorithm unlikely to exist. Usual methods that guarantee to find a solution are enumerative and have an exponential time complexity in the worst case. In order to provide better theoretical complexity bounds, structural decomposition methods have received considerable interest these last decades. Numerous decomposition methods have been successfully used to characterize some tractable classes [4,5,8,11,15,16,28]. From theoretical viewpoint, methods based on (generalized) hypertree decomposition are better than those based on tree decomposition [11]. In addition, theoretical time complexities for resolution algorithms using tree decomposition as well as (generalized) hypertree decomposition can really outperform the classical direct methods. However, except for the recent work on *BTD* (Backtracking on Tree Decomposition)

Keywords. Constraint Satisfaction Problems, Generalized Hypertree Decomposition, Compressed Table Constraints.

* *A preliminary version of this paper was awarded at COSI 2013.*

[1] LCOMS EA 7306, University of Lorraine, Metz, France. zineb.habbas@univ-lorraine.fr; daniel.singer@univ-lorraine.fr

[2] LIMED, Faculté des Sciences Exactes, Université de Bejaia, 06000 Bejaia, Algeria. k_amroun25@yahoo.fr

Article published by EDP Sciences

method [20], the experimental results do not confirm this theoretical gain from practical viewpoint. The memory space consumption problem is the main drawback which prevents the practical efficiency of using structural decomposition methods.

This work is an attempt to exploit efficiently a generalized hypertree decomposition (GHD) for solving non binary CSPs. A basic algorithm is presented in [11] for processing a GHD with Join and Acyclic Solving (JAS). Its main primitive operation is the join of the relations to solve the subproblems associated with the nodes of the hypertree. This operation is the major bottleneck for practical efficiency of JAS when solving large instances. To overcome this drawback, we propose to exploit compressed representation of relations in order to represent an exponential number of tuples in a polynomial space. This compressed representation of table constraints (constraint relations defined in extension) needs to extend the classical join and semi-join operations used in JAS giving rise to the new algorithm $CJAS$ (Compressed Join Acyclic Solving). Notice that compressing table constraints has already been successfully used for improving Generalized Arc Consistency (GAC) algorithms dealing with large extensional constraints [23]. But, as far as we know, this idea has never been explored for improving the algorithms for solving non binary CSPs using a GHD.

Many techniques are proposed in order to reduce the space needed for representing and propagating constraint relations. Katsirelos *et al.* [23] use compressed tuples to improve the algorithm GAC schema [3]. Régin [34] proposed to use compressed tuples for expressiveness of constraint tables. Gent *et al.* [9] proposed to use *tries* to propagate constraint relations and to look for a support in order to enforce GAC property [27]. A trie is a rooted tree storing and retrieving strings over an alphabet. A table can be represented by a trie where levels are associated with the variables in the scope of the constraint. At each level, the alphabet is the domain of the corresponding variable. Kenil *et al.* [24] proposed to use *Multi-valued Decision Diagrams* (MDD) [21] for enforcing GAC. A MDD is a trie where prefix redundancy is eliminated. Another approach [33] uses Deterministic Finite Automaton to enforce GAC determining if a given tuple is accepted (present in the table) or not. Ullmann [38] and Lecoutre [26] proposed to use Simple Table Reduction (STR) to keep supports. STR maintains dynamically the tables of allowed tuples by removing from the tables all tuples that become invalid whenever a value is removed from the domain of a variable.

The rest of the paper is organized as follows. Section 2 is devoted to the presentation of the background: the useful definitions related to Constraint Satisfaction Problems, Tree Decomposition, Generalized Hypertree Decomposition and the main algorithm proposed in the literature for solving CSP instances *via* a generalized hypertree decomposition. In Section 3, we present the definitions related to compressed tuple, compressed relation and compressed CSP. Based on the definition of compressed relation, we extend the classical join and semi-join operations to compressed join and compressed semi-join operations. In Section 4 we present the algorithm $CJAS$ and prove its correctness. Section 5 permits us to validate experimentally our approach by numerous tests on well-known benchmarks. Finally, in Section 6 we conclude this paper.

2. Background

The notion of Constraint Satisfaction Problem (CSP) has been formally defined by Montanari [29]. A CSP instance is defined as a tuple $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R}\rangle$. $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a finite set of n variables and $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of finite domains. Each variable X_i takes its value from its domain D_i. $\mathcal{C} = \{C_1, \ldots, C_m\}$ is a set of m constraints.

A constraint $C_i \in \mathcal{C}$ on an ordered subset of variables, $C_i = (X_{i_1}, \ldots, X_{i_{a_i}})$[3] ($a_i$ is called the arity of the constraint C_i), is defined by an associated relation $R_i \in \mathcal{R}$ of allowed combinations of values for the variables in C_i. Note that we take the same notation for the constraint C_i and its scope. Binary CSPs are those defined where each constraint involves only two variables that is $\forall i \in \{1, \ldots, m\}, |C_i| = 2$. Constraints of arity 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 $t \in R_i$ is a list of values $(v_{i_1}, \ldots, v_{i_{a_i}})$ where $a_i = |C_i|$ such that $v_{i_j} \in D_{i_j} \forall j \in \{1, \ldots, a_i\}$.

[3]For the sake of simplicity, the list of variables in C_i will be used also to mean the set of variables occurring in C_i which is called its scope.

A *solution* to a CSP is an assignment of values to all the variables in $\mathcal{X}$ such that for each constraint C_i the assignment restricted to C_i belongs to R_i.

The *constraint hypergraph* associated with a CSP instance $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$ is the hypergraph $\mathcal{H} = \langle V, E \rangle$ where the set of vertices V is the set of variables $\mathcal{X}$ and the set of hyperedges E are the set of constraint scopes in $\mathcal{C}$. For any hyperedge $h \in E$ we denote by $\mathrm{var}(h)$ the set of vertices of h and for any subset of hyperedges $K \subseteq E$ $\mathrm{var}(K) = \bigcup_{h \in K} \mathrm{var}(h)$. We denote by $\mathrm{var}(\mathcal{H})$ the set of vertices V and by $edges(\mathcal{H})$ the set of hyperedges E. (We use the term var because the vertices of the hypergraph correspond to the variables of the CSP). The *primal graph* of a hypergraph $\mathcal{H} = \langle V, E \rangle$ is a graph whose set of vertices is V and whose edges connect each pair of vertices occurring together in a same hyperedge of $\mathcal{H}$. In this context, tree decomposition and generalized hypertree decomposition play an important role. These two notions are described hereafter.

Definition 2.1 (Tree decomposition [35])**.** A tree decomposition of a graph $G = (V, E)$ is a pair $\langle T, \chi \rangle$ where $T = (N, F)$ is a tree and χ is a labelling function which associates with each vertex $p \in N$ of T a set of vertices $\chi(p)$ such that all the following conditions hold:

(1) For each vertex $v \in V$, there is a vertex $p \in N$ such that $v \in \chi(p)$.
(2) For each edge $\{v, w\} \in E$, there is a vertex $p \in N$, such that $\{v, w\} \subseteq \chi(p)$.
(3) For each vertex $v \in V$, the set $\{p \in N | v \in \chi(p)\}$ induces a (connected) subtree of T.

The *width* of a tree decomposition is equal to $\max_{p \in N} |\chi(p)| - 1$ and the treewidth of a graph is the minimal width over all its tree decompositions.

Note that this notion can be applied on any hypergraph by considering the tree decomposition of its associated primal graph.

Definition 2.2 (Hypertree)**.** Let $\mathcal{H} = \langle V, E \rangle$ be a hypergraph. A *hypertree* for $\mathcal{H}$ is a triple $\langle T, \chi, \lambda \rangle$ where $T = (N, F)$ is a rooted tree, and χ and λ are labelling functions which associate each vertex $p \in N$ with two sets $\chi(p) \subseteq V$ and $\lambda(p) \subseteq E$. If $T' = (N', F')$ is a subtree of T we define $\chi(T') = \bigcup_{v \in N'} \chi(v)$. We denote the set of vertices N of T by $vertices(T)$ and the root of T by $root(T)$. T_p denotes the subtree of T rooted at the node p and $Parent(p)$ is the parent node of p in T.

Definition 2.3 (Hypertree Decomposition [12])**.** A *Hypertree Decomposition* of a hypergraph $\mathcal{H} = \langle V, E \rangle$ is a hypertree $\mathrm{HD} = \langle T, \chi, \lambda \rangle$ which satisfies the following conditions:

(1) For each edge $h \in E$, there exists $p \in vertices(T)$ such that $\mathrm{var}(h) \subseteq \chi(p)$.
(2) For each vertex $v \in V$, the set $\{p \in vertices(T) | v \in \chi(p)\}$ induces a connected subtree of T.
(3) For each vertex $p \in vertices(T), \chi(p) \subseteq \mathrm{var}(\lambda(p))$.
(4) For each $p \in vertices(T)$, $\mathrm{var}(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$.

The *width* of a hypertree $\mathrm{HD} = \langle T, \chi, \lambda \rangle$ is equal to $\max_{p \in vertices(T)} |\lambda(p)|$. The *hypertree-width* ($hw(\mathcal{H})$) of a hypergraph $\mathcal{H}$ is the minimum width over all its hypertree decompositions.

A hyperedge h of a hypergraph $\mathcal{H} = \langle V, E \rangle$ is *strongly covered* in $\mathrm{HD} = \langle T, \chi, \lambda \rangle$ if there exists $p \in vertices(T)$ such that the vertices of h are contained in $\chi(p)$ and $h \in \lambda(p)$. A hypertree decomposition $\mathrm{HD} = \langle T, \chi, \lambda \rangle$ of a hypergraph $\mathcal{H}$ is *complete* if every hyperedge h of $\mathcal{H}$ is strongly covered in HD.

A hypertree $\mathrm{HD} = \langle T, \chi, \lambda \rangle$ is called a *Generalized Hypertree Decomposition* (GHD) [1,13] if the conditions (1), (2) and (3) of Definition 2.3 hold. The width of a Generalized Hypertree Decomposition $\mathrm{HD} = \langle T, \chi, \lambda \rangle$ is equal to $\max_{p \in vertices(T)} |\lambda(p)|$. The *generalized-hypertree-width* ($ghw(\mathcal{H})$) of a hypergraph $\mathcal{H}$ is the minimum width over all its generalized hypertree decompositions.

The pair $\langle T, \chi \rangle$ in a GHD $\langle T, \chi, \lambda \rangle$ of a hypergraph $\mathcal{H}$ is a tree decomposition of $\mathcal{H}$.

Remark 2.4. The terms node and vertex will be used interchangeably to refer to a vertex of T.

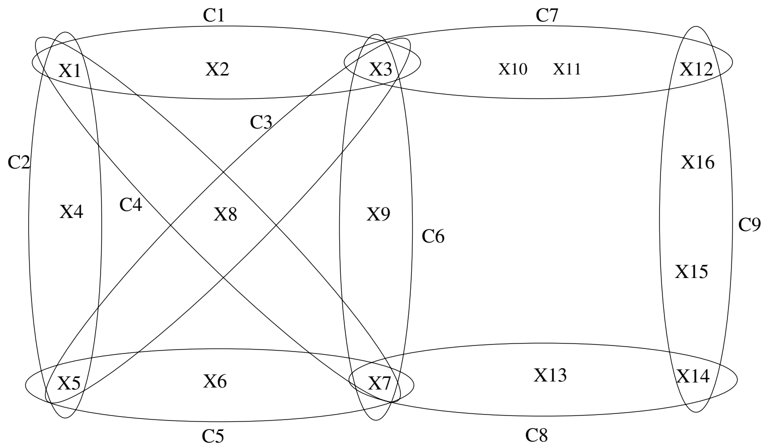

FIGURE 1. The constraint hypergraph of the CSP instance of Example 2.5.

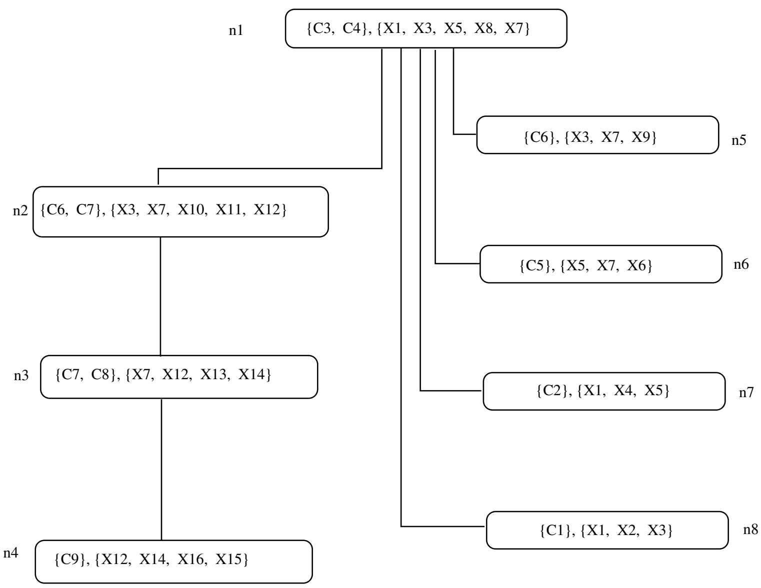

FIGURE 2. A 2-width generalized hypertree decomposition of the constraint hypergraph of Example 2.5.

Example 2.5. Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$ be a CSP instance defined as follows.

