Noname manuscript No.
(will be inserted by the editor)

Compiling CP Subproblems To MDDs and d-DNNFs

Diego de Uña · Graeme Gange · Peter Schachte · Peter J. Stuckey

the date of receipt and acceptance should be inserted later

Abstract Modeling discrete optimization problems is not straightforward. It is often the case that precompiling a subproblem that involves only a few tightly constrained variables as a table constraint can improve solving time. Nevertheless, enumerating all the solutions of a subproblem into a table can be costly in time and space. In this work we propose using Multivalued Decision Diagrams (MDDs) and formulas in Deterministic Decomposable Negation Normal Form (d-DNNFs) rather than tables to compute and store all solutions of a subproblem. This, in turn, can be used to enhance the solver thanks to stronger propagation via specific propagators for these structures. We show how to precompile part of a problem into both these structures, which can then be injected back in the model by substituting the constraints it encodes, or simply adding it as a redundant constraint. Furthermore, in the case of MDDs, they can also be used to create edge-valued MDDs for optimization problems with an appropriate form. From our experiments we conclude that all three techniques are valuable in their own right, and show when each one should be chosen over the others.

1 Introduction

Even when using modern high level constraint programming modeling approaches such as IBM OPL [58], MiniZinc [45] or Essence [26], building a good model of a discrete optimization problem is challenging. Although a modeler may be able to easily write a model that correctly captures the problem at hand, the efficiency of this model may be far from the best possible. Hence model improvement methods are valuable. Approaches such as detecting symmetries [43] or missing global constraints [41] are highly advantageous, but for many problems these refinements may not be applicable. Precompilation [9, 23, 40], where the model is improved during compilation, is an effective approach to model improvement.

Diego de Uña[1] · Graeme Gange[1] · Peter Schachte[1] · Peter J. Stuckey[1,2]
1. University of Melbourne
2. Data61 CSIRO
Email: {d.deunagomez@student.,gkgange@,schachte@,pstuckey@}unimelb.edu.au

One simple precompiling technique is to replace a subproblem by a `table` constraint representing the solutions of this subproblem [20, 21]. We say the subproblem is *tablified*. This can be highly effective both in improving naive models and in improving models where some subproblem over a few variables is highly constrained.

Although tablifying a model has a lot of value, and can be the best way of precompiling part of a problem, it comes with limitations. An obvious one is that the table can be huge, simply because there are too many solutions to the part of the problem being precompiled. Furthermore, tables can contain a lot of repeated "suffixes". For example, many rows of a 100-column table could contain identical values for the last 90 columns, and only the 10 first columns differ between those rows. In that case, finding a data structure that can compress all those rows into an object with only one common "suffix" for all the combinations of the first 10 columns could save up to 90% of memory. The problem with tables in this example is twofold. First, there is a clear redundant use of memory, and second, a solver would actually need to generate all those almost-identical rows, which is slower than a data structure where rows could point to "commonly shared sub-rows".

This prompts the question of what better data structure can be used. In this paper we investigate alternatives to using tables as solution stores for precompiling parts of Constraint Programming models. We choose to compare with Multi-valued Decision Diagrams (MDD), which are well known in the CP community, and Deterministic Decomposable Negation Normal Forms (d-DNNFs), which are widespread in model counting for SAT to store solutions and then perform model counting using these structures. We say that we *mddify* or *d-dnnffy* part of a model. The goal of this compilation is to produce models with constraints that lead to **stronger propagation** without the user needing to construct these complex structures manually. Note that our approach compiles part of a model directly into an MDD or d-DNNF, and no intermediate data structures are needed. It is a general CP model precompilation technique for arbitrary subproblems.

The contributions of this work are:

– The use of problem equivalence detection techniques to a new purpose: model compilations. This is described in Section 3.4.
– A new caching key for the `tree` constraint. This is described in Section 3.4.
– A new way of handling fixed variables when compiling (cost-)MDDs from CP models, compared to previous work. This is described in Section 4.1.
– An new algorithm to construct d-DNNFs from CP models that involve globals. This is described in Section 4.3.
– Subsequently, a new approach to split globals, exemplified by an algorithm we introduce to split `tree` to reduce the size of the d-DNNFs. This is described in Section 4.4.
– An extensive set of experiments comparing tablification with compilation into MDDs and d-DNNFs, as well as the use of Cost-MDDs, where we show the value of this technique. (Section 5).

The paper is organized as follows. Section 2 presents relevant work on compilation of CP (sub)problems in the form of Decision Diagrams (DDs) as well as compilation of mostly SAT problems into Negation Normal Forms (NNFs). Section 3 familiarizes the reader with MDDs, d-DNNFs and subproblem-equivalence detection by hashing, which is a key component of the approach we use to compile subproblems into these structures. Section 4 presents the algorithms to compile (parts of) problems into MDDs or d-DNNFs. Finally Section 5 presents a set of experiments we performed to compare the different precompilation approaches.

2 Related Work

2.1 Precompilation Techniques

As announced in the introduction, Constraint Programming models sometimes lack good modeling choices, and tools can be developed to automatically enhance these models before they are solved, by detecting some features and converting them into a better model for the solver. These are called *precompilation techniques*.

A good example of these techniques is the idea of detecting symmetries to generate symmetry breaking constraints, by Puget [54] and later implemented and improved by Mears et al [43]. This method allows finding symmetries on the variables or values of a CSP, and is applicable either to individual CSP instances or to models (regardless of the data that will be used for the model). Of course, the latter is preferred, as this precompilation effort would only need to be done once. Without entering in details, the method works by creating a novel graph representation, called the *full assignment graph*, of the problem. This representation is then pruned by establishing n-ary arc-consistency across the constraints and by reducing the arity of constraints (using equivalent constraints of smaller arity). It is proven in the paper that every graph automorphism on the full assignment graph corresponds to symmetry in the original problem.

From the same research group comes the idea of *globalizing models* developed by Leo et al [41]. In their paper, the authors develop a MiniZinc "Globalizer", a tool that allows the MiniZinc compiler to detect missed opportunities to use global constraints in models. For example, it is able to detect that a set of dis-equalities can be converted into an `alldifferent` constraint. This technique is nonetheless non-automatic: it does not substitute parts of the model with global constraints, it suggests global constraints that may or may not capture part of the problem to the user through a GUI. For this method to work, the user must provide some data files and the system will solve them to find properties on the solutions that match some global constraints.

Later, the same researchers integrated *whole-program optimization* to their MiniZinc compiler [40]. This is not so much a precompilation technique but rather a precompilation framework. Nowadays, CP is turning towards high-level languages such as IBM OPL [58], MiniZinc [45] or Essence [26] because of their natural modeling approach. Nonetheless, compilers for modeling languages usually perform optimizations of the model while converting the model into a low-level representation ready for the back-end solvers (such as FlatZinc). For that reason, only partial information of the model is known at each step of the low-level model generation. Leo and Tack [40] develop a framework to improve this by allowing whole-program optimizaton in the MiniZinc compiler by compiling in multiple passes. The information gained at each of the passes is used in later passes to improve the model. Ideally, any precompilation technique can be used at each of these passes.

Another precompilation technique is *variable elimination* [23], where variables that are not used or can be subsumed by other variables are eliminated from the model in a precompilation effort. This showed to be beneficial in the experiments by the authors, although this was only tested on SAT problems, to our knowledge. Common expression elimination [16] is also consistently used in precompiling CSPs, although it comes from the general compilers research area.

The most closely related work to this paper is the work on tablifying parts of models by Dekker et al. [20, 21]. In this work, the authors introduce annotations to the MiniZinc language to allow the user to convert a MiniZinc predicate into a `table` constraint. In MiniZinc, predicates are a tool to capture complex constraints in an abstract manner. They are the equivalent to a procedure in imperative programming. They allow the modeler to capture a constraint that is, conceptually, only one constraint but due to its complexity needs to be written as a composition of multiple constraints.

We will mainly compare with this work by Dekker et al. [20, 21], because our framework has the same goal, but uses alternative data structures to store the contents of those tables. In that work, a predicate is solved (all its solutions are determined) and converted into a table constraint that is then injected into the model at each call of the predicate. Their results clearly show how compiling predicates can yield better solving times for the model. While for the examples they examine, building the tables does not require much time, we will show examples where tablifying blows up, and hence the use of more compact representations can be crucial.

We will also see that the idea of converting a table into an MDD [9] is also inapplicable in many cases because building the table is impractical, and compiling directly into an MDD (without intermediate structure) is much more efficient.

2.2 MDD Compilation

The idea of compiling subproblems or entire CP problems into decision diagrams (whether they are BDDs or MDDs) is not new. Indeed, the book by Bergman et al [6] already discusses this idea. In this subsection we discuss several works that are related to compilation of subproblems as Decision Diagrams and we show the differences between their systems and ours.

Perez et al. [51, 49] presented efficient algorithms for MDD-operations with the goal of reducing the time to build MDDs. They show that using MDDs built with these operations can be competitive with dedicated algorithms. This motivates the idea of modeling complex constraints, as we will do in this paper, with MDDs (or d-DNNFs) rather than having to create a specific propagator to achieve good propagation for some constraint. Additionally, the aforementioned authors summarize some approaches for building MDDs. These include building MDDs from a table constraint (originally presented by Cheng and Yap [11]), from an automaton (i.e. a `regular` constraint), a pattern or a trie. None of these approaches are what we are after because we want something as modular as table creation, but more robust in memory consumption (as we will see in the experiments, it not particularly hard to make the table approach explode in size). These papers by Perez et al., although related, do not solve the problem we are tackling: we want to build the MDD from a set of basic constraints that any modeler could write in a modeling language like MiniZinc, not from a specific form of constraint like the `regular`.

The work presented by Koriche et al [38] is much more closely related to our own work here. Indeed, their algorithm for compiling constraint networks into their custom Multivalued Decomposable Decision Graphs (MDDG) language is quite similar to the algorithms we present in this paper. There are some important differences though. Their `cn2mddg` compiler does not generalize to all global constraints. As they mention the paper, they only support three globals (`alldifferent`, linear constraints and `element`). Arguably this limitation can be solved by extending their compiler to other globals but this is a task that needs to be done every time a new global is defined and, as we have seen in recent times, the number of global constraints defined in the literature is constantly growing. In our case, we only need to update the caching capabilities for new globals. Another difference with that work, as we will see later in Section 4.1, is the way variables fixed by propagation are treated compared to their algorithm. We believe our approach for compiling parts of problems into MDDs is more flexible, and we provide a larger number of experiments and comparisons with more competitive approaches (like tables and d-DNNFs, rather than just the original models).

Cheng and Yap [10] introduced a representation for general n-ary constraints called Constrained Decision Diagrams (CDDs). As they mention in their paper, their approach aims for compact representations (even more than canonical MDDs). The use they make of their data structure is to store solutions that can then be presented to a user to filter manually. A good example of this would be to solve a problem for a client, show the solutions and then the client can filter them out based on some criteria that may not have been present in the original model. Their compilation into CDDs is not intended as a precompilation that can be used to enhance search like our work or that of Dekker et al [21]. Although it could be used in this way, it would require a fair amount of work since we would need to create a propagator for this CDD representation in our solver. Sadly, there are no efficient algorithms known for propagating these data-structures.

On the other hand, MDD, cost-MDD and d-DNNF propagators (with explanations) are well understood [27, 28, 29] so we decided to limit our scope to these representations.

Hoda et al [32] create a *pure MDD-based* CP solver. Given a problem, they construct an MDD that takes *all* the variables of the problem. Initially that MDD is of width 1 (i.e., a "stick") that allows more solutions than the problem permits. Then, each constraint is propagated through the MDD to remove arcs that are inconsistent with the constraints. This provides a tractable data structure that models the original problem. The focus of their paper was the construction of the MDD by propagating constraints in the MDD. This technique is presented under the name *compilation by separation* by Bergman et al [6]. Note that constraints are not propagated to a fixed point in that work; only two passes for each constraints are performed. Because they were mapping an entire problem into an MDD, the size of the latter could explode easily. For that reason, they consider approximate MDDs where nodes are merged to keep a predefined width. This of course allows for more solutions in the MDD than there are in the original problem. The approximate MDD is then used as a "domain store".

Earlier work by Andersen et al [1] constructs an MDD *constraint store*, much like Hoda et al [32]. The main difference between these two approaches is that Andersen et al [1] do not construct a pure MDD-based solver, but a hybrid one where the MDD starts by being of width one and as the search advances the MDD is refined by vertex splitting. Similarly, the MDD is always approximated to avoid size explosion.

Our approach differs from these two papers in that we do not build an MDD as a constraint store to capture an entire problem. Rather we focus on a subset of variables to be compiled in the form of MDD or d-DNNF with the hope of enhancing the solving process. The goals are completely different. Our construction method for the MDD is also substantially different, as we do not use construction by separation. Furthermore, our resulting MDD is exact (although it only represents a part of the problem).

Work by Hadzic et al [31] and Bergman et al [3] also investigated compiling problems into MDDs. The initial algorithms used to construct the MDDs in these papers are very similar to the algorithm we use. In both those papers and the present one, the MDD is built top-down using caching to detect equivalence of nodes. The caching used by Hadzic et al [31] is specific to linear constraints and the `alldiff` global (and detects fewer equivalences than the caching we use). The authors propose alternative ways of subsuming this problem with incremental algorithms and vertex-splitting approaches. The resulting MDD is approximated to avoid size explosion as well. Bergman et al [3] apply the same algorithm to the specific case of Set-Covering, where the caching is properly constructed to detect more node equivalences for that specific problem. Even with efficient caching, the MDD needs to be approximated due to its size.

The main difference between these two publications and the current paper is the caching technique. We use the technique for subproblem dominance presented by Chu et al [13]. This generalization solves several problems encountered in previous research. Specifically, it can be applied to any problem, rather than only problems with linear and `alldiff` or set-covering constraints. The combination of using this caching technique and a reordering of the variables in our algorithms (as we will see) allows detecting much more equivalence than their work. Another major difference is that our MDDs are exact, and not approximate.

Other less closely related work [5, 15, 36] builds MDDs to use in Lagrangian Decompositions, or specifically for scheduling problems and bin-packing problems. The MDDs are not built in a generic way that could serve our purpose. We will see examples of where compilation of cost-MDDs can be helpful for problems where Lagrangian Decomposition is appropriate. Binary Decision Diagrams have also been used [2, 4, 7], but either no generic automatic compilation was used, or the compiled BDD was approximate and used only as a bounding technique for branch-and-bound.

