Compiling Constraint Networks into Multivalued Decomposable Decision Graphs

Frédéric Koriche, Jean-Marie Lagniez, Pierre Marquis, Samuel Thomas
CRIL-CNRS and Université d’Artois, Lens, France
{koriche, lagniez, marquis, thomas}@cril.univ-artois.fr

Abstract
We present and evaluate a top-down algorithm for compiling finite-domain constraint networks (CNs) into the language MDDG of multivalued decomposable decision graphs. Though it includes Decision-DNNF as a proper subset, MDDG offers the same key tractable queries and transformations as Decision-DNNF, which makes it useful for many applications. Intensive experiments showed that our compiler cn2mddg succeeds in compiling CNs which are out of the reach of standard approaches based on a translation of the input network to CNF, followed by a compilation to Decision-DNNF. Furthermore, the sizes of the resulting compiled representations turn out to be much smaller (sometimes by several orders of magnitude).

1 Introduction
Constraint Programming (CP) has long been recognized as a paradigm of choice for representing and solving combinatorial problems [Rossi et al., 2006]. Knowledge about the problem is represented in a compact and intuitive way, using a constraint network (CN), which involves a set of variables associated with their domain of values, and a collection of constraints specifying that some subsets of values cannot be used together. Despite its undoubtable success in IA, one of the key remaining challenges of CP is to provide performance guarantees for query answering, which often amounts to solving instances of NP-hard problems. As emphasized in [Freuder and O’Sullivan, 2014], this issue is critical in on-line applications, such as configuration softwares [Junker, 2006] and recommender systems [Cambazard et al., 2010], where queries supplied “on the fly” by users, are to be answered in real time.

The aim of this paper is to address this challenge using knowledge compilation [Darwiche and Marquis, 2002]. The overall idea is to convert a constraint language into a target compilation language that supports inference tasks (often classified as queries and transformations) in polynomial time. Thus, while many queries are intractable when the input is a CN, they become tractable from a compiled representation of it, enabling on-line performance guarantees when the compiled representation remains small enough.

Specifically, we present a top-down algorithm cn2mddg for compiling finite-domain CNs into multivalued decomposable decision graphs. The input of cn2mddg is a CN represented in the XCSP 2.1 format [Roussel and Lecoutre, 2009]. The output of our compilation algorithm is a representation of the solutions of the CN in the language MDDG of multivalued decomposable decision graphs. MDDG is precisely the extension to non-Boolean domains of the language DDG [Fargier and Marquis, 2006] also known as Decision-DNNF [Oztok and Darwiche, 2014]: it is based on decomposable A-nodes and (multivalued) decision nodes. Similarly to Decision-DNNF, the MDDG language offers a number of tractable queries, including (possibly weighted) solution finding and counting, solution enumeration (solutions can be enumerated with polynomial delay), and optimization w.r.t. a linear objective function. It also offers tractable transformations, especially the conditioning one (i.e., the instantiation of variables, and more generally, the addition of unary constraints).

cn2mddg benefits from a specific caching technique, a new variable ordering heuristic based on betweenness centrality and it detects universal constraints during the search in order to perform additional simplifications. We performed an intensive evaluation of cn2mddg on a number of benchmarks (173) from several data sets (15). Given the availability of Decision-DNNF compilers, a way to compile CNs is to follow a translate-then-compile schema: one first encodes the input network into a CNF formula, then one takes advantage of a compiler like c2d [Darwiche, 2004] or Dsharp [Muise et al., 2012] to turn the resulting formula into a Decision-DNNF representation. Based on the results reported from our experiments, it turns out that whatever the encodings used, both the huge number of Boolean variables in the generated CNF formulae, and the structure loss inherent to the CNF format (compared to the constraint network one) make the translate-then-compile approaches impractical in many cases. Contrastingly, our compiler cn2mddg proved much more robust since it succeeded in compiling many CNs which are out of reach of the translate-then-compile approaches. Moreover, the sizes of the resulting compiled representations turn out to be much smaller (sometimes by several orders of magnitude).