- $\mathcal{X} = \{X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}, X_{16}\}$ is the set of variables,
- $\mathcal{D} = \{D_1, \ldots, D_{16}\}$ where $D_i = \{0, 1, 2\}$ is the domain of the variable X_i $\forall i \in \{1, \ldots, 16\}$,
- $\mathcal{C} = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9\}$ is the set of constraints where
  $C_1 = (X_1, X_2, X_3)$, $C_2 = (X_1, X_4, X_5)$, $C_3 = (X_3, X_8, X_5)$, $C_4 = (X_1, X_8, X_7)$, $C_5 = (X_5, X_6, X_7)$,
  $C_6 = (X_3, X_9, X_7)$, $C_7 = (X_3, X_{10}, X_{11}, X_{12})$, $C_8 = (X_7, X_{13}, X_{14})$, $C_9 = (X_{12}, X_{16}, X_{15}, X_{14})$.

Figure 1 shows the constraint hypergraph associated with P and Figure 2 shows one of its generalized hypertree decompositions. The width of the decomposition is 2.

Two approaches have been proposed in the literature for constructing (generalized) hypertree decompositions, namely the exact and heuristic methods for which we give a brief description in the following.

- Exact methods: given a hypergraph $\mathcal{H} = \langle V, E \rangle$, the goal of exact algorithms is to find a hypertree decomposition of width hw less than or equal a constant k if it exists. The first exact algorithm *opt-k-decomp* proposed for the generation of optimal hypertree decomposition is due to Gottlob *et al.* [12]. This algorithm builds a hypertree decomposition in two steps: it finds if a hypergraph $\mathcal{H} = \langle V, E \rangle$ has a hypertree decomposition HD $= \langle T, \chi, \lambda \rangle$ with width less than or equal to a constant k then it tries to find a hypertree decomposition with the smallest possible width. The algorithm *opt-k-decomp* runs in $O(m^{2k}V^2)$ where m is the number of hyperedges, V is the number of vertices and k is a constant. Among the improvements of *opt-k-decomp*, we can cite *Red-k-decomp* [17], *det-k-decomp* [10] and the algorithm proposed by Subbarayan and Anderson [37] which is a backtracking version of *opt-k-decomp*. However up to now, exact methods have an important drawback: they need a huge amount of memory space and runtime. This makes the exact approach inefficient in practice for large instances. To overcome this limitation, some heuristics have been proposed for computing (generalized) hypertree decompositions.
- Heuristics and meta-heuristics: many heuristics and meta-heuristics have been proposed for computing (generalized) hypertree decompositions. Korimort [25] proposed a heuristic based on the vertex connectivity of the given hypergraph. Marko Samer [36] explored the use of branch decomposition heuristics for constructing hypertree decompositions. Musliu and Schafhauser [30] explored the use of Genetic Algorithms for generalized hypertree decompositions. In [2], the authors proposed a heuristic based on separators of the constraint hypergraph, *etc.*

Hereafter, we briefly present the algorithm Bucket Elimination (BE) for generating generalized hypertree decompositions because it is the one used in our experimental study. BE is successfully used to compute a tree decomposition of a given graph (or a primal graph of a hypergraph). BE has been extended by Dermaku *et al.* [6] to compute a generalized hypertree decomposition. The simple idea behind this extension derives from the fact that a GHD satisfies the properties of a tree decomposition. Consequently, for computing a generalized hypertree decomposition, BE proceeds as follows: it builds first a tree decomposition and then it creates the λ-*labels* for each node of this tree in order to satisfy the third condition of a generalized hypertree decomposition. This is done greedily by attempting to cover the variables of each node by hyperedges (constraints). Moreover the BE algorithm requires a variable ordering to be efficient.

2.1. Solving a CSP with a Generalized Hypertree Decomposition

To solve a CSP instance using a GHD, Gottlob *et al.* [11] have proposed the following algorithm (we called GLS for Gottlob, Leone and Scarcello).

Algorithm 1. Algorithm GLS.

Input: a CSP instance $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$
Output: a solution of P

1: Compute a GHD of the constraint hypergraph
2: Complete the obtained decomposition
3: For each node p, compute a new constraint relation R_p which is the projection on the variables in $\chi(p)$ of the join of the constraint relations in $\lambda(p)$
4: Process the obtained CSP by any efficient algorithm solving acyclic instances.

In this article we will consider the Yannakakis algorithm [39] for processing acyclic instances. Algorithm 2 which implements the 3rd and 4th steps of Algorithm 1 is called *Join Acyclic Solving algorithm*.

2.2. Join-Acyclic Solving Algorithm (*JAS*)

The Join Acyclic Solving (JAS) algorithm uses the following database operations for processing a GHD. Let R_i and R_j be two relations associated with the constraints C_i and C_j respectively.

- Join operation of two tuples $\bowtie^t$
  Let t_k and t_l be two tuples in R_i and R_j respectively. $t_k \bowtie^t t_l$ is a tuple t such that $\forall X_r \in C_i \cap C_j$ $t_k[X_r] = t_l[X_r]$, and $t[C_i] = t_k$ and $t[C_j] = t_l$.
- Join operation of relations $\bowtie$
  $R_i \bowtie R_j = \{t \mid t \; is \; a \; tuple \; defined \; on \; C_i \cup C_j \; with \; t[C_i] \in R_i \; and \; t[C_j] \in R_j\}$.[4]
  $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_k = ((\ldots(R_1 \bowtie R_2) \bowtie \ldots) \bowtie R_k)$.
- Semi-join operation $\ltimes$
  $R_i \ltimes R_j = \Pi_{C_i}(R_i \bowtie R_j) = \{t \in R_i \mid \exists t' \in R_j \; s.t \; \forall X_k \in C_i \cap C_j,\, t'[X_k] = t[X_k]\}$.

Let $t \in R_i$ and $t' \in R_j$ be two tuples. t and t' are said *compatible* if $\forall X_k \in C_i \cap C_j$, $t[X_k] = t'[X_k]$.

The Join Acyclic Solving (JAS) algorithm is formally described by Algorithm 2. After selecting an appropriate node ordering (line 1), all the subproblems associated with the nodes of the GHD are solved separately by the *join* and *projection* operations (lines 3–8). The resulting GHD is made *directional arc consistent* by using bottom up semi-join operations (lines 9–14) and the whole problem is finally solved in a backtrack-free way (lines 15–20).

Algorithm 2. Join Acyclic Solving Algorithm (*JAS*).

Input: a complete GHD $\langle T, \chi, \lambda \rangle$ associated with the CSP instance
Output: a solution $\mathcal{A}$ of the CSP if it is consistent

1: $\sigma \leftarrow (p_1, \ldots, p_l)$ /* node ordering with p_1 being the root and each node precedes all its children. l is the number of nodes of the considered GHD.*/
2: $\mathcal{A} \leftarrow \emptyset$
3: **for** each node p_i in σ **do**
4:     $R_{p_i} \leftarrow (\bowtie_{C_j \in \lambda(p_i)} R_j)[\chi(p_i)]$
5:     **if** $R_{p_i} = \emptyset$ **then**
6:         **Exit** /*the problem has no solution*/
7:     **end if**
8: **end for**
9: **for** i = l downto 2 **do**
10:     $R_{Parent(p_i)} \leftarrow R_{Parent(p_i)} \ltimes R_{p_i}$
11:     **if** $R_{Parent(p_i)} = \emptyset$ **then**
12:         **Exit** /*the problem has no solution*/
13:     **end if**
14: **end for**
15: Select a tuple t_{p_1} in R_{p_1}
16: $\mathcal{A} \leftarrow t_{p_1}$
17: **for** i = 2 to l **do**
18:     Select a tuple t_{p_i} in R_{p_i} such that $\mathcal{A}$ is compatible with t_{p_i} /*t_{p_i} necessarily exists*/
19:     $\mathcal{A} \leftarrow \mathcal{A} \bowtie^t t_{p_i}$
20: **end for**
21: Return $\mathcal{A}$

Unfortunately, JAS has not proved its practical efficiency for instances of realistic size because of the *join* operations being the source of memory explosion. In order to cope with this crucial drawback of JAS, we propose in this paper an optimized version based on a compression strategy. This version is referred as $CJAS$

[4] The square brackets denote the projection operator: $t[C_i]$ is the projection of tuple t on variables that belong to C_i.

for Compressed Join Acyclic Solving. Before presenting formally the *CJAS* algorithm in Section 4, we present in Section 3 the compression method used in this paper.

3. The compression strategy

In this section, we present the concepts relating to compressed tuple, compressed relation and compressed CSP. Accordingly to the definition of compressed relation, we introduce formally the concepts of compressed join and compressed semi-join operations.

3.1. Compressed CSP

Definition 3.1 (Compressed tuple [23])**.** Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$ be a CSP instance. Let R_i be the relation associated with the constraint $C_i = (X_{i_1}, \ldots, X_{i_{a_i}})$. A compressed tuple (*ctuple* for short) is a tuple $(D'_{i_1}, \ldots, D'_{i_{a_i}})$ where $D'_{i_j} \subseteq D_{i_j}$, $\forall j \in \{1, \ldots, a_i\}$. A tuple t accepted by a ctuple ct is any tuple $(v_{i_1}, \ldots, v_{i_{a_i}})$ where $v_{i_j} \in D'_{i_j}$ $\forall j \in \{1, \ldots, a_i\}$. D'_{i_j} is called the *c_value* of the variable X_{i_j}.

Notations: in the rest of this paper,

- $c\_value(X_{i_j}, ct)$ will denote the *c_value* of the variable X_{i_j} in the compressed tuple ct.
- $tuples(ct)$ will denote the set of tuples accepted by ct.

Now we present the definition of compressed relation.

Definition 3.2 (Compressed relation [23])**.** Let R_i be the relation associated with the constraint $C_i = (X_{i_1}, \ldots, X_{i_{a_i}})$. A compressed relation (crelation for short) R_i^c associated with R_i is a set of ctuples. Each tuple in R_i is accepted by a ctuple in R_i^c and each ctuple in R_i^c accepts only tuples in R_i.

Note that compressed representation of a relation is not always unique. So, in the sequel, the compressed relation associated with any relation is the one obtained by the compression algorithm presented in the Appendix of this article.

Example 3.3. Let R_1 be the relation associated with the constraint $C_1 = (X_1, X_2, X_3)$.

$$R_1 = \{(0,9,1), (1,9,2), (2,9,0), (3,9,3), (4,9,2), (5,9,3), (6,9,1), (7,9,1), (8,9,1), (9,9,3), (10,9,3)\}.$$

One possible compressed relation associated with R_1 can be:

$$R_1^c = \{(\{3,5,9,10\}, \{9\}, \{3\}), (\{0,6,7,8\}, \{9\}, \{1\}), (\{1,4\}, \{9\}, \{2\}), (\{2\}, \{9\}, \{0\})\}$$

Each ctuple in R_1^c is a compact representation of a subset of 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 R_1.

Definition 3.4 (Compressed CSP)**.** Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$ be a CSP instance, a *compressed CSP* instance associated with P is any $P' = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R}' \rangle$ where $|\mathcal{R}'| = |\mathcal{R}|$ and $\forall R_i \in \mathcal{R}, \exists R_i^c \in \mathcal{R}'$ such that R_i^c is a compressed version of R_i.

3.2. Compressed join and compressed semi-join operations

Now, we introduce the compressed join and compressed semi-join operations in order to manipulate compressed relations. For this purpose, we present the following definitions.

Definition 3.5 (Compatible compressed tuples)**.** Let R_i and R_j be two relations associated respectively with the constraints C_i and C_j. Let R_i^c and R_j^c be two crelations associated with R_i and R_j. Consider ct and ct' two ctuples in R_i^c and R_j^c respectively.

- ct is *all-compatible* with ct' if $\forall X_k \in C_i \cap C_j$, $c\_value(X_k, ct) \subseteq c\_value(X_k, ct')$.

- ct and ct' are *compatible* if there are at least two tuples t and t' accepted respectively by ct and ct' such that t and t' are compatible.
- ct and ct' are *incompatible* otherwise.

Definition 3.6 (Maximal under inclusion ctuple)**.** Let R_i be the relation associated with the constraint C_i. Let R_i^c be a crelation associated with the relation R_i. A ctuple ct is said maximal under inclusion in R_i^c if $\nexists ct'$ in R_i^c such that $\forall X_k \in C_i$, $c\_value(X_k, ct) \subseteq c\_value(X_k, ct')$.

Remark 3.7. In this paper, we will consider only ctuples which are maximal under inclusion.

3.2.1. Compressed join operation

First, we introduce the compressed join of two ctuples.

Definition 3.8 (Compressed join of two ctuples)**.** Let R_i and R_j be two relations associated with the constraints C_i and C_j respectively. Let R_i^c and R_j^c be two crelations associated with R_i and R_j and consider ct and ct' two ctuples of R_i^c and R_j^c respectively.

The *compressed join* of ct and ct', denoted by $ct \bowtie^{ct} ct'$ is a ctuple ct'' defined on $C_k = C_i \cup C_j$ as follows:

- $\forall X_r \in C_k \setminus C_j,\ c\_value(X_r, ct'') = c\_value(X_r, ct)$.
- $\forall X_r \in C_k \setminus C_i,\ c\_value(X_r, ct'') = c\_value(X_r, ct')$.
- $\forall X_r \in C_i \cap C_j,\ c\_value(X_r, ct'') = c\_value(X_r, ct) \cap c\_value(X_r, ct')$.
  If $c\_value(X_r, ct'') = \emptyset$ then $ct \bowtie^{ct} ct' = \emptyset$.

We then present the compressed join of two compressed relations.

Definition 3.9 (Compressed join of two crelations)**.** Let R_i and R_j be two relations associated with the constraints C_i and C_j respectively. Let R_i^c and R_j^c be two crelations associated with R_i and R_j. The *compressed join* of R_i^c and R_j^c is a crelation R_k^c defined on $C_k = C_i \cup C_j$. It will be denoted by $R_i^c \bowtie^{cr} R_j^c$ and defined as the union of all pairs of possible compressed join of ctuples of R_i^c with those of R_j^c. Formally, $R_k^c = R_i^c \bowtie^{cr} R_j^c = \{ct'' = ct \bowtie^{ct} ct' | ct''$ maximal under inclusion in R_k^c, $\forall ct \in R_i^c, \forall ct' \in R_j^c\}$.

