Defining and Evaluating Heuristics for the Compilation of Constraint Networks

Jean-Marie Lagniez, Pierre Marquis, and Anastasia Paparrizou

CRIL, U. Artois & CNRS, Lens, France
lastname@cril.fr

Abstract. Several branching heuristics for compiling in a top-down fashion finite-domain constraint networks into multi-valued decision diagrams (MDD) or decomposable multi-valued decision graphs (MDDG) are empirically evaluated, using the cn2mddg compiler. This MDDG compiler has been enriched with various additional branching rules. These rules can be gathered into two families, the one consisting of heuristics for the satisfaction problem (which are suited to compiling networks into MDD representations) and the family of heuristics favoring decompositions (which are relevant when the MDDG language is targeted). Our empirical investigation on a large dataset shows the value of decomposability (targeting MDDG allows for compiling many more instances and leads to much smaller compiled representations). The well-known (Dom/Wdeg) heuristics appears as the best choice for compiling networks into MDD. When MDDG is the target, a new rule, based on a dynamic, yet parsimonious use of hypergraph partitioning for the decomposition purpose turns out to be the best option. As expected, the best heuristics for the satisfaction problem perform better than the best heuristics favoring decompositions when MDD is targeted, and the converse is the case when MDDG is targeted.

Keywords: Knowledge compilation, top-down compiler, heuristics.

1 Introduction

The objective of this work is to evaluate several branching heuristics (both existing ones but also new ones) which are candidates for compiling in a top-down fashion finite-domain constraint networks into decision diagrams. Two target languages are considered: the language MDD of multi-valued (deterministic) decision diagrams, and its superset, the language MDDG of decomposable multi-valued decision graphs. The significance of those two compilation languages comes from the fact that they support many useful queries in polynomial time. For instance, it is possible to determine in polynomial time whether an MDD representation (or an MDDG representation) is consistent or not, and even to count in polynomial time its number of solutions, or more generally to compute in polynomial time the number of (possibly weighted) solutions compatible with a given (partial) instantiation. It is also possible to enumerate with a polynomial delay all the solutions. Answering such queries is fundamental in a number of applications like

product configuration (see e.g., [1]), where looking for a feasible product given the user choices amounts to decide the consistency of a representation conditioned by the instantiation encoding the user choices, or probabilistic inference in Bayesian networks (computing the probability of a piece of evidence amounts to a weighted model counting query, see e.g., [3]). However, all those queries are NP-hard when the input is a constraint network.

In the Boolean case, the MDD language corresponds to the language FBDD of free binary decision diagrams [13], while MDDG corresponds to the language Decision-DNNF [22, 12]. Roughly, every internal node in an MDD representation is a decision node associated with a variable of the input constraint network and having as many children as the number of elements in the domain of the variable. In MDDG representations, internal nodes can also be decomposable $\wedge$-nodes, i.e., conjunctions of representations based on pairwise disjoint sets of variables. Despite the increase of generality obtained by accepting non-Boolean domains, the key tractable queries and transformations offered by Decision-DNNF are also offered by MDDG and MDD. Note that MDD offers some transformations that MDDG does not, and this is why this subset of MDDG is interesting in its own right. For instance, from an MDD representation of the feasible products of a configuration problem, it is possible to enumerate with a polynomial delay all the full instantiations corresponding to the non-feasible products (while this is impossible from an MDDG representation unless $\mathsf{P} = \mathsf{NP}$).

In order to generate MDDG and MDD representations, one takes advantage of the cn2mddg compiler [17], see http://www.cril.fr/KC/mddg.html. As in the Boolean case [16], an MDDG representation of a constraint network can be generated by recording the trace of a solver (in the Boolean case, a SAT solver and here, a CSP solver). Accordingly, cn2mddg is a top-down constraint network compiler, based on a CSP solver. It exploits constraint propagation and conflict analysis to guide the search. It also benefits from a specific caching technique and it detects universal constraints during the search in order to perform additional simplifications. Though cn2mddg was primarily based on a specific branching heuristics relying on betweenness centrality for promoting decompositions, we have implemented in it a number of additional heuristics for the sake of further evaluations and comparisons.

We have considered two groups of heuristics. The first one is composed of six heuristics for the consistency issue, namely the *Dom/Wdeg* heuristics (Dom/Wdeg) [15], and its by-products (the *Dom* heuristics (Dom), and the *Wdeg* heuristics (Wdeg)), the *impact-based* heuristics (IBS) [23], the *activity-based* heuristics (ABS) [20], and the *conflict-ordering* search heuristics (COS) [18]. All those heuristics were already known.

The second one gathers heuristics for promoting the generation of decomposable $\wedge$-nodes. It is composed of seven heuristics. Three of them are *static heuristics*, meaning that they are used for generating a decomposition tree (dTree) of the input network in a preliminary step, before the compilation phase. The three static heuristics considered for computing a dTree are *Min Degree* (dTree-MD), *Min Fill* (dTree-MF), *Hypergraph Partitioning* (dTree-HP). The four remaining

heuristics are *dynamic ones*, which means that each selected variable is computed during the search from the current network (thus, not prior to the search). Two of them, namely *Closeness Centrality* (CC) and *Betweenness Centrality* (BC), are based on a notion of centrality of the variables in the primal graph of the constraint network, one on the *Hypergraph Partitioning* (HP) of the dual hypergraph of the network, and the remaining one aims at computing a *Cut Set* (CS) of the primal graph. Actually, we have also considered for each of those four heuristics (H) a *parsimonious variant* (H-P) of it, meaning that a branching variable is not computed at each decision step using the heuristics, but instead, one computes a set of variables (containing all the variables that are ranked first by the heuristics) and uses all those variables successively for branching until the set becomes empty. All those heuristics except the one based on betweenness centrality (computed at each decision step) are new in the sense that they have not been tested so far, even if they are based on ingredients which are not brand new.

The branching heuristics from the two groups have been implemented in `cn2mddg`, parameterized in such a way that the compiler computes either `MDDG` representations (its default mode) or `MDD` representations (which can be done easily, by freezing the detection of disjoint components). For evaluating and comparing their relative performances in generating `MDD` representations and `MDDG` representations, `cn2mddg` (either in `MDDG` mode or in `MDD` mode) , equipped with each of the thirteen heuristics under consideration has been run on 546 benchmarks corresponding to several families of instances.

