Constraint Programming with Multi-valued Decision Diagrams: A Saturation Approach

Vince Molnár, István Majzik
Budapest University of Technologies and Economics, Department of Measurement and Information Systems
Email: {molnarv, majzik}@mit.bme.hu

***Abstract*—Constraint programming is a declarative way of modeling and solving optimization and satisfiability problems over finite domains. Traditional solvers use search-based strategies enhanced with various optimizations to reduce the search space. One of such techniques involves multi-valued decision diagrams (MDD) to maintain a superset of potential solutions, gradually discarding combinations of values that fail to satisfy some constraint. Instead of the relaxed MDDs representing a superset, we propose to use exact MDDs to compute the set of solutions directly without search, compactly encoding all the solutions instead of enumerating them. Our solution relies on the main idea of the saturation algorithm used in model checking to reduce the required computational cost. Preliminary results show that this strategy can keep the size of intermediate MDDs small during the computation.**

I. Introduction

Many problems in computer science such as operations research, test generation or error propagation analysis can be reduced to finding at least one (optimal) assignment for a set of variables that satisfies a set of constraints, a problem called *constraint programming* (CP) [9]. CP solvers usually use a search-based strategy to find an appropriate solution, enhanced with various heuristics to reduce the search space.

Multi-valued decision diagrams (MDD) are graph-based representations of functions over tuples of variables with a finite domain [7]. As such, they can be used to compactly represent sets of tuples by encoding their membership function. Set operations computed on MDDs then have a polynomial time complexity in the size of the diagram instead of the encoded elements [4].

One of the heuristics proposed for CP solvers use MDDs to maintain a superset of potential solutions, gradually shrinking the set by discarding tuples failing to satisfy some constraint [1]. These approaches limit the size of MDDs to sacrifice precision for computational cost, which is compensated for by the search strategy. Using an exact representation of solution sets seems to be neglected by the community, except in [6] where special decision diagrams are used to achieve this.

This paper proposes to revisit the idea of exact MDD-based CP solvers, applying a strategy well-known in the model checking community: the saturation algorithm [5]. An efficient implementation of the idea could overcome a common limitation of search-based approaches, i. e., the complexity of computing *every* solution. As opposed to traditional search-based solvers, such a tool could natively compute the MDD representation of all the solutions instead of enumerating them one by one.

This work has been partially supported by the CECRIS project, FP7–Marie Curie (IAPP) number 324334. Special thanks to Prof. András Pataricza and Imre Kocsis for their motivation and support.

II. Preliminaries

A. Multi-valued Decision Diagram

Multi-valued decision diagrams (MDD) offer a compact representation for functions in the form of $\mathbb{N}^K \to \{0,1\}$ [7]. MDDs can be regarded as the extension of binary decision diagrams first introduced in [4]. By interpreting MDDs as membership functions, they can be used to efficiently store and manipulate sets of tuples. Definition of MDDs (based on [8]) and common variants are as follows.

Definition 1 (Multi-valued Decision Diagram) A *multi-valued decision diagram* (MDD) encoding the function $f(x_1, x_2, \dots, x_K)$ (where the domain of each x_i is $D_i \subset \mathcal{N}$) is a tuple $MDD = \langle N, r, \mathit{level}, \mathit{children}, \mathit{value} \rangle$, where:

- $N = \bigcup_{i=0}^{K} N_i$ is a finite set of *nodes*, where items of N_0 are *terminal* nodes, the rest ($N_{>0} = N \setminus N_0$) are *nonterminal* nodes;
- $\mathit{level} : N \to \{0, 1, \dots, K\}$ is a function assigning non-negative *level numbers* to each node ($N_i = \{n \in N \mid \mathit{level}(n) = i\}$);
- $r \in N$ is the *root node* of the MDD ($\mathit{level}(r) = K$);
- $\mathit{children}$: $N_i \times D_i \to N$ is a function defining edges between nodes labeled by elements of D_i, denoted by $n_k[i]$ (i. e., $\mathit{children}(n_k, i) = n_k[i]$);
- $(N, \mathit{children})$ as a directed graph is acyclic (DAG);
- $\mathit{value} : N_T \to \{0,1\}$ is a function assigning a *binary value* to each terminal node (therefore $N_0 = \{\mathbf{0}, \mathbf{1}\}$, where $\mathbf{0}$ is the terminal zero node ($value(\mathbf{0}) = 0$) and $\mathbf{1}$ is the terminal one node ($value(\mathbf{1}) = 1$).