The run-time code of our compiler, as well as the translators and the benchmarks used in our experiments, and additional empirical results, can be downloaded from www.cril.fr/KC/
2 Formal Preliminaries

A finite-domain constraint network (CN) is a triple \( \mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C}) \) consisting of a set \( \mathcal{X} = \{X_1, \ldots, X_n\} \) of variables, a set \( \mathcal{D} = \{D_1, \ldots, D_m\} \) of domains, and a set \( \mathcal{C} = \{C_1, \ldots, C_m\} \) 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 \( S_j = \{X_{j_1}, \ldots, X_{j_k}\} \) 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:

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

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

For a state \( s \) and a set of variables \( T \subseteq \text{scope}(s) \), \( s[T] \) denotes the restriction of \( s \) to \( T \), i.e., \( s[T] = \{X_i, x_j\} \in s \mid X_i \in T \). An instantiation \( s \) satisfies a constraint \( C_i = (S_i, R_i) \) if \( S_i \subseteq \text{scope}(s) \) and \( R_i(x_j, \ldots, x_{k_j}) = 1 \), where \( \forall i \in 1, \ldots, k, \langle x_j, \ldots, x_{k_j} \rangle \in s[S_i] \). A solution of a CN \( \mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C}) \) is a full instantiation \( s \) satisfying all constraints \( C_j \) in \( \mathcal{C} \). For example, \( s = \{\{X_1, 1\}, \{X_3, 0\}, \{X_3, 1\}, \{X_4, 0\}\} \) is a solution of the CN given at Example 1.

Given a CN \( \mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C}) \) and a state \( s \) over a subset of \( \mathcal{X} \), the conditioning \( \mathcal{N} \mid s \) of \( \mathcal{N} \) by \( s \) is the CN \( (\mathcal{X}', \mathcal{D}', \mathcal{C}') \) defined as follows: \( \mathcal{X}' = \mathcal{X} \setminus \text{single}(s) \); with each domain \( D_i \) in \( \mathcal{D}' \), one associates the domain \( D'_i = D_i \) if \( X_i \notin \text{scope}(s) \) and \( D'_i = s(D_i) \) otherwise; finally, with each constraint \( C_i' = (S_j', R_j') \) in \( \mathcal{C}' \), one associates the constraint \( C_j' = (S_j, R_j') \) in \( \mathcal{C} \), where \( S_j' = S_j \setminus \text{single}(s) \) 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 \( G \) with vertex set \( \mathcal{X} \) and edge set \( \mathcal{E} \), such that \( \{x_p, x_q\} \in \mathcal{E} \) if and only if \( \{x_p, x_q\} \) 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 1.

3 The MDDG Language

Let us first consider the language MDDG of multivalued decision graphs:

Definition 1 (MDDG) Given a finite set \( \mathcal{X} \) of finite-domain variables, the (read-once) MDDG language over \( \mathcal{X} \) is the set of all single-rooted directed acyclic graphs \( \Delta \), where leaf nodes are labelled by \( \top \) (true) or \( \bot \) (false), and every internal node is either a \( \land \)-node \( N = \land(N_1, \ldots, N_k) \) or a decision node \( N \) associated with variable \( X_i \in \mathcal{X} \), i.e., a deterministic \( \lor \)-node \( N = \lor(N_1, \ldots, N_j) \) such that \( D_i = \{x_1, \ldots, x_j\} \) and the arc from \( N \) to \( N_k \) (\( k \in 1, \ldots, j \)) is labelled by the elementary assignment \((\langle X_i, x_i \rangle) \). The paths of \( \Delta \) must satisfy the read-once property: for every path from the root of \( \Delta \) to a \( \top \) leaf node, and for any variable \( X_i \in \mathcal{X} \), no more than one arc can be labelled by an elementary assignment over \( X_i \).