2.3 NNF Compilation

Deterministic Decomposable Negation Normal Forms (d-DNNFs), as well as other forms of NNFs, are well known and widely used in model counting (#SAT problems) and knowledge compilation work, but unlike MDDs they have had much less attention from the CP community. There is, nonetheless, a propagator for d-DNNF constraints with explanations [27] available in the Chuffed solver, which we will be using.

Darwiche [17] worked on compiling CNF to d-DNNF. The algorithm presented in his paper is somewhat similar to ours for d-DNNFs, but it is much simpler as its scope is limited to CNF formulae. Later on, Huang and Darwiche [33] worked on compiling propositional theories (again CNF formulae) into different tractable target languages, including d-DNNFs. This work differs from their previous work in that they used the exhaustive version of the DPLL algorithm from SAT solvers in order to construct a Free Binary Decision Diagram (FBDD) that is then converted into a d-DNNF. This research yielded the `c2d` compiler from CNFs into d-DNNFs. The work by Muise et al [44] starts from that research and further improved it, yielding the `Dsharp` compiler.

Sang et al [56] published an improvement to model counting using the ZCHAFF solver where they used caching and clause learning, just like we do (since CHUFFED also implements clause learning, and we used caching). With an extra effort, their work could be converted in a precompilation technique like ours, but this was not their goal: their objective was to perform model counting, so the solutions were never stored in a data structure. Furthermore, their model counting and caching was targeted to SAT problems, and not to Constraint Programming, therefore it could not have been used directly to achieve our task, due to the presence of global constraints in CP models.

Another interesting application of knowledge compilation by Jha and Suciu [34] looked into compiling database queries into d-DNNFs. Work in knowledge compilation has also been used for probabilistic reasoning and inference via model counting using d-DNNFs or related languages [39, 8].

A common component in all these works in knowledge compilation is the use of caching very similar to the one we use ourselves [13]. Nonetheless, because all this work is directed to propositional formulae and SAT, they did not need to deal with global constraints as we have to. Furthermore, we have not found any work that uses this compilation into d-DNNFs for the purpose we are using them, that is, precompilation to enhance the solving process of a CP solver.

3 Preliminaries

Let $\Rightarrow$ denote logical entailment and $vars(O)$ the set of variables of object O. A *constraint problem* P is a tuple (C, D), where D, the *domain*, is a set of unary constraints, and C is a set of constraints such that $vars(C) \subseteq vars(D)$. Each set D and C is logically interpreted as the conjunction of its elements. We define D_V, the restriction of D to a set of variables V, as $\{c \in D | vars(c) \subseteq V\}$. We note $D(x) = \{d \mid x = d \Rightarrow D_{\{x\}}\}$ the set of values that x can take in domain D.

A *vmap* of $P = (C, D)$ is of the form $x \mapsto d$, where $d \in D(x)$. A *valuation* θ *of* P *over set of variables* $V \subseteq vars(D)$ is a set of vmaps of P with exactly one vmap per variable in V. It is a mapping of variables to values.

The *projection* of valuation θ over a set of variables $U \subseteq vars(\theta)$ is the valuation $\theta_U = \{x \mapsto \theta(x) \mid x \in U\}$. We denote by $fixed(D)$ the set of fixed variables in D and by $fx(D)$ the associated valuation. We define $fixed(P) = fixed(D)$ and $fx(P) = fx(D)$ when $P = (C, D)$.

A valuation θ is a *solution* of a constraint c if $\theta(c)$ holds, that is replacing the variables in c with the values given by θ gives a true statement. A *solution* of P is a valuation over $vars(P)$ that satisfies every constraint in C. We let $\mathsf{solns}(P)$ be the set of all its solutions.

Finally, we use $\exists_V.F$ to denote $\exists v_1.\exists v_2 \cdots \exists v_n.F$ where F is a formula and V is the set of variables $\{v_1, v_2, \ldots, v_n\}$. We define $\forall_V.F$ similarly. Formally, a predicate $p(\boldsymbol{x})$ is a "macro" for a constraint formula which has a meaning defined by $\forall \boldsymbol{x}.(p(\boldsymbol{x}) \Leftrightarrow \exists_{vars(F)\setminus \boldsymbol{x}}.F)$ for some formula F. We assume (for the paper) that fixing all variables $\boldsymbol{x}$ will cause propagation to fix the remaining variables. This restriction is not required in the implementation but simplifies the algorithms.

3.1 Multivalued Decision Diagrams

Multivalued Decision Diagrams [6, 57] can be thought of as "compressed" decision trees. An MDD m is a connected directed acyclic graph where nodes are layered by their depth from the root. An MDD node n is either one of the terminals True or False, or is of the form $(x, [(v_1, n_1), ..., (v_k, n_k)])$ where x is a variable, $v_1, \ldots, v_k$ are integer values and $n_1, \ldots, n_k$ are MDD nodes. This represents k labeled edges $n \xrightarrow{x=v_1} n_1, \ldots, n \xrightarrow{x=v_k} n_k$. We let ϕ define the semantics of an MDD node as $\phi((x, [(v_1, n_1), ..., (v_k, n_k)])) \equiv (x = v_1 \wedge \phi(n_1)) \vee \cdots \vee (x = v_k \wedge \phi(n_k))$ and $\phi(\text{True}) \equiv true$ and $\phi(\text{False}) \equiv false$.

We will restrict attention to layered MDDs where each node n is in layer i (the shortest path from the root to n) and each edge in a given layer is labeled by the same variable. Each path from the root r of an MDD to True represents a solution of $\phi(r)$. To simplify the presentation, we elide the False node and all edges to it.

Example 1 Figure 1 shows an example of an MDD that captures the constraints $x_1 + 2x_2 \leq 5 \wedge x_1 + x_2 + 2x_3 \leq 8$ for $\{x_1, x_2, x_3\} \subseteq 1..3$. □

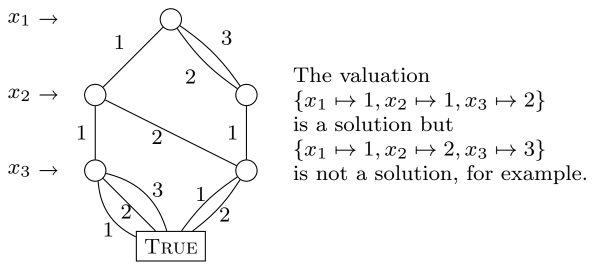

$x_1 \rightarrow$
$x_2 \rightarrow$
$x_3 \rightarrow$
1
2
3
1
2
1
1
2
3
1
2
True
The valuation
$\{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 2\}$
is a solution but
$\{x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto 3\}$
is not a solution, for example.

Fig. 1: Example of an MDD.

Global propagation algorithms for constraints represented by MDDs are well understood (see e.g. [9, 48, 50, 52]) including versions with explanations [28].

3.2 Edge-valued MDDs

Cost-MDDs (or edge-valued MDDs, or weighted-MDDs) are a variation of MDDs where each edge has a weight associated with it. The *cost* of the solution to an MDD is the sum of the weights on the edges representing the path from the root to True. Cost-MDDs allow better branch-and-bound search when the cost is part of the objective since paths that cannot lead to better solutions can be pruned. An example of cost-MDD is provided in Figure 2.

Similarly to regular MDDs, propagation algorithms for cost-MDDs already exist (see e.g [22, 29]).

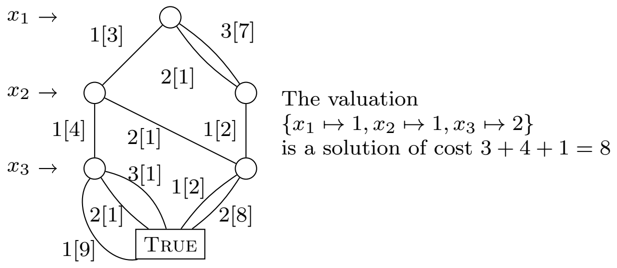


Fig. 2: Example of a cost-MDD (costs are indicated in brackets).

3.3 Deterministic Decomposable Negation Normal Form Formulae (d-DNNFs)

d-DNNFs are a less common data structure in the CP community, but widely used in solution counting literature [44].

A formula is considered to be in *Negational Normal Form* (NNF) if the only logic operators used are $\wedge$, $\vee$ and $\neg$ and the latter (logical negation) is only applied to elementary variables (rather than sub-formulas).

An NNF is *decomposable* (DNNF) if, for all $\wedge$ operators, no variable appears in both operands. That is, a formula $F_{left} \wedge F_{right}$ is DNNF if, and only if, both formulas F_{left} and F_{right} are DNNF and $vars(F_{left}) \cap vars(F_{right}) = \emptyset$.

A DNNF is *deterministic* (d-DNNF) if, for all $\vee$ operators, the conjunction of its operands is unsatisfiable. That is, a formula $F_{left} \vee F_{right}$ is d-DNNF if, and only if, both formulas F_{left} and F_{right} are d-DNNF and $F_{left} \wedge F_{right}$ is unsatisfiable.

For clarity, d-DNNFs are often represented as diagrams. Figure 3 gives an example of one.

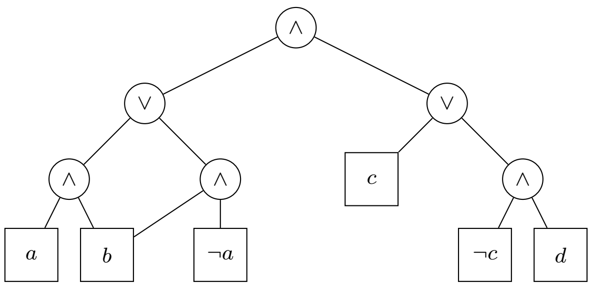


Fig. 3: Example of an d-DNNF (formula $(a \wedge b) \vee (\neg a \wedge b) \wedge (c \vee (\neg c \wedge d)))$.

Propagators for this type of formulae exist in CP. In our experiments we use the one presented by Gange and Stuckey [27].

3.4 Detecting Subproblem Equivalence by Hashing

A key component of the algorithm to build structures that can store all the solutions to a CSP efficiently is equivalence detection: we will need to detect when we reach a state where the "remaining subproblem" is equivalent to a previously seen "remaining subproblem". If we detect this we can re-use parts of the data structures we constructed previously.

As far as we are aware, the state of the art in subproblem dominance detection in CP is the method of Chu et al [13]. In their paper, the authors describe how to map a subproblem into a concise key that can be used to compare two subproblems. The goal of their work was detecting dominance between subproblems, in order to avoid exploring a subproblem that was already proved to contain no solutions. We use their dominance detection to efficiently construct our solution stores.

The key definition and proposition we use are as follows. A subproblem $P = (C, D)$ *dominates* a subproblem $P' = (C, D')$ if $\mathit{fixed}(D) = \mathit{fixed}(D') = F$ and $\exists_F C \wedge D' \Rightarrow \exists_F C \wedge D$. We say that P and P' are *equivalent* if they dominate each other.

Proposition 1 (from [13]) *If P is equivalent to P' then $\theta \in \mathsf{solns}(P)$ iff $\theta_{vars(P) \setminus fixed(P)} \cup fx(P')_{fixed(P)} \in \mathsf{solns}(P')$.* □

Hence if we build an MDD/d-DNNF encoding P, we can reuse the same structure to encode P'. In order to efficiently check that $\exists_F C \wedge D \Rightarrow \exists_F C \wedge D'$, we use Theorem 1.

Theorem 1 (from [13]) *Let $P = (C, D)$ and $P' = (C, D')$ be subproblems where $\mathit{fixed}(D) = \mathit{fixed}(D') = F$ and $U = vars(C) \setminus F$, Then if $\forall c \in C$ we have that $\exists_F c \wedge D' \Rightarrow \exists_F c \wedge D$ and $D'_U \Rightarrow D_U$ then P dominates P'.* □

What this means is that to detect that two remaining subproblems are equivalent we need to make sure that:

- The fixed variables are the same on both. We do not care about their value, only whether they are fixed.
- The domains of the unfixed variables in both problems need to be the same.
- The projection of each constraint onto the unfixed variables has to be the same in both problems.

Thus, any subproblem P can be projected into a key $(F, [\mathit{key}(c, D) \mid c \in C], D_U)$. The construction of those keys is described in detail by Chu et al [13], and thus we will only discuss here the ones that we will be using. In addition, we will present our key for the `tree` constraint, as it did not exist in previous works.

3.4.1 Key Construction

The examples that we will see in the experiments in Section 5 will use three types of constraints: binary, linear and the `tree` constraint. For the two first ones, Chu et al [13] provided the keys as follows.

Binary Constraints Assuming that the propagator used in the solver has the property that once one of the two variables is fixed, the domain of the other variable is modified in a manner that the constraint will always be satisfied, i.e. the propagator enforces arc consistency, then no key is needed for these constraints. This is the case because *i)* if neither variable is fixed, this information is already available in the key *ii)* if neither variable is fixed, the binary constraint at hand is satisfied (otherwise it would have caused a failure) and *iii)* if at least one variable is fixed, the constraint is satisfied (by the propagation rules). Thus, in all cases, the key would be the same, and therefore, no key is needed.

Linear Constraints The key for linear constraints $c \equiv \sum_{i=1}^{n} a_i x_i \leq a_0$ is defined by $a_0 - \sum_{i \in F_c} a_i x_i$ where F_c is the fixed variables involved in c. The intuition behind this is that the gap between the right-hand-side and the current partial sum is what matters to detect equivalence between two subproblems. Thus, if the gap between the right hand side is 5, for example, it does not matter how the left hand side reached the value $a_0 - 5$, only the fact that it reached that value. Example 2 illustrates this.

Example 2 Consider the problem of Example 1. The key for the subproblem $P = (C, D)$ where $D = \{x_1 = 1, x_2 = 2\}$ would be $k_P = (\{x_1, x_2\}, [true, 8 - 1 - 2 = 5], \{x_3 \in 1..3\})$. Now consider the subproblem $P' = (C, D')$ where $D' = \{x_1 = 2, x_2 = 1\}$. The key for P' is $k_{P'} = (\{x_1, x_2\}, [true, 8 - 2 - 1 = 5], \{x_3 \in 1..3\})$. Thus P and P' are equivalent, as it can be seen in Figure 1. □

Tree Constraint No key representation of this constraint was ever introduced, so we will be presenting it here. The motivation for choosing this global constraint to be represented as a key is that later on (in Section 4.4.2) we will see how we can split globals and we will use the `tree` constraint to demonstrate the value of this technique in the experimental section (Section 5).