An MDD is *ordered* iff for each node $n \in N$ and value $i \in D_{level(n)}$: $\mathit{level}(n) > \mathit{level}(n[i])$. An ordered MDD is *quasi-reduced* (QROMDD) iff the following holds: if $n \in N$ and $m \in N$ are on the same level and all their outgoing edges are the same, then $n = m$. An ordered MDD is *fully reduced* (ROMDD) if it is quasi-reduced and there is no node $n \in N$ such that every children of n are the same node.

The width of an MDD is the maximum number of nodes belonging to the same level: $w(MDD) = max_{1 \le i \le K}(|N_i|)$.

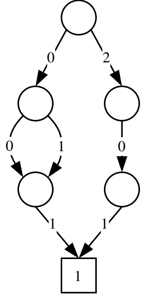

Fig. 1: An ordered MDD.

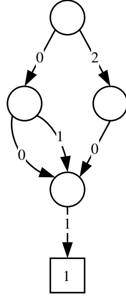

Fig. 2: A quasi-reduced MDD.

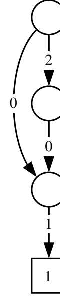

Fig. 3: A fully-reduced MDD.

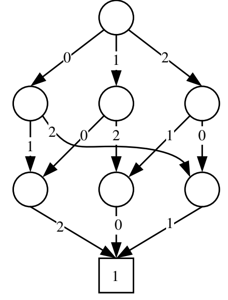

Fig. 4: MDD representation of ALLDIFFERENT(x_1, x_2, x_3).

The height of an MDD is the highest level to which any node belongs: $h(MDD) = max_{n \in N}(level)$. Note that only ROMDDs can have a lower height than K.

The semantics of a *quasi-reduced* MDD rooted in node r in terms of the encoded function f is the following: $f(v_1, v_2, \ldots v_K) = \mathit{value}((((r[v_K])[v_{K-1}]) \cdots)[v_1])$. When interpreted as a set, the set of all tuples encoded in an MDD rooted in node r is $S(r) = \{\mathbf{v} \mid \mathit{value}((((r[v_K])[v_{K-1}]) \cdots)[v_1]) = \mathbf{1}\}$.

In case of ROMDDs, a reduced node is assumed to have all edges connected to the target of its incoming edge.

Figures 1–3 illustrate an ordered but not reduced, a quasi-reduced and an ROMDD respectively, both encoding the set of tuples $\{(0,0,0),(0,1,0),(0,0,1)\}$ over the domain $\{0,1,2\} \times \{0,1\} \times \{0,1\}$. For the sake of simplicity, the terminal **0** node is omitted in figures.

An advantage of decision diagrams is the ability to compute set operations such as union and intersection with polynomial time complexity in the number of nodes in the operands [4].

B. Constraint Programming

Constraint programming (CP) is a framework for modeling and solving continuous or discrete optimization or satisfiability problems over finite domains [9]. The main advantage of CP over the similar SAT or ILP problems is that it can handle arbitrary finite domains and virtually any type of constraints: they can be any relation over the set of defined variables. The subset of CP problems addressed in this paper is *constraint satisfaction problems* (CSP), where the goal is to find at least one tuple that satisfies all the defined constraints.

Definition 2 (Constraint satisfaction problem) A *constraint satisfaction problem* (CSP) is defined over a set of *variables* $X = \{x_1, \ldots, x_K\}$ with finite domains $D = \{D(x_1), \ldots, D(x_K)\}$ and set of arbitrary relations over the variables (called *constraints*) $C = \{c_i \mid c_i \in D(x_{i_1}) \times \cdots \times D(x_{i_k})\}$, where $S(c_i) = \{x_{i_1}, \ldots, x_{i_k}\} \subseteq X$ is the support of the constraint, i. e., the variables over which the relation is defined. The question is whether there exists a tuple $\mathbf{v} \in D(x_1) \times \cdots \times D(x_K)$ that satisfies all constraints in C.