Example 3.10. Let R_1, R_2 and R_3 be three relations associated with the constraints C_1, C_2 and C_3 respectively.

$C_1 = (X_1, X_2, X_3)$,
$C_2 = (X_1, X_2, X_4)$,
$C_3 = (X_1, X_2, X_3, X_4)$,
$R_1 = \{(0,9,1),(1,9,2),(2,9,0),(3,9,3),(4,9,2), (5,9,3), (6,9,1),(7,9,1),(8,9,1),(9,9,3)\}$,
$R_2 = \{(0,9,3),(1,8,3),(1,9,3),(5,8,3),(6,9,3), (7,9,3)\}$,
$R_3 = R_1 \bowtie R_2 = \{(0,9,1,3),(6,9,1,3),(7,9,1,3),(1,9,2,3)\}$
Let R_1^c, R_2^c and $R_3^c = R_1^c \bowtie^{cr} R_2^c$ be three crelations associated respectively with R_1, R_2 and R_3.
$R_1^c = \{(\{3,5,9\},\{9\},\{3\}),(\{0,6,7,8\},\{9\},\{1\}), (\{1,4\},\{9\},\{2\}),(\{2\},\{9\},\{0\})\}$,
$R_2^c = \{(\{0,1,6,7\},\{9\},\{3\}),(\{1,5\},\{8\},\{3\})\}$ and
$R_3^c = \{(\{0,6,7\},\{9\},\{1\},\{3\}),(\{1\},\{9\},\{2\},\{3\})\}$.

Observe that $tuples((\{0, 6, 7\}, \{9\}, \{1\}, \{3\})) \cup tuples((\{1\}, \{9\}, \{2\}, \{3\}) = R_3$.

Remark 3.11. $R_1^c \bowtie^{cr} R_2^c \bowtie^{cr} \ldots \bowtie^{cr} R_k^c = (\ldots(R_1^c \bowtie^{cr} R_2^c) \bowtie^{cr} \ldots) \bowtie^{cr} R_k^c$.

Remark 3.12. The compressed join operation generalizes naturally the classical join operation. Indeed, a classical tuple is a compressed tuple where the c_value of each variable contains only one value.

3.2.2. Compressed semi-join operation

We present the compressed semi-join of two crelations and an algorithm for computing this operation.

Definition 3.13 (Compressed semi-join of two crelations)**.** Let R_i and R_j be two relations associated with the constraints C_i and C_j respectively. Let R_i^c and R_j^c be two crelations associated respectively with R_i and R_j. The *compressed semi–join* of R_i^c and R_j^c, denoted by $R_i^c \ltimes^{cr} R_j$, is a crelation $R_i'^c$ defined on C_i and containing a set of ctuples representing all and only the tuples, accepted by the ctuples in R_i^c, which have compatible ones in the tuples accepted by the ctuples in R_j^c. More formally, the compressed join of two crelations R_i^c and R_j^c can be written as follows: $R_i'^c{=}R_i^c \ltimes^{cr} R_j^c = \{ct|\ ct$ is a ctuple defined on C_i such that: (ct is maximal under inclusion in $R_i'^c$) and $(\exists ct' \in R_i^c\,|(\forall X_k \in C_i, c\_value(X_k, ct) \subseteq c\_value(X_k, ct'))$ and $(\exists\, ct'' \in R_j^c\,|ct\, is\, all$-$compatible\, with\, ct'')\}$.

Example 3.14. Consider the constraints C_1 and C_2 of Example 3.10.

$$R_1^c \ltimes^{cr} R_2 = \Pi_{C_1}(R_1^c \bowtie^{cr} R_2) = \{(\{0,6,7\},\{9\},\{1\}),(\{1\},\{9\},\{2\})\}$$

3.2.3. Algorithm for computing compressed semi-join of two crelations

In order to present Algorithm 3 computing the compressed semi-join operation, we have to introduce the notion of a support of a tuple.

Definition 3.15 (A support for a tuple in a ctuple)**.** Let R_i^c and R_j^c be two crelations. Let ct be a ctuple in R_j^c and let t be a tuple accepted by a ctuple in R_i^c. ct will be said a *support* for t if ct accepts at least one tuple t' such that t and t' are compatible.

Algorithm 3. Compressed semi-join operation.

Input: two crelations R_i^c and R_j^c
Output: $R_i'^c = R_i^c \ltimes^{cr} R_j^c$

$R_i'^c \leftarrow \emptyset$
while $R_i^c \neq \emptyset$ **do**
  $ct_i \leftarrow head(R_i^c)$ /* The function *head* returns the first ctuple of R_i^c */
  $R_i^c \leftarrow R_i^c - \{ct_i\}$
  $ct_j \leftarrow head(R_j^c)$
  $found \leftarrow false$
  while $(notfound)$ and $(ct_j \neq NIL)$ **do**
    if compatible(ct_i, ct_j) **then**
      $const\_all\_compatible\_ctuple(ct_i,\ ct_j,\ ct_{\text{comp}})$
      $R_i'^c \leftarrow R_i'^c \cup \{ct_{\text{comp}}\}$
      if $next(ct_j) \neq NIL$ **then**
        $DIFF \leftarrow$ difference(ct_i, ct_{comp})
        $R_i^c \leftarrow R_i^c \cup\ DIFF$
      end if
      $found \leftarrow true$
    else
      $ct_j \leftarrow next(ct_j)$
    end if
  end while
end while

In Algorithm 3, the crelation R_i^c is managed as a stack. The crelation $R_i'^c$ is initially empty. It will contain, at the end, a set of ctuples accepting all the tuples represented by the ctuples of R_i^c which have a support in R_j^c. At each step, the first ctuple of R_i^c is moved to ct_i and the current ctuple of R_j^c is saved in ct_j. The main difference between computing the compressed semi-join and computing the semi-join of two relations comes from the fact that a ctuple is not a simple tuple but indeed an abstraction of a subset of simple tuples. So, a ctuple ct_i of R_i^c can be compatible but not all-compatible with a ctuple ct_j of R_j^c. This means that some tuples accepted by ct_i must be derived to make the resulting ctuple ct_{comp} all-compatible with ct_j and to be inserted in $R_i'^c$. However, a set of ctuples representing the other tuples of ct_i must be created and pushed in R_i^c because they can be compatible with other ctuples (after ct_j) in R_j^c if such ctuples exist.

To deal with this technical particularity related to the definition of a ctuple, Algorithm 3 proceeds as follows. If the ctuples ct_i and ct_j are compatible then, according to Algorithm 4, a ctuple ct_{comp} derived from ct_i and corresponding to a ctuple all-compatible with ct_j is built and pushed in $R_i'^c$ (lines 8–10).

Algorithm 4. const_all_compatible_ctuple.

Input: two ctuples $ct_i \in R_i^c$ and $ct_j \in R_j^c$
Output: a ctuple ct_{comp}

for each variable X_k in C_i **do**
  if $X_k \in C_i \cap C_j$ **then**
    $c\_value(X_k, ct_{\text{comp}}) \leftarrow c\_value(X_k, ct_i) \cap c\_value(X_k, ct_j)$
  else
    $c\_value(X_k, ct_{\text{comp}}) \leftarrow c\_value(X_k, ct_i)$
  end if
end for

ct_{comp} represents all the tuples accepted in ct_i and having compatible tuples accepted in ct_j.

If ct_j is not the last ctuple of R_j^c, then Algorithm 5 builds a set *DIFF* of ctuples incompatible with the current ctuple ct_j (lines 11-12) and representing all and only the tuples of ct_i which have not compatible ones in the tuples accepted by ct_j. These new ctuples are pushed in R_i^c (line 13) in order to be tested against the next ctuples of R_j^c after ct_j.

To build the set *DIFF*, Algorithm 5 is called giving rise to the ctuples derived from ct_i and incompatible with ct_j w.r.t. each variable belonging both to C_i and C_j.

Algorithm 5. Difference.

Input: two ctuples ct_i and ct_{comp}
Output: a set *DIFF* of ctuples accepting the tuples of ct_i which are not accepted in ct_{comp}

$DIFF \leftarrow \emptyset$
for each variable $X_k \in C_i \cap C_j$ **do**
  if $(c\_value(X_k, ct_i) \setminus c\_value(X_k, ct_{\text{comp}})) \neq \emptyset$ **then**
    create_incompatible_ctuple(ct_i, ct_{comp}, X_k, ct_{X_k})
    $DIFF \leftarrow DIFF \cup \{ct_{X_k}\}$
  end if
end for
return *DIFF*

Remark 3.16. The compressed semi-join operation generalizes the classical semi-join operation on relations. Indeed, a classical tuple is a compressed tuple with all the c_values having only one value. So, when computing the compressed semi-join operation of two classical tuples t and t', ct_{comp} accepts only one tuple which is t if t and t' are compatible else ct_{comp} is empty. The set *DIFF* is always empty because there is no derived ctuples when t and t' are compatible.

Algorithm 6. create_incompatible_ctuple

Input: two ctuples ct_i and ct_{comp}
Input: a variable $X_k \in C_i \cap C_j$
Output: a ctuple ct_{X_k}

$c\_value(X_k, ct_{X_k}) \leftarrow c\_value(X_k, ct_i) \setminus c\_value(X_k, ct_{\text{comp}})$
for each variable $X_l \in C_i \setminus C_i \cap C_j$ **do**
  $c\_value(X_l, ct_{X_k}) \leftarrow c\_value(X_l, ct_i)$
end for
for each variable $X_l \in C_i \cap C_j$ and $X_l \neq X_k$ **do**
  if X_l is before X_k in C_i **then**
    $c\_value(X_l, ct_{X_k}) \leftarrow c\_value(X_l, ct_{\text{comp}})$
  else
    $c\_value(X_l, ct_{X_k}) \leftarrow c\_value(X_l, ct_i)$
  end if
end for

Example 3.17 (Computing a compressed semi-join operation)**.** Let C_1 and C_2 be two constraints where:

$C_1 = (X_1, X_2, X_3, X_4)$,

$C_2 = (X_5, X_6, X_3, X_4, X_7)$,

$S = C_1 \cap C_2 = (X_3, X_4)$.

Let R_1^c and R_2^c be two crelations associated with R_1 and R_2 respectively:

$R_1^c = \{(\{3,5,9\},\{9,2\},\{3,4\},\{1,2,4\}),(\{0,6,7,8\},\{9\},\{1\},\{3,4\})\}$;

$R_2^c = \{(\{0,1,6,7\},\{9\},\{3,5\},\{1,2,3\},\{3,4\}),(\{1,5\},\{8\},\{3,4\}, \{1,2,4\},\{2,4\})\}$;

$R_1'^c = R_1^c \ltimes^{cr} R_2^c$ defined on C_1 is computed as follows.

(1) (line 1) Initially, $R_1'^c = \emptyset$.
(2) (lines 2–20) the first ctuple $(\{3,5,9\},\{9,2\},\{3,4\},\{1,2,4\})$ in R_1^c is moved to ct_i and the first ctuple $(\{0,1,6,7\},\{9),\{3,5\},\{1,2,3\},\{3,4\})$ in R_2^c is saved in ct_j. $found$ is assigned to false.
(3) (lines 7–19) ct_i and ct_j are compatible then
- (lines 9–18) a ctuple ct_{comp} all-compatible with ct_j is derived from ct_i. Variables X_3 and X_4 belonging to S take the c_values: $c\_value(X_3, ct_i) \cap c\_value(X_3, ct_j) = \{3,4\} \cap \{3,5\} = \{3\}$ and $c\_value(X_4, ct_i) \cap c\_value(X_4, ct_j) = \{1,2,4\} \cap \{1,2,3\} = \{1,2\}$ respectively. The other variables in $C_1 \setminus S$ keep their corresponding c_values in ct_i. The derived compressed ctuple $ct_{\text{comp}} = (\{3,5,9\},\{9,2\},\{3\},\{1,2\})$ is then pushed in $R_1'^c$.
- (lines 11-14) as ct_i and ct_j are not all-compatible and ct_j is not the last ctuple of R_2^c, the set DIFF of ctuples, incompatible with ct_j, is computed thanks to Algorithm 5. There are two intersection variables X_3 and X_4 making the set DIFF containing ctuples ct_{X_3} and ct_{X_4}. DIFF $= \{ct_{X_3}, ct_{X_4}\} = \{(\{3,5,9\},\{9,2\},\{4\},\{1,2,4\}),(\{3,5,9\},\{9,2\},\{3,4\},\{4\})\}$. Because ctuple $(\{3,5,9\},\{9,2\},\{4\},\{4\})\}$ is already present in ct_{X_3} hence DIFF becomes $\{(\{3,5,9\},\{9,2\},\{4\},\{1,2,4\}),(\{3,5,9\},\{9,2\},\{3\},\{4\})\}$. DIFF is then pushed in R_1^c.