Recall the `tree` constraint is given a graph $\mathcal{G}$ and intends to build a subgraph G of $\mathcal{G}$ such that G is a tree. To represent G, two sets of Boolean variables are used: one for the nodes (whether a node of $\mathcal{G}$ is selected or not to be in G) and another for the edges (whether an edge of $\mathcal{G}$ is selected or not to be in G). Namely the tree constraint is:

$$tree(vs, es, \mathcal{G})$$

where vs is the set of Boolean variables for the nodes, es the set of Boolean variables for the edges and $\mathcal{G}$ is a graph (that can be represented in any arbitrary manner).

Following the notation by De Uña et al [19], during search, some edges/nodes will be *in*-nodes/edges if they are chosen to be in the tree, *out*-nodes/edges if they are chosen not to be in the tree or *unfixed* otherwise.

Let T_1 and T_2 be two partial assignments for the Boolean variables of the `tree` constraint. The subproblems in the search tree below these partial assignments are called, respectively, P_1 and P_2. We want to create a key that allows us to identify whether P_1 and P_2 are equivalent.

Intuitively, we need all the in-nodes to be part of the key. This is because if T_1 contains node n, and n is still unfixed in T_2, there may be solution of P_2 that does not involve n, but those solutions will not appear in P_1. So this information needs to be in the key.

Furthermore, let T_1 have a set of in-edges connecting some in-nodes (they have to be in-nodes by the tree propagation [19]). This creates a connected component of in-nodes C_1. The edges involved in this connected component are irrelevant. The intuition behind this claim is that if we compress C_1 into a "meta-node", the problem remains the exact same: the rest of the tree needs to be connected regardless of what is inside C_1. For this reason in-edges do not need to appear in the key. This implies that in-nodes should appear in the key, grouped by connected components, to be able to identify the "meta-nodes" both in T_1 and T_2. Both should have the same "meta-nodes" in order for P_1 and P_2 to be equivalent. We need to carefully represent a "meta-node" in a canonical way: we choose to represent *each* 'meta-node" with a list $\langle|A|\rangle + + A$ where A is the list of nodes in the connected component sorted by a unique integer identifier. $++$ is the concatenation operator for lists. The representation of each "meta-node" is then concatenated. An example of this can be seen in 4a, where the three nodes labeled 1, 2 and 3 become a "meta-node" regardless of how they were connected. Node 4 remains on its own. Clearly, the two problems on the left of subfigure 4a are equivalent. Their keys are $\langle \mathbf{3}, 1, 2, 3, \mathbf{1}, 4\rangle$. The bold numbers correspond to the sizes of each connected component, for clarity.

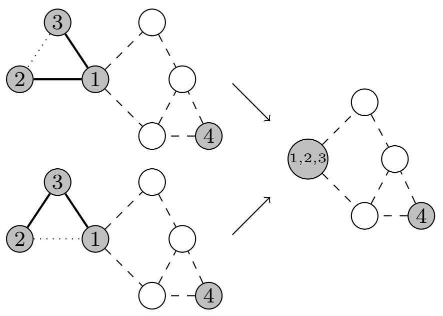
The tree propagator removes edges that could form a cycle if they were in-edges. These edges do not need to be in the key. The reason is that, for an edge e to potentially create a cycle, the extremities (a, b) of such an edge need to be connected to each other via in-edges. That is, the a and b are in the same connected component, which is already represented in the key. As we said before, we do not care about what edges are used in a connected component, since we can regard it as a "meta-node". Following the same reasoning we do not care about which edges are not in the connected component.

Nonetheless, if T_1 has an out-edge e that would not have formed a cycle (this edge could not have been removed by the tree propagator given the propagation rules [19]), and T_2 still has e available, there may be solutions to P_2 that do not exist in P_1. Therefore we need to keep this information in the keys. To represent these edges in the key, we use unique identifiers for them (positive integers) that we negate to distinguish them for the part of the key corresponding to nodes. For example, in Figure 4b, the key for the top figure is $\langle 3, 1, 2, 3, 1, 4, \mathbf{-5}\rangle$, but for the bottom one is $\langle 3, 1, 2, 3, 1, 4, \mathbf{-6}\rangle$. So, even if the rest of the key is the same, these two problems are deemed not to be equivalent.

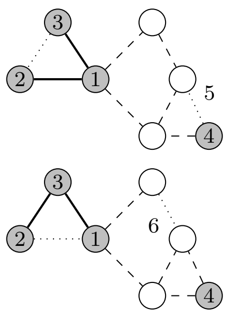
To sum up, the final key follows is a list matching $\langle(\langle|A|\rangle + + A)^*\rangle + + \langle(-I)^*\rangle$ where I are integer identifiers for edges, and A are sorted lists of integer identifiers for nodes that are connected by in-edges.

4 Compiling Subproblems

Our goal is to compile part of a problem into a language (MDD or d-DNNF) which represents all the solutions to this subproblem, much like the work by Dekker et al [21] which converts subproblems into `table` constraints. Our thesis is that using a tablification approach can be impractical in some applications because the table can explode in size, and can be slower to build than other alternatives. Our choice of structures can build a smaller representation of the solutions of the subproblem, and using caching we can do this with less work.



(a) Connected components need to be represented in the key through the nodes they contain, not the edges involved in them.



(b) Removed edges (edge 5 or 6) that do not form a cycle need to appear in the key to distinguish these two cases.

Fig. 4: Example of key for the *tree* constraint. ('——','- - -','......', '●', '○' are in-edges, unfixed edges, out-edges, in-nodes and unfixed nodes respectively.)

4.1 Compiling to an MDD

Given a subproblem $P = (C, D)$, Algorithm 1 describes our compilation of an MDD for that problem. The function MDDIFY takes three arguments. The first argument C is the set of constraints being considered for building the MDD. The second, D, is the current domain of all variables. The third argument, $X \subseteq vars(C)$, is a list of the variables in the order in which they should appear in the MDD. The algorithm returns an MDD which represents the formula $\exists_{vars(C) \setminus X} C \wedge D$. For clarity, assume that the variable *problem_store* is a global hash-table (initially empty).

Algorithm 1 Constructing MDD via propagation.

```
 1: procedure MDDIFY($C, D, X$)
 2:     $D \leftarrow propagate(C, D)$                  ▷ Propagate constraints $C$ to fixed point
 3:     if $D$ is a false domain then return FALSE
 4:     $fbp \leftarrow fixed(D) \cap X$                ▷ Variables fixed by propagation at this level
 5:     $p \leftarrow key(C, D)$                        ▷ Hash the remaining subproblem
 6:     if $p \in problem\_store$ then
 7:         $m \leftarrow problem\_store[p]$            ▷ MDD representing remaining subproblem
 8:         $m \leftarrow m$ & MDD($D_{fbp}$)           ▷ Conjoin $m$ with a "stick" MDD of fixed variables
 9:         return $m$
10:     $Y \leftarrow X \setminus fbp$                  ▷ Remove fixed variables
11:     if $Y = []$ then $m \leftarrow$ TRUE
12:     else
13:         $x \leftarrow head(Y), R \leftarrow tail(Y), m \leftarrow$ FALSE
14:         for all $d \in D(x)$ do
15:             $m \leftarrow m$ | ( MDD($x = d$) & MDDIFY($C,\ D \cup \{x = d\}, R$) )
16:     $problem\_store[p] \leftarrow m$                ▷ Associate the remaining subproblem with the MDD
17:     $m \leftarrow m$ & MDD($D_{fbp}$)               ▷ Conjoin fixed variables
18:     return $m$
```

The operations & and | over MDDs are the classical *conjoin* and *disjoin* operations defined for this data structure. The constructor MDD for an MDD simply takes a mathematical expression and builds an MDD for it.

The algorithm works by keeping a map from keys for subproblems into already computed MDDs for the remaining subproblem, or filling that same map if the remaining subproblem has never been encountered before.

First, it propagates the last decision made (or propagates root-level information in the first call). If this propagation results in unsatisfiability, the corresponding MDD is FALSE. Otherwise, the problem is mapped into a key. This is done by using the domains of variables and the constraints; details of this are presented by Chu et al [13]. If the key matches a problem in the table, then the MDD associated to that remaining subproblem is returned (line 9). If the key does not match a subproblem (line 10), then we choose the next unfixed variable x in X. If there are no more unfixed variables, then the remaining subproblem is empty and we return the MDD TRUE. If not, we branch on each possible value of x and create an MDD is a disjunction of the MDDs corresponding to each assignment. Note that the MDD returned by the recursive call only considers the variables in R.

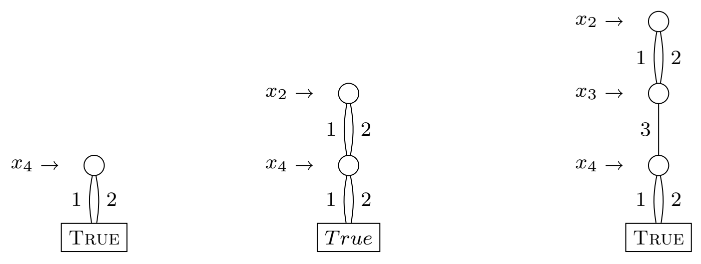
After branching on all values for x we store the MDD encoding the remaining subproblem in the map.

There is one important detail in the algorithm regarding fixed variables. The cache key $key(C, D)$ only considers which variables are fixed by D and not their values (except how they affect the constraints C). When variables are fixed by propagation, they may be fixed out of the sequence of variables X we are using to build the MDD, but eventually the MDDs returned must constrain these variables to their fixed values, respecting the variable order (by the definition of an MDD). The MDD m we attach to $key(C, D)$ represents the solutions for the remaining subproblem (and hence only considers the variable sequence Y that are not fixed). To return an MDD that considers the whole sequence X we build a "stick" MDD representing the fixed value for each newly fixed variable (line 17), and conjoin this with the MDD for the remaining subproblem.

A more straightforward approach to constructing MDDIFY is to attach the returned MDD to $key(C, D)$. But if we do this we may miss opportunities to reuse MDDs as we will see in Example 3.

Example 3 Consider a subproblem with $C = \{-x_1 + x_3 = 2\}$ and $D_0 = \{x_1 \in 1..2, x_2 \in 1..2, x_3 \in 3..4, x_4 \in 1..2\}$ using the order $X = [x_1, x_2, x_3, x_4]$.

The initial call MDDIFY(C, D_0, X) finds no propagation, and calculates key $(\emptyset, [], D_0)$, because binary constraints have no key [13], which has no entry in *problem_store*. We choose $x_1 = 1$ and make a recursive call MDDIFY$(C, \{x_1 = 1, x_2 \in 1..2, x_3 \in 3..4, x_4 \in 1..2\}, [x_2, x_3, x_4])$. Now propagation computes $D_1 = \{x_1 = 1, x_2 \in 1..2, x_3 = 3, x_4 \in 1..2\}$ and $fbp = \{x_3\}$; the key is $k_1 = (\{x_1, x_3\}, [], \{x_2 \in 1..2, x_4 \in 1..2\})$ which again has no entry. The remaining unfixed variables are $Y = [x_2, x_4]$. We then choose $x_2 = 1$ and make a recursive call MDDIFY$(C, \{x_1 = 1, x_2 = 1, x_3 = 3, x_4 \in 1..2\}, [x_4])$. After examining both possibilities for x_4 we build the MDD shown in Figure 5a attached to key $k_2 = (\{x_1, x_2, x_3\}, [], \{x_4 \in 1..2\})$. We then choose $x_2 = 2$ and the recursive call MDDIFY$(C, \{x_1 = 1, x_2 = 2, x_3 = 3, x_4 \in 1..2\}, [x_4])$ reuses the MDD attached to k_2. After exploring all values of x_2, the MDD of Figure 5b is attached to k_1, but the returned one is shown in Figure 5c, with the fixed value for $x_3 = 3$. When trying $x_1 = 2$, the recursive call MDDIFY$(C, \{x_1 = 2, x_2 \in 1..2, x_3 \in 3..4, x_4 \in 1..2\}, [x_2, x_3, x_4])$ computes $D_2 = \{x_1 = 2, x_2 \in 1..2, x_3 = 4, x_4 \in 1..2\}$ by propagation. The key is



(a) MDD for $[x_4]$ (b) MDD for $[x_2, x_4]$. (c) MDD for $[x_2, x_3 = 3, x_4]$.

Fig. 5: Construction of an MDD with caching.

$(\{x_1, x_3\}, [], \{x_2 \in 1..2, x_4 \in 1..2\})$, which matches k_1, thus returning the MDD in Figure 5b to which the value $x_3 = 4$ is inserted (line 8, with $fbp = \{x_3\}$).

If we instead did not delay inserting the fixed variable x_3 we would not discover a cache hit, and indeed the MDD of sub-figure 5c is not reusable. □

The example clearly shows how delaying the insertion of propagated variables is key to reuse MDDs. This optimization can result in important differences in MDD construction (e.g. ≈481k vs. ≈1 million search nodes for one instance).

Note that the algorithms by Bergman et al [3] and Hadzic et al [31] are not able to exploit this equivalence. First, their keys for D_1 and D_2 would not have matched unlike the ones we use, developed by Chu et al [13], because their linear constraint keys could not match (since in the approach we use, linear constraints do not have keys, this problem is avoided). Secondly, if they had used the more effective keys we use, their algorithms would need to be modified as ours, since the stored MDDs could not have been reused: they would have stored the MDD from Figure 5c which, because it contains the assignment $x_3 = 3$, cannot be reused in situations where $x_3 \neq 3$ (when $x_3 = 4$ and we compute the key D_2 in our example).

The key benefit of mddification compared to tablification [21, 20], is that we have to explore much less search space because of caching. Indeed, to construct a table, all solutions must be retrieved, thus a much broader region of the search space (all regions leading to satisfiable solutions at least) must be explored. In our approach, the stored MDDs represent parts of solutions that can be "plugged-in" to other partial assignments. As we saw in Example 3 we never had to explore $x_1 = 1 \wedge x_2 = 2$ or $x_1 = 2$ to retrieve the 6 solutions in those subspaces. In Section 4.3 we will see we also have this same advantage for the compilation of d-DNNFs.