Due to the rich modeling opportunities, solvers cannot exploit any uniform semantics of the constraints. CP solvers therefore employ a systematic search, supported by *propagation strategies* that transfer the knowledge inferred in a constraint to other constraints to reduce the search space [9]. *Constraint propagation* is the process of discarding as many potential solutions as possible before stepping in the search. One extremity is the explicit enumeration of every possible tuple, gradually shrinking the set by discarding those that violate some constraint. In this case, a search in the traditional sense is not necessary, as after the restrictions, every tuple is a valid solution to the problem. Since this approach is generally considered infeasible or not scalable, the CP literature proposed various relaxations that are still useful to reduce the search space with an acceptable cost. Note however, that most of these solutions are therefore limited when it comes to computing *every* solution of a problem.

1) Domain-based Constraint Propagation: The traditional constraint propagation approach for CSP solving is built around *domain stores* [9]. Domain stores maintain the current domain for every variable of the problem, propagating inferred knowledge by the means of *domain consistency* [9].

Definition 3 (Domain consistency) A constraint C is *domain consistent* with the current variable domains D if for every value $v \in D(x_i)$ of every variable $x_i \in X$, there exists a tuple $\mathbf{v}$ with $\mathbf{v}[i] = v$ that satisfies C.

A constraint C can be made domain consistent by discarding values from the domains that cannot be extended to a tuple that satisfies C. Constraint propagation then consists of making constraints domain consistent until every constraint is domain consistent. If any domain becomes empty, the problem is unsatisfiable. If every domain contains a single value only, a solution is found and returned. In any other case, the search strategy binds the value of a variable and repeats the process.

2) MDD-based Constraint Propagation: One weakness of domain-based constraint propagation approaches is the lack of interaction between variables, i. e., every subset of the Cartesian product of the domains is domain consistent. For example, in the case of the ALLDIFFERENT constraint, which demands that all the variables in the tuple should assume different values, domain consistency fails to express the connection

between values of the variables.

To address this, [1] introduced the notion of *MDD consistency* and enhanced the domain store with an *MDD store*. MDDs can efficiently encode the various interactions between variables (see Figure 4 for the MDD representation of the ALLDIFFERENT constraint for three variables). One or more MDDs can then be used to communicate the restrictions between different constraints.

Definition 4 (MDD consistency) A constraint C is *MDD consistent* with an MDD rooted in node r if every edge in $children$ belongs to at least one path leading from r to the terminal $\mathbf{1}$ representing a tuple $\mathbf{v} \in S(r)$ that satisfies C.

MDD-based constraint propagation approaches usually use limited-width MDDs to reduce the complexity of MDD operations at the cost of losing some information. As previously noted, the spurious solutions introduced by the relaxation are eliminated with a search strategy that will eventually concretize solutions to obtain an exact result.

Note that in this form, MDDs are used as a relaxed set of potential solutions, a superset of the actual solutions. Domain stores can be regarded as the special case when the width of the MDD is fixed in one [2]. In this case, every domain is represented by the edges starting from the node on the corresponding level and the MDD encodes the Cartesian product of the domains.

III. SATURATION-BASED CONSTRAINT PROPAGATION

As presented in Section II, the CP community have embraced limited-width MDDs as a means to enhance the traditional domain store for more efficient constraint propagation. However, the literature rarely mentions the possibility of using explicit MDD representations (with unlimited width) and MDD operations to compute the set of solutions directly without relaxations and searching. As it seems, researchers of the community consider this approach infeasible or not scalable, which can explain the lack of corresponding results.

This paper proposes to revisit the idea mentioned as an extremity in Section II-B, that is, enumerating every potential solution and discarding those that fail to satisfy some of the constraints. In this setting, *fully reduced* MDDs provide an efficient encoding as only the levels corresponding to variables in the support of a constraint will contain nodes. The set of all tuples, for example, is encoded simply by the terminal $\mathbf{1}$ node, as none of the variables are bound by any constraint.