Our experiments show the value of decomposability (targeting `MDDG` allows for compiling many more instances and leads to much smaller compiled representations). The well-known (Dom/Wdeg) heuristics appears as the best choice for compiling networks into `MDD`. When `MDDG` is the target, the new rule (HP-P), based on a dynamic, yet parsimonious use of hypergraph partitioning for the decomposition purpose turns out to be the best option. As expected, the best heuristics for the satisfaction problem perform better than the best heuristics favoring decompositions when `MDD` is targeted and the converse is the case when `MDDG` is targeted.

Previous work on AND/OR search has shown the impact of variable ordering on performance. AND/OR search is a framework for solving optimization tasks in graphical models by detecting independencies in the model (decompositions). Marinescu and Dechter [19] have shown that combining static and dynamic variable orderings with problem decomposition principles results in exponential savings. Variable orderings based on decompositions are also important for compiling constraint networks in other graphical representations. Narodytska and Walsh [21] proposed heuristics to reduce the time and space requirements for compiling configuration problems into `BDD`s. These heuristics are based on the distinctive clustered and hierarchical structure of the constraint graphs and are used for a bottom-up compilation.

2 Formal Preliminaries

A *finite-domain constraint network (CN)* is a triple $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ consisting of a set $\mathcal{X} = \{X_1, \cdots, X_n\}$ of *variables*, a set $\mathcal{D} = \{D_1, \cdots, D_n\}$ of *domains*, and a set $\mathcal{C} = \{C_1, \cdots, C_m\}$ of *constraints*. Each domain D_i is a finite set containing the possible values of X_i. Each constraint C_j characterizes the combinations of values satisfying it. Formally, $C_j = (S_j, R_j)$, where $S_j = \{X_{j_1}, \cdots, X_{j_k}\}$ is a subset of variables from $\mathcal{X}$, called the *scope* of C_j, and R_j is a predicate over the Cartesian product $D_{j_1} \times \cdots \times D_{j_k}$, called the *relation* of C_j. R_j can be represented extensionally by the list of its satisfying tuples (or dually, by the list of its forbidden tuples), or intensionally by an oracle, i.e., a mapping from $D_{j_1} \times \cdots \times D_{j_k}$ to $\{0, 1\}$ which is supposed to be computable in time polynomial in its input size. The *arity* of a constraint is given by the size of its scope. Constraints of arity 2 are called *binary* and constraints of arity greater than 2 are called *non-binary*.

Example 1. Let $\mathcal{N}$ be the CN given by four variables X_1, X_2, X_3, and X_4, each of them being defined on the same domain $\{0, 1, 2\}$, and three constraints C_1, C_2, and C_3, specified by the following mathematical statements:

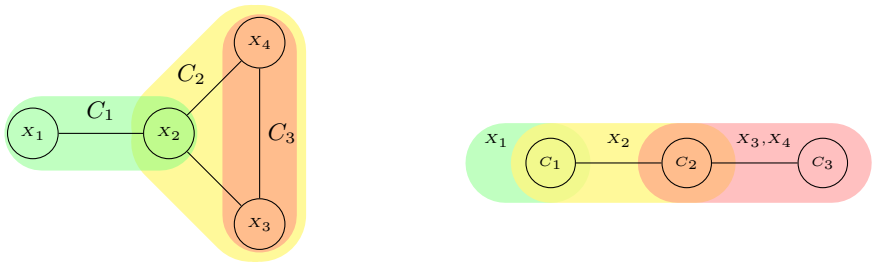
- $C_1 = (X_1 \neq X_2)$;
- $C_2 = (X_2 = 0) \vee (X_2 = 1) \vee (X_2 = X_3 + X_4 + 1)$;
- $C_3 = (X_3 > X_4)$.

Given a subset S of variables from $\mathcal{X}$, a *(decision) state* $\boldsymbol{s}$ over S is a mapping that associates with each variable X_i in S a subset $\boldsymbol{s}(X_i)$ of values in D_i. In what follows, states are often noted as union of elementary assignments, i.e., sets of the form $\{\langle X_i, x_j\rangle\}$, where $x_j \in \boldsymbol{s}(X_i)$. $scope(\boldsymbol{s})$ denotes the set S of variables over which $\boldsymbol{s}$ is defined. A state $\boldsymbol{s}$ is *partial* if $scope(\boldsymbol{s})$ is a proper subset of $\mathcal{X}$; otherwise, $\boldsymbol{s}$ is called a *full* state. A variable X_i in $scope(\boldsymbol{s})$ is *instantiated* if $\boldsymbol{s}(X_i)$ is a singleton set. The set of instantiated variables in $\boldsymbol{s}$ is noted $single(\boldsymbol{s})$. As usual, a state $\boldsymbol{s}$ is called an *instantiation* when all its variables are instantiated, i.e., $scope(\boldsymbol{s}) = single(\boldsymbol{s})$.

For a state $\boldsymbol{s}$ and a set of variables $T \subseteq scope(\boldsymbol{s})$, $\boldsymbol{s}[T]$ denotes the *restriction* of $\boldsymbol{s}$ to T, i.e., $\boldsymbol{s}[T]$ is the set $\{\langle X_i, x_j\rangle \in \boldsymbol{s} \mid X_i \in T\}$. An instantiation $\boldsymbol{s}$ *satisfies* a contraint $C_j = (S_j, R_j)$ if $S_j \subseteq scope(\boldsymbol{s})$ and $R_j(x_{j_1}, \ldots, x_{j_k}) = 1$, where $\forall l \in 1, \ldots, k$, $\langle X_{j_l}, x_{j_l}\rangle \in \boldsymbol{s}[S_j]$. A *solution* of a CN $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is a full instantiation $\boldsymbol{s}$ satisfying all constraints C_j in $\mathcal{C}$. For example, $\boldsymbol{s} = \{\langle X_1, 1\rangle, \langle X_2, 0\rangle, \langle X_3, 1\rangle, \langle X_4, 0\rangle\}$ is a solution of the CN given at Example 1.

Given a CN $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and a state $\boldsymbol{s}$ over a subset of $\mathcal{X}$, the *conditioning* $\mathcal{N} \mid \boldsymbol{s}$ of $\mathcal{N}$ by $\boldsymbol{s}$ is the CN $(\mathcal{X}', \mathcal{D}', \mathcal{C}')$ defined as follows: $\mathcal{X}' = \mathcal{X} \setminus single(\boldsymbol{s})$; with each domain D_i in $\mathcal{D}$, one associates the domain $D'_i \in \mathcal{D}'$, where $D'_i = D_i$ if $X_i \notin scope(\boldsymbol{s})$ and $D'_i = \boldsymbol{s}(D_i)$ otherwise; finally, with each constraint $C_j = (S_j, R_j)$ in $\mathcal{C}$, one associates the constraint $C'_j = (S'_j, R'_j)$ in $\mathcal{C}'$, where $S'_j = S_j \setminus single(\boldsymbol{s})$ and R'_j is the restriction of R_j to S'_j.