It may not be obvious but Algorithm 1 does not require that X includes all variables in $vars(C)$. In this case the algorithm produces an MDD that encodes $\bar{\exists}_{vars(C) \setminus X} C \wedge D$, where $\bar{\exists}$ is the *quasi-projection* introduced by Chu and Stuckey [12]. Now $\exists_{vars(C) \setminus X} C \wedge D \Rightarrow \bar{\exists}_{vars(C) \setminus X} C \wedge D$, hence the resulting MDD does not remove any solutions of the original problem, and hence can be safely added to the original model. It is also guaranteed that $\bar{\exists}_{vars(C) \setminus X} C \wedge D \Rightarrow c$ for all c where $vars(c) \subseteq X$, so when adding the quasi-projection we can remove all constraints all of whose variables appear in X since they will be made redundant by the MDD of the quasi-projection.

4.2 Compiling a Cost-MDD

We can reuse Algorithm 1 for compiling a Cost-MDD under certain circumstances. Given an objective expression $o = \sum_{x \in X, d \in D(x)} (w_{xd} \times (x = d))$ if we build an MDD for the problem (C, D) for variable sequence X, we can convert this to a cost-MDD for calculating the value of o by adding weight w_{xd} to each edge labeled $x = d$ in the resulting MDD.

4.3 Compiling to d-DNNFs

The algorithm to build d-DNNFs from a set of constraints and variables is similar to the one to build MDDs. The major differences is that a fixed order of variables is not required (unlike in the case of MDDs). Therefore we do not need to worry about reinserting the set of variables *fbp* in the data structure as we did in Algorithm 1. The variables fixed by propagation will instead just be immediately added to the d-DNNF.

Algorithm 2 provides pseudocode for the compilation of d-DNNFs from a set of variables and constraints. The arguments are the same as for Algorithm 1.

Algorithm 2 Constructing d-DNNF via propagation.

```
1: procedure DDNNFFY($C, D, X$)
2:     $node \leftarrow$ TRUE
3:     $D \leftarrow propagate(C, D)$                 ▷ Propagate constraints $C$ to fixed point
4:     if $D$ is a false domain then return FALSE
5:     $fbp \leftarrow fixed(D) \cap X$               ▷ Variables fixed by propagation at this level
6:     for all $f \in fbp$ do
7:         $node \leftarrow node \wedge [\![f = val(f)]\!]$   ▷ Append leaves enforcing propagated values
8:     $p \leftarrow key(C, D)$                        ▷ Hash the remaining subproblem
9:     if $p \in problem\_store$ then
10:        $d \leftarrow problem\_store[p]$          ▷ d-DNNF representing remaining subproblem
11:        $node \leftarrow node \wedge d$
12:        return $node$
13:    $x \leftarrow$ NEXTVAR($X$)                    ▷ Next variable to branch on
14:    $children \leftarrow$ FALSE
15:    if $x = \bot$ then
16:        return $node$
17:    else
18:        for all $d \in D(x)$ do
19:            $children \leftarrow children \vee$ (DDNNFFY($C, D \cup \{x = d\}, X$) $\wedge [\![x = d]\!]$)
20:        $problem\_store[p] \leftarrow children$
21:    return $node \wedge children$
```

The algorithm constructs a d-DNNF *node* and returns it. First, we propagate the current decisions. If this produces a false domain, then there is no solution, and the returned node is simply FALSE. If this was conjoined to some other formula, then the conjoined formula would obviously become false as well. The variables that have been fixed at the current decision level are then retrieved and directly injected in the *node* d-DNNF. For MDDs we needed to inject them afterwards (in lines 8 and 17 of Algorithm 1). Here, the propagated assignments are just conjoined to the *node* in line 7.

Similarly to the MDD compilation, a lookup is done in the problem cache, and if found, the corresponding d-DNNF is returned, as it was already constructed. Notice how the returned d-DNNF is first conjoined with the *node* d-DNNF, which at this point may contain the assignments of variables fixed by propagation (in line 11).

Otherwise, a new variable x is chosen to be branched on. The choice of variable is wide open here, as there is no strict ordering in d-DNNFs. If no variable remains, then *node* can be returned. Otherwise, we iteratively assign a different value to x and recurse in the construction of the d-DNNF. The d-DNNF corresponding to the disjunction of recursive calls is stored for possible reuse. The returned d-DNNF is that same disjunction conjoint with the fixed variables (already in *node*).

Example 4 Consider the same problem as in 3: $C = \{-x_1 + x_3 = 2\}$ and $D_0 = \{x_1 \in 1..2, x_2 \in 1..2, x_3 \in 3..4, x_4 \in 1..2\}$. For simplicity we will use the same order $X = [x_1, x_2, x_3, x_4]$, but note that this is not necessary.

The initial call $\textsc{ddnnffy}(C, D_0, X)$ finds no propagation, and calculates key $(\emptyset, [], D_0)$, which has no entry in *problem_store* (just as earlier). We first branch on $x_1 = 1$ and make a recursive call $\textsc{ddnnffy}(C, \{x_1 = 1, x_2 \in 1..2, x_3 \in 3..4, x_4 \in 1..2\}, X)$. Now propagation computes $D_1 = \{x_1 = 1, x_2 \in 1..2, x_3 = 3, x_4 \in 1..2\}$ and $fbp = \{x_3\}$. The partially constructed d-DNNF is shown in Figure 6a (the recursive call will attach d-DNNFs to the edge shown in dashes); the key is $k_1 = (\{x_1, x_3\}, [], \{x_2 \in 1..2, x_4 \in 1..2\})$ which again has no entry.

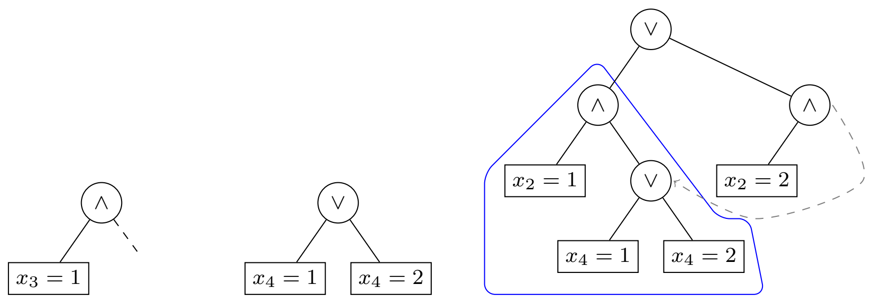
We then choose $x_2 = 1$ and make a recursive call $\textsc{ddnnffy}(C, \{x_1 = 1, x_2 = 1, x_3 = 3, x_4 \in 1..2\}, X)$. After examining both possibilities for x_4 we build the d-DNNF shown in Figure 6b attached to key $k_2 = (\{x_1, x_2, x_3\}, [], \{x_4 \in 1..2\})$. When the recursive call ends, this d-DNNF is conjoined to $[x_2 = 1]$ yielding the circled part of Figure 6c. We then choose $x_2 = 2$ and the recursive call $\textsc{ddnnffy}(C, \{x_1 = 1, x_2 = 2, x_3 = 3, x_4 \in 1..2\}, X)$ reuses the d-DNNF attached to k_2. This is shown in Figure 6c by a gray dashed arrow. After exploring all values of x_2, the d-DDNF of Figure 6c is attached to k_1, but the returned one is shown in Figure 6a, with the fixed value for x_3 (the dashed line connects to the root of Figure 6c). When trying $x_1 = 2$, the recursive call $\textsc{ddnnffy}(C, \{x_1 = 2, x_2 \in 1..2, x_3 \in 3..4, x_4 \in 1..2\}, [x_2, x_3, x_4])$ computes $D_2 = \{x_1 = 2, x_2 \in 1..2, x_3 = 4, x_4 \in 1..2\}$ by propagation. The key is $(\{x_1, x_3\}, [], \{x_2 \in 1..2, x_4 \in 1..2\})$, which matches k_1, thus returning the d-DNNF in Figure 6c to which the value $x_3 = 4$ is conjoined.

The final d-DNNF will have a $\vee$ root with two $\wedge$ children, each of them with 3 children, two of which are the values of x_1 and x_3 on each branch and the third being the d-DNNF in Figure 6c.

Notice that similar behavior as for MDDs, where the d-DNNF stored for reuse is not necessarily the same as the one returned by the algorithm. □

4.4 Splitting Subproblems for d-DNNFs

Because d-DNNFs are decomposable, it is natural to try to decompose the problem. Imagine a CSP composed of two completely independent parts: then the solution to the CSP is the conjunction of the solutions of all the parts. d-DNNFs are the perfect structure for this kind of behavior, if we can detect this independence of problems.



(a) Partial d-DNNF after propagation of x_3.

(b) d-DNNF for x_4

(c) d-DNNF for $[x_2, x_4]$.

Fig. 6: Construction of d-DNNF with caching

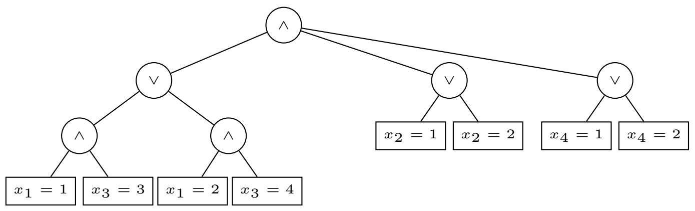


Fig. 7: Example of d-DNNF when splitting subproblems.

Example 5 Looking back at Example 4, variables x_2 and x_4 are completely independent from the rest of the problem. Therefore there are 3 independent sets of variables to this problem $\{x_1, x_3\}$, $\{x_2\}$ and $\{x_4\}$. The d-DNNF we would want for this would be as shown below in Figure 7. □

4.4.1 Detecting Independent Subproblems

A simple way to detect subproblems as being independent is to simply build the Constraints Graph after each decision. The Constraints Graph is a graphical representation of the Constraint Problem at hand. Nodes are variables of the problem, and edges connect variables that are related through constraints in the problem. This is commonly used in the compilation of d-DNNFs that we have seen in Section 2.

In our implementation, we consider all propagators and clauses to be *keyable* objects. All keyable objects produce a key (possibly empty) for subproblem equivalence (as seen in Section 3.4). In addition, we make all keyable objects update a disjoint-set data structure (also known as union-find) of variables at the time of producing a key. This introduces little overhead, since both tasks can usually be performed at the exact same time. By default, all keyable objects will unite all variables involved with the keyable itself. For instance, for a clause, all Boolean variables involved in the clause are united. Similarly, all variables in each propagator are united. In addition, as our implementation is in the CHUFFED solver, which implements no-good learning, integer variables are also united with the literals associated to them. Thanks to this, we build an internal Constraints Graph after each decision.

The goal is to then consider each connected component of the Constraints Graph separately, as they are indeed independent problems. The expected d-DNNFs will look similar to the one in Figure 7, where the independent subproblems are all children of an "$\wedge$" node.

4.4.2 Splitting Global Constraints

Unlike CNFs or SAT problems, CP has global constraints. Sometimes, after some decisions are made, it is possible that some global constraints can split their variables into independent connected components of the Constraints Graph. Because these constraints are "opaque" they cannot be split directly as we do with other constraints. That is, we can separate them from each other, but it is not obvious how to split the inside of a global constraint.

Example 6 Consider the constraint $\mathit{alldifferent}([a, b, c, d, e])$ where $D(a) = D(b) = [0..5]$, $D(d) = D(e) = [6..10]$ and $D(c) = [0..10]$. If we make the decision that $c < 5$ then the constraint can be split into two globals: $\mathit{alldifferent}([a, b, c]) \wedge \mathit{alldifferent}([d, e])$. $\square$

This is something that each global needs to implement, as it cannot be generalized for all globals.

Fages et al [25] formalized the idea of splitting globals. They showed it is very useful even simply for reducing the propagation time, arguing that propagating on many "smaller" constraints is faster than propagating on few "big" constraints. They showed how to split the `alldifferent` constraint as well as the `cumulative` constraint. In their paper, though, their algorithm does not require that the splitting is into disjoint sets of variables. That is, they can split the variables involved in an `alldifferent` in a set of sets $G = \{S_1, S_2, ..., S_n\}$ such that $\exists i, j, S_i \cap S_j \neq \emptyset$. In our case we can't have this, since the d-DNNF needs to split into independent set of variables. But this is a minor difference.

We have implemented the global splitting for three globals, as follows.

1. *alldifferent* [25]: the variables are split into groups by overlapping domains. That is, if the intersection of the domain of two variables is empty, then the variables are not united at this stage (note that some other constraint may unite them).
2. *minimum*: this constraint enforces that a variable y takes the value of the smallest variable in an array x of integer variables. The split here is also trivial: only the variables in x whose domain overlaps the domain of y are united.
3. *tree* [19]: this constraint enforces a graph to be a tree. An articulation node is a node that, if removed, splits a graph into at least two "induced" disconnected subgraphs. As shown by De Uña et al [19], these nodes need to be in the solution tree. The solution to a *tree* constraint with an articulation node is the conjunction of the solutions to the *tree* constraint on each "induced" subgraph. Therefore, the nodes in each "induced" subgraph are connected to each other, but not across subgraphs.

As an anecdotal result, for one `alldifferent` constraint with 10 variables where the domains of the even-numbered variables is 1..5 and the domain of the odd-numbered variables is 6..10 except that the first variable had domain 1..10 we saw a huge difference in size. The d-DNNF built using splitting had 635 nodes, whereas the one without splitting the `alldifferent` global had 5461 nodes! In the first case, the d-DNNF split in 4 rapidly (first in 2, depending on the value of the 1..10 variable, and once that decision was made, there are two independent conjoined d-DNNFs). In the second case, the d-DNNF was much more intricate and hard to visualize.

Once we have this ability to split a problem into smaller subproblems, we can also cache them independently, as shown in Algorithm 3. Notice in this version, there is a loop through all the independent problems, and each one of them is treated individually (each has a key and its own set of variables). The call to *key* was changed to a call to a *split* function that, in turn, calls a function for each propagator that produces a key and updates a union-find structure. We saw no overhead from updating the union-find. The results section backs this claim.