This loop leads to the following configuration:
$R_1'^c = \{(\{3,5,9\},\{9,2\},\{3\},\{1,2\})\}$;
$R_1^c = \{(\{3,5,9\},\{9,2\},\{4\},\{1,2,4\}),(\{3,5,9\},\{9,2\},\{3\},\{4\}),$
$(\{0,6,7,8\},\{9\},\{1\},\{3,4\})\}$;

(4) The same process (beginning in 2) is repeated for the new ctuple $ct_i = (\{3,5,9\},\{9,2\},\{4\},\{1,2,4\})$ removed from R_1^c compatible with the next ctuple $ct_j = (\{1,5\},\{8\},\{3,4\},\{1,2,4\},\{2,4\})$ of R_2^c then
   - $ct_{\text{comp}} = (\{3,5,9\},\{9,2\},\{4\},\{1,2,4\})$ all-compatible with ct_j is pushed in $R_1'^c$.
   - the set DIFF is not created because ct_j is the last ctuple of R_2^c. This loop leads to the following configuration:
     $R_1'^c = \{(\{3,5,9\},\{9,2\},\{3\},\{1,2\}),(\{3,5,9\},\{9,2\},\{4\},\{1,2,4\})\}$;
     $R_1^c = \{(\{3,5,9\},\{9,2\},\{3\},\{4\}),(\{0,6,7,8\},\{9\},\{1\},\{3,4\})\}$;

(5) The first ctuple $(\{3,5,9\},\{9,2\},\{3\},\{4\})$ is removed from R_1^c to ct_i which is compatible with $ct_j = \{1,5\},\{8\},\{3,4\},\{1,2,4\},\{2,4\})$ in R_2^c then
   - ctuple $ct_{\text{comp}} = \{(\{3,5,9\},\{9,2\},\{3\},\{4\})\}$ all-compatible with ct_j is created.
   - $DIFF = \emptyset$ because ct_i is all-compatible with ct_j. Then ct_{comp} is moved to $R_1'^c$. This loop leads to the following configuration:
     $R_1'^c = \{(\{3,5,9\},\{9,2\},\{3\},\{1,2\}),(\{3,5,9\},\{9,2\},\{4\},\{1,2,4\}),$
     $(\{3,5,9\},\{9,2\},\{3\},\{4\})\}$; $R_1^c = \{(\{0,6,7,8\},\{9\},\{1\},\{3,4\})\}$.

(6) Finally the last unique ctuple $ct_i = (\{0,6,7,8\},\{9\},\{1\},\{3,4\})$ in R_1^c is considered. It has no compatible ctuple in R_2^c making $R_1'^c$ unchanged and $R_1^c = \emptyset$.

The result of the compressed semi-join operation is then
$R_1'^c = \{(\{3,5,9\},\{9,2\},\{3\},\{1,2\}),(\{3,5,9\},\{9,2\},\{4\},\{1,2,4\}),(\{3,5,9\},\{9,2\},\{3\},\{4\})\}$.

Remark 3.18. Observe that the tuples accepted by the ctuples in $R_1'^c$ are exactly the tuples accepted by the ctuples in R_1^c which have a support in the ctuples of R_2^c.

4. *CJAS*: The Compressed version of *JAS*

In this section we present the compressed version of *JAS* called *CJAS*. Both algorithms are composed of the same main steps, but the ones in *CJAS* are submitted to some modifications deriving from the compressed representation of relations. The compressed version is formally described by Algorithm 7.

4.1. Presentation of *CJAS*

This algorithm proceeds as follows:

(1) The procedure *Compress_Csp* (line 1) transforms each relation R_i of a constraint C_i in $\mathcal{C}$ into a compressed relation R_i^c according to Definitions 3.1 and 3.2. The method proposed by Katsirelos *et al.* in [23] is used to compress the relations and it is presented in Appendix.
(2) The nodes of T are organized in a list σ according to the depth-first (pre-order) traversal (line 2).
(3) At each node p_i of T, the subproblem is separately solved (lines 4–9) by computing, according to the Definition 3.9, the compressed join of the constraint crelations in $\lambda(p_i)$. The obtained crelation is then projected on the variables in $\chi(p_i)$ and each ctuple of this crelation represents a set of solutions for this subproblem.
(4) Once all the subproblems have been solved, the resulting GHD is made *directional arc consistent* in the downto loop (lines 10–15) using the compressed semi-join operations as follows.
Let p and q be two nodes of GHD such that q is the parent of p. R_p^c (resp. R_q^c) is the crelation obtained by the compressed join of the crelations associated with the constraints in $\lambda(p)$ (resp. $\lambda(q)$). The compressed semi-join of R_q^c and R_p^c consists of removing from the ctuples in R_q^c all the tuples which have no compatible ones (tuples) accepted in the ctuples of R_p^c.
(5) Finally, the whole CSP instance is solved in *backtrack-free* way (lines 16–21).

Algorithm 7. Compressed Join Acyclic Solving (*CJAS*).

Input: a complete GHD $\langle T, \chi, \lambda \rangle$ associated with a given CSP
Output: a solution $\mathcal{A}$ of the CSP if it is consistent

1: *Compress_Csp (CSP)*
2: $\sigma = (p_1, p_2, \ldots, p_l)$ /* a node ordering of T, p_1 is the root and each node precedes all its children */
3: $\mathcal{A} \leftarrow \emptyset$
4: **for** $i = 1$ to l **do**
5: $\quad R^c_{p_i} \leftarrow (\bowtie^{\mathbf{cr}}{}_{C_j \in \lambda(p_i)} R^c_j)[\chi(p_i)]$ /* $R^c_{p_i}$ is a compressed relation associated with the node p_i*/
6: $\quad$ **if** $R^c_{p_i} = \emptyset$ **then**
7: $\quad\quad$ **Exit** /*the problem has no solution*/
8: $\quad$ **end if**
9: **end for**
10: **for** $i = l$ downto 2 **do**
11: $\quad R^c_{Parent(p_i)} \leftarrow R^c_{Parent(p_i)} \ltimes^{cr} R^c_{p_i}$
12: $\quad$ **if** $R^c_{Parent(p_i)} = \emptyset$ **then**
13: $\quad\quad$ **Exit** /*the problem has no solution */
14: $\quad$ **end if**
15: **end for**
16: Select a tuple t_{p_1} accepted by one ctuple in $R^c_{p_1}$ /*t_{p_1} necessarily exists*/
17: $\mathcal{A} \leftarrow t_{p_1}$
18: **for** i = 2 to l **do**
19: $\quad$ Select a tuple t_{p_i} accepted by one ctuple in $R^c_{p_i}$ such that $\mathcal{A}$ is compatible with t_{p_i}
20: $\quad \mathcal{A} \leftarrow \mathcal{A} \bowtie^t t_{p_i}$
21: **end for**
22: return $\mathcal{A}$

The *CJAS* efficiency depends clearly on the quality of the compression (line 1). Indeed, when the compression ratio is bad, *CJAS* behaves like *JAS*. In this work, we have considered the algorithm proposed in [23] with the MaxFreq heuristic for compressing constraint relations because it gives acceptable compression ratio for the benchmarks used in our experimental study.

4.2. Theoretical properties of *CJAS*

4.2.1. Correctness of CJAS

In order to prove the correctness of the *CJAS* algorithm, we have to establish the following lemmas. First, we prove that the set of the tuples accepted by a compressed join of two ctuples ct_i and ct_j is exactly the join of tuples accepted by ct_i with those accepted by ct_j.

Lemma 4.1. *Let R^c_i and R^c_j be two crelations associated with the constraints C_i and C_j respectively. Let ct_i and ct_j be two compressed tuples in R^c_i and R^c_j respectively. If $T_i = tuples\ (ct_i)$ and $T_j = tuples\ (ct_j)$ then $T_i \bowtie T_j = tuples\ (ct_i \bowtie^{ct} ct_j)$.*

Proof.

(i) Let $t \in T_i \bowtie T_j$ then according to the definition of join operation there exist $t_1 \in T_i$ and $t_2 \in T_j$ such that $t_1 = t[C_i]$, $t_2 = t[C_j]$ and $\forall X_k \in C_i \cap C_j, t_1[X_k] = t_2[X_k]$. Then t_1 is accepted by ct_i and t_2 is accepted by ct_j. Following the definition of the compressed join of two ctuples, t is accepted by $ct_i \bowtie^{ct} ct_j$ and then $t \in tuples(ct_i \bowtie^{ct} ct_j)$.

(ii) Let $t \in tuples(ct_i \bowtie^{ct} ct_j)$ then t is accepted by the ctuple $ct = ct_i \bowtie^{ct} ct_j$ defined on $C_i \cup C_j$. According to the definition of $\bowtie^{ct}$, there exist $t_1 \in tuples(ct_i)$ and $t_2 \in tuples(ct_j)$ such that $t_1 = t[C_i]$, $t_2 = t[C_j]$ and $\forall X_k \in C_i \cap C_j, t_1[X_k] = t_2[X_k]$. So, $t \in T_i \bowtie T_j$. □

Lemma 4.2 ensures that at each iteration of Algorithm 3, the tuples accepted by the ctuple ct_{comp} together with the ones accepted by the ctuples in *DIFF* are exactly the tuples accepted by the ctuple ct_i. Moreover, ct_{comp} is incompatible with all the ctuples in *DIFF* and is all-compatible with ct_j.

Lemma 4.2. *Let ct_i and ct_j be two compressed tuples in R_i^c and R_j^c respectively.*
If ct_i and ct_j are compatible and ct_{comp} is the ctuple built by Algorithm 4 then:

(1) *ct_{comp} is all-compatible with ct_j.*
(2) *All the ctuples in* DIFF *(computed by Algorithm 5) are incompatible with ct_j.*
(3) *The set of tuples accepted by ct_i is the union of the tuples accepted by ct_{comp} and those accepted by the ctuples in* DIFF.

Proof.

(1) By construction ct_{comp} is all-compatible with ct_j.
(2) By construction each compressed tuple ct_k in DIFF is incompatible with ct_j.
(3) Let t_i be a tuple accepted by ct_i. Consider that t_i is not accepted by ct_{comp} nor by all the ctuples in *DIFF*. If t_i is not accepted by ct_{comp} then there is in t_i a value v for a variable $X_{i_k} \in C_i \cap C_j$ such that $v \notin c\_value(X_{i_k}, ct_j)$. However, by construction there is a ctuple in *DIFF* (see Algorithm 5) associated with X_{i_k} which accepts any combination of the value v for X_{i_k} with all other possible values for all other variables in $C_i \setminus \{X_{i_k}\}$. Consequently, t_i is either accepted in ct_{comp} or accepted by a ctuple in *DIFF*. So, the union of the tuples accepted by ct_{comp} and all those accepted by all the ctuples in *DIFF* is exactly the set of tuples accepted by ct_i. □

In Lemma 4.3, we prove that the tuples which are removed by the compressed semi-join operation (tuples which are not accepted in $R_i'^c = R_i^c \ltimes^{cr} R_j^c$) are incompatible with all the tuples accepted by the ctuples of the second relation R_j^c and hence cannot be part of any global solution for the CSP instance.

Lemma 4.3. *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$ be a CSP instance. Let R_i^c and R_j^c be two crelations associated respectively with two constraints C_i and C_j in $\mathcal{C}$. Let $R_i'^c$ be a crelation such that $R_i'^c = R_i^c \ltimes^{cr} R_j^c$. Let M be the set of tuples accepted in R_i^c and not in $R_i'^c$. For each solution **sol** of the CSP instance, $\boldsymbol{sol}[C_i] \notin M$.*

Proof. Suppose that there is a global solution **sol** for $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$ such that $sol[C_i] = t_i$ and $sol[C_j] = t_j$. Hence, there is in R_i^c a ctuple ct which accepts t_i and there is in R_j^c a ctuple ct' which accepts t_j such that t_i is compatible with t_j. So, we have to show that there is necessarily a ctuple ct_{comp} accepting t_i in $R_i'^c$. At each iteration of Algorithm 3 where ct_i (the head of R_i^c) accepts t_i, two situations are possible for ct_j.

- $ct_j = ct'$, in this case a ctuple ct_{comp} accepting t_i is created and inserted in $R_i'^c$ (lines 9–10).
- $ct_j = ct_j'$ (ct_j' before ct' in R_j^c), we distinguish three cases.
  (1) ct_j' is incompatible with ct_i. In this case, the next ctuple ct_j'' after ct_j' in R_j^c is considered and thanks to the loop while (line 7) the same process is repeated with the configuration $\langle ct_i, ct_j'' \rangle$.
  (2) ct_j' is a support for t_i. In this case a ctuple ct_{comp} accepting t_i is built and inserted in $R_i'^c$.
  (3) ct_j' is not a support for t_i and ct_i is compatible with ct_j'. In this case a new ctuple ct_i' accepting t_i is derived from ct_i and pushed in R_i^c (*via* the set *DIFF*). Thanks to Lemma 4.2 and thanks to the loop while (line 2) we will reach a configuration such that we have at the head of R_i^c a ctuple $ct_i = ct_i'$ accepting t_i and the same process is then repeated.

So at the end, there is necessarily a ctuple $ct_{\text{comp}} \in R_i'^c$ accepting t_i and then (thanks to Lem. 4.2) $t_i \notin M$. Finally, all the tuples which are in R_i^c and not in $R_i'^c$ cannot be part of a global solution for the CSP instance. □

Proposition 4.4. CJAS *is correct w.r.t. the* JAS *algorithm.*

Proof. In order to prove the correctness of *CJAS w.r.t. JAS*, we have to prove the correctness of each step.

(1) **Step 1:** The procedure Compress_Csp is correct. The proof is given in [23].
(2) **Step 2:** The compressed join of compressed relations is correct.
Since $tuples((\bowtie^{\mathbf{cr}}{}_{C_j \in \lambda(p_i)} R_j^c)[\chi(p_i)]) = (\bowtie_{C_j \in \lambda(p_i)} R_j)[\chi(p_i)]$ (thanks to Lem. 4.1), the set of the tuples obtained by the join operation on the relations in the λ label of each node p_i is equivalent to the set of the tuples accepted by the ctuples of the crelations computed by the compressed join operation of the crelations in the λ label of p_i. Hence at the end of this step, the obtained acyclic compressed CSP instance has the same solutions as the original CSP instance P.
(3) **Step 3:** The compressed semi-join of crelations is correct. Indeed, thanks to Lemma 4.3, all the tuples removed at the crelation associated with the parent node of each node p_i cannot be part of a global solution of the acyclic CSP instance represented by the GHD.
(4) **Step 4:** This step is obviously correct. It consists just of choosing a tuple t compatible with the previous ones at each node, t necessarily exists thanks to the compressed semi-join operation.