Discarding invalid solutions is then equivalent to computing the intersection of the current set of potential solutions with the set of solutions of the next constraint c_i. The set of all actual solutions is therefore obtained by taking the intersection of all the MDDs representing every constraint in the problem:

$$\mathcal{S} = \bigcap_{c_i \in C} \mathcal{S}(c_i) \tag{1}$$

The usual pitfall in MDD-based set operations is that the decision diagrams tend to grow very large during computation. Such computations usually aim to reach a fixed point, thus the number of encoded tuples constantly rises or falls during the computation. *Denser* or *sparser* sets usually have a more compact MDD representation than those encoding around half of all the possible tuples, thus the final size of the decision diagram is usually in an acceptable range. Intermediate results, however, can be exponentially larger (multiple orders of magnitude in practice as shown in Section IV).

The symbolic model checking community uses an efficient strategy to combat this phenomenon: *saturation* [5].

A. The Saturation Approach

Originally, saturation has been proposed as an iteration strategy tailored to work with decision diagrams to perform least fixed point computation with the transition relation of state-based behavioral models for state space exploration [5]. In other words, its original purpose is to efficiently compute the reflexive transitive closure of a relation on a set of initial values (initial states), minimizing the size of intermediate decision diagrams during the computation.

The essential idea of saturation is to keep the intermediate decision diagrams as dense as possible by applying relations affecting only the lower levels first. Relations are therefore applied in the order of the highest level that they affect.

In the CSP setting, transition relations are replaced with constraints and instead of the reflexive transitive closure, the intersection of all constraints must be computed. Nevertheless, the idea of ordering the constraints by the "highest" variable they affect (highest in terms of the decision diagram level that encodes the variable) is applicable. Moreover, it should carry the same benefits since the number of currently encoded tuples is again monotonic during the computation – this time converging towards the empty set.

To formally describe the proposed solution, we have to assume that a variable ordering is given, i. e., there is a bijective function $l : X \to \{1, \ldots, K\}$. This function describes the relationship between the variables and the encoding MDD as well: every variable x_i is represented by the level $l(x_i)$. W.l.o.g., we will assume that $l(x_i) = i$, unless otherwise noted.

Definition 5 (Level of constraint) The *level* of a constraint c_i is $Top(c_i) = max\{l(x_i) \mid x_i \in S(c_i)\}$, i. e., the largest level number assigned to the variables in the support of the constraint. Let $C_i = \{c_j \mid Top(c_j) = i\}$ then denote the set of constraint belonging to level i.

Ordering the constraints by the assigned *Top* level modifies Equation 1 as follows:

$$\mathcal{S} = \Bigg(\bigg(\Big(\big(\underbrace{c_1^1 \cap \ldots}_{c_j^1 \in C_1}\big) \cap \underbrace{c_1^2 \cap \ldots}_{c_j^2 \in C_2}\Big) \cap \ldots\bigg) \underbrace{c_1^K \cap \ldots}_{c_j^K \in C_K}\Bigg) \tag{2}$$

B. Discussion

The strategy of the proposed saturation approach can be characterized by two goals:

- In every step, minimize the height of the resulting (fully-reduced) MDD.

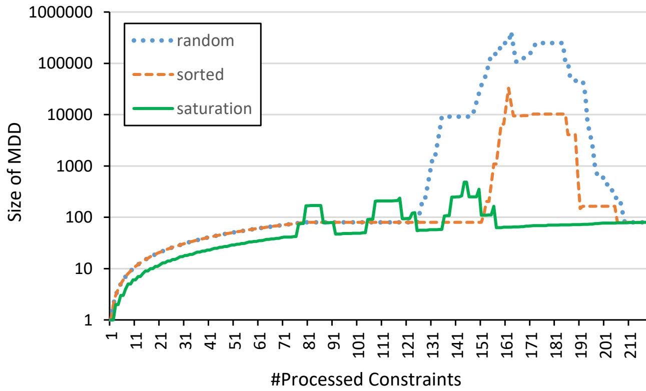

Fig. 5: MDD size during computation.

- If a new node n is introduced on a new level, minimize the number of tuples n encodes.

Both of these goals are accomplished with the ordering of constraints. The following lemmas provide the rationale.

Lemma 1 The width of an ROMDD cannot be higher than the number of tuples it encodes.