The *primal graph* of a CN $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is the undirected graph $PG(\mathcal{N})$ with vertex set $\mathcal{X}$ and edge set $\mathcal{E}$, such that $\{X_p, X_q\} \in \mathcal{E}$ if and only if $\{X_p, X_q\}$



(a) Primal graph (b) Dual hypergraph

Fig. 1: Graph representations of the CN given at Example 1.

is a subset of the scope S_j of some constraint C_j in $\mathcal{C}$. For instance, the primal graph of the CN given at Example 1 is depicted on Figure 1a.

The *dual hypergraph* of a CN $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is the undirected hypergraph $DH(\mathcal{N})$ with vertex set $\mathcal{C}$ and hyperedge set $\mathcal{H}$, such that $\mathcal{H} = \{H \subseteq \mathcal{C} \mid \exists X_i \in \mathcal{X} \text{ s.t. } \forall C_j \in \mathcal{C}, C_j \in H \text{ iff } X_i \in S_j\}$. Thus, every hyperedge corresponds precisely to a variable $\mathcal{X}$ which belongs to the scopes of all the constraints of the hyperedge, but does not belong to the scope of any other constraint of the input network. For instance, the dual hypergraph of the CN given at Example 1 is depicted on Figure 1b. For this example, every hyperedge contains two constraints only, but of course this is not the case in general.

Let us now introduce a few definitions suited to the target languages considered for compiling CNs.

Definition 1 (MDG). *Given a finite set $\mathcal{X}$ of finite-domain variables, the (read-once)* MDG *language over $\mathcal{X}$ is the set of all single-rooted directed acyclic graphs Δ, where leaf nodes are labelled by $\top$ (true) or $\bot$ (false), and every internal node is either a $\wedge$-node $N = \wedge(N_1, \ldots, N_i)$ or a decision node N associated with variable $X_i \in \mathcal{X}$, i.e., a deterministic $\vee$-node $N = \vee(N_1, \ldots, N_j)$ such that $D_i = \{x_{i_1}, \ldots, x_{i_j}\}$ and the arc from N to N_k ($k \in 1, \ldots, j$) is labelled by the elementary assignment $\{\langle X_i, x_{i_k}\rangle\}$. The paths of Δ must satisfy the* read-once *property: for every path from the root of Δ to a $\top$ leaf node, and for any $X_i \in \mathcal{X}$, no more than one arc can be labelled by an elementary assignment over X_i.*

For every node N in an MDG representation Δ, $Var(N)$ is defined inductively as follows:

- if N is a leaf node, then $Var(N) = \emptyset$;
- if N is a $\wedge$-node $N = \wedge(N_1, \ldots, N_i)$, then $Var(N) = \bigcup_{k=1}^{i} Var(N_i)$;
- if N is a decision node $N = \vee(N_1, \ldots, N_j)$ associated with variable X, then $Var(N) = \{X\} \cup \bigcup_{k=1}^{j} Var(N_k)$.

Let $\mathtt{s}$ be a full instantiation over $\mathcal{X}$ and let Δ be a MDG representation over $\mathcal{X}$, rooted at node N. Let $eval(N, \mathtt{s})$ be the MDG representation without any decision node, defined inductively by:

- if N is a leaf node, then $eval(N, \mathtt{s}) = N$;
- if N is a $\wedge$-node $N = \wedge(N_1, \ldots, N_i)$, then $eval(N, \mathtt{s}) = \wedge(eval(N_1, \mathtt{s}), \ldots, eval(N_i, \mathtt{s}))$;
- if N is a decision node $N = \vee(N_1, \ldots, N_j)$ associated with variable X_i, then $eval(N, \mathtt{s}) = eval(N_k, \mathtt{s})$, where $\langle X_i, x_{i_k} \rangle \in \mathtt{s}$.

$\mathtt{s}$ is a *solution* of Δ if and only $eval(N, \mathtt{s})$ evaluates to true.

The language MDDG we are interested in is the subset of MDG consisting of *decomposable* representations, those where the children of any $\wedge$-node do not share any variable. MDD is the subset of it, containing the representations without $\wedge$-nodes.

Definition 2 (MDDG, MDD). *Given a finite set $\mathcal{X}$ of finite-domain variables:*

- *the* MDDG *language over $\mathcal{X}$ is the subset of* MDG *representations Δ, where each $\wedge$-node $N = \wedge(N_1, \ldots, N_i)$ is decomposable, i.e., $\forall k, l \in 1, \ldots, i$, if $k \neq l$, then $Var(N_k) \cap Var(N_l) = \emptyset$.*
- *the* MDD *language over $\mathcal{X}$ is the subset of* MDDG *representations containing no $\wedge$-nodes.*

MDD can also be viewed as a restriction of the language of non-deterministic multi-valued decision diagrams considered in [2].

3 Heuristics for Compiling CNs

In this section, we briefly describe the branching heuristics which have been considered in our top-down constraint network compiler which targets the MDDG language (but can also be downsized to target the MDD language).

3.1 Heuristics Targeting the MDD Language

Dom/Wdeg. With (Dom/Wdeg), one selects a variable with minimum ratio of current domain size to weighted degree [15]. Each variable is associated with a weighted degree (Wdeg), which is the sum of the weights over all constraints involving the variable and at least another (unassigned) variable. A weight, initially set to one, is given to each constraint and each time a constraint causes a domain wipeout its weight is incremented by one. It is a generic state-of-the-art heuristics and the interesting is that it is adaptive, with the expectation to focus on the hard part(s) of the problem.

Dom. (Dom) is a first by-product of dom/wdeg. With (Dom), variables are ordered by considering their current domain size (the smallest cardinalities first).

Wdeg. (Wdeg) is a second by-product of (Dom/Wdeg). With (Wdeg), variables are ordered by considering their weighted degrees (the largest values first). The Wdeg score is close to the VSADS score used in model counters and compilers for propositional CNF formulae [24].