Hence the correctness of *CJAS* derives from the correctness of *JAS*. □

4.2.2. Complexity analysis of CJAS

In this subsection, we assess the space and time complexities of the *CJAS* algorithm.

Notations

- S is the size of the compressed CSP instance.
- cr is the size of the largest compressed relation.
- $ghtw$ is the generalized hypertree width of the decomposition *GHD* returned by *BE*.
- l is the number of nodes of the hypertree.
- $cval$ is the size of the largest c_value.
- a is the largest arity of constraints.
- m is the number of constraints.
- r is the size of the largest relation (maximum number of classical tuples).
- d is the size of the largest domain of a variable.
- s' is the size of the largest separator between two successive nodes in the hypertree.
- cr_{node} is the size of the largest crelation associated with nodes.

Theorem 4.5. *The space complexity of* CJAS *is in* $O(S \cdot cr^{ghtw})$.

Proof. Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$ be a CSP instance and let $P' = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R}' \rangle$ be one of its compressed representations. Let C_i be a constraint and let ct_j be a compressed tuple in R_i^c (R_i is the relation associated with C_i).

- The size of ct_j is $s_{ct_j} = \sum_{k=1}^{|C_i|} |c\_value(X_k, ct_j)| \leq a \cdot cval$.
- The size of R_i^c is $s_{C_i} = \sum_{j=1}^{|R_i^c|} s_{ct_j} \leq a \cdot cval \cdot cr$.
- The cost of the compressed relations is bounded by $\sum_{i=1}^{i=m} s_{C_i} \leq a \cdot cval \cdot cr \cdot m$.
- The number of ctuples (which are the solutions of the subproblem) at each node of the *GHD is bounded by* cr^{ghtw}. *Hence, the size of the compressed relation at each node is bounded by* $a \cdot cval \cdot cr^{ghtw}$.

Finally, the space complexity of *CJAS* is bounded by $(l \cdot a \cdot cval \cdot cr^{ghtw} + a \cdot cval \cdot cr \cdot m)$=O$(S \cdot cr^{ghtw})$. □

Theorem 4.6. *The time complexity of* CJAS *is in* $O(l \cdot s' \cdot cval^{s'+2} \cdot cr^{2ghtw})$.

Proof.

- The first step (line 1) of *CJAS*, dealing with compressing relations can be done in O$(m.r.a^2.d)$. Indeed, compressing one relation with the method used in this article can be done in O$(r.a^2.d)$ [23].
- The second step (line 2) can be done in O(l).

- Since the cost of computing intersection of two c_values can be done in $O(cval^2)$, then the cost of the third step (lines 4–9) of *CJAS* dealing with compressed join operation of the compressed constraint relations at each node can be done in $O(a \cdot cval^2 \cdot cr^{ghtw})$.
- Let R_p^c and R_q^c two compressed relations computed for the node p and q such that q is the parent of p. In the worst case, each ctuple in R_q^c can give rise to $cval^{s'}$ other ctuples. The cost of the compressed semi-join operation of R_q^c and R_p^c (fourth step lines 10-15) is in $O(cval^{s'} \cdot s' \cdot cval^2 \cdot cr_{node}^2)$. This cost is bounded by $\mathrm{O}(cval^{s'} \cdot s' \cdot cval^2 \cdot cr^{2ghtw})$.
  So, the time complexity of the third step of *CJAS* is bounded by $O(l \cdot cval^{s'+2} \cdot s' \cdot cr^{2ghtw})$.
- The last step consists of building a solution (lines 16–21).
  The cost of this operation is in $\mathrm{O}(cval \cdot s' \cdot cr \cdot l)$.

Finally, the time complexity of *CJAS* is in $O(l \cdot cval^{s'+2} \cdot s' \cdot cr^{2ghtw})$. □

5. Experimental results

5.1. Environment considerations

The Compressed Join Acyclic Solving (*CJAS*) and the Join Acyclic Solving (*JAS*) algorithms are implemented using the C++ language and the Bucket Elimination (*BE*) algorithm [6] is used to compute a *GHD* for each CSP instance. *BE* was evaluated in [6] as the best algorithm giving a nearly optimal generalized hypertree decomposition within a reasonable CPU time. For making complete the decompositions, we have used the method proposed in [11]: for each constraint C_i, not strongly covered in the hypertree, choose a node p of the hypertree such that the *scope* of C_i is a subset of $\chi(p)$ (this node must exist by condition (1) of Def. 2.3) and create a special leaf node q as a child of p with $\chi(q)$ is the set of the variables in the scope of C_i and $\lambda(q) = \{C_i\}$. The experiments were run on a Core (TM) 2 Duo CPU T5670 @ 1.80 GHZ with 2 GB of RAM under Linux. These tests have been conducted with benchmarks presented at the Third International CSP 2008 Solver Competition[5]. Some of these problems are original CSP problems and some are CSP formulation of Satisfiability problems from the DIMACS Benchmark challenge on cliques, satisfiability and coloring problems[6].

In all the tables of results, $|\mathcal{X}|$ is the number of variables, $|\mathcal{C}|$ is the number of constraints, *ghtw* is the width of the GHD returned by *BE* for the constraint hypergraph, *nb_nodes*[7] is the number of nodes of the GHD and r is the maximum cardinality of the constraint relations. MO means that the method runs out of memory for the considered instance. All the times are given in seconds and for each instance the time out (TO) was fixed to 1800 s. The symbol / means that *BE* cannot decompose the constraint hypergraph. For the *CJAS* and *JAS* algorithms, all the reported times include the decomposition time of the *BE* algorithm to build the GHD and the time of making complete[8] the obtained decomposition. The times reported for *CJAS* include also the time spent by the compression algorithm for the considered CSP instance.

5.2. Description of benchmarks

The following benchmarks presented at the Third International CSP Solver Competition are considered:

(1) original CSP problems:
   - Renault and Modified Renault series: The original "Renault problem" is obtained from a Renault Megane configuration and appears under two forms: normalized and simple. This series involves large relations of high arity. The "Modified Renault" series contains 50 structured instances involving domains with up to 42 possible values. The largest constraint relation contains 48 721 tuples.

[5] http://www.cril.univ-artois.fr/CPAI08/.
[6] http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html.
[7] The special nodes added to complete GHD are not included in this number.
[8] Unless otherwise stated this time is $\simeq 0$.

- Large bdd class: this series contains 35 quasi random instances (random containing a small structure). The maximum arity of the constraints is 15 and the maximum number of tuples for each constraint relation is 7086.

(2) CSP formalization of Satisfiability problems from DIMACS (domain of all variables is Boolean):
- Ssa series: This series contains 8 instances encoding circuit fault analysis: single-stuck-at fault – 4 instances are satisfiable, 4 instances are unsatisfiable. The maximum arity of the constraints is less than 6 and the largest constraint relation contains 63 tuples.
- Pret series: This series contains 8 instances encoding 2-coloring forced to be unsatisfiable with either 60 or 150 variables. The constraint arity of each constraint is 3 (3-SAT) and the maximum number of tuples of each constraint relation is 4.
- Dubois series: This series contains 13 randomly generated unsatisfiable 3-SAT instances. For each instance the maximum number of tuples of each constraint relation is 4.
- VarDimacs series: This series comes from the original Sat formalization of BF (Bridge Fault): circuit fault analysis (4 unsatisfiable instances), and from the Pigeon-hole problem (5 unsatisfiable instances). For each instance, the maximum arity of the constraints is greater than 2 and the maximum number of tuples of each constraint relation is 511 (normalized-hole-10_ext).
- aim-50 series: This series contains 24 artificially generated random-3-SAT instances. For each instance the maximum number of tuples of each constraint relation is 7.

5.3. Performance measures

To validate our contribution from a practical point of view, we consider both performance measures: the *CPU time* and the *memory space*. For this purpose, we present the following notions.

Definition 5.1 (Compression gain)**.** Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$ be a CSP instance and let $P' = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R}' \rangle$ be one of its compressed representations.

- the compression gain: $G_P = 1 - \frac{\sum_{i=1}^{m} s_{C_i}}{\sum_{i=1}^{m} |C_i| \times |R_i|}$, where m is the number of constraints and s_{C_i} is the size of the constraint C_i (see Sect. 4.2.2, Proof 4.2.2).

G_P (also called memory gain) measures the memory savings achieved when the relations are compressed. The closer to 0 this gain is, the worse is the compression. Notice that we suppose that each relation R_i is associated with only one constraint.

Definition 5.2 (Compression ratio)**.** Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R} \rangle$ be a CSP instance and let $P' = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R}' \rangle$ be one of its compressed representations. Let R_i be the relation of C_i and R_i^c one of its compressed versions. The compression ratio ρ measures the performance of the compression algorithm *w.r.t.* the total number of tuples covered by the instance.

- the compression ratio: $\rho = \frac{\sum\limits_{i=1}^{m} |R_i^c|}{\sum\limits_{i=1}^{m} |R_i|}$, where m is the number of constraints.

ρ measures the degree of compression *w.r.t.* constraint relations. Closer to 0 this number is, better is the compression. The optimal ratio is reached when $\rho = \frac{|C|}{\sum_{i=1}^{|C|} |R_i|}$ meaning that each compressed relation is represented by one unique compressed tuple.

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

5.4. Comparing *CJAS* with *JAS*

We compare *CJAS* and *JAS* on all the benchmarks of Section 5.2.

- Table 1 compares *JAS* and *CJAS* on the Modified-Renault and Renault series. These results clearly show the benefit of our approach. The low value of ρ means that all the instances are well compressed. It explains the good behaviour of *CJAS* in terms of time resolution while *JAS* fails to solve the most part of the two series.
- Table 2 reports the results on the Large bdd series. The good results obtained by *CJAS* are due to the small widths of the GHD (2) (because two constraints of arity 15 are sufficient to recover all the 21 variables in the χ label of the unique node of the GHD) and the good compression ratio ($\rho \simeq 0.79$). Note that the number of special nodes added (as children of the unique node returned by *BE*) to complete the decomposition is 2711 for each instance. For each instance of this series, the time of making complete the decomposition is less than 1 s. This time is included in the times of *CJAS* and *JAS* in Table 2. Note that for this series, all the solutions of each CSP instance are computed after the semi-join operation because all the variables of each instance are in the root node of the GHD decomposition, so each tuple accepted by a ctuple in the obtained crelation satisfies all the 2713 constraints.
- Table 3 presents the results obtained on the ssa series. The ssa-6288-047_ext instance is defined on 10 408 variables and 23 563 constraints. *BE* and *det-k-decomp* fail to decompose its constraint hypergraph into a GHD because of its size. Regarding the instances unsolved by *JAS*, although their relations are small, the widths of their GHDare too large (between 11 and 25). This explains why *JAS* could not solve these instances due to the memory explosion problem. But, thanks to the high-compression ratio (≃0.4) for all instances, *CJAS* succeeds to solve three of these instances in a short time.
- Table 4 shows the results obtained on the Pret series. Both relation sizes (4) and the widths of the GHD decompositions returned by BE (5) are very small for all instances. This explains why the two algorithms succeed to solve very quickly all these instances even if no compression gain is observed.
- Table 5 shows the results on the Dubois series. *CJAS* and *JAS* succeed to solve all the instances in short times because relation sizes (4) and the widths of the GHD decompositions are very small for all instances.
- Table 6 reports the comparison results of the *CJAS* and *JAS* algorithms on some instances of VarDimacs series. Excepted for *normalized-bf-432-007_ext* instance for which both algorithms have been confronted to the memory explosion problem due to the fact that the width of the GHD is very high (29), for other instances *CJAS* is more efficient thanks to the good compression ratio ($\rho \simeq 0.50$).
- Table 7 reports some significant results on the aim-50 series. For this series again, *CJAS* solved all the tested instances while *JAS* fails to solve them because of the memory explosion. The large widths of the decompositions GHD returned by BE (between 9 and 12) are the main reasons of the failure of *JAS*.

5.5. Comparing *CJAS* with *BTD*

This subsection compares *CJAS* with the four variants of *BTD* used in [20]

- $BTD\text{-}09_{MF(TD)}$
- $BTD\text{-}09_{MCS(TD)}$
- $BTD\text{-}HD_{BE(HD)}$
- $BTD\text{-}09_{BE(HD)}$

on some instances of the "modified Renault series" used in [20].

Note that it is very difficult to compare the practical behavior of two algorithms developed by different teams. Comparison of a new algorithm with previous ones is made possible by means of published results when the environment (operating system, CPU, language, RAM, *etc.*) and the search/inference procedure is known [26]. Another strategy is to reimplement previous algorithms but with the problem of choosing the right optimized data structures.

In our case, since we did not have the binaries of *BTD*, we opted for the first strategy. This is not an ideal strategy to compare two methods, but we believe that it can provide a general indication about the practical behavior of *CJAS* and *BTD*. For the *BTD* variants, the resolution times are the ones reported in [20] where

TABLE 1. Comparing *CJAS* with *JAS*: modified Renault and Renault instances.

