STAIRS 2012
K. Kersting and M. Toussaint (Eds.)

doi:10.3233/978-1-61499-096-3-216

Set-Labeled Diagrams for CSP Compilation

Alexandre NIVEAU [a], Hélène FARGIER [b] and Cédric PRALET [c]
[a] *CRIL–Université d'Artois, F-62307 Lens Cedex, France*
[b] *IRIT–Université Paul Sabatier, F-31062 Toulouse Cedex 9, France*
[c] *Onera—The French Aerospace Lab, F-31055, Toulouse, France*

Abstract. Knowledge compilation structures such as MDDs have been proposed as a way to compile CSPs, to make requests tractable online, in cases where solving is not possible. This paper studies the interest in relaxing two assumptions usually imposed on MDDs, static ordering and read-once property, using a new compilation structure called Set-labeled Diagrams, which are compiled by tracing the search tree explored by a CSP solver. The impact of read-once and static ordering is assessed by simply playing on the variable choice heuristics used during search in the CSP solver.

1. Introduction

Constraint Satisfaction Problems (CSPs) offer a powerful framework for representing a great variety of problems, e.g. planning or configuration problems. Different kinds of requests can be posted on a CSP, such as extraction of a solution (the most classical request), strong consistency of the domains, addition or retraction of new constraints (dynamic CSP), counting of the number of solutions, and even combinations of these requests. For instance, the interactive solving of a configuration problem amounts to a series of (unary) constraints additions and retractions while maintaining the strong consistency of the domains, i.e. each value in a domain is involved in at least one solution.

Most of these requests are NP-hard. They must however sometimes be addressed online. A possible way of solving this contradiction consists in representing the set of solutions of the CSP as a Multivalued Decision Diagram [1,2,3], i.e. as a graph whose nodes are labeled by variables and whose edges represent assignments of the variables. In such diagrams, each path from the root to the sink represents a solution of the CSP. They allow several operations, like those previously cited, to be achieved in time polynomial w.r.t. the size of the diagram. This size can theoretically be exponentially higher than the one of the original CSP, but it remains low in many applications. Indeed, as they are graphs, MDDs can take advantage of the (conditional) interchangeability of values and save space by merging identical subproblems. As a matter of fact, decision diagrams have been used in various contexts, e.g. in product configuration [4], in recommender systems [5], or, in their original Boolean form, in planning [6,7] and diagnosis [8].

Up to our knowledge, these works always consider *read-once* and *ordered* graphs, that is, graphs such that variables are not repeated along a path, and such that the order in which variables are encountered along a path is fixed (x cannot appear before y in one path and after y in another path). However, this is not a requirement for many applications; as for the Boolean case, it has been shown [9] that OBDDs can often be advantageously replaced by FBDDs (free BDDs), that are read-once, but not ordered. In this paper, we use the language of Set-labeled Diagrams [10], which generalize MDDs by relaxing both requirements.

Note that we do not consider using MDDs to encode single constraints and use them for CSP-solving purposes. In this case, requests are mainly propagation. What we want is to compile the solution set of a whole CSP, in order to answer to a series of online requests.

The goal of this paper is to study a method of compilation of set-labeled diagrams, namely applying Huang and Darwiche's "DPLL with a trace" [11] to a CSP solver. We present a generic compilation algorithm for building set-labeled diagrams while benefiting from constraint programming techniques available in CP engines. With this compilation algorithm, relaxing the "ordered" or "read-once" assumptions depends on the choice of variable choice and branching heuristics to be used during the search by the CP engine. We study different possible heuristics and present experimental results, obtained with our implementation of the algorithm on the Choco solver [12].

The paper is organized as follows: We present in Section 2 the formal framework of Set-labeled Diagrams, that generalize MDDs. Then in Section 3, we describe our compilation algorithm. Section 4 contains details about the heuristics we used, and the results of our experiments.

2. Set-labeled Diagrams

2.1. Structure and Semantics

We first give a very general definition of set-labeled diagrams, and restrict it afterwards to a specific framework.

Definition 1 (Set-labeled diagram)**.** Let $\mathcal{V}$ be a set of variables, and let $\mathcal{E}$ be a set of sets. A *set-labeled diagram* (SD) is a directed acyclic graph with at most one root and at most one leaf (the *sink*). Non-leaf nodes are labeled by a variable of $\mathcal{V}$. Each edge is labeled by a set in $\mathcal{E}$.