Proof *Assume there is an ROMDD with width w encoding $w-1$ tuples. Having a width w means there is at least one level with w nodes. Every node in any ROMDD has to be on a path leading from the root node to the terminal* **1***, so the MDD must encode at least w tuples, as opposed to $w-1$.*

Lemma 2 The number of nodes in an ROMDD cannot be higher than $w(MDD) \cdot h(MDD)$.

From the lemmas we can conclude that the saturation strategy aims to minimize the size of the resulting MDD in every step. Note, however, that the number of encoded tuples is not in a direct relationship with the width of the MDD. As stated before, the two extremities are the empty set and the universe, but in between the size can grow exponentially. The saturation strategy can be therefore considered as a "best effort" heuristic rather than an optimal algorithm.

IV. Results

The proposed approach has been implemented in Java as a CSP solver processing problems given in the XCSP3 format [3]. As a proof of concept, a small experiment has been carried out where a simple model encoding error propagation in a railway system has been solved by three different strategies.

The first "strategy" applied the constraints in the order of declaration in the problem definition (which can be considered more-or-less random). The second one orders the constraints by the number of variables supporting the constraint, while the third one is the proposed saturation approach. Figure 5 shows the size of the solution MDD after the processing of each constraint for the three strategies on a logarithmic scale.

The experiment demonstrates the potential benefits of using the saturation approach in an explicit MDD-based CSP solver. Compared to the "random" strategy, the peak size of the MDD was almost three orders of magnitude smaller with the saturation approach, at most 6 times more than the final size. Ordering the constraints by the number of supporting variables yields a better result than the random strategy, but it is still far worse than the saturation approach. The remaining peaks correspond to the inclusion of complex constraints when a new variable is processed and the size of the MDD usually normalizes before processing the next variable.

V. Conclusion and Future Work

This paper proposed to revisit a seemingly undiscussed topic of applying exact MDD-based methods to compute the solution set of finite-domain constraint programming problems. Although the approach of compiling the MDD representation of constraints and computing their intersection may seem "brute-force", it is worth exploring the solutions employed in related research areas such as symbolic model checking.

In this spirit, we have applied the strategy of the saturation algorithm well-known in the model checking community. Saturation orders the relations by the highest level assigned to one of their supporting variables, keeping the intermediate MDD relatively compact compared to other approaches.

We have demonstrated the benefits of the strategy in a small experiment, which provided promising results. Once a larger set of benchmark models are available after the first XCSP3 competition[1], we plan to systematically evaluate and fine-tune our solution to see if it can match the performance of traditional tools employing the domain- and MDD-store approaches. Regardless of the performance, it is noteworthy that the proposed solution can natively compute the set of *all solutions* in a compact representation, which poses a great challenge to traditional tools that can only enumerate the solutions one by one.

References

[1] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *Proc. of the 13th Internation Conference on Principles and Practice of Constraint Programming*, pages 118–132. Springer, 2007.

[2] D. Bergman, A. A. Cire, W. J. van Hoeve, and J. Hooker. *MDD-Based Constraint Programming*, pages 157–181. Springer, 2016.

[3] F. Boussemart, C. Lecoutre, and C. Piette. XCSP3: an integrated format for benchmarking combinatorial constrained problems. *CoRR*, abs/1611.03398, 2016.

[4] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[5] G. Ciardo, R. Marmorstein, and R. Siminiceanu. The saturation algorithm for symbolic state-space exploration. *International Journal on Software Tools for Technology Transfer*, 8(1):4–25, 2006.

[6] R. Mateescu and R. Dechter. Compiling constraint networks into AND/OR multi-valued decision diagrams (AOMDDs). In *Proc. of the 12th Int. Conf. on Principles and Practice of Constraint Programming*, pages 329–343, 2006.

[7] D. M. Miller and R. Drechsler. Implementing a multiple-valued decision diagram package. In *Proc. of the 28th IEEE Int. Symp. on Multiple-Valued Logic*, pages 52–57, 1998.

[8] V. Molnár, A. Vörös, D. Darvas, T. Bartha, and I. Majzik. Component-wise incremental LTL model checking. *Formal Aspects of Computing*, 28(3):345–379, 2016.

[9] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier, 2006.

[1]http://xcsp.org/competition