Algorithm 3 Constructing d-DNNF via propagation.

```
1: procedure DDNNFFY($C, D, X$)
2:     $node \leftarrow$ TRUE
3:     $D \leftarrow propagate(C, D)$                    ▷ Propagate constraints $C$ to fixed point
4:     if $D$ is a false domain then return FALSE
5:     $fbp \leftarrow fixed(D) \cap X$                 ▷ Variables fixed by propagation at this level
6:     for all $f \in fbp$ do
7:         $node \leftarrow node \wedge [\![f = val(f)]\!]$     ▷ Append leaves enforcing propagated values
8:     $pbs \leftarrow split(C, D)$                      ▷ Split into independent subproblems
9:     for all $pb \in pbs$ do
10:        if $pb.key \in problem\_store$ then
11:            $d \leftarrow problem\_store[pb.key]$
12:            $node \leftarrow node \wedge d$
13:            continue
14:        $x \leftarrow$ NEXTVAR($pb.vars$)          ▷ Next variable to branch on for this subproblem
15:        if $x = \bot$ then                              ▷ No more variables
16:            return $node$
17:        else
18:            $children \leftarrow$ FALSE
19:            for all $d \in D(x)$ do
20:                $children \leftarrow children \vee$ (DDNNFFY($C, D \cup \{x = d\}, X$) $\wedge [\![x = d]\!]$)
21:            $problem\_store[p.key] \leftarrow children$
22:            $node \leftarrow node \wedge children$
23:    return $node$
```

4.4.3 Variable Selection

As said in the introduction, d-DNNFs are a well known data structure in the fields of model counting. The notion of splitting the problem into independent subproblems is also well known in that field. It is therefore natural that researchers have investigated search strategies to be able to enhance this splitting of problems.

We implement the Variable State Aware Decaying Sum (VSADS) strategy, which is a combination of Variable State Independent Decaying Sum (VSIDS) and Dynamic Largest Combined Sum (DLCS). It was introduced by Davies and Bacchus [18] and a thorough explanation can be found there. All our experiments used this strategy to generate the d-DNNFs (using the same strategy meta-parameters as Davies and Bacchus [18]). The call to NEXTVAR() in Algorithm 2 uses this strategy.

5 Experiments Descriptions and Results

This section first describes the problem chosen to evaluate these precompilation techniques, then follows the experimental results. The results are divided in three major parts. First, we evaluate compilation time in Section 5.2.1, that is the time to precompile part of a model as a table, MDD or d-DNNF. Section 5.2.2 studies the benefits of splitting subproblems and globals as described in Section 4.4. Finally we compare the net benefit of the three precompilation techniques in Section 5.2.3. All instances are available online.[1]

For most of our experiments, we start with a MiniZinc model to which we added an annotation `mddify` or `ddnnffy` in the declaration of the variables to be precompiled. At this stage, this is an ad-hoc solution that is only supported in Chuffed (since this is the solver we use in our prototype) which reads this annotation in the generated FlatZinc file and will do precompilation rather than solving. Ideally, this process will be automated with an annotation within the MiniZinc compiler much like the work by Dekker et al [21].

Most of our test cases come from the paper by Dekker et al [21] and are a direct comparison to their work. The "Shift Scheduling" and "Concert Hall" problems (Section 5.1.6) were chosen as examples of scheduling problem, which is a typical application of Constraint Programming. The "All-Trees" (Section 5.1.8) problem was created to demonstrate the value of splitting globals. The instances, though, are standard amongst the Steiner Tree problem instances. "Fox Geese Corn" and Progressive Graph Coloring (Sections 5.1.3 and 5.1.5) were chosen to showcase how for even simple problems, the use of tables can be inadequate.

5.1 Problems Description

5.1.1 Black-Hole

The well known Black-Hole [30, 47] problem can be stated as follows. We are given 17 piles of 3 cards, and the Ace of Spades as the starting point of the "black-hole". The player needs to move the top of a pile onto the black hole until all the cards are on the black-hole. The card to be moved must be one less or one more than the current top of the black-hole, regardless of suit. The predicate ensuring that two cards are adjacent is shown in Figure 8.

For this problem, we separate the predicate from the problem and build an MDD or d-DNNF that ensures two cards are adjacent. The variables of our solution store are the parameters of the predicate. The body of the predicate is then substituted by the call to the global corresponding to either encoding. The

[1] https://people.eng.unimelb.edu.au/pstuckey/mdd_ddnnf_precomp/mdd_ddnnf_precomp.zip

```
    predicate adjacent(var 1..52: a, var 1..52: b) =
              ((a-b) in {13*i+1 | i in -4..3} union {13*i-1 | i in -3..4});
```

Fig. 8: MiniZinc Predicate for Black-Hole

MDD/d-DNNF has two variables of domains 1..52. We used 21 instances from Dekker et al [21].

5.1.2 Block Party Metacube

The Block-Party Metacube Problem [21] can be defined as follows. Cubes have *icons* on each corner of each side, which have 3 attributes: shape, color and pattern. A metacube is an arrangement of 8 of these cubes that forms a cube (i.e. $2 \times 2 \times 2$ cubes). A *block-party* metacube is formed when the 4 icons on the center of each face of the metacube are all identical (in all 3 attributes) or have all attributes different. The full model for this problem can be found in [20]. As part of the model, a predicate is defined to link the identifier of each cube to the three icons of that cube that are on the center of a face of the metacube, for all the possible rotations of each cube. The predicate was presented and tablified by Dekker et al [21] and is shown in Figure 9. The annotated variables are the arguments, so there will be 4 variables in the MDD/d-DNNF, each with a domain 0..63 (there are 64 possible symbols). We used 14 instances from [21].

```
    predicate link_cube_and_symbols(array [1..4] of var 0..63: cs) =
              let { var 1..24: pos, var int: cube = cs[1],} in
                  forall ( i in 1..3 ) ( data[cube, pp[pos, i]]==cs[i+1] );
```

Fig. 9: MiniZinc Predicate for Block-Party

5.1.3 Fox Geese Corn (FGC)

This is a generalization of the famous Fox-Goose-Corn puzzle. In this version, a farmer wants to transport f foxes, g geese and c bags of corn from the west to the east side of a river. She has a boat with a capacity available for her to move *some* of the goods at once while the rest remain on shore. She can go back and forth to bring as many goods as she wants to the east. Nonetheless, some rules apply to the goods that are not being supervised on either side while the farmer is on the boat: *i)* If only foxes and bags of corn are sitting on a shore, then a fox dies by eating a bag of corn; *ii)* If there are foxes and geese, and the foxes outnumber the geese, one fox dies; *iii)* On the other hand, if the geese are not outnumbered, each fox kills one goose; *iv)* If there is no fox, and the geese outnumber the bags, a goose dies and one bag is eaten; *v)* On the other hand, if the corn is not outnumbered, each goose eats a bag.

The farmer must maximize the profit (there is a price for each good) from the surviving goods on the east. Although she could do any number of trips, there is an optimal number of trips t after which it is worth abandoning goods on the west and continue her journey with the goods on the east. The natural way of modeling this is by defining a predicate for the above rules and apply the predicate for trips before t. It will update the number of foxes, geese and corn bags for the next time slice. After time t, the predicate is not applied, as she is not crossing the river anymore. Thus the predicate is reified by a condition dependent on t. This is precisely the model we used. The precompiled predicate has 7 variables (3 for the state before the travel, 3 for the state after the travel, and a Boolean to turn on and off the constraint). Their domains will be highly dependent on the instances: we use 11 instances with domains ranging between 5 and 27 for each variable. The predicate's definition is in Figure 10.

```
predicate alone(var bool: reif, var 0..f: fox0, var 0..f: fox1,
                var 0..g: geese0, var 0..g: geese1, var 0..c: corn0,
                var 0..c: corn1) =
          if reif == true then
              if fox0 = 0 /\ geese0 = 0
                 \/ fox0 = 0 /\ corn0 = 0
                 \/ geese0 = 0 /\ corn0 = 0
                 \/ fox0 < 0 \/ geese0 < 0\/ corn0 < 0  then
                    fox1 = fox0 /\ geese1 = geese0 /\ corn1 = corn0
              elseif fox0 > 0 /\ geese0 > 0 then
                    corn1 = corn0
                    /\ if fox0 > geese0 then
                            fox1 = fox0 - 1 /\ geese1 = geese0
                       else
                          fox1 = fox0 /\ geese1 = geese0 - fox0
                       endif
              elseif geese0 = 0 /\ fox0 > 0 /\ corn0 > 0 then
                    fox1 = fox0 - 1 /\ geese1 = 0 /\ corn1 = corn0 - 1
              else
                    fox1 = 0
                    /\ if geese0 <= corn0 then
                       corn1 = corn0 - geese0 /\ geese1 = geese0
                    else
                       corn1 = corn0 - 1 /\ geese1 = geese0 - 1
                    endif
              endif
         else true endif;
```

Fig. 10: MiniZinc Predicate for Fox-Geese-Corn

5.1.4 Water Bucket

This is another classic puzzle. Given a set of buckets (or jars) of water (some initially filled, some not), and a target water level for each bucket, transfer the initially contained water into other buckets to reach the target levels. The difficulty lies in the fact that we cannot stop pouring water anytime we please, but only when either the receiving bucket is full or the pouring bucket is empty. The objective is to achieve the final levels in as few transfers as possible.

The model uses a predicate that represents the transition of states of the buckets at a given time when choosing which buckets to transfer, Figure 11. There will be $2 \times j + 2$ variables (where j is the number of buckets). The domains depend on each instance, but the biggest domain for any variable across all 4 instances we used is 0..12 with up to 4 buckets.

```
predicate transfer(array[BUCKET] of var PINT: state_b, array[BUCKET] of var
    PINT: state_a, var BUCKET: from, var BUCKET: to) =
          (state_b = final /\ state_a = final /\ from = 1 /\ to = 1) \/
          (forall(b in BUCKET where b != from /\ b != to)
                 (state_a[b] = state_b[b]) /\
                 [state_a[from],state_a[to]] = pour(state_b[from], capacity
   [from], state_b[to], capacity[to]));

function array[1..2] of var PINT: pour(var int: from_b, var int: from_cap,
                                       var int: to_b, var int: to_cap) =
        let { var PINT: amount = min(from_b,to_cap - to_b); } in
             [from_b - amount, to_b + amount];
```

Fig. 11: MiniZinc Predicate for Water-Bucket

5.1.5 Progressive Graph Coloring (PGC)

We introduced this problem for our tests. Given a graph of n nodes, an initial coloring with c colors and a target coloring, change the color of at most k nodes at each step to reach the target coloring in as few steps as possible maintaining a valid coloring at all times. A valid coloring is one where any two adjacent nodes are not assigned the same color.

This is modeled with a “valid coloring” predicate, called at each step, shown in Figure 12. The variables taken in the predicate will be one for each node, all with domains $1..c$. We build the graphs using Erdös-Rényi’s model [24] with the probability of an edge existing being 0.5 for 15-node graphs, and 0.2 for 25-node graphs.

```
predicate valid_coloring(array [NODES] of var COLORS: coloring,
                         array [EDGES] of NODES: xs,
                         array [EDGES] of NODES: ys) =
          forall (e in EDGES) (coloring[xs[e]] != coloring[ys[e]]);
```

Fig. 12: MiniZinc Predicate for Valid Coloring

5.1.6 Shift Scheduling

This problem was first introduced by Demassey et al [22]. It consists in allocating n workers in 15 minute shifts to a activities such that all activity has the minimum required number of workers at all times. The objective is to minimize the number of shifts worked. The constraints are: *i)* Workers must work on a task at least 1 hour, and cannot switch tasks without a 15 minute break; *ii)* Part-time workers work between 3 and 5.75 hours, with one 15 minute break; *iii)* Full-time workers work 6 to 8 hours, with one hour for lunch, and 2 breaks (one before and one after lunch); *iv)* Workers can only be working after the first activity is started, and before the last activity finishes.

This problem can be modeled using a `grammar` constraint as described by Gange et al [29]. We used their exact same model, with 5 variations to it:

– ShiftReg uses regular constraints (implemented by MDD propagators) to enforce the "shape" of a valid shift.
– ShiftDec uses a decomposition into clauses to enforce valid shifts.
– ShiftNNF and ShiftGCC use grammar propagators [35] to enforce valid shifts.
– ShiftWRg uses cost-regular constraints (implemented by cost-MDD propagators) to enforce the shape of shifts, and compute their costs (the objective).

ShiftGCC uses a `gcc` constraint [55] to ensure the demand of workers is met at each shift, whereas the other models simply use a linear for this. The cost function is modeled with a linear constraint in all cases (except ShiftWRg).

For this problem, we choose the variables of the MDD/d-DNNF to be the task allocation of all workers in windows of 3 adjacent shifts. That is, one for each 45 minutes. The ordering of the variables puts first the allocations of the first worker for three shifts, then the allocations of the second worker for the same three shifts, and so on. We used 8 instances with up to 2 activities and 5 workers. In the original paper by Gange et al [29], the instances were directly modeled in Chuffed. We also worked directly on Chuffed for ease of implementation. The model is available on the official release of Chuffed.[2]

5.1.7 Concert Hall Scheduling Problem

In this problem we have n concert halls and m concerts to schedule each in one of the concert halls. Concerts can be chosen to not be scheduled by scheduling them in a special hall numbered 0. Each concert has fixed start and end times and a given profit that we acquire when we schedule the concert. The concerts need to be scheduled such that they do not overlap. The goal is to minimize the set of unscheduled concerts taking into account the profit they would have brought.

The MiniZinc model can be seen in Figure 13.

For this problem we use the Boolean variables indicating whether a concert is scheduled or not as the variables for MDD/d-DNNF. Nonetheless, we do not put all the variables into the same data structure of choice. Instead, we split them evenly in 4 parts by start date. That is, in an instance with 40 concerts, we would get 4 MDDs (or d-DNNFs), each with 10 variables, such that the first MDD (or d-DNNF) would contain the 10 concerts with earlier starting date, the second one would contain the next 10 concerts chronologically and so on. All constraints in the model will eb active when doing the compilation.

We used 20 instances, all with $n = 8$ concert halls and $m = 45$ concerts. The number of times slots ranges between 52 to 63.

[2] https://github.com/chuffed/chuffed/blob/master/chuffed/examples/shift.cpp

```
int: n;  % number of halls
int: m;  % number of concerts
set of int: CONCERT = 1..m;
array[CONCERT] of int: profit;

int: o; % number of interesting times;
set of int: TIME = 1..o;
array[TIME] of set of CONCERT: starts;
array[TIME] of set of CONCERT: ends;

array[0..o] of var 0..n: usage;
array[CONCERT] of var 0..1: scheduled;

constraint usage[0] = 0;
constraint forall(t in 1..o) (usage[t] = usage[t-1]
                              + sum(c in starts[t])(scheduled[c])
                              - sum(c in ends[t])(scheduled[c]));

var int: objective = sum(c in CONCERT)((1 - scheduled[c]) * profit[c]);

solve minimize objective;
```

Fig. 13: MiniZinc Model for the Concert Hall Problem

5.1.8 All-Trees

To study whether splitting globals is worthwhile or not, we created a very simple model to collect in a d-DNNF all the solutions to the Steiner Tree Problem in a set of graphs.