This definition contains no requirements on nodes' and edges' labels. SDs thus generalize a number of structures representing solution sets, such as BDDs [13] (binary decision diagrams, using Boolean variables and $\mathcal{E} = \{\{\bot\}, \{\top\}\}$), MDDs [14,1,15,3] (multivalued decision diagrams, using discrete variables and $\mathcal{E}$ a set of singletons), and interval automata [16] and interval diagrams [17] (with $\mathcal{E}$ a set of intervals).

In the following, we restrict our framework to that of MDDs: variable domains are finite parts of $\mathbb{Z}$. However, contrary to MDDs, edges are not labeled with singletons but with finite sets of $\mathbb{Z}$. We introduced these restricted set-labeled

diagrams in a previous work [10]. Note that we consider *explicitly enumerated sets*; the use of sets does not allow us to save space (the point is actually to define a new structural restriction, namely focusingness). SDs are nonetheless more general than MDDs, notably because variable ordering and repetition along a path are not restricted, but also because they can be *non-deterministic*: label sets of edges going out of a given node need not be disjoint. We call deterministic SDs, dSDs.

Definition 1 allows the graph to be empty (no node at all) or to contain only one node (together root and sink). Figure 1 gives an example of an SD.

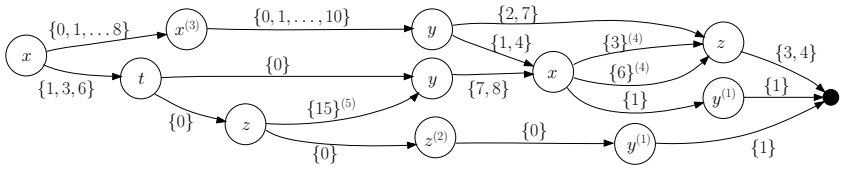

Figure 1. An example of non-reduced SD. Variable domains are all $\{0, 1, \ldots, 10\}$. As for reduction properties [10]: The two nodes marked $^{(1)}$ are isomorphic; node $^{(2)}$ is stammering; node $^{(3)}$ is undecisive; the edges marked $^{(4)}$ are contiguous; edge $^{(5)}$ is dead.

We use the following notation: For $x \in \mathcal{V}$, $\mathrm{Dom}(x)$ denotes the *domain* of x. We suppose that $\mathcal{V}$ is totally ordered, and that in a set denoted $X = \{x_1, \ldots, x_k\} \subseteq \mathcal{V}$, variables are sorted in ascending order. Then $\mathrm{Dom}(X)$ denotes $\mathrm{Dom}(x_1) \times \cdots \times \mathrm{Dom}(x_k)$, and $\vec{x}$ denotes an X-*assignment* of variables from X, i.e. $\vec{x} \in \mathrm{Dom}(X)$. Last, $\vec{x}_{|x_i}$ denotes the value assigned to x_i in $\vec{x}$. The cardinal of a set S is denoted $|S|$.

Let φ be a set-labeled diagram, N a node and E an edge in Γ; we denote $\mathrm{Var}(\varphi)$ the set of all variables mentioned in φ; $\mathrm{Root}(\varphi)$ the root of φ and $\mathrm{Sink}(\varphi)$ its sink; $\|\varphi\|$ the *size* of φ, i.e. the sum of the cardinalities of all labels in φ plus the cardinalities of the variables' domains; $\mathrm{Var}(N)$ the variable labeling N; $\mathrm{Lbl}(E)$ the set labeling E; and $\mathrm{Var}(E)$ the variable labeling the source of E.

An SD is a compact representation of a Boolean function over discrete variables. This function is the *interpretation* of the set-labeled diagram:

Definition 2 (Semantics of an SD)**.** Let φ be an SD, and $X = \mathrm{Var}(\varphi)$. The *interpretation of* φ is the function $[\![\varphi]\!]$, from $\mathrm{Dom}(X)$ onto $\{\bot, \top\}$, and defined as follows: for every X-assignment $\vec{x}$, $[\![\varphi]\!](\vec{x}) = \top$ if and only if there exists a path p from the root to the sink of φ such that for each edge E along p, $\vec{x}_{|\,\mathrm{Var}(E)} \in \mathrm{Lbl}(E)$.

We say that $\vec{x}$ is a *model* of φ whenever $[\![\varphi]\!](\vec{x}) = \top$. $\mathrm{Mod}(\varphi)$ denotes the set of models of φ. φ is said to be *consistent* if and only if $\mathrm{Mod}(\varphi) \neq \varnothing$.

Note that the interpretation function of the empty SD always returns $\bot$, since it contains no path from the root to the sink. Conversely, the interpretation function of the one-node SD always returns $\top$, since in the one-node SD, the only path from the root to the sink contains no edge.