For every node \( N \) in an MDDG representation \( \Delta \), \( \text{Var}(N) \) is defined inductively as follows:

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

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

- if \( N \) is a leaf node, then \( \text{eval}(N, s) = N \);
- if \( N \) is a \( \land \)-node \( N = \land(N_1, \ldots, N_k) \), then \( \text{eval}(N, s) = \land(\text{eval}(N_1, s), \ldots, \text{eval}(N_k, s)) \);
- if \( N \) is a decision node \( N = \lor(N_1, \ldots, N_j) \) associated with variable \( X_i \), then \( \text{eval}(N, s) = \text{eval}(N_k, s) \), where \( \langle X_i, x_i \rangle \in s \).

\( s \) is a solution of \( \Delta \) if and only \( \text{eval}(N, s) \) evaluates to true.

The language MDDG we are interested in is the subset of MDDG consisting of decomposable representations, those where the children of any \( \land \)-node do not share any variable:

Definition 2 (MDDG) Given a finite set \( \mathcal{X} \) of finite-domain variables, the MDDG language over \( \mathcal{X} \) is the subset of MDDG representations \( \Delta \), where each \( \land \)-node \( N = \land(N_1, \ldots, N_k) \) is decomposable, i.e., \( \forall k, l \in 1, \ldots, i, \text{if } k \neq l \text{, then } \text{Var}(N_k) \cap \text{Var}(N_l) = \emptyset \).

The MDDG representation reported on Figure 2 is equivalent to the CN given at Example 1, i.e., they have the same solution set. Nodes with a single child are shunted, and their
labels (in the case of decision nodes) are gathered with those of their incoming arcs. The corresponding elementary assignments typically result from propagation (the V-nodes are not created in that case, we mention them in the definition of MDDG for the ease of exposure). The arcs leading to ⊥ are not depicted. The T leaf is duplicated for readability reasons.

Figure 2: An MDDG equivalent to the CN given at Example 1.

Decision-DNNF [Oztok and Darwiche, 2014; Fargier and Marquis, 2006] corresponds to the proper subset of MDDG where each variable has a Boolean domain. Despite the increase of generality obtained by accepting non-Boolean domains, the key tractable queries and transformations (weighted solution counting, solution enumeration, optimization w.r.t. a linear objective function and conditioning) offered by Decision-DNNF are also offered by MDDG (the polynomial-time (or polynomial-delay) algorithms used to achieve those queries and transformations in the Decision-DNNF case can be extended in a trivial way to the MDDG case).

MDDG is also close to the MDD language considered in [Amilhatre et al., 2014], and to the AOMDD language considered in [Mateescu et al., 2008], but it does not coincide with any of them. Thus, MDD and MDDG are not comparable w.r.t. set-inclusion; on the one hand, MDD consists of non-deterministic structures (more than one outgoing arc of a decision node labelled by a variable X, can be labelled by the same value from the domain D of X), while the decision nodes in an MDDG representation are always deterministic ones; on the other hand, MDDG representations include V-nodes, while the internal nodes of any MDDG representation are decision nodes. Similarly, AOMDD and MDDG are not comparable w.r.t. set-inclusion; on the one hand, AOMDD is suited to the compilation of graphical models (or weighted constraint networks), and as such, it enables the representation of functions the co-domain of which is not Boolean in essence (like utility or cost functions, probability distributions, etc.) while MDDG cannot do it; on the other hand, AOMDD representations are ordered structures (they respect a pseudo-tree induced by a preset elimination order over the variables – this is a crucial requirement for canonicity) while in MDDG there is no such a requirement (MDDG representations are not canonical ones).

4 A Top-Down MDDG Compiler

We developed a top-down compiler cn2mddg which takes as input a CN represented in the XCSP 2.1 format [Roussel and Lecoutre, 2009], and generates a MDDG representation equivalent to it, i.e., having the same solutions. All the basic features offered by the XCSP 2.1 format are taken into account; especially, the constraints can be represented in intension (as “predicates”) or in extension (as “relations”); however, only three global constraints are supported by our compiler in its current turn; namely allDifferent, weightedSum (i.e., linear constraints), and element.