Given a graph $G = (V, E)$ and a set of nodes called *terminals* $T \subseteq V$, a Steiner Tree is a tree S_T that is a subgraph of G and that contains at least the nodes in T. That is, S_T spans the terminals. The Steiner Tree Problem consists in finding the Steiner Tree of minimum cost given a weighting function $w : E \mapsto \mathbb{R}$.

Here we are not concerned with the Steiner Tree Problem, instead what we want is to simply collect all the Steiner Trees of a graph given a set of terminals into a d-DNNF. This will help us understand whether splitting the `tree` constraint was worthwhile. The d-DNNF can then be used in a variety of problems involving Steiner Trees.

We used instances from the SteinLib [37], from the dataset ES10FST and ES20FST (15 instances each). The model is shown in Figure 14.

```
predicate all_trees(array[NODES] of var bool: in_nodes,
                    array[EDGES] of var bool: in_edges,
                    set of NODES: terminals,
                    array[NODES,NODES] of bool: adjacency) =
          forall(t in terminals) (in_nodes[t])
          /\ tree(in_nodes,in_edges,adjacency);
```

Fig. 14: MiniZinc Predicate for Steiner Tree without costs

5.2 Experimental Results

We present here our results. The experiments were done on an Intel® Core™ i7-4770 CPU @ 3.40GHz running Linux 3.16, with the CP solver Chuffed [14].

The table propagator implemented in Chuffed is a decomposition in clauses of the table. The propagators for MDDs, cost-MDDs and d-DNNFs were described by Gange et al. [28, 29, 27]. Note that the results shown for tablification also use Chuffed for precompilation (that is building the table) and for solving the problem.

5.2.1 Compilation

We first compare the results of the first 5 problems against the tablification approach [21]. Let us first compare the total precompilation time for each problem (all instances summed up) in Table 1. For Black-Hole, the same compilation could be recycled for all instances. All the others are instance dependent.

Problem	Instances	Table construction	MDD construction	d-DNNF construction
Black-Hole	21	**0.01s**	0.02s	0.02s
Block-Party	14	0.30s	**0.28s**	0.28s
FGC (reif)	(11) 3	($+\infty$) 9.06s	(6.40s) 0.22s	(6.65s) **0.19s**
FGC (no reif)	11	0.60s	0.27s	**0.21s**
Water Bucket	4	173.15s	0.45s	**0.44s**
PGC ($n = 15, c = 6$)	30	10.89s	**7.06s**	7.74s
PGC ($n = 25, c = 5$)	(31) 25	($+\infty$) 437.68s	(90.24s) **77.67s**	(121.63s) 105.03s
Total	122	636.1s	**85.97s**	113.93s

Table 1: Total time spent compiling Tables, MDDs or d-DNNFs. For problems where the Table construction failed we srestrict to the instances where Table construction succeeded. The statistics for all instances of these problems are given in parentheses.

There are two things worth noting from this table. First building a table can behave poorly when there are too many solutions to a problem. Indeed, building a table requires finding all the solutions to a problem. We noticed how for some of the instances of PGC with 25 nodes, there are 65k solutions, including an instance with more than 13M solutions. That explains the result in the last row.

More importantly, we had to rewrite the model for FGC. In the original model, with the reified predicate, building the table is simply impossible. For 8 of the 11 instances used, the compilation step ran out of memory in our machine. The reason is simple: if the reifying Boolean is turned off, all valuations of the other 6 arguments are valid. The memory explosion happens even with relatively small domains (6 to 8 fox, geese or corn bags), but it does not happen when building MDDs or d-DNNFs. In the case of the MDD this is because the MDD under the decision “turn off Boolean” is simply a multi-edged stick (much like the one in Figure 5b) where all problems are equivalent. In the case of the d-DNNF, the part under the “turn off Boolean” decision is compressed into multi-edged stick with intermediate layers for the assignments of each variable. This can be seen in Figure 15.

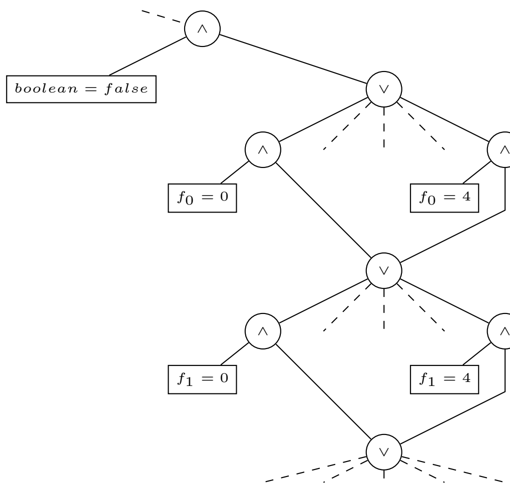


Fig. 15: d-DNNF for the FGC example when the Boolean `reif` in the reification is set to false (dashed lines link to other "$\wedge$" nodes, not shown for lack of space)

The time to build the MDDs and d-DNNFs of the 3 instances of FGC that could be tablified is shown in the table (the time for all 11 instances is in parenthesis). The rewritten model for FGC is basically the same as the first one, except that we remove the reification argument from the predicate and apply the constraint to the leftover foxes, geese and corn after the optimal number of trips. The same problem occurred with six of the bigger instances of the PGC problem.

This problem also arises if we construct our data structures without the use of the equivalence keys shown in Section 3.4. For example, for a FGC instance with domains 0..50, it took ≈ 16.5 hours to construct the MDD without using cache keys, but only 61 seconds when using them. Both yielded the same MDD, of course. The same behavior happens for d-DNNFs.

From these experiments we see that building compact structures as MDDs or d-DNNFs can be substantially faster than building the tables, and is rarely slower. More importantly, it is safer in terms of resource consumption. This is why compiling a table and then converting it into an MDD [9], for example, is not as good as building an MDD via caching: the table needs to be built anyway, which is the bottleneck in this case.

5.2.2 d-DNNF Independent Subproblems

Here we investigate whether the effort made for splitting subproblems is actually worthwhile. Recall that, from what we saw in Section 4.3, this comes practically for free, so we don't expect any noticeable loss in time. Table 2 shows a comparison of the construction of d-DNNFs.

Looking at each individual instance, the splitting was not useful for all instances. In fact, for some instances, there were no independent subproblems found at all. This is likely due to the order of the decisions. Indeed, how often a problem can be separated into independent subproblems is very much determined by the order in which decision are made (some decisions induce more splitting).

	Time		Average d-DNNF Nodes	
Problem	Using Splitting	No Splitting	Using Splitting	No Splitting
Black-Hole	0.02s	0.02s	64.00	64.00
Block-Party	**0.28s**	0.42s	207.00	207.00
FGC (reif)	**6.65s**	8.18s	337.27	337.27
FGC (no reif)	**0.21s**	0.22s	273.63	273.63
Water Bucket	0.44s	0.44s	1599.25	1599.25
PGC ($n = 15, c = 6$)	**7.74s**	84.24s	**6436.19**	31281.20
PGC ($n = 25, c = 5$)	**121.63s**	280.67s	**37778.80**	139267.60
Total	**136.97s**	374.19s		

Table 2: Comparison of time spent to build and size of d-DNNFs with or without splitting independent subproblems.

However, we can clearly see in the PGC problem how beneficial splitting can be. This saved us around 65% of the time and produced d-DNNFs almost 4 times smaller in average.

We could not identify any cue as to when splitting will or not happen, apart from obvious cases where a single variable unites two very distinct problems. For example, looking at one instance of the PGC problem with 25 nodes that takes 39s with splitting but 93s without it, we could not see anything special about it compared to instances that took similar times with or without splitting. We believe that the VSADS search can sometimes pick the right variables to branch on to create more splits, but as usual with search strategies, they don't always work perfectly on all instances.

We look now at the task of collecting all Steiner Trees of a graph into a d-DNNF (see Section 5.1.8). To do so, we write a MiniZinc model using the `tree` constraint [19] and we d-dnnffy the binary variables corresponding to edges of the graph. Table 3 shows the results for all the instances we tested.

The table does not show instances where the resulting d-DNNFs are identical and therefore splitting was not beneficial. In those instances times were always identical (and the construction was actually immediate). As can be seen, there is never a loss from implementing this splitting technique within global constraints. For example, for the instance es10fst14 we saved 96% of the time, and 95% of nodes. Big gains can also be seen in other instances like es10fst{08,09} or es20fst{09,13,14,15}. For instance es20fst07, the version with splitting the `tree` global constraint took 14 minutes, whereas the non-splitting version does not terminate within one hour.

This shows the value of implementing the independent subproblem detection *inside* global constraints. We conclude this is a good implementation choice since, when it pays off, it does so greatly, and when it does not, there is no loss.

5.2.3 Solving CSPs and COPs

We now compare how much having compiled subproblems helps in solving the problem, depending on the data structure used. All instances ran with a time limit of one hour.

Figure 16 shows a comparison in solving time between the original model and having mddified or d-dnnffied a predicate (including compilation time). The plots are split in two groups of problems for clarity. The right panels shows results for the 4 sets of instances of PGC, and the left show the other tests.

Problem	Vars.	Time		d-DNNF Nodes		Ratios	
		Splitting	No Splitting	Splitting	No Splitting	#Nodes	Time
es10fst01	20	0.01s	0.01s	**416**	461	0.90	1.00
es10fst03	20	0.01s	0.01s	**330**	548	0.60	1.00
es10fst04	20	0.01s	0.01s	**206**	237	0.87	1.00
es10fst06	20	**0.01s**	0.02s	**474**	854	0.56	0.50
es10fst08	28	**0.29s**	2.97s	**3277**	56109	0.06	0.10
es10fst09	29	**0.30s**	13.38s	**3087**	162988	0.02	0.02
es10fst10	21	0.01s	0.01s	**78**	370	0.21	0.00
es10fst13	21	0.01s	0.01s	**310**	561	0.55	1.00
es10fst14	32	**0.57s**	13.85s	**7419**	157085	0.05	0.04
es10fst15	18	0.01s	0.01s	**150**	175	0.86	1.00
es20fst04	83	>3600.00s	>3600.00s	—	—	—	—
es20fst05	77	>3600.00s	>3600.00s	—	—	—	—
es20fst07	59	**857.64s**	>3600.00s	**664054**	—	—	—
es20fst08	74	>3600.00s	>3600.00s	—	—	—	—
es20fst09	42	**1.84s**	9.05s	**22523**	125954	0.18	0.20
es20fst10	67	>3600.00s	>3600.00s	—	—	—	—
es20fst11	36	**0.02s**	0.03s	**409**	801	0.51	0.67
es20fst12	36	**0.10s**	0.14s	**2657**	3574	0.74	0.71
es20fst13	40	**0.09s**	0.47s	**1284**	7194	0.18	0.19
es20fst14	44	**2.32s**	371.01s	**14795**	758384	0.02	0.01
es20fst15	43	**0.10s**	1.30s	**1072**	17766	0.06	0.08

Table 3: Comparison of time spent to build and size of d-DNNFs with or without splitting independent subproblems for a problem where the split is done in a `tree` constraint.

For MDDs, as we see for the first 5 problems, compiling is clearly beneficial, as the solving time is generally smaller. For the PGC problem we notice a more scattered plot. An interesting point that we show is that there are two sets of instances of 25 nodes. The first one (marked ◦) averages 184450 nodes for each MDD, whereas the second averages 9402 nodes. This indicates that, despite a fast compilation step (total of 23.3 seconds for all 11 instances on the ◦ set), the sizes of the MDDs are impractical for efficient propagation. This suggests that, after compiling, the user may want to avoid using the resulting MDD if it is too big. Since compiling takes less than ≈ 3 seconds for each of these instances, it is reasonable to try mddifying and then ignore the result if it is too big, or simply abort the compilation. As a comparison, the number of nodes in d-DNNFs of the PGC instances with 25 nodes were, on average 50835 and 2488 for "◦" instances and unmarked instances respectively.

Leaving aside these instances, most instances are solved quicker when the MDD is added, and the ones that are not tend to be solved in under 1 second in any case.

For d-DNNFs, the results are a bit less clear. For the first 5 problems, around 57% of the problems are solved faster using d-DNNFs. It is not clear that d-DNNFs would actually be useful in this case. On the other hand, looking at the hardest instances of PGC, we see that d-DNNF pays off compared to MDDs: most of the hard instances are now solved within 10 seconds with d-DNNFs whereas the original model would solve them in between 100 seconds and 1 hour. We conclude that d-DNNFs are probably more practical for hard problems. Indeed it seems like the biggest payoff (up to 3 orders of magnitude) appears only in the instances that originally took the longest to solve.

Figure 17 compares the solving time (including compilation time) when using an MDD or d-DNNF against using a table.

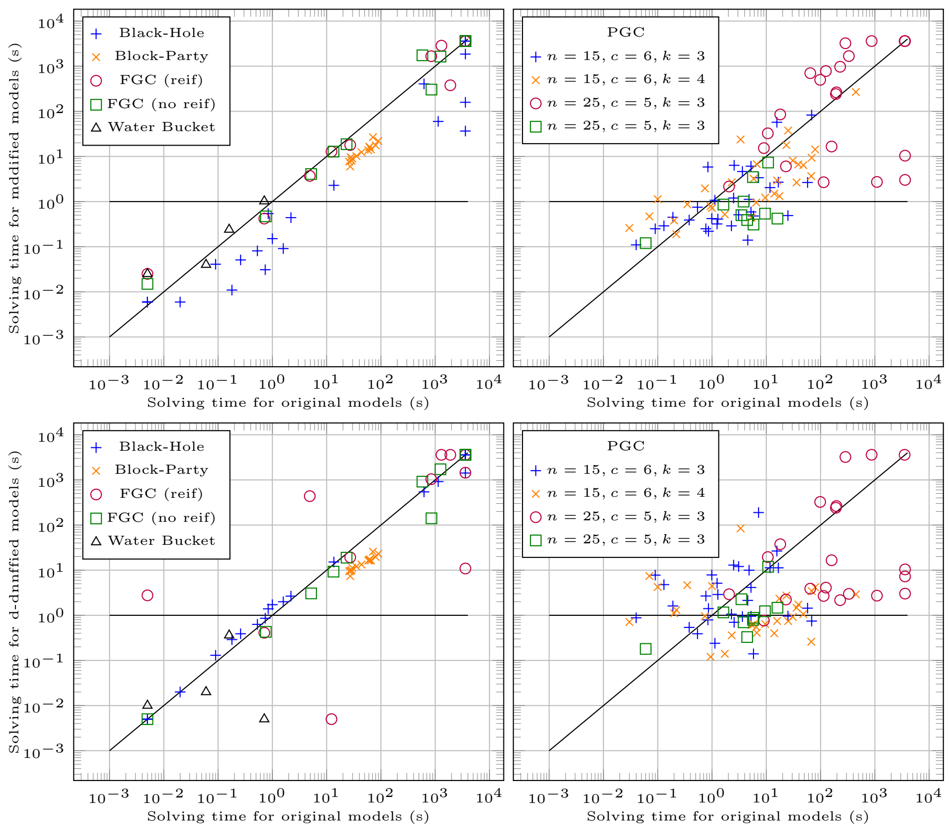


Fig. 16: Time to solve mddified/d-dnnfied instances, including compilation time (y-axis) vs. time to solve original models (x-axis).