Like BDDs, SDs can be reduced in size (potentially by an exponential factor) without changing their semantics, as addressed in a previous work [10]. This is done notably by the merging of isomorphic nodes, that is, nodes rooting identical

subgraphs. For space reasons, we do not detail reduction operations; they are nonetheless hinted in Figures 1 and 2.

Contrary to the case of MDDs, deciding whether an SD is consistent is not tractable [10]. One of the reasons is that the sets restricting a variable along a path can be disjoint, which implies that this path is not associated with any model. It is thus necessary to check each path of an SD before being able to decide of its consistency. To avoid this, a possibility is to consider SDs in which sets related to a given variable can only *shrink* along a path. We call this property *focusingness*; it is illustrated in Figure 2. Focusingness generalizes the "read-once" property defined for free BDDs: if no path in the SD uses a variable twice, the automaton is trivially focusing.

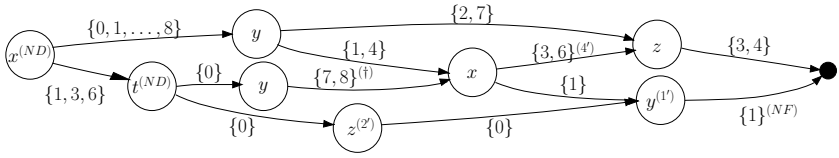

Figure 2. In this SD, all edges are focusing but the one marked $^{(NF)}$ (it is not included in the one marked $^{(\dagger)}$), and all nodes are deterministic but the ones marked $^{(ND)}$. This SD is the reduced form [10] of the SD presented in Figure 1: isomorphic nodes marked $^{(1)}$ have been merged into node $^{(1')}$, stammering node $^{(2)}$ has been collapsed into node $^{(2')}$, contiguous edges marked $^{(4)}$ have been merged into edge $^{(4')}$, and undecisive node $^{(3)}$ and dead edge $^{(5)}$ have been removed.

Definition 3 (Focusing and read-once SD)**.** A *focusing* edge in a set-labeled diagram φ is an edge E such that all edges E' on a path from the root of φ to the source of E such that $\mathrm{Var}(E) = \mathrm{Var}(E')$ verify $\mathrm{Lbl}(E) \subseteq \mathrm{Lbl}(E')$.

A *focusing* set-labeled diagram (FSD) is an SD containing only focusing edges.

A a set-labeled diagram φ is *read-once* (RSD) iff it contains no path p such that two nodes along p are labeled by the same variable.

We refer to as dFSD (resp. dRSD) an SD both deterministic and focusing (resp. read-once). Finally, we can impose an order on the variables encountered along the paths, and recover MDDs in their practical acceptation[1] [14,1,3].

Definition 4 (Ordered SD)**.** Let X be a set of variables and $<$ be a total order on X. An SD is said to be *ordered w.r.t.* $<$ iff for each couple of nodes (N, M) such that N is an ancestor of M, it holds that $\mathrm{Var}(N) < \mathrm{Var}(M)$.

A dSD ordered w.r.t. $<$ is called an $\mathtt{MDD}_<$. The language `MDD` is the union of all $\mathtt{MDD}_<$ languages.[2]

Before going on to the compilation section, let us stress here that further information about the SD family, including a knowledge compilation map (results about relative succinctness of languages and their support of various requests) can be found in a previous paper [10].

[1] The original definition of MDDs requires neither determinism nor ordering. Nevertheless, papers resorting to these structures work only with ordered and deterministic MDDs; that is why we abusively designate ordered dSDs as MDDs.

[2] A *language* is a set of graph structures, fitted up with an interpretation function. We denote `SD` the language of SDs, `dSD` the language of dSDs, and so on.

3. Compilation Algorithm: Choco with a Trace

3.1. State of the Art

Knowledge compilation is a domain that is mainly investigated from the theoretical point of view. A few compilers have been implemented, mainly in the case of Boolean domains, i.e., their inputs are Boolean functions over Boolean domains. Let us cite the OBDD packages (Buddy [18] and CUDD [19]) and the more recent DPLL with a trace proposed by [11]. The first series of packages compiles any elementary formula (and elementary constraint) as a (Boolean) decision diagram, and incrementally combines the resulting graphs through AND operations. It is always possible to go from multivalued domains to Boolean ones, using a Boolean encoding of the domains of the CSP [2]; nevertheless, it has been experimentally shown [20] that on real-world instances (namely configuration problems), MDDs are often smaller than log-BDDs.