Impact-Based Search (IBS). With (IBS), one selects a variable with the highest impact, where impact measures the importance of a variable in reducing the search space [23]. An estimation of the size of the search space $S(P)$ is the product of every variable domain size:

$$S(P) = \prod_{x \in X} |D_x|$$

The impact of a variable assignment at a decision node k is computed by the ratio of the search space reduction as:

$$I(x = a) = 1 - \frac{S(P^k)}{S(P^{k-1})}$$

Note that if $x = a$ leads to a failure, then $I(x = a) = 1$, which is the maximum impact as $S(P^k) = 0$. It is easy to see that this heuristics can be used for value selection as well. For variable selection, the average impact is preferred, computed over the remaining values in its domain divided by its current domain size. For more accurate results, a forgetting strategy is used in order to give less importance to past variable assignments.

Activity-Based Search (ABS). With this heuristics, one selects a variable with the highest activity, where activity is measured by the times the domain of each variable is reduced during the search [20]. This heuristics is motivated by the key role of propagation in constraint programming and relies on a decaying sum to forget the oldest statistics progressively. The activities are initialized by making random probing in the search space.

More formally, the activity $A(x)$, of each variable x is updated at each node k of the search tree regardless of the outcome (success or failure) by the following two rules:

i. $A^k(x) = A^{k-1}(x) * \gamma$, where $0 \leq \gamma \leq 1$, $|D_x^k| > 1$ and $D_x^k = D_x^{k-1}$
ii. $A^k(x) = A^{k-1}(x) + 1$, where $D_x^k \subset D_x^{k-1}$

Conflict-Ordering Search (COS). This heuristics is considered more as a repairing mechanism than as a heuristics, that can be combined with any (underlying) variable ordering heuristics (e.g., (Dom/Wdeg)) [18]. When the solver needs to backtrack, the last conflicting variables, recorded during search, are selected in priority until they are all instantiated without causing any failure. Otherwise, in normal mode, the variable ordering heuristics is the one that decides the next variable.

3.2 Heuristics Targeting the `MDDG` Language

We have also considered eleven heuristics targeting the `MDDG` language, thus promoting the generation of decomposable $\wedge$-nodes. The rationale for it is the gain in succinctness which mainly results in the generated representation when such nodes are allowed. Indeed, consider a constraint network $\mathcal{N}$ with x variables

each of them having a domain of size $d > 1$ (for simplicity reasons). Suppose that every MDD representation of $\mathcal{N}$ has a size which is a fraction k ($0 < k \leq 1$) of the search space of all instantiations explored for generating it (which implies that the corresponding compilation time will be at least as high). Suppose now that a decomposition $(\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3)$ of the set $\mathcal{X}$ of variables of $\mathcal{N}$ has been found. Such a decomposition is a tripartition of $\mathcal{X}$ such that for every assignment $\boldsymbol{x_1}$ over $\mathcal{X}_1$, the conditioned network $\mathcal{N} \mid \boldsymbol{x_1}$ has (at least) two disjoint components, one over the variables of $\mathcal{X}_2$ and one over the variables of $\mathcal{X}_3$. Then a decomposable $\wedge$-node can be generated. With $|\mathcal{X}_i| = x_i$ ($i \in \{1, 2, 3\}$), the size of the resulting MDDG representation of $\mathcal{N}$ will be at most $d^{x_1} \times (k \times d^{x_2} + k \times d^{x_3})$, which is always strictly smaller than the size $k \times d^{x_1+x_2+x_3}$ of the MDD representation of $\mathcal{N}$, unless $x_2 = x_3 = 1$ (in which case the decomposition is trivial). One can also easily check that the size of the resulting MDDG representation of $\mathcal{N}$ is as small as the decomposition is balanced, i.e., as x_2 and x_3 are close. More formally, $x_2^* = \lfloor \frac{x_2+x_3}{2} \rfloor$ and $x_3^* = \lceil \frac{x_2+x_3}{2} \rceil$ minimize the value of $d^{x_2} + d^{x_3}$ when the sum $x_2 + x_3$ is fixed (which is the case here whenever $\mathcal{X}_1$ has been set since $x_1 + x_2 + x_3 = x$). Accordingly, finding a "good" decomposition $(\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3)$, i.e., a decomposition leading to an MDDG representation of "small" size) amounts to minimizing x_1 while making x_2 and x_3 as close as possible. It turns out that those two objectives can be antagonistic so that a trade-off must be looked for.

Static Heuristics. Static heuristics consist in generating a decomposition tree (dTree) prior to the compilation, which will be used to make precise the branching variables to be considered at each step. This approach is similar to the one used by the C2D compiler for propositional CNF formulae, which targets the Decision-DNNF language [8, 9], see reasoning.cs.ucla.edu/c2d/. Roughly, a dTree for a constraint network is a full binary tree which induces a recursive decomposition of the network (for more details, see e.g., [10]).

The three static heuristics considered for computing a dTree are *Min Degree (dTree-MD)*, *Min Fill* (dTree-MF), and *Hypergraph Partitioning* (dTree-HP).

Min Degree (dTree-MD). The (dTree-MD) heuristics is used for generating a dTree in a bottom-up way and is driven by a variable elimination ordering: the variables are ordered by increasing degrees in the primal graph of the input network. The generation of the dTree starts with the leaves (each of them being associated with the scope of a constraint of the input network), and the initial forest (set of trees) to be dealt with is composed of those leaves. Then the variables of the network are considered according to the elimination ordering, and each time a variable is picked up, a tree is computed so that all the trees of the current forest which contain this variable are children of the resulting tree. Once all the variables have been processed, a forest of dTrees is generated (it consists of a single dTree when the primal graph of the network considered at start is connected).

Min Fill (dTree-MF). The (dTree-MF) heuristics is also used for generating a dTree in a bottom-up way and is driven by a variable elimination ordering

(which is different from the one corresponding to the *Min Degree* heuristics): the variables are ordered by increasing numbers of non-connected neighbours in the primal graph of the input network. Then the generation of the dTree is made as in the case of *Min Degree* but considering the *Min Fill* elimination ordering instead.