The architecture of our cn2mddg compiler is somehow “standard”, i.e., close to the one of a top-down compiler suited to Boolean domains, like the c2d compiler (reasoning.cs.ucla.edu/c2d/), or the Dsharp compiler (www.haz.ca/research/dsharp), both targeting the Decision-DNNF language. Especially, our compiler is search-based: it follows the trace of a search engine [Huang and Darwiche, 2007]. It covers similar techniques as those used in c2d and in Dsharp, including conflict analysis for guiding the search, constraint propagation for simplification purpose, component caching in order to avoid the duplication of identical subparts of the compiled representation, and a dynamic variable ordering heuristic (as in Dsharp which takes advantage of the vsads variable selection heuristic). Both the caching technique and the variable ordering heuristic used in cn2mddg are specific to the nature of the input (a CN), which exhibits much more structure than “flat” CNF formulae. Furthermore, our algorithm exploits a specific method for handling universal constraints, enabling additional simplifications to be performed.

Universal constraints. Universal constraints are constraints which are necessarily satisfied whatever the values (in the current domains of the variables) given to the variables of their scopes. Thus, for the CN given at Example 1, contraint C2 when conditioned by any of the elementary assignments \{ ⟨X2, 0⟩ \} or \{ ⟨X2, 1⟩ \} becomes universal. At every step of the compilation (i.e., whatever the current decision state), universal constraints are detected. Every constraint \( C_j = (S_j, R_j) \in C \) for which there exists \( X_i \in S_j \) such that \( D_i \) has been reduced by propagation after the last elementary assignment, is checked for universality. One looks for an instantiation \( s \) of the variables of the current scope of \( C_j \) to values in their current domains such that \( s \) violates \( R_j \); \( C_j \) is valid iff one cannot find such an instantiation \( s \). For efficiency reasons, \( s \) is searched in a lazy way: when found, \( s \) is stored and the next time \( C_j \) is checked for universality, \( s \) is considered in priority. Once detected, a universal constraint \( C_j \) is simply deleted from the current network; obviously, this simplifies the forthcoming treatments (no need to take those constraints into account), favors decomposability (due to the

\[^1\] These constraints can be encoded as “predicates” as well, but then one cannot take advantage of their dedicated propagator.

\[^2\] Contrastingly, in c2d the variable ordering is static.
edge deletion it leads to on the primal graph of the CN), and impacts the variable ordering heuristic. Note that in Decision-DNNF compilers the handling of universal constraints simply amounts to ignoring every clause sharing a literal with the current partial interpretation.

**Caching.** Caching is a key technique of any compiler computing DAG-based representations. It aims at refraining from solving the same subproblem twice or more, and duplicating parts of the compiled representation. Indeed, due to the (conditional) interchangeability of values in many networks, it is often the case that two distinct decision states \( s_1 \) and \( s_2 \) considered successively during the search give rise to the same problem, i.e., \( N \mid s_1 \) and \( N \mid s_2 \) are equivalent. In such a case, instead of compiling both networks, it can prove much better to compile \( N \mid s_1 \) only, then to store in a cache an entry corresponding to \( N \mid s_1 \) associated with the root node \( N \) of its MDDG representation, and to detect that \( N \mid s_2 \) is equivalent to \( N \mid s_1 \) by looking at each step into the cache: in this case, instead of performing the computationally demanding compilation of \( N \mid s_2 \), it is enough to create an arc pointing to \( N \) to do the job. Thus, for the CN \( N \) given at Example 1, the component about \( \{X_3, X_4\} \) obtained by dynamic decomposition is the same one for the states \( \{X_2, 0\} \) and \( \{(X_2, 1)\} \), so there is no need to duplicate it.

However, testing the equivalence of two CNs under states is computationally hard and an exponential number of subproblems have to be considered in general. For these reasons, it is not possible to perform brute-force caching where all non-equivalent \( N \mid s \) networks encountered during the search would be considered (this would require unmanageable compilation times). Thus, \( N \mid s_1 \) and \( N \mid s_2 \) are detected as “equivalent” when they are identical.