The drawback of this kind of method is that it can generate intermediate data structures, that are space consuming and can even be exponentially larger than the final decision diagram. A second approach has been proposed with Huang and Darwiche's *DPLL with a trace* algorithm [11] for SAT compilation. The latter, which has proven very efficient in practice, builds decision diagrams (and d-DNNFs) by tracing the search tree of a DPLL algorithm enumerating all solutions of a CNF. This idea has been adapted in [21] to build approximate MDDs (i.e. MDDs whose model set is an approximation of the solution set of the input CSP) tracing a depth-first search algorithm. A similar technique is used in [22], where AND/OR MDDs (MDDs with AND nodes) are built following the trace of an AND/OR search.

All the preceding approaches use a predetermined variable order (in the case of AOMDDs, this is a tree order). We will relax this assumption here: the choice of the next variable to branch on can be done dynamically, depending on an heuristics. We present here a general description of our algorithm, that we implemented on top of the Choco CSP solver [12].

3.2. General description

Let $\mathcal{C} = (X, C)$ be a CSP defined by a set of constraints C over a set of variables X. Extending *DPLL with a trace* principles from the Boolean domain to the integer domain, we introduce mechanisms for building dFSDs by tracing the search tree of a CSP solver. The approach corresponds to Algorithm 1. It is implemented on top of the Choco CSP solver [12] and provides us with a *Choco with a trace* algorithm. Main function `SD-Builder(`$\mathcal{C}$`,` X_a`)` takes as input a constraint network $\mathcal{C}$ and the set of variables X_a currently assigned in $\mathcal{C}$. It returns a compiled representation of the set of solutions of CSP $\mathcal{C}$, in the form of a dFSD. The initial call is `SD-Builder(`$\mathcal{C}_0$`,` $\varnothing$`)` with $\mathcal{C}_0$ the initial CSP to be compiled.

3.2.1. Standard Search

The standard part of procedure `SD-Builder(`$\mathcal{C}$`,` X_a`)` is a generic depth-first search able to enumerate the set of all solutions of CSP $\mathcal{C}$. It behaves as fol-

lows: function `Propagate` first applies constraint propagation to the input constraint network $\mathcal{C}$, in order to remove inconsistent values from the variables' domain; if the reduced CSP obtained is consistent (no empty domain), (a) function `Choose_unassigned_var` selects an unassigned variable x, (b) function `Divide_domain` partitions the current domain of x into several non-empty disjoint subsets, and (c) the main search procedure is called successively on each of the subproblems defined by the partition. Any implementation of functions `Propagate`, `Divide_domain`, and `Choose_unassigned_var` available in existing CSP solvers can be used at that point.

Algorithm 1 `SD-Builder`$(\mathcal{C}, X_a)$: returns a set-labeled diagram that represents the solution set of constraint network $\mathcal{C}$. X_a is the set of currently assigned variables.

```
1: Propagate(C)
2: k := Compute_key(C, X_a)
3: if there is an entry for the key k in the cache then
4:    return the SD corresponding to key k in the cache
5: if C is proven inconsistent then
6:    return the empty graph
7: if X_a = X (all variables of C are assigned) then
8:    return the sink-only graph
9: Ψ := ∅
10: x := Choose_unassigned_var(C)
11: R := Divide_domain(C, x)
12: for all r ∈ R do
13:    X'_a ← X_a
14:    if r is reduced to a singleton then
15:       X'_a ← X'_a ∪ {x}
16:    let ψ_r := SD − Builder(C|_{Dom(x)←r}, X'_a)
17:    Ψ := Ψ ∪ {ψ_r}
18: let node N := Get_node(x, Ψ)
19: let φ be the graph rooted at N
20: store φ at the key k in the cache
21: return φ
```

3.2.2. Additions for Compilation

Some additions are made to the basic algorithm in order to build a dFSD representing the set of solutions of the initial CSP. These additions are framed in Algorithm 1. The basic idea is to compute a trace of the search tree using different mechanisms:

Internal node. Let n be an internal node in the search tree; this node is associated with a current CSP denoted $\mathcal{C}(n)$; let x be the unassigned variable chosen at node n and let R be the partition of $\mathrm{Dom}(x)$ computed to branch on the domain of x (one branch per element in the partition); the idea is that the exploration

associated with each subdomain $r \in R$ returns an SD ψ_r; this SD represents the set of solutions of subproblem $\mathcal{C}(n)|_{\mathrm{Dom}(x)\leftarrow r}$ obtained by reducing the domain of x to r; the set of solutions of CSP $\mathcal{C}(n)$ over $X \setminus X_a$ is then an SD $\varphi(n)$ whose root is labeled by x and which contains, for each $r \in R$ such that ψ_r is not the empty SD, an arc from the root to ψ_r.