Hypergraph Partitioning (dTree-HP). The hypergraph partitioning heuristics considers the dual hypergraph of the input network and generates a dTree for it in a top-down way. It looks for a subset of the set of hyperedges of the current network containing as few elements as possible such that removing them leads to a hypergraph containing (at least) two disjoint components having sizes as close as possible. When the variables corresponding to the selected hyperedges are instantiated (whatever the way they are assigned) it is guaranteed that the current network conditioned by the corresponding assignment has at least two disjoint components, so that a decomposable $\wedge$-node can be generated in the compiled form. This set of variables is the cut set of the root of the dTree, and then the hypergraph partitioning approach proceeds recursively considering the disjoint hypergraphs which are generated by removing from the current hypergraph the hyperedges which have been selected at the previous step. One takes advantage of the partitioner `PaToH` – Partitioning Tools Hypergraph, v. 3.2 (http://bmi.osu.edu/∼umit/software.html) [7] to do the job.

As explained above, hypergraph partitioning goes further than minimal cutting by taking account of the sizes of the subgraphs which are generated, which must be balanced, i.e., their difference in terms of numbers of vertices must be below a preset bound. This comes with a significant complexity increase since determining whether there exists a hypergraph partition of $DH(\mathcal{N})$ corresponding to a decomposition $(\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3)$ of $\mathcal{N}$ such that $\#(\mathcal{X}_1) \leq c$ and $|\#(\mathcal{X}_2) - \#(\mathcal{X}_3)| \leq d$ (where c and d are two given bounds) is NP-complete. This departs deeply from the other heuristics considered in the paper which can be computed in polynomial time. In our experiments, we looked for 2-partitionings (i.e., one does not try to split the given hypergraph into more than two disjoint components) and used the default setting of `PaToH`.

Dynamic Heuristics. The four (pairs of) remaining heuristics we have considered are *dynamic ones*: each selected variable is computed during the search from the current network (thus, not prior to the search, from the input network), such that the propagations resulting from the previous variable assignments and leading to simplify the network, are taken into account. This can have a huge impact on the compilation process (both on the compilation times and on the sizes of the compiled forms).

We have considered two heuristics based on a notion of centrality (*Betweenness Centrality* (BC) vs. *Closeness Centrality* (CC)), which favor the decomposition of the current CN into components of balanced sizes by targeting the variables which are in some sense the central ones in its primal graph. We have also considered a heuristics based on *Hypergraph Partitioning* (HP) but this time

it is not in the objective of generating first a dTree from the dual hypergraph of the input network but considering instead the dual hypergraph of the current network. Finally, we have also taken into account a *Cut Set* heuristics (CS), which focuses on identifying variables to be instantiated in order to split the current network into (at least) two disjoint components (whatever their sizes).

Each of those four heuristics (H) may point out several branching variables as the best ones. When they are several best variables, they are ordered following their decreasing (Dom/Wdeg) score. When (H) is used in its default mode, the first variable w.r.t. this ordering is selected. When (H) is run in a parsimonious mode (H-P), the score of each variable from the current set of best candidates is not re-computed after each variable assignment but all the variables from this set are successively considered as branching variables up to exhaustion.

Closeness Centrality (CC). Closeness centrality is a measure of the centrality of a node in a graph [4]. Given a node X_i in a graph (here, the primal graph of the current CN in which the nodes can be identified as with the variables labelling them), the score $cc(X_i)$ is calculated as the sum of the lengths of the shortest paths between X_i and all other vertices in the graph. Thus the more central a vertex is, the closer it is to all other vertices. Formally:

$$cc(X_i) = \frac{1}{\Sigma_{X_j \neq X_i} d(X_i, X_j)}$$

where $d(X_i, X_j)$ is the geodesic distance between nodes X_i and X_j which belong to the same connected component of the graph. We also assume that the set of vertices of the primal graph of the component of the current CN which contains X_i is not a singleton, so that $\Sigma_{X_j \neq X_i} d(X_i, X_j) \neq 0$ (indeed in the remaining case, there is no option: X_i must be chosen in the component). The computation of the values of all $cc(X_i)$ when X_i varies in the set of vertices of the primal graph of the current CN can be done in time polynomial in the size of this graph, via a repeated use of breadth-first search from X_i or using Floyd-Warshall algorithm. We took advantage of the code of Floyd-Warshall algorithm from Boost Graph Library http://www.boost.org/doc/libs/1_64_0/libs/graph/doc/ for implementing *cc*. Clearly enough, by instantiating first the most central variables, the objective is to find out a decomposition $(\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3)$ of the set of variables of the current network for which the cardinalities of $|\mathcal{X}_2|$ and $|\mathcal{X}_3|$ are close, however the number of variables in $\mathcal{X}_1$ can be large and one does not try to minimize it.

Betweenness Centrality (BC). Betweenness centrality is another measure of the centrality of a node in a graph [5, 6]. The score $bc(X_i)$ is equal to the number of shortest paths from all nodes to all others that pass through X_i. Formally,

$$bc(X_i) = \Sigma_{X_j \neq X_i \neq X_k} \frac{\sigma_{X_i}(X_j, X_k)}{\sigma(X_j, X_k)}$$

where X_i, X_j, X_k are nodes of the same connected component of the given network, $\sigma(X_j, X_k)$ is the number of shortest paths from X_j to X_k, and $\sigma_{X_i}(X_j, X_k)$

are the number of those paths passing through X_i. Thus, for the CN $\mathcal{N}$ given at Example 1, X_2 is the unique variable maximizing the value of bc.

Interestingly, computing the betweenness centralities of all nodes in $(\mathcal{X}, \mathcal{E})$ can be done in time $\mathcal{O}(n.m)$, where n is the cardinality of $\mathcal{X}$ and m is the cardinality of $\mathcal{E}$. In practice, the computation of $bc(X_i)$ for each node X_i of the primal graph $(\mathcal{X}, \mathcal{E})$ of a CN is efficient enough so that it can be computed dynamically, i.e., for each network encountered during the compilation process. Again, we took advantage of the implementation of betweenness centrality available in the Boost Graph Library.

The rationale for instantiating first the most central variables (as to (BC)) is the same as the one for (CC), i.e., to split the network into two parts of similar sizes. (BC) can also be seen as an alternative of community structure [14], where instead of constructing communities by adding the strongest edges to an initially empty vertex set, it constructs them by progressively removing edges from the original graph. The community method detects which edges are most central to communities, while betweenness finds those edges that are most "between" communities. The community structure is thus more relevant for the bottom-up approaches to knowledge compilation like the one presented in [21].

Hypergraph Partitioning (HP). The approach is the same as the one described above, except that one does not compute a full dTree of the given hypergraph, but only its root. Note that when the current hypergraph has several disjoint components, the cut set returned by `PaToH` is empty so that no branching variable is defined. This is harmless when `MDDG` is targeted since this problem cannot happen in this case (a decomposable $\wedge$-node would have been introduced before a branching variable is sought). However, this is still problematic in the case `MDD` is targeted. To deal with it, we switch to the (Dom/Wdeg) heuristics when such a pathological situation occurs.