A main issue to be addressed for an efficient caching in practice concerns the size of the entries; preferably, one must keep them as small as possible. In our cache, one first stores the current domains of the current variables, i.e., \( s \) restricted to its unassigned variables. Storing all the current constraints would be too space demanding. Fortunately, this is useless in general. Indeed, every constraint \( C_j = (S_j, R_j) \) such that \( S_j \cap \) single(\( s \)) = \( \emptyset \) does not need to be saved (provided that the initial constraint \( C_j \) is available). Furthermore, no constraint \( C_j = (S_j, R_j) \) which is binary in the input network needs to be saved, provided that the current network is arc consistent: if no variable from \( S_j = \{X_i, X_k\} \) has been instantiated, then the previous case is recovered; if both variables \( X_i, X_k \) from \( S_j \) have been instantiated, then either \( C_j \) is universal or \( C_j \) is inconsistent, and there is no need to store it whatever the case; finally if only one variable \( X_i \) from \( S_j \) has been instantiated (say, to value \( x_i \)), then the projection on the remaining (uninstantiated) variable \( X_k \) of the restriction of \( R_j \) for which \( X_i = x_i \) coincides with the restriction of \( s \) to \( \{X_k\} \) when the current network is arc consistent.\(^3\) Similarly, there is no need to store the allDifferent constraints which can be viewed as conjunctions of binary constraints. The remaining constraints are saved in our cache: for those represented in extension, the variables instantiated in \( s \) are replaced by their values in the predicates, and a simplification step is performed in order to possibly reduce the representations; those represented in extension are stored explicitly; finally, for each weightedSum constraint, the variables instantiated in \( s \) are replaced by their values, the constraint is simplified and only the resulting constant term needs to be stored; each element constraint \( R \), when binary, does not need to be stored; in the remaining case, one stores \( \{(X_i, x_i) \in s \mid X_i \in S_j\} \) into the cache.

**Variable ordering heuristic.** Our variable ordering heuristic \( bc \) is based on the concept of betweenness centrality [Brades, 2008] which has been used in many network applications.

Given a node \( X_i \) in a graph (in our case, the primal graph of the current CN in which the nodes can be identified as with the variables labelling them), \( bc(X_i) \) is equal to the number of shortest paths from all nodes to all others that pass through \( X_i \). Formally,

\[
bc(X_i) = \sum_{X_j \neq X_i \neq X_k} \frac{\sigma_{X_j}(X_j, X_k)}{\sigma(X_j, X_k)}
\]

where \( X_i, X_j, X_k \) are nodes of the given network, \( \sigma(X_j, X_k) \) is the number of shortest paths from \( X_j \) to \( X_k \), and \( \sigma_{X_j}(X_j, X_k) \) are the number of those paths passing through \( X_i \). Thus, for the CN \( N \) given at Example 1, \( X_2 \) is the unique variable maximizing the value of \( bc \). Clearly enough, assigning first the most central variables of the primal graph \( (X, E) \) of a CN is a way to promote the generation of disjoint connected components of similar sizes, allowing the decomposition of the network into independent networks (i.e., bearing on pairwise disjoint sets of variables) of close sizes, which can be compiled separately and gathered using a \( \land \) node in the resulting MDDG representation. Interestingly, computing the betweenness centralities of all nodes in \( (X, E) \) can be done in time \( O(n.\text{p}) \), where \( n = \#(X) \) and \( p = \#(E) \). In practice, the computation of \( bc(X_i) \) for each node \( X_i \) of the primal graph \( (X, E) \) of a CN is efficient enough so that we can achieve it dynamically, i.e., for each network encountered during the compilation.