In procedure `SD-Builder(`$\mathcal{C}$`,` X_a`)`, diagram $\varphi(n)$ is obtained via call `Get_node`$(x, \{\psi_r \mid r \in R\})$. In particular, function `Get_node` checks whether diagram $\varphi(n)$ (or a diagram isomorphic to it) already exists in the so-called *unique node table* [23]; this table contains all SD nodes created during search; if there exists an isomorphic diagram, it is directly returned; otherwise, node $\varphi(n)$ is created and added to the unique node table.

Leaf node. When n is a leaf node of the search tree, it corresponds either to a solution or to a dead-end; in the former case, the algorithm returns a sink-only SD (Line 8), to represent that any assignment of the current problem is a solution; in the latter, it returns an empty SD (Line 6).

Caching. The two previous points suffice to get a compiled dFSD representing the set of solutions of the initial CSP. We additionally maintain a cache during search to avoid equivalent subproblems to be re-explored. More precisely, each time a subproblem $\mathcal{C}$ is solved, a cache key $k(\mathcal{C})$ (function of the current domains of variables) together with the SD node produced for that subproblem are stored (Line 20). Then, prior to computing any new subproblem $\mathcal{C}'$, we compute its key $k(\mathcal{C}')$ and check whether it is already present in the cache (Lines 2–3); if yes, we directly return the SD associated with cache key $k(\mathcal{C}')$ (Line 4). The key is a list of the current domains of all variables that are either not assigned yet, or involved in constraints not yet satisfied in the current subproblem (sometimes called *universal* or *entailed* constraints). It has been proven [24] that all subproblems sharing such a key have the same solution set.

3.2.3. Structure of the SDs Obtained

The set-labeled diagrams returned by Algorithm 1 always satisfy the focusing property. Indeed, the variables' domains are systematically reduced, either by domain splitting or by constraint propagation. The set-labeled diagrams returned are also always deterministic, since function `Divide_domain` computes a partition, hence containing only disjoint subsets of the considered domain.

However, depending on the branching and variable-choice heuristics (i.e. on functions `Divide_domain` and `Choose_unassigned_var`), resulting dFSDs may differ:

- We obtain dRSDs if `Divide_domain` splits the domain into singletons, that is, if the algorithm enumerates each possible value during the search. On the contrary, a dichotomic branching search (splitting each current domain in two) will result in non read-once dFSDs.
- We obtain MDDs if `Choose_unassigned_var` follows a static ordering; but using heuristics to guide variable choice (like `MinDomain`) will often lead to non-ordered dFSDs.

3.3. Heuristics

We now detail the alternatives we considered for `Choose_unassigned_var`.

3.3.1. Standard Heuristics

We used the following standard CSP branching heuristics: **Min-domain**, which chooses the variable with the smallest domain; and **Dom/WDeg**, which choses the variable minimizing the ratio $|\operatorname{Dom}(x)|/\deg(x)$, where $\deg(x)$ is the number of constraints x is involved in (the constraints being weighted according to the conflicts) [25]. We also considered a **Random** heuristics, which chooses a random variable.

3.3.2. Constraint Graph–Based Heuristics

We used heuristics based on the constraint graph of the CSP, in which each variable is linked to another if and only if there exists a constraint involving them both. For a given variable x, we denote $\mathrm{N}(x)$ the set of variables that are linked with x in the graph.

Algorithm 2 `Next_var`(O) chooses the next variable, given a current order $O = \{o_1, \ldots, o_k\}$.

```
1: if O = ∅ then
2:     return Argmax_{x∈X} |N(x)|
3: let x := Argmax_{x∈X\O} H_S(x)
4: add x as the last value of order O
5: return x
```

The following heuristics are based on the scheme presented in Algorithm 2, varying the criterion $\mathcal{H}_S(x)$. They have been introduced in [26]. Intuitively, we try to group together variables that are strongly related, hoping that it could limit the number of edges. For **HBW**, $\mathcal{H}_S(x) = \max_{1 \le i \le |O|, o_i \in \mathrm{N}(x)} |O| - i$ (it chooses a neighbor of o_1 first, then of o_2, etc). For **HSBW**, $\mathcal{H}_S(x) = \sum_{1 \le i \le |O|, o_i \in \mathrm{N}(x)} |O| - i$ (it chooses a neighbor of the first chosen variables). Last, for **MCSInv**, $\mathcal{H}_S(x) = |\mathrm{N}(x) \cap O|$ (it chooses the variable most linked to those already chosen).