Cut Set (CS). A (2-way) cut of a (undirected) graph is a partition of its vertices into two, non-empty sets. The corresponding cut set is the set of all edges between the two sets of vertices. A minimal cut set is a cut set of minimal size. Removing all the edges of the cut set of a graph leads to split it into two disjoint components. A minimal cut set of a graph $(\mathcal{X}, \mathcal{E})$ can be computed in time $\mathcal{O}(n.m^2)$ where n is the number of vertices of $\mathcal{X}$ and m is the number of edges in $\mathcal{E}$ [11]. Once a cut set of the primal graph of the current CN has been computed, one selects one variable per edge in the cut set. By construction, eliminating the variables of the resulting set in the primal graph is enough for ensuring that the resulting graph contains two disjoint components.

Using the cut set heuristics on the primal graph of the current constraint network, one computes decompositions $(\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3)$ of the set of variables of the current network. One does not take care at all of balancing $|\mathcal{X}_2|$ and $|\mathcal{X}_3|$. In our implementation, we exploited the Stoer / Wagner algorithm [25] for min-cut available in the Boost Graph Library. As for (HP), if $\mathcal{X}_1$ turns out to be empty and `MDD` is targeted, then we switch to the (Dom/Wdeg) heuristics.

4 Empirical Evaluation

Setup. We have considered 546 CNs from 15 data sets.[1] Those data sets correspond to several families of problems, including configuration problems, graph coloring, scheduling problems, frequency allocation problems. For some instances, the constraints are represented extensionally, by the list of satisfying tuples or by the list of forbidden tuples; for other instances, they are given in intension. Each instance has been compiled using `cn2mddg` equipped with the various heuristics we focused on.

Our experiments have been conducted on a Quadcore Intel XEON X5550 with 32GiB of memory. A time limit of 1800s for the off-line compilation phase (including the dTree generation when relevant) and a total amount of 8GiB of memory for storing the resulting compiled representation have been considered for each instance.

Results. We have first evaluated a random heuristics serving as base line for compiling the 546 benchmarks (for this heuristics, the decision variables are selected at random under a uniform distribution). Based on the random heuristics, `cn2mddg` has been able to compile 271 instances when `MDDG` was targeted and 238 instances when `MDD` was targeted.

We have then evaluated all the heuristic methods discussed before. In Figure 2, we report their performances on a cactus plot where the x axis represents the number of "solved" (i.e., compiled) instances (numbers are displayed in the legend) and the y axis the CPU time needed per method, in logarithmic scale. Dotted lines correspond to `MDD` representations and solid to `MDDG` representations.

The general picture is that compiling constraint networks into `MDDG` allows to solve constantly more instances than when compiling to `MDD`, independently of the heuristics involved. The performance shift between the best approaches (ABS) and (Dom/Wdeg) targeting `MDD` and the one targeting `MDDG` (HP-P) is equal to 133-134 instances (over 546), which is significant. In more detail, when `MDD` is targeted, (ABS) solves 258 instances in the time and memory given and (Dom/Wdeg) solved 257 instances; the number of instances solved by the virtual best solver induced by the set of heuristics is 388. Its performance shift with (ABS) and/or (Dom/Wdeg) is thus large. When `MDD` is targeted, heuristics having instances solved by them and only them are (ABS), (IBS) and (dTree-MF). When `MDDG` is targeted, the best heuristics is (HP-P) with 391 instances solved; in this case, the number of instances solved by the virtual best solver induced by the set of heuristics is 404. Its performance shift with (HP-P) is thus quite limited (13 instances only). When `MDDG` is targeted, heuristics having instances solved by them and only them are (HP-P), (BC), (CC) and (dTree-MF).

When the target is `MDD`, as expected, heuristics that target to `MDD` work better than heuristics targeting `MDDG`. However, the performance shift is not that huge (it amounts to 27 instances, the worst heuristics for `MDD` being (CC-P) with 231

[1] From www.cril.fr/~lecoutre/benchmarks.html, github.com/MiniZinc/minizinc-benchmarks, and www.itu.dk/research/cla/externals/clib/

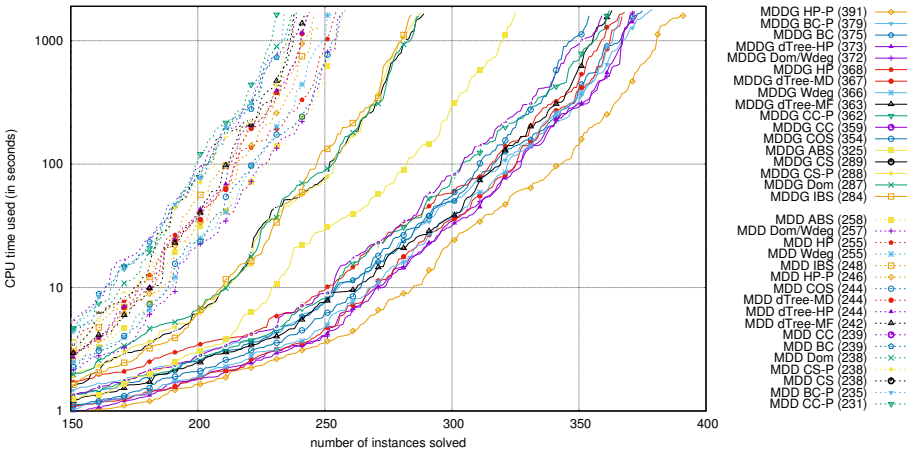


Fig. 2: Number of instances "solved" per method as the time allowed increases.

instances solved). Globally speaking, heuristics from the first group (the search-based ones) appear as slightly better than the other heuristics but the difference is not tremendous. Thus, one can also observe that the (HP) heuristics performs quite well when `MDD` is targeted, with 255 instances solved, despite the fact that it is designed for promoting decompositions. Contrastingly, some heuristics performed quite bad, actually as bad as the random heuristics ((Dom), (CS), (CS-P)) or even worse than it ((BC-P) and (CC-P)).