**The cn2mddg compiler.** Algorithm 1 provides the pseudo-code for the compiler cn2mddg. The compilation of a given CN \( N \) is achieved by calling cn2mddg on it and on the decision state \( s = \{\langle X_i, x_i \rangle \mid X_i \in X, x_i \in D_i\} \). First of all (line 1), the arc consistency of \( N \) under \( s \) is established (the values of the variables occurring in \( s \) which are not supported in \( N \) are removed from the state). For efficiency reasons, arc consistency is ensured at start (i.e., at the first call) and then maintained dynamically each time a new elementary assignment is considered (at line 13). Then, \( N \) is conditioned by the resulting state (line 2).\(^4\) At line 3, a CSP solver

\(^3\)This is reminiscent to the treatment of binary clauses in Dsharp, which do not need to be cached provided that unit propagation has been performed [Muise et al., 2012].

\(^4\)In the implementation, the conditioning \( N \mid s \) at line 2 is not performed explicitly (it is presented as such for the sake of clarity); only the list of uninstantiated variables and the current domains are updated at each step (the constraints themselves are never modified for efficiency reasons).
is used to determine whether the resulting \( \mathcal{N} \) is consistent or not. We developed our own solver, based on chronological backtracking and using the (now standard) conflict-directed dom\textasciitilde\text{deg} heuristic for selecting variables [Boussemart et al., 2004]. Arc consistency is maintained at every choice step. Every constraint \( C_j \) of \( \mathcal{C} \) is associated with a weight, which is incremented each time a conflict is detected. If \( \mathcal{N} \) is inconsistent, then it is equivalent to the \text{MDDG} representation reduced to a leaf labelled by \( \bot \), as returned by the algorithm. Line 4 concerns the other base case, when all the variables of \( \mathcal{X} \) have been considered; in this situation, \( \mathcal{N} \) is equivalent to the \text{MDDG} representation \( \top \), as returned by the algorithm. In the remaining case (line 5), one first determines whether the current network \( \mathcal{N} \) has already been encountered or not during the search. One takes advantage of the cache function which associates networks with \text{MDDG} representations given by their root nodes. If \( \mathcal{N} \) has already been found, then the algorithm simply returns the root node of its \text{MDDG} compilation. Otherwise, \( \mathcal{N} \) is first simplified by removing from it the universal constraints it may contain (this is achieved by the removeUniversal function, at line 6). Then (line 7) the connected components of the resulting network are looked for. The function connectedComponents returns a partition \( \text{CoCo} \) of the current set of variables \( \mathcal{X} \) corresponding to the connected components of the primal graph of \( \mathcal{N} \) (a simple breadth-first search is performed to find them). Each element \( \text{Co} \) of \( \text{CoCo} \) is considered successively (line 9); \( \text{Co} \) is a set of variables which are independent from the other elements of \( \text{CoCo} \) and each network corresponding to \( \mathcal{N} \) restricted to \( \text{Co} \) can be compiled separately, leading to a set of nodes \( \mathcal{N}_\text{\text{Co}} \) which is initialized to the empty set at line 8. For each \( \text{Co} \), the current decision state \( \mathfrak{s} \) can be restricted to the variables occurring in \( \text{Co} \). A variable \( X_i \) from \( \text{Co} \) is picked up using the function selectVariable at line 10. Then the values \( x_i \) from the current domain of \( X_i \) are successively considered (line 12); each of them corresponds to an elementary assignment \( \{ \langle X_i, x_i \rangle \} \) and the current network conditioned by \( x_i \) is compiled recursively (line 13); the set \( \mathcal{N}_\mathfrak{s} \) of resulting nodes, initialized to the empty set at line 11 is updated at line 13. When all the values \( x_i \) have been considered, a new decision node labelled by \( X_i \) is created at line 14, and added to the set \( \mathcal{N}_\mathfrak{s} \). When all the connected components of \( \text{CoCo} \) have been considered, the elements of \( \mathcal{N}_\mathfrak{s} \) are gathered conjunctively to form a \( \lor \)-node \( N \) at line 15 thanks to the function aNode. This node is added to the cache associated with the entry \( \mathcal{N} \) (line 16) and finally returned (line 17) as the root node of the \text{MDDG} representation of \( \mathcal{N} \).