3.3.3. Cache-Based Heuristic

We implemented an heuristics aiming at maximizing the use of the cache. The idea is that a variable choice leading to already treated subproblems limits the number of new nodes. It is executed as follows: for each unassigned variable x, we count the number $n_{\mathrm{new}}(x)$ of branching values for which it will be necessary to open a new search node (that is, it does not lead to a cached subproblem or to an inconsistent one). We then choose the variable x minimizing $n_{\mathrm{new}}(x)$, using **HBW** to break ties. We call this heuristics **MaxHashUse**.

3.3.4. Static and Dynamic Versions

MinDom, **Dom/Wdeg** and **MaxHashUse** are always dynamic: they generally lead to non-ordered dFSDs. As for **HBW**, **HSBW**, and **MCSInv**, it is possible to compute a static order prior to the solving, and thus obtain MDDs, or to let the variable choice be dynamic, by using an up-to-date version of the contraint graph, in which constraints being entailed in the current problem are not considered. We call **DynHBW**, **DynHSBW**, and **DynMCSInv** the corresponding heuristics that do not compute a static order, and we also consider **DynRandom**.

4. Experiments

We considered the following problems in our experiments on our "Choco with a trace" compiler. *ObsToMem* is a reconfiguration problem; it represents a controller managing connections between the observation device and the mass memory of a satellite. *Drone* is a planning problem, in which a drone must achieve different goals on a number of zones in limited time. *NQueens* is the standard CSP representing the "n queens" problem. Last, *Star* is the CSP representing the problem of coloring a star graph (a center variable linked to other variables independant from each other).

Let us recall beforehand that heuristic efficiency is here estimated with respect to the size of the resulting graph (as defined in 2.1), and *not* to the time needed to find the first solution, as it is often the case in constraint programming. Thus, we do not expect classically "good" heuristics and domain-splitting functions to be particularly efficient.

4.1. Variable Choice Heuristics

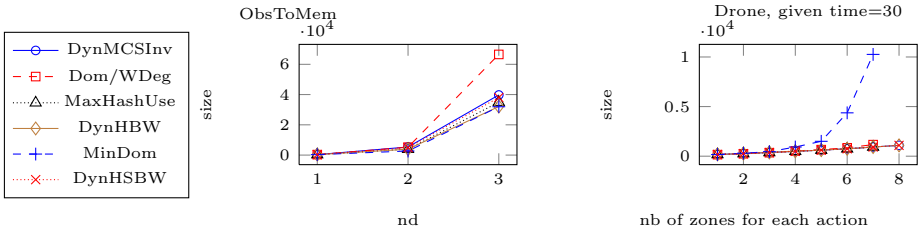

Figure 3. Comparison between the dynamic heuristics for *ObsToMem* and *Drone*.

We first compare the different variable choice heuristics, setting `Divide_domain` to split the domains into singletons, and thus always obtaining dRSDs. Since some of them have no static version, we only compare the dynamic ones. Results about *Drone* and *ObsToMem* can be found in Figure 3; for *NQueens* and *Star*, all heuristics gave very similar results.

Random is not included here (but is in the next section) to improve readability of the graphs, because it does far worse than the other heuristics.

It is interesting to notice that **MinDom** seems to be the best heuristics for *ObsToMem*, but is far worse than the others for *Drone*. **Dom/WDeg** is not really

interesting in any of our problems. Among the heuristics based on the variable graph, **HBW** seems to be the best. **MaxHashUse** is not bad, but does not outclass **HBW** (on which it is based); it seems that looking only one step ahead to maximize the use of the cache is not sufficient (we choose the best variable w.r.t. the next node, and open less new subgraphs; but these subgraphs are bigger).

4.2. Comparison between Static and Dynamic Orders

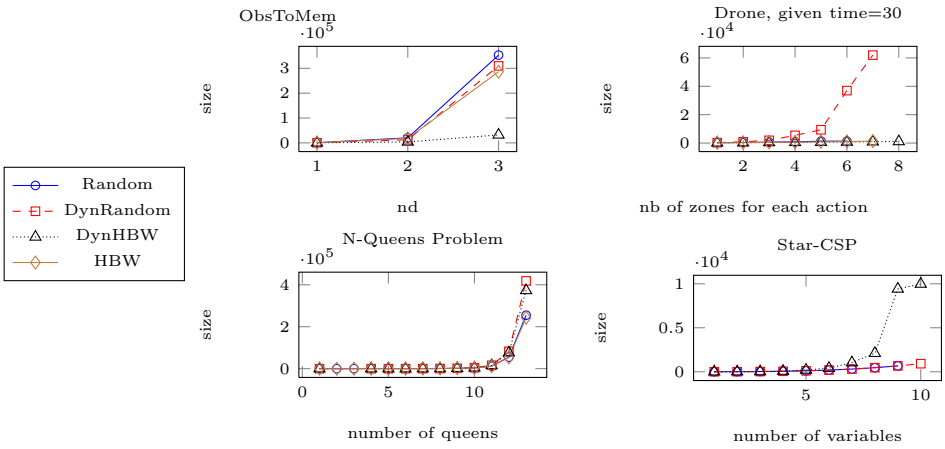

Figure 4. Comparison between the static and dynamic versions of **HBW** and **Random**. **DynRandom** results are not shown for *Star*, since they are far worse than the others (the size of the resulting dRSD exceeds 100,000 for 7 variables).