When `MDDG` is targeted, heuristics that take into account the graph structure are more suited. However, in that case, the performances shifts between the heuristics from the two groups and between the heuristics from the second group are important. Thus, the worst heuristics for `MDDG` is (IBS) which solves only 284 instances, showing a shift of 107 instances with (HP-P). Other search heuristics like (IBS), (Dom) and (ABS) also behave very poorly. Nevertheless, all the heuristics which have been considered performed better than the random one (with a shift at least equal to 13 instances). Search heuristics that take into account structural information, namely the degree like (Wdeg) are more efficient. It is interesting to note that the efficiency of (Dom/Wdeg) is mainly due to its second by-product (Wdeg), since when we separately try (Dom) and (Wdeg), there are 79 instances of difference in favor of (Wdeg). Such a difference does not appear when `MDD` is the target, demonstrating that structure plays a critical role for producing meaningful decompositions. Focusing now on the heuristics designed for promoting decompositions, significant shifts can also be observed, (CS-P) and (CS) with 288 and 289 instances solved, respectively, being far below the other heuristics (the next worst is (CC) with 359 instances solved). The best members from this second group are (HP-P), and then (BC-P), (BC), and (dTree-HP). Compared to (CS-P), (CS), (dTree-MD) and (dTree-MF), this suggests that the fact that the generated decompositions are balanced has a major

A / B	Dom/Wdeg	Dom	Wdeg	IBS	ABS	COS	dTree-MD	dTree-MF	dTree-HP	CC-P	BC-P	HP-P	CS-P	CC	BC	HP	CS
Dom/Wdeg	-	85	8	89	48	19	7	12	1	27	17	5	85	30	20	5	84
Dom	0	-	0	11	3	0	1	4	0	12	8	0	16	13	9	0	16
Wdeg	2	79	-	83	46	15	8	14	3	27	17	5	82	29	19	4	81
IBS	1	8	1	-	3	2	2	5	1	14	9	0	16	15	9	1	16
ABS	1	41	5	44	-	5	6	10	2	20	15	5	45	22	17	5	44
COS	1	67	3	72	34	-	6	13	2	21	15	5	69	23	17	4	68
dTree-MD	2	81	9	85	48	19	-	10	0	24	14	3	81	27	17	4	80
dTree-MF	3	80	11	84	48	22	6	-	1	20	11	2	78	23	14	6	77
dTree-HP	2	86	10	90	50	21	6	11	-	27	17	5	86	30	20	5	85
CC-P	17	87	23	92	57	29	19	19	16	-	1	3	77	3	5	19	76
BC-P	24	100	30	104	69	40	26	27	23	18	-	4	93	21	4	26	92
HP-P	24	104	30	107	71	42	27	30	23	32	16	-	103	35	19	27	102
CS-P	1	17	4	20	8	3	2	3	1	3	2	0	-	5	4	1	0
CC	17	85	22	90	56	28	19	19	16	0	1	3	76	-	3	19	75
BC	23	97	28	100	67	38	25	26	22	18	0	3	91	19	-	25	90
HP	1	81	6	85	48	18	5	11	0	25	15	4	81	28	18	-	80
CS	1	18	4	21	8	3	2	3	1	3	2	0	1	5	4	1	-

Table 1: Dominance matrix (`MDDG`).

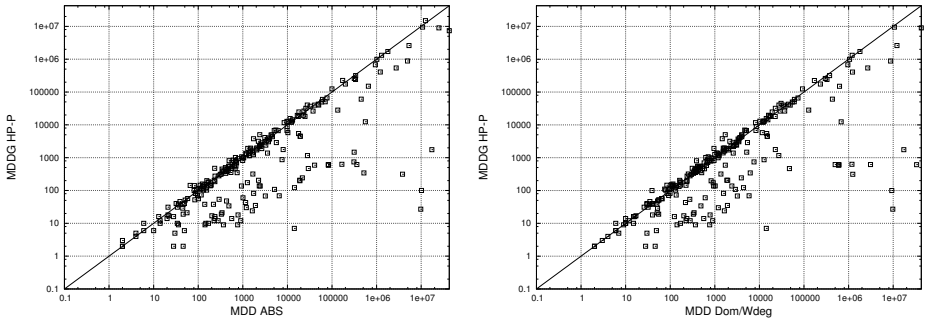
A / B	Dom/Wdeg	Dom	Wdeg	IBS	ABS	COS	dTree-MD	dTree-MF	dTree-HP	CC-P	BC-P	HP-P	CS-P	CC	BC	HP	CS
Dom/Wdeg	-	19	4	10	2	1	13	16	13	26	22	11	19	19	19	2	19
Dom	0	-	0	3	0	0	0	3	0	11	7	0	12	11	7	0	12
Wdeg	2	17	-	11	2	1	12	15	12	24	20	9	20	17	16	4	20
IBS	1	13	4	-	1	1	10	13	10	22	18	8	15	16	16	1	15
ABS	3	20	5	11	-	2	15	18	15	28	24	13	21	19	19	4	21
COS	2	20	4	11	2	-	14	17	14	27	23	12	20	19	19	3	20
dTree-MD	0	6	1	6	1	0	-	3	0	13	9	1	15	14	10	0	15
dTree-MF	1	7	2	7	2	1	1	-	1	11	7	2	13	12	8	1	13
dTree-HP	0	6	1	6	1	0	0	3	-	13	9	1	15	14	10	0	15
CC-P	0	4	0	5	1	0	0	0	0	-	0	0	3	1	1	0	3
BC-P	0	4	0	5	1	0	0	0	0	4	-	0	6	5	1	0	6
HP-P	0	8	0	6	1	0	3	6	3	15	11	-	15	14	10	0	15
CS-P	0	12	3	5	1	0	9	9	9	10	9	7	-	4	7	0	0
CC	1	12	1	7	0	0	9	9	9	9	9	7	5	-	4	1	5
BC	1	8	0	7	0	0	5	5	5	9	5	3	8	4	-	1	8
HP	0	17	4	8	1	0	11	14	11	24	20	9	17	17	17	-	17
CS	0	12	3	5	1	0	9	9	9	10	9	7	0	4	7	0	-

Table 2: Dominance matrix (`MDD`).

impact, which is more significant than the fact that the decomposition is computed *ex ante* via the generation of a dTree, or on the fly. However, this does not explain the deceiving performances of (CC-P) and (CC) which try as well to split the network in two balanced parts by cutting it "in the middle".

Let us now provide a more detailed, pairwise comparison of the performances of the various heuristics. Tables 1 and 2 report (respectively) the dominance matrices of the heuristics under consideration for the two target languages. Each dominance matrix M gives for every pair of heuristics A, B, the number $M[A, B]$ of instances solved by A and not by B. Thus, when $M[A, B] = 0$ and $M[B, A] \neq 0$, A is strictly dominated by B.