Algorithm 1 is guaranteed to terminate since at each recursive step at least one variable of the initial CN is instantiated. By construction, the resulting \text{MDDG} representation is equivalent to \( \mathcal{N} \mid \mathfrak{s} \).

5 Experiments

While compiling CNs has been an issue considered for years (see e.g., [Vempaty, 1992; Amilhastre et al., 2002]), we are not aware of any available compiler suited to CNs over non-Boolean domains and handling constraints which are intensionally represented. Especially, the \text{AOMDD} compiler available at http://graphmod.ics.uci.edu/group/aomdd assumes that each constraint of the input CN is represented extensionally by the list of its satisfying tuples. Fortunately, many SAT-encodings of CNs have been pointed out so far (see among others [de Kleer, 1989; Iwama and Miyazaki, 1994; Walsh, 2000; Gent, 2002]), rendering feasible a comparison with Decision-\text{DNNF} compilations of CNF translations of such CNs.

Setup. We have considered 173 CNs from 15 data sets, downloaded from github.com/MiniZinc/minizinc-benchmarks. www.cril.univ-artois.fr/~lecoutre/benchmarks. html, and www.itu.dk/research/cla/externe\textsc{s}/clib/. Those data sets correspond to several families of problems, including configuration problems, 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.

Our purpose was to compile each input CN into an \text{MDDG} representation using \text{cn2mddg}, and into Decision-\text{DNNF} representations, using first a translation of it into \text{CNF}, then the Decision-\text{DNNF} compiler Dsharp. Two CNF encodings have been considered in our experiments: the sparse encoding \( S \) of the domains together with a conflict encoding of the constraints (i.e., each constraint is encoded using the support encoding or the conflict encoding, in order to minimize the number of generated clauses), and the log encoding \( L \) of the domains together with a conflict encoding of the constraints.\(^3\)

\(^3\)Some of the available translators, like Sugar [Tamura et al., 2009] or Azucar [Tanjo et al., 2012] lead to encodings which do not preserve equivalence (they are oriented to solve the satisfaction problem), and cannot be used as such for our compilation purpose.
For each instance, we computed the compilation time (in seconds) and the size of the compiled representation (number of arcs in the DAG). For translation-based approaches, we also computed the translation time and the size of the resulting CNF formula (number of variables and number of clauses). Our experiments have been conducted on a Quad-core Intel XEON X5550 with 32GB of memory. A time limit of 3600s for the CNF translation phase (resp. the off-line compilation phase) and a total amount of 8GB of memory for storing the resulting CNF formula (resp. the compiled representation) have been considered for each instance.

### Some Results.

Within the time and memory limits we set, cn2mddg succeeded in compiling 131 instances over 173; the computation aborted with a time-out (TO) for 32 instances, and with a memory-out (MO) for 10 instances. This highly contrasts with Dsharp which succeeded in compiling only 83 instances when the $S$ encoding was used, and 61 instances when the $L$ encoding was used. More in details, the CNF translation using $S$ (resp. $L$) led to a memory-out for 27 (resp. 35) instances over 173; over the 146 (resp. 138) remaining instances, the DNNF compilation using Dsharp aborted with a time-out for 24 (resp. 21) instances, and with a memory-out for 39 (resp. 56) instances.

Table 1 presents a selection of the results. Each line corresponds to a constraint network $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ identified by the leftmost column. The next columns give respectively the “type” of CN ($\mathcal{X}$ extension or $\mathcal{X}$ Intension), the number $|\mathcal{X}|$ of variables, the number $|\mathcal{C}|$ of constraints, the maximal arity $\maxA$ of the constraints, the maximal size $\maxD$ of the domains, a upper bound $\tw$ of the treewidth of its primal graph,$^7$ the time needed to get the MDDG representation using cn2mddg, and the size of it. For the two CNF encodings under consideration, one can find the number $\#pv$ of propositional variables in it, the number $\#pcl$ of clauses in it, the time needed to get the DNNF representation using Dsharp, and the size of it. The reported results illustrate the benefits offered by cn2mddg over the translate-then-compile approaches, about both the number of benchmarks for which the compilation succeeded, and the sizes of the compiled representations.