Problems	Size							Time (s)			
	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	r	*ghtw*	*nb_nodes*	ρ	G	*CJAS*	*JAS*	*BE*	Observation
normalized-renault-mod-0_ext	111	147	48 721	4	84	0.02	0.94	9.36	MO	0	Consistent
normalized-renault-mod-1_ext	111	147	48 721	2	85	0.03	0.93	4.33	883.71	0	Inconsistent
normalized-renault-mod-2_ext	111	147	48 721	3	84	0.03	0.93	606.52	MO	0	Consistent
normalized-renault-mod-3_ext	111	147	48 721	3	82	0.03	0.94	5.39	MO	0	Inconsistent
normalized-renault-mod-4_ext	111	147	48 721	4	84	0.03	0.94	7.51	MO	0	Consistent
normalized-renault-mod-5_ext	111	147	48 721	3	83	0.03	0.94	4.75	TO	0	Inconsistent
normalized-renault-mod-6_ext	111	147	48 721	3	82	0.03	0.94	4.98	1799.66	0	Inconsistent
normalized-renault-mod-7_ext	111	147	48 721	3	83	0.03	0.94	6.52	TO	0	Consistent
normalized-renault-mod-8_ext	111	147	48 721	3	87	0.02	0.94	4.54	TO	0	Inconsistent
normalized-renault-mod-9_ext	111	147	48 721	3	86	0.04	0.93	8.06	MO	0	Consistent
normalized-renault-mod-10_ext	111	149	48 721	3	80	0.04	0.93	4.83	56.79	0	Inconsistent
normalized-renault-mod-11_ext	111	149	48 721	3	84	0.03	0.93	9.75	TO	0	Consistent
normalized-renault-mod-12_ext	111	149	48 721	3	85	0.02	0.94	4.83	TO	0	Inconsistent
normalized-renault-mod-13_ext	111	149	48 721	3	79	0.03	0.93	7.01	TO	0	Consistent
normalized-renault-mod-14_ext	111	149	48 721	3	84	0.03	0.93	4.67	TO	0	Inconsistent
normalized-renault-mod-15_ext	111	149	48 721	3	79	0.03	0.93	5.10	1296.43	0	Inconsistent
normalized-renault-mod-16_ext	111	149	48 721	3	80	0.03	0.93	4.61	55.39	0	Inconsistent
normalized-renault-mod-17_ext	111	149	48 721	4	82	0.02	0.94	5.03	MO	0	Inconsistent
normalized-renault-mod-18_ext	111	149	48 721	3	81	0.02	0.94	4.28	MO	0	Inconsistent
normalized-renault-mod-19_ext	111	149	48 721	3	83	0.03	0.94	4.79	839.71	0	Inconsistent
normalized-renault-mod-20_ext	111	159	48 721	3	82	0.04	0.92	6.03	MO	0	Inconsistent
normalized-renault-mod-21_ext	111	159	48 721	3	84	0.04	0.92	8.62	MO	0	Inconsistent
normalized-renault-mod-22_ext	111	159	48 721	3	82	0.04	0.93	4.40	TO	0	Inconsistent
normalized-renault-mod-23_ext	111	159	48 721	3	84	0.04	0.92	4.51	MO	0	Inconsistent
normalized-renault-mod-24_ext	111	159	48 721	4	79	0.04	0.92	4.80	MO	0	Inconsistent
normalized-renault-mod-25_ext	111	159	48 721	3	82	0.04	0.93	4.48	323.02	0	Inconsistent
normalized-renault-mod-26_ext	111	159	48 721	3	81	0.05	0.92	4.68	MO	0	Inconsistent
normalized-renault-mod-27_ext	111	159	48 721	4	85	0.04	0.93	4.35	MO	0	Inconsistent
normalized-renault-mod-28_ext	111	159	48 721	3	79	0.04	0.92	4.46	TO	0	Inconsistent
normalized-renault-mod-29_ext	111	159	48 721	4	78	0.05	0.92	5.15	MO	0	Inconsistent
normalized-renault-mod-30_ext	111	154	48 721	3	86	0.03	0.93	33.50	MO	0	Inconsistent
normalized-renault-mod-31_ext	111	154	48 721	3	85	0.04	0.93	6.02	TO	0	Consistent
normalized-renault-mod-32_ext	111	154	48 721	5	86	0.03	0.94	122.62	MO	0	Consistent
normalized-renault-mod-33_ext	111	147	48 721	3	83	0.03	0.93	14.56	TO	0	Inconsistent
normalized-renault-mod-34_ext	111	154	48 721	4	85	0.03	0.93	8.81	MO	0	Consistent
normalized-renault-mod-35_ext	111	154	48 721	4	84	0.03	0.93	16.22	MO	0	Inconsistent
normalized-renault-mod-36_ext	111	154	48 721	4	82	0.03	0.93	8.04	MO	0	Consistent
normalized-renault-mod-37_ext	111	154	48 721	4	80	0.02	0.94	4.30	MO	0	Inconsistent
normalized-renault-mod-38_ext	111	149	48 721	4	77	0.03	0.93	6.21	MO	0	Consistent
normalized-renault-mod-39_ext	111	154	48 721	5	76	0.02	0.94	4.92	MO	0	Inconsistent
normalized-renault-mod-40_ext	108	149	48 721	4	82	0.02	0.94	4.46	MO	0	Inconsistent
normalized-renault-mod-41_ext	108	149	48 721	4	81	0.03	0.94	29.50	MO	0	Consistent
normalized-renault-mod-42_ext	108	149	48 721	3	82	0.02	0.94	10.98	TO	0	Inconsistent
normalized-renault-mod-43_ext	108	149	48 721	3	78	0.02	0.94	18.40	MO	0	Consistent
normalized-renault-mod-44_ext	108	149	48 721	4	83	0.02	0.94	11.30	MO	0	Consistent
normalized-renault-mod-45_ext	108	149	48 721	5	83	0.02	0.94	8.43	MO	0	Consistent
normalized-renault-mod-46_ext	108	149	48 721	4	78	0.02	0.94	10.04	MO	0	Consistent
normalized-renault-mod-47_ext	108	149	48 721	4	80	0.02	0.94	4.80	MO	0	Inconsistent
normalized-renault-mod-48_ext	108	149	48 721	4	82	0.03	0.93	9.21	MO	0	Consistent
normalized-renault-mod-49_ext	108	149	48 721	4	78	0.02	0.94	5.89	MO	0	Consistent
normalized-renault_ext	101	134	48 721	4	79	0.018	0.95	4.69	TO	0	Consistent
normalized-renault-mgd_ext	101	113	48 721	2	81	0.018	0.95	4.51	1418.79	0	Consistent

Table 2. Comparing *CJAS* and *JAS*: large bdd instances.

Problems	Size							Time (s)			
	$\|V\|$	$\|E\|$	r	*ghtw*	*nb_nodes*	ρ	G	*CJAS*	*JAS*	*BE*	Observation
normalized-bdd-21-2713-15-79-1_ext	21	2713	6892	2	1	0.79	0.19	114.20	363.12	95	Inconsistent
normalized-bdd-21-2713-15-79-2_ext	21	2713	6836	2	1	0.79	0.19	109.88	328.31	95	Inconsistent
normalized-bdd-21-2713-15-79-3_ext	21	2713	6910	2	1	0.79	0.19	114.42	266.61	96	Inconsistent
normalized-bdd-21-2713-15-79-4_ext	21	2713	6993	2	1	0.79	0.19	154.20	280.30	97	Consistent
normalized-bdd-21-2713-15-79-5_ext	21	2713	6825	2	1	0.79	0.19	124.02	263.09	95	Consistent
normalized-bdd-21-2713-15-79-6_ext	21	2713	6935	2	1	0.79	0.19	112.55	277.34	94	Inconsistent
normalized-bdd-21-2713-15-79-7_ext	21	2713	6944	2	1	0.78	0.19	110.24	277.18	95	Inconsistent
normalized-bdd-21-2713-15-79-8_ext	21	2713	6939	2	1	0.78	0.20	120.39	299.12	98	Consistent
normalized-bdd-21-2713-15-79-9_ext	21	2713	6934	2	1	0.79	0.19	113.32	293.70	96	Inconsistent
normalized-bdd-21-2713-15-79-10_ext	21	2713	6717	2	1	0.79	0.18	119.54	285.93	103	Inconsistent
normalized-bdd-21-2713-15-79-11_ext	21	2713	6925	2	1	0.79	0.19	120.05	269.12	101	Inconsistent
normalized-bdd-21-2713-15-79-12_ext	21	2713	6851	2	1	0.79	0.19	124.47	287.13	101	Inconsistent
normalized-bdd-21-2713-15-79-13_ext	21	2713	6794	2	1	0.79	0.19	131.16	261.63	99	Consistent
normalized-bdd-21-2713-15-79-14_ext	21	2713	6923	2	1	0.79	0.19	118.67	294.59	99	Inconsistent
normalized-bdd-21-2713-15-79-15_ext	21	2713	6900	2	1	0.78	0.19	125.48	287.65	102	Inconsistent
normalized-bdd-21-2713-15-79-16_ext	21	2713	6901	2	1	0.79	0.19	126.09	274.56	102	Inconsistent
normalized-bdd-21-2713-15-79-17 _ext	21	2713	6972	2	1	0.78	0.20	122.32	288.26	101	Inconsistent
normalized-bdd-21-2713-15-79-18_ext	21	2713	6907	2	1	0.79	0.19	120.16	293.69	103	Inconsistent
normalized-bdd-21-2713-15-79-19_ext	21	2713	6829	2	1	0.79	0.19	142.36	305.60	102	Consistent
normalized-bdd-21-2713-15-79-20_ext	21	2713	7092	2	1	0.79	0.19	126.96	325.08	102	Inconsistent
normalized-bdd-21-2713-15-79-21_ext	21	2713	6879	2	1	0.78	0.19	115.06	273.53	100	Inconsistent
normalized-bdd-21-2713-15-79-22_ext	21	2713	6796	2	1	0.79	0.19	148.73	264.01	102	Consistent
normalized-bdd-21-2713-15-79-23_ext	21	2713	6869	2	1	0.79	0.19	118.55	287.09	103	Inconsistent
normalized-bdd-21-2713-15-79-24_ext	21	2713	7086	2	1	0.78	0.19	126.02	287.71	101	Consistent
normalized-bdd-21-2713-15-79-25_ext	21	2713	6945	2	1	0.78	0.19	126.05	335.73	105	Inconsistent
normalized-bdd-21-2713-15-79-26_ext	21	2713	6861	2	1	0.79	0.19	117.02	258.62	95	Inconsistent
normalized-bdd-21-2713-15-79-27_ext	21	2713	6979	2	1	0.78	0.19	121.93	301.57	99	Inconsistent
normalized-bdd-21-2713-15-79-28_ext	21	2713	6775	2	1	0.80	0.18	124.78	300.79	102	Inconsistent
normalized-bdd-21-2713-15-79-29_ext	21	2713	6954	2	1	0.79	0.19	119.03	285.08	100	Inconsistent
normalized-bdd-21-2713-15-79-30_ext	21	2713	6807	2	1	0.79	0.19	119.72	260.44	101	Inconsistent
normalized-bdd-21-2713-15-79-31_ext	21	2713	6898	2	1	0.78	0.20	140.84	253.40	101	Consistent
normalized-bdd-21-2713-15-79-32_ext	21	2713	6858	2	1	0.79	0.19	130.23	311.63	102	Consistent
normalized-bdd-21-2713-15-79-33_ext	21	2713	6806	2	1	0.79	0.18	117.17	298.57	103	Inconsistent
normalized-bdd-21-2713-15-79-34_ext	21	2713	7036	2	1	0.79	0.19	124.49	264.08	105	Consistent
normalized-bdd-21-2713-15-79-35_ext	21	2713	6814	2	1	0.79	0.19	121.49	286.95	102	Consistent

Table 3. Comparing *CJAS* with *JAS*: ssa instances.

Problems	Size							Time (s)			
	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	r	*ghtw*	*nb_nodes*	ρ	G	*CJAS*	*JAS*	*BE*	Observation
ssa-0432-003_ext	435	738	31	16	283	0.52	0.43	3.80	MO	1	Inconsistent
ssa-2670-130_ext	1359	2366	31	25	655	0.51	0.44	MO	MO	1	Inconsistent
ssa-2670-141_ext	391	177	15	2	166	0.47	0.46	0.01	0.05	0	Consistent
ssa-6288-047_ext	10 408	23 563	63	/	/	/	/	/	/	/	Consistent
ssa-7552-038_ext	1501	2444	63	18	1071	0.53	0.43	MO	MO	11	Consistent
ssa-7552-158_ext	1363	1985	31	11	955	0.57	0.39	6.94	MO	6	Consistent
ssa-7552-159_ext	1363	1983	31	11	1012	0.57	0.39	8.05	MO	7	Consistent
ssa-7552-160_ext	757	847	7	4	332	0.48	0.38	0.13	0.07	0	Consistent

TABLE 4. Comparing *CJAS* with *JAS*: Pret instances.

Problems	Size			*ghtw*	*nb_nodes*	ρ	*G*	Times			Observation
	$\lvert\mathcal{X}\rvert$	$\lvert\mathcal{C}\rvert$	*r*					*CJAS*	*JAS*	*BE*	
normalized-Pret -60-25_ext	60	40	4	5	25	1	0	0.008	0.007	0	Inconsistent
normalized-Pret -60-40_ext	60	40	4	5	25	1	0	0.006	0.006	0	Inconsistent
normalized-Pret -60-60_ext	60	40	4	5	27	1	0	0.009	0.007	0	Inconsistent
normalized-Pret -60-75_ext	60	40	4	5	26	1	0	0.08	0.01	0	Inconsistent
normalized-Pret -150-25_ext	150	100	4	5	69	1	0	0.01	0.009	0	Inconsistent
normalized-Pret -150-40_ext	150	100	4	5	68	1	0	0.01	0.01	0	Inconsistent
normalized-Pret -150-60_ext	150	100	4	5	69	1	0	0.01	0.009	0	Inconsistent
normalized-Pret -150-75_ext	150	100	4	5	68	1	0	0.01	0.01	0	Inconsistent

TABLE 5. Comparing *CJAS* with *JAS*: Dubois instances.