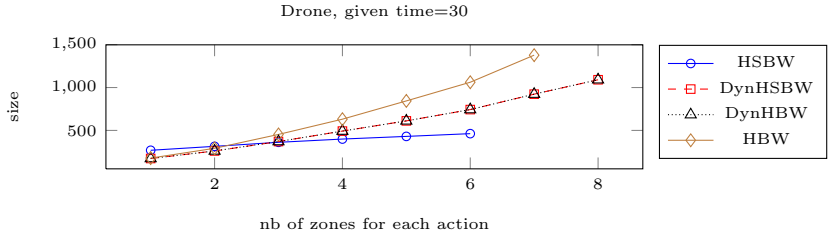

Figure 5. Comparison between the static and dynamic versions of **HBW** and **HSBW** for the *Drone* problem.

Let us now compare results obtained by using static and dynamic versions of a given heuristics, with the same domain splitting as in the previous section: we respectively obtain MDDs and dRSDs. Results for **HBW** (the best heuristics in the previous section) and **Random** can be found on Figure 4. We see that **DynRandom** is far worse than its static counterpart; indeed, using a static order, we increase the probability of getting isomorphic nodes. Results for **DynHBW** are better than static **HBW** for the real-world problems, but it is not the case for the smaller ones (for *NQueens*, the resulting MDDs are even smaller than the dRSDs).

However, this highly depends on the heuristics and on the problem considered; on Figure 5, we see that for the *Drone* problem, while **DynHBW** and **DynHSBW** coincide, their static versions are either better (**HSBW**) or worse (**HBW**).

4.3. Influence of the Domain-Splitting Function

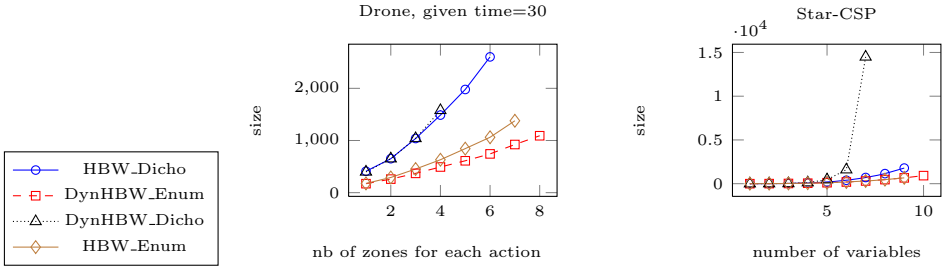

Figure 6. Comparison between the dichotomic and enumerating domain-splitting functions.

We now show to what extent the choice of the domain-splitting function affects the resulting graph. Using the enumerating domain-splitting function (that splits the domain into singletons) allows to compile dRSDs, whereas the dichotomic one (that splits the domain in two parts) allows to compile dFSDs. Results can be found on Figure 6; they are better in the first case. This does not imply that dRSDs are always smaller than equivalent dFSDs, but that our heuristics seems to be particularly adapted to the compilation of dRSDs. A method allowing to efficiently compile pure, non-read-once dFSDs is still to be found.

5. Conclusion

In this paper, we introduced set-labeled diagrams, that generalize MDDs by relaxing the properties of ordering, read-once and determinism. We presented an algorithm able to compile various kinds of deterministic SDs (dFSDs, dRSDs and MDDs), applying the idea of "DPLL with a trace" to a CSP solver. We showed how the choice of certain search parameters (variable-choice heuristics and domain-splitting function) affects the structure of the resulting dSD. Using our implementation of the compiler, based on the Choco CSP solver, on two real-world problems and two standard CSPs, we presented experimental results about the ifluence of these search parameters. Our results show that none of our variable-choice heuristics outclasses the others for all problems; a dominating heuristics is still to be found. They also show that our heuristics seem to be interesting only for compiling read-once diagrams.