Note that the time differences in favor of cn2mddg would be even larger if the time needed to translate the CN into CNF would have been taken into account.

#### Table 1: An excerpt of our empirical results.

| Name | type | $|\mathcal{X}|$ | $|\mathcal{C}|$ | $\maxA$ | $\maxD$ | $\tw$ | size |
|-------|------|--------------|--------------|--------|--------|------|------|
| rect-packing/rect-packingrpp09-true | 1 | 2196 | 2533 | 10 | 30 | 19 | 1675.35 | 571478 |
| ghoulomb/ghoulomb3-4-5 | I | 2033 | 2051 | 11 | 26 | 31 | 15.17 | 5162 |
| driver/normalized-driverlowg-08c-sat-ext | E | 408 | 9321 | 2 | 11 | 92 | 15.63 | 5162 |
| fapp/fapp19/normalized-fapp19-0350-6 | I | 325 | 352 | 46 | 316 | 52 | 1277.21 | 404437 |
| costa/Array/Costas/Array10 | I | 110 | 338 | 4 | 19 | 23 | 10.39 | 1340 |
| photo/photo2 | I | 189 | 133 | 21 | 11 | 21 | 499.93 | 9564220 |
| rfpap/normalized-scen4 | I | 680 | 967 | 2 | 44 | 30 | 3.47 | 5226 |
| rengn/normalized-rengn-mod-32-ext | E | 111 | 154 | 22 | 29 | 18 | 28.84 | 556051 |
| stl-life/stl-life7x7 | I | 198 | 110 | 27 | 11 | 21 | 499.93 | 9564220 |
| configit/Aralia/edfpa14q | I | 505 | 194 | 22 | 24 | 34 | 16.22 | 41919 |
| configit/Aralia/edfpa15r | I | 600 | 324 | 8 | 2 | 15 | 8.94 | 45853 |

<table>
<thead>
<tr>
<th>CIN</th>
<th>#pv</th>
<th>#pcl</th>
<th>time</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>1e+06</td>
<td>1e+07</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: cn2mddg vs. Dsharp: comparing the compilation times and the sizes of the compiled forms.
sentations. They also show the feasibility of MDDG compilation of CNs corresponding to real applications, and of significant complexity (often out of reach of tree clustering compilations [Dechter and Pearl, 1989], given the sizes of their domains and the treewidth of their primal graphs).

We also performed a differential evaluation for assessing the impact of each technique used within cn2mddg. Let \( \text{dom/wdeg+noU} \) be the version of cn2mddg for which the \( \text{dom/wdeg} \) heuristic is used (instead of \( \text{bc} \)), and the handling of universal constraints is disabled. \( \text{dom/wdeg+noU} \) solved only 101 instances (over 173) within the time and memory limits. Besides, the number of instances (over 101) for which the size of the MDDG representation obtained by cn2mddg (resp. \( \text{dom/wdeg+noU} \)) is lower than \( p = \frac{1}{3} \) times the size of the MDDG representation obtained by \( \text{dom/wdeg+noU} \) (resp. cn2mddg) is 35 (resp. 6). The corresponding number for the proportion \( p = \frac{1}{10} \) (instead of \( \frac{1}{2} \)) is 12 (resp. 0).

6 Conclusion

The contribution of the paper is a top-down algorithm cn2mddg for compiling finite-domain CNs into multivalued decomposable decision graphs. cn2mddg takes advantage of a specific caching technique, a new variable ordering heuristic based on betweenness centrality, and the handling of universal constraints. Intensive experiments showed that cn2mddg succeeds in compiling CNs which cannot be compiled into Decision-DNNF via a preliminary translation into CNF, and leads to compiled forms which are typically much smaller.

References