We notice that for all the Block-Party instances, a table is better. Overall the trend is that using an MDDs can save a lot of time, and occasionally lose little time: when tablifying performs better, the gain is marginal or the instance was solved rather quickly anyway (under 1 second). When the MDD wins, it can make a difference of up to 3 orders of magnitude (c.f. Water Bucket, FGC and some of the 25 node PGC instances).

The results for d-DNNFs are more scattered, but we still see a similar pattern as before: for very big instances, d-DNNFs perform better than tables or original models. This can be clearly seen in the biggest PGC instances, as they are solved 3 to 4 orders of magnitude faster with d-DNNF than with tables.

Let us now look at a direct comparison between MDDs and d-DNNFs in Figure 18. Once again, we notice that MDDs perform better in general, but d-DNNFs seem to be more appropriate for very large instances (like the ∘ instances of the PGC problem).

Furthermore, out of the 122 instances, the mddified models solved 111, and the d-dnnfied models solved 112. The tablified models solved only 87 (out of 108, since 14 of the 122 could not be tablified). The original models solved 108.

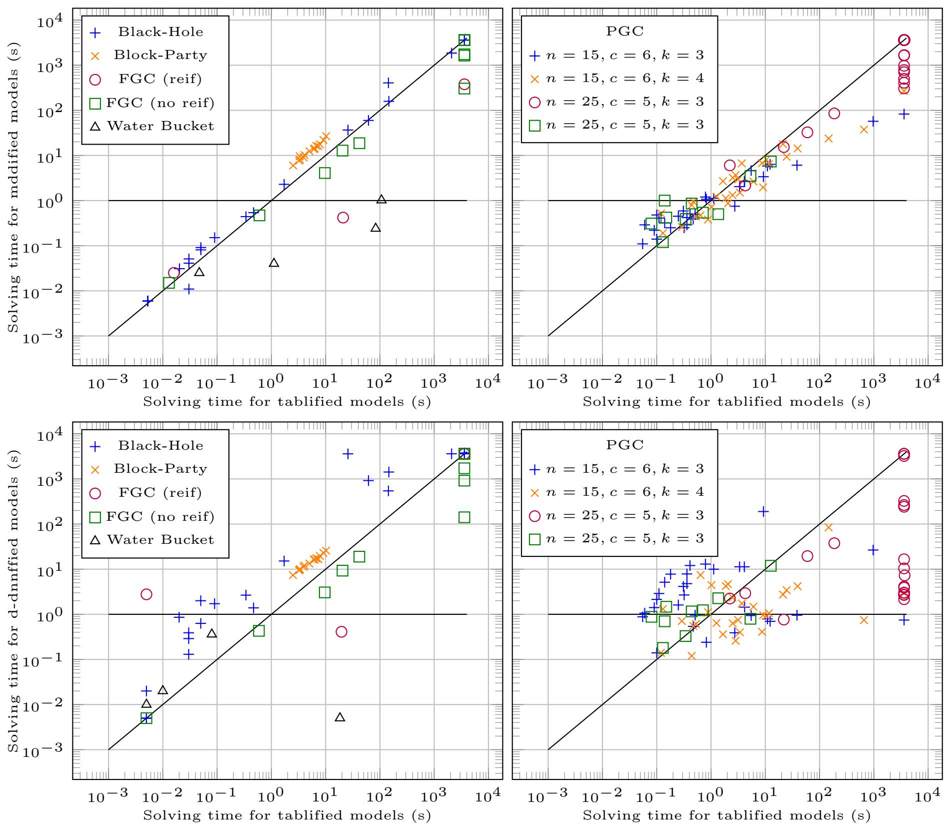


Fig. 17: Time to solve mddified/d-dnnffied instances, including compilation time (y-axis) vs. time to solve tablified instances, including compilation time (x-axis).

Overall, we notice two main results. First, mddifying can yield big MDDs that are impractical. Second, for easy instances, having the extra MDD propagators can be overkill. This technique is therefore more tailored for hard problems that do not result in huge MDDs. The user may also want to experiment with instance-dependent ordering of the variables to achieve more compact MDDs. Interestingly enough, it seems that the pitfall of MDDs is the strength of d-DNNFs: d-DNNFs seem to perform better than the other alternatives when the set of solutions is too big ($\geq 100k$). This brings the possibility of a more robust precompilation technique, where one of the 3 approaches (tablifying, mddifying or d-dnnffying) is used depending on an estimate of the size of the solution set for the subproblem to compile. Such estimate could be obtained, for example, using a structural approach [53].

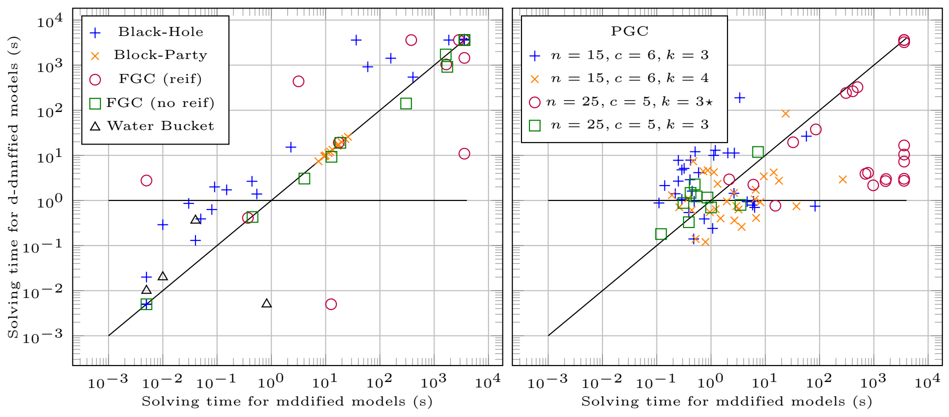


Fig. 18: Time to solve d-dnnffied instances, including compilation time (y-axis) vs. time to solve mddified instances, including compilation time (x-axis).

5.2.4 Using MDDs as Cost-MDDs

In this section we investigate the advantage that cost-MDDs can have for specific problems over other structures. To do so, we look at the Shift Scheduling and Concert Hall problems.

Shift Scheduling We used 8 instances solved with the 5 models described earlier. We ran the original models, mddified models and *cost-mddified* (using the same MDDs as cost-MDDs) as well as d-dnnffied models. For the sake of brevity, we only show geometric means of the ratios of conflicts, nodes and time of the compiled versions over the original version, and the total solving time. The first 5 rows of both Tables 4a and 4b correspond to the use of MDDs constructed while all constraints were active (i.e. a quasi-projection).

For the last 5 rows only the constraints ensuring the demand of workers is met were active upon mddifying. Columns labeled "Time" correspond to solving time; "Total" shows compilation plus solving time.

As can be seen, the use of the Cost-MDD completely dominates the other options. The total solving time shown in 4b is enormously decreased, making it worth paying the overhead of constructing the MDDs. For 5 instances where ShiftWRg performed extremely well (solved in $\leq 3s$), cost-mddifying was not worth it (c.f. geometric means of "Total"), but it still paid-off when measuring the time to solve all instances. This indicates, once again, that for individual instances on which a given model performs well, this technique might not be so valuable. The Cost-MDD version always solved all the instances, whereas the other two versions failed to solve between 1 and 3 of them (depending on the model). Comparing the first and last 5 rows of Tables 4a and 4b shows that building MDDs for the quasi-projection reduces the solving time, although computing those MDDs is more costly as there are fewer equivalences. Tablifying this problem produced tables with more than a billion entries, making them impractical.

We believe this result to be very interesting. As we saw in the presentation of the models we used earlier, the ShiftWRg uses a cost-MDD to enforce the shape

Model	MDD				Cost-MDD				d-DNNF			
	Confl.	Nodes	Time	Total	Confl.	Nodes	Time	Total	Confl.	Nodes	Time	Total
ShiftReg	87.4	87.3	97.9	146.7	**0.6**	**0.7**	**1.3**	**11.2**	95.7	95.7	116.2	441.6
ShiftDec	100.3	100.3	93.5	139.2	**1.2**	**1.4**	**1.1**	**8.2**	54.2	54.6	421.6	905.2
ShiftNNF	90.3	83.7	96.5	144.5	**2.7**	**3.5**	**4.7**	**31.0**	88.9	88.72	108.1	613.7
ShiftGCC	93.2	84.2	92.5	134.8	**2.5**	**3.2**	**4.0**	**26.8**	95.4	95.2	113.7	564.2
ShiftWRg	52.1	64.9	68.1	1052.9	**12.9**	**14.8**	**18.1**	674.9	57.9	77.6	103.5	13931.8
ShiftReg	97.6	97.4	103.7	169.4	**1.1**	**1.3**	**2.2**	**10.2**	0.3	0.4	0.9	64.0
ShiftDec	98.1	98.1	93.6	151.4	**2.2**	**2.7**	**2.2**	**9.1**	0.7	0.7	7.3	203.3
ShiftNNF	97.1	96.1	104.8	155.5	**3.9**	**5.5**	**6.7**	**20.1**	47.3	49.57	65.6	210.3
ShiftGCC	93.5	93.2	99.8	146.4	**3.6**	**5.2**	**6.2**	**18.0**	3.8	5.15	6.8	123.7
ShiftWRg	58.7	76.0	74.3	619.6	**16.4**	**18.9**	**29.7**	305.7	91.3	102.6	127.1	2886.0

(a) Geometric mean (%) when comparing to original models (> 100% indicates that the original model is better)

Model	Original	MDD		Cost-MDD		d-DNNF	
		Time	Total	Time	Total	Time	Total
ShiftReg	9624.4	10617.6	11480.3	285.5	**1148.2**	10320.3	42732.0
ShiftDec	13160.1	12969.7	13832.3	522.5	**1385.2**	19169.8	51581.5
ShiftNNF	7657	7785.7	8648.3	101.9	**964.5**	7707.6	40119.3
ShiftGCC	7897.9	7910.1	8772.8	93.7	**956.3**	7913.9	40325.6
ShiftWRg	3723.5	3726.3	4589.0	29.2	**891.9**	4526.2	36937.8
ShiftReg	9624.4	11274.9	11360.5	549.7	**635.3**	4343.0	4980.0
ShiftDec	13160.1	13164.6	13250.2	1217.3	**1302.9**	14470.0	15107.0
ShiftNNF	7657	7687.4	7773	117.9	**203.6**	7648.2	8285.3
ShiftGCC	7897.9	7817.9	7903.5	126.5	**212.1**	3658.7	4295.7
ShiftWRg	3723.5	3725.0	3810.6	82.5	**168.1**	3915.9	4553.0

(b) Sum of time to solve all instances in seconds.

Table 4: Comparison of conflicts, nodes and time for the shift scheduling problem.

of a shift. That is, there are already cost-MDDs present in that model, but one per worker. Thanks to our approach, we could create (cost-)MDDs for sets of days, across all workers. This is not trivial to do by hand, and it is not obvious that it would produce such a big advantage as it does, so a modeler might decide to not make that effort. Thanks to our approach, it was possible to build those MDDs and we show that they have a huge value for the model when used as cost-MDDs.

Concert Hall To further confirm our findings, we used the Concert Hall Problem described earlier as another benchmark for cost-MDDs. Table 5 presents the results of such experiments. As it can be seen in the results, the use of MDDs or d-DNNFs does not pay off, similarly to the Shift Scheduling problem. Nonetheless, these experiments confirm our findings about cost-MDDs: they are about 11 times faster than the original model and have 22 times less nodes to explore.

This, once again, shows the great value of cost-MDDs and the precompilation technique shown in this paper. Furthermore, the compilation itself was almost immediate, averaging 0.5s on each instance (for all 4 MDDs), and 1.3s for the compilation of the d-DNNFs (for each instance).

It would be possible that this finding also translates to a weighted version of d-DNNFs, but sadly we do not have such a propagator with explanations available in Chuffed. But this gives a very interesting direction for future work.