Problems	Size			*ghtw*	*nb_nodes*	ρ	*G*	Time (s)			Observation
	$\lvert\mathcal{X}\rvert$	$\lvert\mathcal{C}\rvert$	*r*					*CJAS*	*JAS*	*BE*	
normalized- Dubois-20_ext	60	40	4	2	25	1	0	0.002	0.001	0	Inconsistent
normalized-Dubois-21_ext	63	42	4	2	25	1	0	0.002	0.002	0	Inconsistent
normalized-Dubois-22_ext	66	44	4	2	27	1	0	0.005	0.004	0	Inconsistent
normalized-Dubois-23_ext	69	46	4	2	28	1	0	0.005	0.004	0	Inconsistent
normalized-Dubois-24_ext	72	48	4	2	28	1	0	0.006	0.004	0	Inconsistent
normalized-Dubois-25_ext	75	50	4	2	31	1	0	0.006	0.002	0	Inconsistent
normalized-Dubois-26_ext	78	52	4	2	31	1	0	0.006	0.005	0	Inconsistent
normalized-Dubois-27_ext	81	54	4	2	33	1	0	0.006	0.005	0	Inconsistent
normalized-Dubois-28_ext	84	56	4	2	39	1	0	0.007	0.006	0	Inconsistent
normalized-Dubois-29_ext	87	58	4	2	38	1	0	0.003	0.002	0	Inconsistent
normalized-Dubois-30_ext	90	60	4	2	36	1	0	0.003	0.002	0	Inconsistent
normalized-Dubois-50_ext	150	100	4	2	63	1	0	0.006	0.005	0	Inconsistent
normalized-Dubois-100_ext	300	200	4	2	128	1	0	0.01	0.01	0	Inconsistent

TABLE 6. Comparing *CJAS* with *JAS*: some instances of VarDimacs series.

Problems	Size			*ghtw*	*nb_nodes*	ρ	*G*	Time (s)			Observation
	$\lvert\mathcal{X}\rvert$	$\lvert\mathcal{C}\rvert$	*r*					*CJAS*	*JAS*	*BE*	
normalized-bf-0432-007_ext	970	1943	31	29	1788	0.53	0.38	MO	MO	16	Consistent
normalized-bf-1355-075_ext	1818	2049	15	5	1766	0.44	0.44	9.24	19.04	8	Consistent
normalized-bf-1355-638_ext	532	339	31	2	322	0.44	0.51	0.045	0.35	0	Consistent
normalized-bf-2670-001_ext	1244	1354	31	6	1288	0.62	0.34	1.36	1.34	1	Consistent
normalized-hole-06_ext	42	133	63	7	121	0.35	0.70	62.99	MO	1	Inconsistent

the authors used a PC Pentium IV, 3.2 GHZ with 1 GB of RAM and running under Linux. Note that for this comparison, we have executed *CJAS* on a machine with similar configuration (PC Pentium IV, 3.2 GHZ with 1 GB of RAM and running under Linux).

The results of this comparison are reported in Table 8. Note that the sizes of the instances are the ones in Table 1.

We can observe that for these 17 instances:

- *CJAS* is the fastest for 6 instances.
- $BTD\text{-}09_{BE(HD)}$ is the fastest for 4 instances.
- $BTD\text{-}09_{MF(TD)}$ is the fastest for 4 instances.
- $BTD\text{-}09_{MCS(TD)}$ is the fastest for 2 instances.

TABLE 7. Comparing *CJAS* with *JAS*: some instances of aim-50 series.

Problems	Size			ghtw	nb_nodes	ρ	G	Time (s)			Observation
	$\lvert\mathcal{X}\rvert$	$\lvert\mathcal{C}\rvert$	r					CJAS	JAS	BE	
normalized-aim-50-1-6-sat-1_ext	50	77	7	10	53	0.42	0.43	9.66	MO	0	Consistent
normalized-aim-50-1-6-sat-2_ext	50	76	7	9	62	0.42	0.43	0.69	MO	0	Consistent
normalized-aim-50-1-6-sat-3_ext	50	78	7	10	57	0.42	0.42	1.85	MO	0	Consistent
normalized- aim-50-1-6-sat-4_ext	50	77	7	10	58	0.42	0.42	23.15	MO	0	Consistent
normalized- aim-50-1-6-unsat-1_ext	50	69	7	9	49	0.42	0.43	0.55	MO	0	Inconsistent
normalized- aim-50-1-6-unsat-2_ext	50	77	7	10	20	0.42	0.43	2.95	MO	0	Inconsistent
normalized- aim-50-1-6-unsat-3_ext	50	70	7	10	55	0.41	0.43	4.58	MO	0	Inconsistent
normalized- aim-50-1-6-unsat-4_ext	50	76	7	11	55	0.42	0.43	3.33	MO	0	Inconsistent
normalized- aim-50-2-0-sat-1_ext	50	94	7	11	19	0.42	0.43	234.66	MO	0	Consistent
normalized- aim-50-2-0-sat-2_ext	50	96	7	11	70	0.42	0.43	16.76	MO	0	Consistent
normalized- aim-50-2-0-sat-4_ext	50	94	7	12	23	0.42	0.43	42.70	MO	0	Consistent
normalized- aim-50-2-0-unsat-1_ext	50	97	7	13	21	0.42	0.43	20.69	MO	0	Inconsistent
normalized- aim-50-2-0-unsat-2_ext	50	94	7	12	21	0.42	0.43	8.46	MO	0	Inconsistent
normalized- aim-50-2-0-unsat-4_ext	50	94	7	11	65	0.42	0.42	0.44	MO	0	Inconsistent

TABLE 8. Comparing *BTD* variants [20] and *CJAS*: some instances of modified Renault series.

Problems	Time (s)				
	CJAS	$BTD-09_{MF(TD)}$	$BTD-09_{MCS(TD)}$	$BTD-HD_{BE(HD)}$	$BTD-09_{BE(HD)}$
normalized-renault-mod-3_ext	**6.39**	10.67	11.15	42.34	20.56
normalized-renault-mod-6_ext	5.26	3.71	3.75	1279.55	**2.70**
normalized-renault-mod-12_ext	**5.34**	11.64	11.85	134.24	10.49
normalized-renault-mod-16_ext	5.98	6.04	10.36	**3.65**	6.43
normalized-renault-mod-17_ext	5.53	9.59	5.42	145.06	**3.41**
normalized-renault-mod-18_ext	**4.92**	12.23	12.09	39.34	10.46
normalized-renault-mod-19_ext	**4.96**	7.74	12.07	49.25	16.36
normalized-renault-mod-23_ext	5.17	12.87	**2.81**	28.82	3.79
normalized-renault-mod-24_ext	5.19	7.97	8.05	35.01	**7.63**
normalized-renault-mod-30_ext	39.89	**3.80**	9.81	202.05	8.25
normalized-renault-mod-35_ext	17.35	**7.32**	12.28	57.33	13.19
normalized-renault-mod-36_ext	8.71	3.80	**1.78**	225.76	4.88
normalized-renault-mod-37_ext	**5.05**	13.68	17.01	13.64	21.16
normalized-renault-mod-39_ext	5.52	13.45	35.55	746.24	**1.79**
normalized-renault-mod-40_ext	5.21	**5.86**	8.60	65.55	9.01
normalized-renault-mod-42_ext	13.98	**2.48**	3.44	TO	2.50
normalized-renault-mod-47_ext	**7.77**	53.71	21.31	324.20	80.25
Cumulative runtime	**152.22**	186.56	187.33	3392.03	222.86

- BTD-$HD_{BE(HD)}$ is the fastest for 1 instance.

Furthermore, the cumulative runtime of *CJAS* on all instances is better.

5.6. Comparing *CJAS* with the Abscon 109 solver

In this subsection we compare *CJAS* with the Abscon 109 [9] solver on some instances considered in this paper. Abscon 109 is an efficient solver using no-decomposition.

Table 9 presents the comparison results between *CJAS* and Abscon 109 on the ssa series. On this series, Abscon 109 succeeds to solve all the instances while *CJAS* fails to solve two instances because of the memory

[9]Available at http://www.cril.univ-artois/~lecoutre/software.html.

TABLE 9. Comparing *CJAS* with *Abscon 109*: ssa instances.

Problems	Size							Time (s)		
	$\vert\mathcal{X}\vert$	$\vert\mathcal{C}\vert$	r	*ghtw*	*nb_nodes*	ρ	G	*CJAS*	*Abscon 109*	Observation
normalized-ssa-0432-003_ext	435	738	31	16	283	0.52	0.43	3.80	7.83	Inconsistent
normalized-ssa-2670-130_ext	1359	2366	31	25	655	0.51	0.44	MO	17.66	Inconsistent
normalized-ssa-2670-141_ext	391	177	15	2	166	0.47	0.46	0.01	0.69	Consistent
normalized-ssa-6288-047_ext	10 408	23 563	63	/	/	/	/	/	107.56	Consistent
normalized-ssa-7552-038_ext	1501	2444	63	18	/	0.53	0.43	MO	2.60	Consistent
normalized-ssa-7552-158_ext	1363	1985	31	11	955	0.57	0.39	6.94	2.05	Consistent
normalized-ssa-7552-159_ext	1363	1983	31	11	1012	0.57	0.39	8.05	2.03	Consistent
normalized-ssa-7552-160_ext	757	847	7	4	332	0.48	0.38	0.13	0.81	Consistent

TABLE 10. Comparing *CJAS* with *Abscon 109*: Pret instances.

Problems	Size							Times		
	$\vert\mathcal{X}\vert$	$\vert\mathcal{C}\vert$	r	*ghtw*	*nb_nodes*	ρ	G	*CJAS*	*Abscon 109*	Observation
normalized-Pret -60-25_ext	60	40	4	5	25	1	0	0.008	0.92	Inconsistent
normalized-Pret -60-40_ext	60	40	4	5	25	1	0	0.006	3.42	Inconsistent
normalized-Pret -60-60_ext	60	40	4	5	27	1	0	0.009	0.93	Inconsistent
normalized-Pret -60-75_ext	60	40	4	5	26	1	0	0.08	1.76	Inconsistent
normalized-Pret -150-25_ext	150	100	4	5	69	1	0	0.01	23.58	Inconsistent
normalized-Pret -150-40_ext	150	100	4	5	68	1	0	0.01	13.04	Inconsistent
normalized-Pret -150-60_ext	150	100	4	5	69	1	0	0.01	5.34	Inconsistent
normalized-Pret -150-75_ext	150	100	4	5	68	1	0	0.01	6.37	Inconsistent

explosion problem. For the instances where Abscon 109 is faster, the main reason is the time decomposition of *BE*.

Table 10 shows clearly that *CJAS* outperforms Abscon 109 on the Pret series. This is because the width of the GHD decomposition returned by BE for each instance is small and the size of each constraint relation is also small.

Table 11 presents the comparison results between *CJAS* and Abscon 109 on the Dubois series. We observe that *CJAS* behaves better on this series. The good behaviour of *CJAS* can be explained by the fact that the width of the GHD decomposition returned by BE for each instance is small and the size of each constraint relation is also small.

5.7. Discussion

Like *JAS*, the *CJAS* method is more suitable for computing all the solutions of a CSP instance. This is especially true when the number of nodes (without the ones added for making complete the decomposition) of the generalized hypertree decomposition is 1 as it is the case in the Large bdd series where all the solutions of the whole instance are computed after the compressed semi-join step. *CJAS* highly depends on the quality of the compression. When the compression ratio ρ is equal to 1, both algorithms *CJAS* and *JAS* have the same behaviour. On the contrary, when the compression ratio ρ is smaller, *CJAS* outperforms *JAS* in general in term of CPU time. This is not only due to the memory gain obtained by the compression step but also to the small CPU time needed to compute the compressed join and compressed semi-join operations.

From another point of view, the overall performance of *CJAS* highly depends on the quality of the decomposition. To illustrate this fact, we have tested the decomposition returned by *det-k-decomp* [10] for the modified Renault 30 instance (*normalized-renault-mod*-30*_ext* in Tab. 1). We have obtained a resolution time of 14 s with the decomposition returned by *det-k-decomp* instead of 33.50 s required by the decomposition returned by *BE*.

TABLE 11. Comparing *CJAS* with *Abscon 109*: Dubois instances.

Problems	Size			*ghtw*	*nb_nodes*	ρ	G	Time (s)		Observation
	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	r					*CJAS*	*Abscon 109*	
normalized-Dubois-20_ext	60	40	4	2	25	1	0	0.002	4.23	Inconsistent
normalized-Dubois-21_ext	63	42	4	2	25	1	0	0.002	0.89	Inconsistent
normalized-Dubois-22_ext	66	44	4	2	27	1	0	0.005	0.97	Inconsistent
normalized-Dubois-23_ext	69	46	4	2	28	1	0	0.005	0.94	Inconsistent
normalized-Dubois-24_ext	72	48	4	2	28	1	0	0.006	0.78	Inconsistent
normalized-Dubois-25_ext	75	50	4	2	31	1	0	0.006	1.05	Inconsistent
normalized-Dubois-26_ext	78	52	4	2	31	1	0	0.006	1.28	Inconsistent
normalized-Dubois-27_ext	81	54	4	2	33	1	0	0.006	0.85	Inconsistent
normalized-Dubois-28_ext	84	56	4	2	39	1	0	0.007	1.56	Inconsistent
normalized-Dubois-29_ext	87	58	4	2	38	1	0	0.003	1.49	Inconsistent
normalized-Dubois-30_ext	90	60	4	2	36	1	0	0.003	1.43	Inconsistent
normalized-Dubois-50_ext	150	100	4	2	63	1	0	0.006	1.98	Inconsistent
normalized-Dubois-100_ext	300	200	4	2	128	1	0	0.01	407.37	Inconsistent

Furthermore, *CJAS* fails to solve many instances of the series Normalized-aim 100 and Normalized-aim 200 because the width of the GHD decomposition returned by BE for each instance is very high. With *CJAS*, the number of ctuples (which are the solutions of the subproblem) at each node of the *GHD* is bounded by $(cr)^{ghtw}$ where *cr* is the maximum number of compressed tuples in a crelation, *ghtw* is the GHD width. Let a be the highest arity of the constraints and let *cval* be the size of the largest c_value. Hence, to use the *CJAS* method, it is better that $a.cval.cr^{ghtw}$ be smaller than a given threshold that depends on the characteristics of the machine used.