For space reasons, we cannot detail the results obtained about the compilation times and the sizes of the compiled forms for the various heuristics. The benchmarks used and a detailed comparison in terms of compilation times and sizes of the compiled forms, reported in a number of scatter plots, of all the heuristics used can be found at http://www.cril.fr/KC/mddg.html. Roughly speaking, it turns out that the best heuristics in terms of number of instances



(a) (ABS) vs. (HP-P) (b) (Dom/Wdeg) vs. (HP-P)

Fig. 3: Comparing the sizes of the MDD representations.

solved are also the best heuristics when those two measures are considered instead (the intuition being that one obtains less time-out and/or memory-out precisely because the compilation times and the sizes of compiled forms are shorter ones). As expected, targeting MDDG leads to more compact representations than when MDD is targeted instead. The scatter plots on Figure 3 illustrate it by focusing on the best heuristics for MDD (ABS) and (Dom/Wdeg) and the best heuristics for MDDG (HP-P). Each dot represents an instance. The size (in number of arcs) of the resulting compiled form, using the compiler cn2mddg equipped with the heuristics corresponding to the x-axis (resp. y-axis) is given by its x-coordinate (resp. y-coordinate). Logarithmic scales are used for both coordinates. These empirical results clearly illustrates the value of the notion of decomposition in the compilation process from the practical side.

Again, due to space limitations, we cannot provide a differential analysis of the heuristics performances depending on the family of benchmarks. However, the family chosen has a clear impact on the performances. For instance, when MDDG is targeted, (HP-P) proved better than (Dom/Wdeg) for compiling Bayesian networks and (Dom/Wdeg) proved better than (HP-P) for instances of the frequency allocation problem (FAP) which are difficult to decompose.

5 Conclusion

In this work, we have evaluated several branching heuristics for the top-down compilation of constraint networks into the MDD language and into the MDDG language, through the use of the cn2mddg compiler. Our evaluation on a large dataset demonstrated that the decomposability of the constraint graph allows to compile many more instances and offers much smaller compiled representations. In particular, when compiling networks into MDD, the (Dom/Wdeg) heuristics proved to be the best, while for MDDG representations, a new heuristics (HP-P) based on dynamic, yet parsimonious hypergraph partitioning was the best performer.

References

1. Amilhastre, J., Fargier, H., Marquis, P.: Consistency restoration and explanations in dynamic CSPs application to configuration. Artificial Intelligence 135(1-2), 199–234 (2002)
2. Amilhastre, J., Fargier, H., Niveau, A., Pralet, C.: Compiling CSPs: A complexity map of (non-deterministic) multivalued decision diagrams. International Journal on Artificial Intelligence Tools 23(4) (2014)
3. Bart, A., Koriche, F., Lagniez, J.M., Marquis, P.: An improved CNF encoding scheme for probabilistic inference. In: Proc. of ECAI'16. pp. 613–621 (2016)
4. Bavelas, A.: Communication patterns in task-oriented groups. Jounal of the Acoustical Society of America 22(6), 725730 (1950)
5. Brandes, U.: A faster algorithm for betweenness centrality. Journal of Mathematical Sociology 25(2), 163–177 (2001)
6. Brandes, U.: On variants of shortest-path betweenness centrality and their generic computation. Social Networks 30(2), 136–145 (2008)
7. Catalyürek, U., Aykanat, C.: PaToH (Partitioning Tool for Hypergraphs), pp. 1479–1487. Encyclopedia of Parallel Computing (2011)
8. Darwiche, A.: Decomposable negation normal form. Journal of the ACM 48(4), 608–647 (2001)
9. Darwiche, A.: New advances in compiling CNF into decomposable negation normal form. In: Proc. of ECAI'04. pp. 328–332 (2004)
10. Darwiche, A., Hopkins, M.: Using recursive decomposition to construct elimination orders, jointrees, and dtrees. In: Symbolic and Quantitative Approaches to Reasoning with Uncertainty, 6th European Conference, ECSQARU 2001, Toulouse, France, September 19-21, 2001, Proceedings. pp. 180–191 (2001)
11. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. J. ACM 19(2), 248–264 (1972), http://doi.acm.org/10.1145/321694.321699
12. Fargier, H., Marquis, P.: On the use of partially ordered decision graphs in knowledge compilation and quantified Boolean formulae. In: Proc. of AAAI'06. pp. 42–47 (2006)
13. Gergov, J., Meinel, C.: Efficient analysis and manipulation of OBDDs can be extended to FBDDs. IEEE Transactions on Computers 43(10), 1197–1209 (1994)
14. Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. Proc. of the National Academy of Sciences 99(12), 7821–7826 (2002)
15. Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proc. of ECAI'04. pp. 146–150 (2004)
16. Huang, J., Darwiche, A.: The language of search. Journal of Artificial Intelligence Research 29, 191–219 (2007)
17. Koriche, F., Lagniez, J.M., Marquis, P., Thomas, S.: Compiling constraint networks into multivalued decomposable decision graphs. In: Proc. of IJCAI'15. pp. 332–338 (2015)
18. Lecoutre, C., Sais, L., Tabary, S., Vidal, V.: Reasoning from last conflict(s) in constraint programming. Artif. Intell. 173(18), 1592–1614 (2009)
19. Marinescu, R., Dechter, R.: Dynamic orderings for AND/OR branch-and-bound search in graphical models. In: Proc. of ECAI '06. pp. 138–142 (2006)
20. Michel, L., Hentenryck, P.V.: Activity-based search for black-box constraint programming solvers. In: Proc. of CPAIOR'12. pp. 228–243 (2012)

21. Narodytska, N., Walsh, T.: Constraint and variable ordering heuristics for compiling configuration problems. In: Proc. of IJCAI '07. pp. 149–154 (2007)
22. Oztok, U., Darwiche, A.: On compiling CNF into decision-DNNF. In: Proc. of CP'14. pp. 42–57 (2014)
23. Refalo, P.: Impact-based search strategies for constraint programming. In: Proc. of CP'04. pp. 557–571 (2004)
24. Sang, T., Beame, P., Kautz, H.A.: Performing Bayesian inference by weighted model counting. In: Proc. of AAAI'05. pp. 475–482 (2005)
25. Stoer, M., Wagner, F.: A simple min-cut algorithm. J. ACM 44(4), 585–591 (1997)