Future work includes further study of variable-choice heuristics, especially of **MaxHashUse**. More generally, we want to think about some ways to efficiently compile non-read-once dFSDs and non-deterministic SDs. It should also be interesting to add "AND" nodes in the language, which would allow to compare static-order AOMDDs with dynamic-order ones.

References

[1] Vempaty, N.R.: Solving Constraint Satisfaction Problems Using Finite State Automata. In: AAAI. (1992) 453–458

[2] Kam, T., Villa, T., Brayton, R., Sangiovanni-Vincentelli, A.: Multi-valued Decision Diagrams: Theory and Applications. Multiple-Valued Logic **4**(1–2) (1998) 9–62

[3] Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemann, P.: A Constraint Store Based on Multivalued Decision Diagrams. In: CP. (2007) 118–132

[4] Amilhastre, J., Fargier, H., Marquis, P.: Consistency Restoration and Explanations in Dynamic CSPs — Application to Configuration. AIJ **135**(1–2) (2002) 199–234

[5] Cambazard, H., Hadzic, T., O'Sullivan, B.: Knowledge Compilation for Itemset Mining. In: ECAI. (2010) 1109–1110

[6] Giunchiglia, F., Traverso, P.: Planning as Model Checking. In: ECP. (1999) 1–20

[7] Hoey, J., St-Aubin, R., Hu, A.J., Boutilier, C.: SPUDD: Stochastic Planning Using Decision Diagrams. In: UAI. (1999) 279–288

[8] Torasso, P., Torta, G.: Model-Based Diagnosis Through OBDD Compilation: A Complexity Analysis. In: Reasoning, Action and Interaction in AI Theories and Systems. (2006) 287–305

[9] Bern, J., Gergov, J., Meinel, C., Slobodová, A.: Boolean manipulation with free bdd's. first experimental results. In: EDAC-ETC-EUROASIC. (1994) 200–207

[10] Niveau, A., Fargier, H., Pralet, C.: Representing CSPs with set-labeled diagrams: A compilation map. In: Proc. of the 2nd International Workshop on Graph Structures for Knowledge Representation and Reasoning (GKR). (2011)

[11] Huang, J., Darwiche, A.: DPLL with a Trace: From SAT to Knowledge Compilation. In: IJCAI. (2005) 156–162

[12] choco Team: choco: an open source java constraint programming library. Research report 10-02-INFO, Ecole des Mines de Nantes (2010)

[13] Bryant, R.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers **35**(8) (1986) 677–691

[14] Srinivasan, A., Ham, T., Malik, S., Brayton, R.: Algorithms for discrete function manipulation. In: ICCAD-90. (November 1990) 92 –95

[15] Amilhastre, J., Vilarem, P., Vilarem, M.C.: FA Minimisation Heuristics for a Class of Finite Languages. In: WIA. (1999) 1–12

[16] Niveau, A., Fargier, H., Pralet, C., Verfaillie, G.: Knowledge compilation using interval automata and applications to planning. In: ECAI. (2010) 459–464

[17] Strehl, K., Thiele, L.: Symbolic model checking of process networks using interval diagram techniques. In: Proc. of the 1998 IEEE/ACM international conference on Computer-aided design. (1998) 686–692

[18] Lind-Nielsen, J.: BuDDy : Binary Decision Diagrams Library Package, release 2.4 (2002) `http://sourceforge.net/projects/buddy/`.

[19] Somenzi, F.: CUDD : Colorado University Decision Diagram package, release 2.4.1 (2005) `http://vlsi.colorado.edu/~fabio/CUDD/`.

[20] Hadzic, T., Hansen, E., B. O'Sullivan, B.: On Automata, MDDs and BDDs in Constraint Satisfaction. In: ECAI Workshop on Inference methods based on Graphical Structures of Knowledge (WIGSK). (2008)

[21] Hadzic, T., Hooker, J.N., O'Sullivan, B., Tiedemann, P.: Approximate compilation of constraints into multivalued decision diagrams. In: CP. (2008) 448–462

[22] Mateescu, R., Dechter, R., Marinescu, R.: AND/OR multi-valued decision diagrams (AOMDDs) for graphical models. J. Artif. Intell. Res. (JAIR) **33** (2008) 465–519

[23] Wegener, I.: Branching Programs and Binary Decision Diagrams. SIAM (2000)

[24] Lecoutre, C., Sais, L., Tabary, S., Vidal, V.: Transposition tables for constraint satisfaction. In: AAAI. (2007) 243–248

[25] Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI. (2004) 146–150

[26] Amilhastre, J.: Représentation par automate d'ensemble de solutions de problèmes de satisfaction de contraintes. PhD thesis, Université Montpellier II (1999)