Compared to *BTD*, *CJAS* is competitive for the benchmarks tested in this paper.

Compared to the direct resolution algorithms non based on decomposition, *CJAS* performs well on structured instances, and on instances where the width of the GHD is not too high and the constraint relations are not very large.

6. Conclusion

To cope with the problem of memory explosion of the Join Acyclic Solving (*JAS*) algorithm, we have presented in this paper, a new algorithm called *CJAS*. It is a compressed version of *JAS* and it is based on the compression of the constraint relations. We have mainly introduced the compressed join and compressed semi-join operations to work with the compressed tuples in the relations. We have evaluated the *CJAS* method on benchmarks selected for the CSP 2008 competition. Our experimental results confirm the fact that *JAS* behaves well when the maximum number of tuples of the relations and the width of the decomposition of the constraint hypergraph are small, but it struggles or fails to solve CSP instances when the width is too high. On its side, *CJAS* depends on the quality of the compression. Indeed, when the compression ratio ρ is 1 (no compression at all), it behaves like *JAS*, but when ρ is small (non-negligible) *CJAS* clearly outperforms *JAS*. Compared to the related method *BTD*, our experiments have shown that *CJAS* is competitive. Compared to the methods non-based-decomposition, *CJAS* behaves well on structured instances and on instances where the width of the GHD is not too high and the constraint relations are not very large. Future works will include a larger study of compression and decomposition algorithms.

Appendix. Computing a compressed relation

In Section 3 we noticed that compressed representation of a relation is not always unique. In this Appendix, the compression algorithm due to Katsirelos *et al.* [23] is presented and it is the one used for the experimental

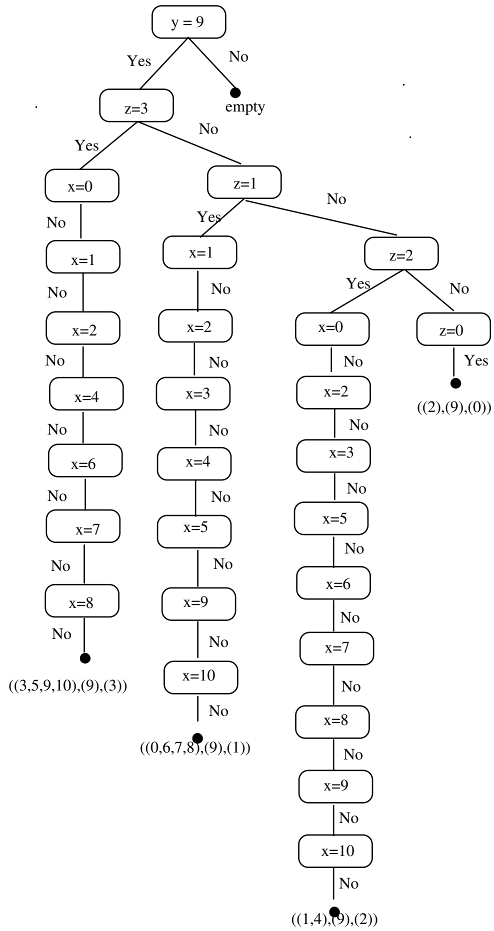

FIGURE A.1. A Decision Tree associated with the relation R_1 of Example 3.3.

study. A compressed relation associated with any relation is a set of compressed tuples obtained from a *Decision Tree* representing all the original tuples of the relation.

As illustrated by Figure A.1, a Decision Tree T representing the tuples of a given relation is a binary tree where each node (v) is labelled with $l(v)$: a possible assignment of a value to a variable (*a literal*). Each edge from a node to its child is labelled with *Yes* if the literal is true and with *No* if the literal is false. Any tuple t in the relation is associated with a node v if it satisfies all the literals on the path from the root of T to v. The set of all the tuples in the relation associated with a node v is denoted by $U(v)$. For any node v in T, v is said *empty* if $U(v) = \emptyset$. For any node v in T, if $U(v)$ contains all the possible tuples that could be associated with v then v is said *complete*. A literal s (resp $\neg s$) is said *implied* in a node v if all the tuples associated with v include (resp. do not include) s.

Unfortunately, constructing an optimal Decision Tree is a NP-Complete problem [18] and a heuristic approach proposed in [23] is outlined by Algorithm 8.

At each node v of the Decision Tree T, this algorithm checks if *implied literals* exist. If it is the case, it extends T with one new node v' for each implied literal. If it is not the case, the function *Choose_Literal*$(U(v))$ selects a literal for the node v and then expands each of the two new nodes v_1 and v_2. If an empty node $(U(v) = \phi)$ or a complete node is created then it stops.

Once the Decision Tree T is built, for each complete leaf node v, a compressed tuple ct representing $U(v)$ is created as follows. Let $S(v)$ be the set of literals labelling the nodes in the path from the root of T to v.

For each variable X_i with domain D_i, let D'_i be the c_value of X_i in ct. Initially D'_i=D_i, if there is a node in the path from the root of T to v labelled with $(X_i = d_i)$ and the outcoming edge is labelled with "Yes", then $D'_i = \{d_i\}$ else $D'_i = D'_i - \{d_j\}$ for each literal $(X_i = d_j) \in S(v)$. At the end, the compressed tuple ct accepting exactly the same tuples as v is $(D'_1, \ldots, D'_n)$.

For the choice of a literal by the function *Choose_Literal*, a number of splitting heuristics are proposed in [23]. In this work, the *MaxFreq* heuristic is used because of its good behavior on the benchmarks experimented for this article.

Algorithm 8. The compression algorithm.

TableToDecisionTree (Set of tuples: U, Node: v)

if v is *empty* or v is *complete* **then**
  return
end if
if $\exists$ a literal s: s is *implied* in v **then**
  v' $\leftarrow$ {Parent:v, Edgeliteral: s}
  TableToDecisionTree (U(v'),v')
else
  s $\leftarrow$ *Choose_Literal* (U(v))
  $v_1 \leftarrow$ {Parent:v, Edgeliteral: s}
  $v_2 \leftarrow$ {Parent:v, Edgeliteral: $\neg s$}
  TableToDecisionTree (U(v_1),v_1)
  TableToDecisionTree (U(v_2),v_2)
end if

Acknowledgements. We would like to thank the anonymous reviewer for his helpful comments and valuable suggestions.

References

[1] I. Adler, G. Gottlob and M. Grohe, Hypertree-width and related hypergraph invariants. In *Proc. of the 3rd European Conference on Combinatorics, Graph Theory, and Applications (EUROCOMB'05), DMTCS Proceedings Series*, vol. AE (2005) 5–10.

[2] A. Ait-Amokhtar, K. Amroun and Z. Habbas, Hypertree Decomposition for Solving Constraint Satisfaction Problems. In *Proc. of International conference on Agents and Artificial Intelligence, ICAART'2009* (2009) 398–404.

[3] C. Bessière and J.C. Régin, Arc Consistency for General Constraint Networks: Preliminary Results. In *Proc. of the Sixtenteenth International Joint Conference on Artificial Intelligence* (1997) 398–404.

[4] D. Cohen, P. Jeavons and M. Gyssens, A unified theory of structural tractability for Constraint Satisfaction Problems. *J. Comput. System Sci.* **74** (2008) 721–743.

[5] R. Dechter and J. Pearl, Tree clustering for constraint networks. *J. Artif. Intell.* **38** (1989) 353–366.

[6] A. Dermaku, T. Ganzow, G. Gottlob, B. McMahan, N. Musliu and M. Samer, Heuristic Methods for Hypertree Decompositions. In *Proc. of the 7th. Mexican Int. Conf. on Artificial Intelligence: Advances in Artificial Intelligence* (2008) 1–11.

[7] E.C. Freuder, A sufficient condition for Backtrack-free search. *J. ACM* **29** (1982) 24–32.

[8] E.C. Freuder, A sufficient condition for backtrack-bounded search. *J. ACM* **32** (1985) 755–761.

[9] I.P. Gent, C. Jefferson and P. Nightingale, Data Structures for Generalized arc Consistency for Extensional Constraints. In *Proc. of AAAI'07* (2007) 191–197.

[10] G. Gottlob and M. Samer, A Backtracking-based algorithm for hypertree decomposition. *ACM J. Exp. Algorithmics* (*JEA*) **13** (2009).

[11] G. Gottlob, N. Leone and F. Scarcello, A comparison of structural CSP decomposition methods. *Artif. Intell.* **124** (2000) 243–282.

[12] G. Gottlob, N. Leone and F. Scarcello, Hypertree decompositions and tractable queries. *J. Comput. System Sci.* **64** (2002) 579–627.

[13] G. Gottlob, N. Leone and F. Scarcello, Robbers, marshals, and guards: game theoretic and logical characterisations of hypertree width. *J. Comput. System Sci.* **66** (2003) 775–808.

[14] G. Gottlob, Z. Miklos and T. Schwentick, Generalized hypertree decompositions: NP – hardness and tractable variants. *J. ACM* **56** (2009).

[15] M. Grohe and D. Marx, Constraint Solving via Fractional Edge Covers. In *Proc. of SODA 2006* (2006) 289–298.

[16] M. Gyssens, P.G. Jeavons and D.A. Cohen, Decomposing constraint satisfaction problems using database techniques. *Artif. Intell. J.* **66** (1994) 57–89.

[17] P. Harvey and A. Ghose, Reducing Redundancy in The Hypertree Decomposition Scheme. In *Proc. of ICTAI'03*, Montreal (2003) 474–481.

[18] L. Hyafil and R.L. Rivest, Constructing optimal binary decision trees is NP-complete. *Inf. Process. Lett.* **5** (1976) 15–17.

[19] P. Jégou and C. Terrioux, Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artif. Intell. J.* **146** (2003) 43–75.

[20] P. Jégou, S.N. Ndiaye and C. Terrioux, Combined Strategies for Decomposition-based Methods for Solving CSPs. In *Proc. of ICTAI'09* (2009) 184–192.

[21] T. Kam, T. Villa, R.K. Brayton and A.L. Sangiovanni-Vincentelli, Multivalued decision diagrams: Theory and applications. *Int. J. Multiple Valued Logic* **4** (1998) 9–62.

[22] G. Katsirelos and F. Bacchus, Generalized Nogoods in CSPs. In *Proc. of AAAI'05*, Pittsburgh (2005) 390–396.

[23] G. Katsirelos and T. Walsh, A Compression Algorithm for Large Arity Extensional Constraints. In *Proc. of CP'07* (2007) 379–393.

[24] C.K. Kenil Cheng and R.H.C. Yap, An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints* **15** (2010) 265–304.

[25] T. Korimort, *Heuristic hypertree decomposition*. Ph. D. thesis, Vienna University of Technology (2003).

[26] C. Lecoutre, STR2: optimized simple table reduction for table constraints. *Constraints* **16** (2011) 341–371.

[27] A. Mackworth, On Reading Sketch Maps. In *Proc. of IJCAI-77* (1977) 598–606.

[28] Z. Miklos, *Understanding Tractable Decompositions for Constraint Satisfaction*. Ph. D. thesis, University of Oxford (2008).

[29] U. Montanari, Networks of constraints: fundamental properties and applications to pictures processing. *Inf. Sci. J.* **7** (1974) 95–132.

[30] N. Musliu and W. Schafhauser, Genetic algorithms for Generalized hypertree decompositions. *Eur. J. Ind. Eng.* **1** (2005) 317–340.

[31] W. Pang and S.D. Goodwin, Constraint-directed backtracking. Vol. 1342 of *Lect. Notes Comput. Sci.* (1997) 47–56.

[32] W. Pang and S.D. Goodwin, A graph based backtracking algorithm for solving general CSPs. *Lect. Notes Comput. Sci. of AI* (2003) 114–128.

[33] G. Pesant, A Regular Language Membership Constraint for Finite Sequences of Variables. In *Proc. of CP'04* (2004) 482–495.

[34] J.-C. Régin, Improving the Expressiveness of Table Constraints. In *Proc. of workshop ModRef 11 at CP'11* (2011).

[35] N. Robertson and P.D. Seymour, Graph minors .II. algorithmic aspects of treewidth. *J. Algorithms* **7** (1986) 309–322.

[36] M. Samer, Hypertree-Decomposition *via* Branch-Decomposition. In *Proc. of IJCAI'05* (2005) 1535–1536.

[37] S. Subbarayan and H. Reif Anderson, Backtracking Procedures for Hypertree, Hyperspread and Connected Hypertree Decomposition of CSPs. In *Proc. of IJCAI'07* (2007) 180–185.

[38] J.R. Ullmann, Partition search for non binary constraint satisfaction. *Inf. Sci. J.* **177** (2007) 3639–3678.

[39] M. Yannakakis, Algorithms for Acyclic Database Schemes. In *Proc. of VLDB'81*. Edited by C. Zaniolo and C. Delobel, Cannes, France (1981) 82–94.