6 Conclusions and Future Work

Our experiments and those by Dekker et al [21] show that compiling part of a problem to an MDD, d-DNNF or table can be beneficial for the total solving

Instance	Original			MDD			Cost-MDD			d-DNNF		
	Confl.	Nodes	Time	Confl.	Nodes	Time	Confl.	Nodes	Time	Confl.	Nodes	Time
concert-01	18.9	21.7	2471.2	21.0	23.8	2746.8	0.79	0.84	201.3	22.0	25.1	2975.8
concert-02	21.1	24.5	2700.5	23.9	27.6	3255.4	1.52	1.71	368.5	21.1	24.6	2852.4
concert-03	28.4	31.8	3116.7	31.0	35.8	3600.4	1.64	1.86	347.5	31.3	35.6	3600.3
concert-04	11.5	13.6	1592.4	12.2	14.6	1447.6	0.62	0.66	187.2	8.19	9.79	1090.3
concert-05	8.35	9.86	1037.1	8.86	10.6	1100.9	0.61	0.70	139.4	9.96	11.7	1266.1
concert-06	32.8	35.7	4135.8	31.3	35.5	3653.2	1.66	1.82	375.9	31.3	34.8	3600.4
concert-07	14.3	17.0	1765.6	23.6	27.6	2981.4	0.33	0.38	69.8	17.4	20.5	2150.4
concert-08	24.5	28.8	3092.4	21.1	24.9	2721.1	2.20	2.34	668.8	21.0	24.8	2737
concert-09	14.5	16.9	1867.0	14.9	17.3	1939.3	0.30	0.34	65.5	13.0	15.2	1708.0
concert-10	2.47	2.99	307.8	1.03	1.21	123.9	0.36	0.40	85.2	1.57	1.90	197.5
concert-11	6.89	7.97	839.8	6.91	7.89	888.2	0.27	0.29	69.0	5.90	7.00	775.3
concert-12	15.4	17.9	1961.1	13.4	15.5	1729.3	1.94	2.07	527.3	9.06	10.5	1187.1
concert-13	27.2	30.5	3600.8	27.1	30.7	3600.9	0.40	0.45	81.5	27.4	31.3	3600.3
concert-14	25.8	30.9	3106.9	29.6	35.0	3600.4	0.36	0.40	76.0	27.5	31.9	3600.6
concert-15	4.41	5.24	524.1	8.76	10.1	1123.8	0.49	0.51	117.3	4.49	5.26	577.1
concert-16	16.2	18.8	2202.1	8.84	10.3	1167.8	0.54	0.58	134.9	24.1	27.5	3349.8
concert-17	14.3	16.2	1980.4	15.6	18.1	2193.4	0.46	0.53	106.8	15.9	18.3	2258.2
concert-18	6.85	8.12	926.1	5.41	6.55	729.3	0.90	1.00	246.8	6.43	7.90	859.8
concert-19	2.98	3.90	346.4	4.24	5.37	505.6	0.21	0.22	46.9	3.27	4.28	407.2
concert-20	19.6	23.0	2590.9	15.7	18.6	2064.5	0.28	0.31	52.8	14.8	17.2	1962.0
Average	15.8	18.3	2008.3	1.62	18.8	2058.7	0.79	0.87	198.4	1.58	18.2	2037.8
Total	317	365	40165.9	324	377	41174.1	15.9	17.5	3969.0	315	365	40756.6

Table 5: Number of conflicts (in millions), nodes (in millions) and solving time (in seconds) for the Concert Hall Problem with different precompilation techniques.

time of the problem. Indeed, these compilations into some data structures can be used by specific propagators (table, MDD and d-DNNF propagators) to produce stronger propagation, thus reducing the search space. Using our choice of data structures is not always better than using tables (there is certainly a role for tablification), but can be substantially more efficient.

We have identified the limitations of using tables as the resource for compilation, and proposed the use of precompilation with MDDs and d-DNNFs. We show how this technique is flexible, since the MDDs can be reused as Cost-MDDs, as well as robust. Indeed, our approaches never ran out of memory, and the compilation times are better than with tables. One downside of our approach is the amount of work needed in the solver. On the one hand, it is necessary to have an MDD propagator, a cost-MDD propagator, or a d-DNNF propagator for the precompilation to be used. On the other hand, in order to implement a precompiler like the one described here, one needs to implement the caching technique by Chu et al [13] which, for most constraints, is easy to implement (c.f. Section 6 of their paper) but for more complex constraints like `cumulative` can be more intricate. These keys must exist for all constraints that appear in the model provided by the user.

The result shows that for small problems, building tables is the right choice. For bigger problems, MDDs offer a scalable approach that, combined with a good MDD propagator, can be beneficial to the solving time. For the biggest instances, we saw that d-DNNFs are a better choice, as they are generally smaller. Cost-MDDs showed their value through two scheduling instances. We showed that, when it is possible to integrate an objective function in a cost-MDD, the improvements in time are substantial.

These results open questions for more applications. We believe it would be very valuable for the community to see more real applications of this prototypes on industrial problems with real data.

We believe it would be interesting to explore the possibility of using the same problem splitting from Section 4.4 with MDDs. When doing it for d-DNNFs, we can take advantage of the inherent data-structure of d-DNNFs to conjoin independent subproblems. It would also be worth investigating the possibility of using the same splitting to separate a problem in different MDDs. It is rather obvious how this can be achieved if a split is detected at the root level, but more intricate algorithms would be needed if the split appears in some branch during the construction of the MDD.

It would also be worthwhile to investigate the possibility of automatically detecting which variables can benefit from these precompilation techniques (for tables, MDDs or d-DNNFs) and, in the case of MDDs, which variable ordering could be the most adequate. Techniques similar to the ones used to pick solvers in portfolio solvers may be an approach to this [42, 46, 59].

References

1. Andersen HR, Hadzic T, Hooker JN, Tiedemann P (2007) A constraint store based on multivalued decision diagrams. In: International Conference on Principles and Practice of Constraint Programming, Springer, pp 118–132
2. Bergman D, Cire AA (2016) Decomposition based on decision diagrams. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, Springer, pp 45–54
3. Bergman D, van Hoeve WJ, Hooker JN (2011) Manipulating MDD relaxations for combinatorial optimization. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, Springer, pp 20–35
4. Bergman D, Cire AA, Hoeve WJv, Hooker JN (2013) Optimization bounds from binary decision diagrams. INFORMS Journal on Computing 26(2):253–268
5. Bergman D, Cire AA, van Hoeve WJ (2015) Improved constraint propagation via lagrangian decomposition. In: International Conference on Principles and Practice of Constraint Programming, Springer, pp 30–38
6. Bergman D, Cire AA, van Hoeve WJ, Hooker J (2016) Decision diagrams for optimization. Springer
7. Bergman D, Cire AA, van Hoeve WJ, Hooker JN (2016) Discrete optimization with decision diagrams. INFORMS Journal on Computing 28(1):47–66
8. Van den Broeck G, Taghipour N, Meert W, Davis J, De Raedt L (2011) Lifted probabilistic inference by first-order knowledge compilation. In: Proceedings of International Joint Conference on Artificial Intelligence, AAAI Press/International Joint Conferences on Artificial Intelligence, pp 2178–2185
9. Cheng KC, HC YR (2008) Maintaining generalized arc consistency on ad hoc r-ary constraints. In: International Conference on Principles and Practice of Constraint Programming, Springer, pp 509–523
10. Cheng KC, Yap RH (2005) Constrained decision diagrams. In: Proceedings of the National Conference on Artificial Intelligence, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, vol 20, p 366

11. Cheng KC, Yap RH (2010) An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. Constraints 15(2):265–304
12. Chu G, Stuckey PJ (2016) Lagrangian decomposition via subproblem search. In: Quimper CG (ed) International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, no. 9676 in LNCS, pp 65–80
13. Chu G, De La Banda MG, Stuckey PJ (2012) Exploiting subproblem dominance in constraint programming. Constraints 17(1):1–38
14. Chu GG (2011) Improving combinatorial optimization. PhD thesis, The University of Melbourne
15. Cire AA, van Hoeve WJ (2013) Multivalued decision diagrams for sequencing problems. Operations Research 61(6):1411–1428
16. Cocke J (1970) Global common subexpression elimination. In: ACM Sigplan Notices, ACM, vol 5, pp 20–24
17. Darwiche A (2002) A compiler for deterministic, decomposable negation normal form. In: Proceedings of the National Conference on Artificial Intelligence, AAAI Press, pp 627–634
18. Davies J, Bacchus F (2007) Using more reasoning to improve #SAT solving. In: Proceedings of the National Conference on Artificial Intelligence, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, vol 22, p 185
19. De Uña D, Gange G, Schachte P, Stuckey PJ (2016) Steiner tree problems with side constraints using constraint programming. In: Proceedings of the National Conference on Artificial Intelligence, AAAI Press, pp 3383–3389
20. Dekker JJ (2016) Sub-Problem Pre-Solving in MiniZinc, master's thesis. Master's thesis, Uppsala Universitet
21. Dekker JJ, Björdal G, Carlsson M, Flener P, Monette JN (2017) Auto-tabling for subproblem presolving in MiniZinc. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, vol 22, Springer, pp 512–529
22. Demassey S, Pesant G, Rousseau LM (2006) A cost-regular based hybrid column generation approach. Constraints 11(4):315–333
23. Eén N, Biere A (2005) Effective preprocessing in SAT through variable and clause elimination. In: International conference on theory and applications of satisfiability testing, Springer, pp 61–75
24. Erdös P, Rényi A (1959) On random graphs, I. Publicationes Mathematicae (Debrecen) 6:290–297
25. Fages JG, Lorca X, Petit T (2014) Self-decomposable global constraints. In: Proceedings of the European Conference on Artificial Intelligence, pp 297–302
26. Frisch AM, Harvey W, Jefferson C, Martínez-Hernández B, Miguel I (2008) Essence: A constraint language for specifying combinatorial problems. Constraints 13(3):268–306, DOI 10.1007/s10601-008-9047-y, URL http://dx.doi.org/10.1007/s10601-008-9047-y
27. Gange G, Stuckey PJ (2012) Explaining propagators for s-DNNF circuits. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, Springer, pp 195–210

28. Gange G, Stuckey PJ, Szymanek R (2011) MDD propagators with explanation. Constraints 16(4):407–429
29. Gange G, Stuckey PJ, Van Hentenryck P (2013) Explaining propagators for edge-valued decision diagrams. In: International Conference on Principles and Practice of Constraint Programming, Springer, pp 340–355
30. Gent IP, Jefferson C, Kelsey T, Lynce I, Miguel I, Nightingale P, Smith BM, Tarim SA (2007) Search in the patience game black hole. AI Communications 20(3):211–226
31. Hadzic T, Hooker JN, OSullivan B, Tiedemann P (2008) Approximate compilation of constraints into multivalued decision diagrams. In: International Conference on Principles and Practice of Constraint Programming, Springer, pp 448–462
32. Hoda S, Van Hoeve WJ, Hooker JN (2010) A systematic approach to MDD-based constraint programming. In: International Conference on Principles and Practice of Constraint Programming, Springer, pp 266–280
33. Huang J, Darwiche A (2005) DPLL with a trace: From SAT to knowledge compilation. In: Proceedings of the International Joint Conference on Artificial Intelligence, vol 5, pp 156–162
34. Jha A, Suciu D (2013) Knowledge compilation meets database theory: compiling queries to decision diagrams. Theory of Computing Systems 52(3):403–440
35. Jung JC, Barahona P, Katsirelos G, Walsh T (2008) Two encodings of DNNF theories. In: ECAI workshop on Inference methods based on Graphical Structures of Knowledge
36. Kell B, van Hoeve WJ (2013) An MDD approach to multidimensional bin packing. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, Springer, pp 128–143
37. Koch T, Martin A, Voß S (2000) SteinLib: An updated library on Steiner tree problems in graphs. Tech. Rep. ZIB-Report 00-37, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustr. 7, Berlin, URL http://elib.zib.de/steinlib
38. Koriche F, Lagniez JM, Marquis P, Thomas S (2015) Compiling constraint networks into multivalued decomposable decision graphs. In: Proceedings of the International Joint Conference on Aritificial Intelligence, pp 332–338
39. Latour AL, Babaki B, Dries A, Kimmig A, Van den Broeck G, Nijssen S (2017) Combining stochastic constraint optimization and probabilistic programming. In: International Conference on Principles and Practice of Constraint Programming, Springer, pp 495–511
40. Leo K, Tack G (2015) Multi-pass high-level presolving. In: Proceedings of the International Joint Conference on Aritificial Intelligence, pp 346–352
41. Leo K, Mears C, Tack G, de la Banda MG (2013) Globalizing constraint models. In: Schulte C (ed) International Conference on Principles and Practice of Constraint Programming, Springer, Lecture Notes in Computer Science, vol 8124, pp 432–447
42. Loreggia A, Malitsky Y, Samulowitz H, Saraswat VA (2016) Deep learning for algorithm portfolios. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA., pp 1280–1286, URL http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12274

43. Mears C, de la Banda MG, Wallace M (2009) On implementing symmetry detection. Constraints 14(4):443–477
44. Muise C, McIlraith S, Beck JC, Hsu E (2010) Fast d-DNNF compilation with sharpSAT. In: Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence
45. Nethercote N, Stuckey P, Becket R, Brand S, Duck G, Tack G (2007) MiniZinc: Towards a standard CP modelling language. In: International Conference on Principles and Practice of Constraint Programming, Springer-Verlag, LNCS, vol 4741, pp 529–543
46. OMahony E, Hebrard E, Holland A, Nugent C, OSullivan B (2008) Using case-based reasoning in an algorithm portfolio for constraint solving. In: Irish Conference on Artificial Intelligence and Cognitive Science, pp 210–216
47. Parlett D (1980) The Penguin Book of Patience. Penguin Books
48. Perez G, Régin JC (2014) Improving GAC-4 for table and MDD constraints. In: International Conference on Principles and Practice of Constraint Programming, Springer, pp 606–621
49. Perez G, Régin JC (2016) Constructions and in-place operations for MDDs based constraints. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, Springer, pp 279–293
50. Perez G, Régin JC (2017) MDDs: Sampling and probability constraints. In: International Conference on Principles and Practice of Constraint Programming, Springer, pp 226–242
51. Perez G, Régin JC, Antipolis U, Umr I (2015) Efficient operations on mdds for building constraint programming models. In: Proceedings of International Joint Conference on Artificial Intelligence, pp 374–380
52. Pesant G (2004) A regular language membership constraint for finite sequences of variables. In: Wallace M (ed) International Conference on Principles and Practice of Constraint Programming, Springer-Verlag, LNCS, vol 3258, pp 482–495
53. Pesant G (2005) Counting solutions of CSPs: A structural approach. In: Proceedings of the International Joint Conference of Artificial Intelligence, pp 260–265
54. Puget JF (2005) Automatic detection of variable and value symmetries. In: International Conference on Principles and Practice of Constraint Programming, Springer, pp 475–489
55. Régin JC (1996) Generalized arc consistency for global cardinality constraint. In: Proceedings of the National Conference on Artificial Intelligence, AAAI Press, pp 209–215
56. Sang T, Bacchus F, Beame P, Kautz HA, Pitassi T (2004) Combining component caching and clause learning for effective model counting. SAT 4:7th
57. Srinivasan A, Ham T, Malik S, Brayton RK (1990) Algorithms for discrete function manipulation. In: Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on, IEEE, pp 92–95
58. Van Hentenryck P (1999) The OPL optimization programming language. MIT Press
59. Xu L, Hutter F, Hoos HH, Leyton-Brown K (2008) Satzilla: Portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research (JAIR) 32:565–606