

Layer Compression in Decision Diagrams

Tarik Hadzic
University College Cork
t.hadzic@4c.ucc.ie

Esben Rune Hansen
IT University of Copenhagen
esben@itu.dk

Barry O’Sullivan
University College Cork
b.osullivan@cs.ucc.ie

Abstract

A number of compact representation forms that are investigated in the knowledge compilation community are utilized in interactive configuration and other forms of decision support. Among different representations multi-valued decision diagrams (MDDs) are particularly well suited for interactive configuration. However, for large variable domains MDDs can be unnecessarily large if many values are repeating on different edges. In this paper we suggest exploiting the repetitive occurrences of values through introduction of pseudo-nodes. The technique can be easily applied over MDDs as well as their more succinct counterpart, interval decision diagrams (IDDs). The compactness of the resulting representations, layered-compressed MDDs (lcMDDs) and layer-compressed IDDs (lcIDDs) is demonstrated empirically on artificial and real-world instances.

1 Introduction

A number of compact representation forms that are investigated in the knowledge compilation community [2] are utilized in interactive configuration and other forms of decision support [10, 3, 8]. If a set of all solutions is compiled into a compact representation (e.g. during some off-line processing prior to user interaction), users can explore the solution space in real-time in much richer ways since compiled representation supports efficient execution of otherwise intractable queries: valid domains computations, multi-dimensional projections, additive cost processing etc [7, 6, 5].

Among different representations *multi-valued decision diagrams* (MDDs) are demonstrated to be particularly well suited for configuration instances [4]. However, for large variable domains and constraints allowing for allot of value sharing, MDDs can be unnecessarily large. In this paper we try to remedy this by exploiting repetitive occurrences of values within variable domains. We propose an easy-to-implement compression scheme, based on introducing

pseudo-nodes in each MDD layer, to get *layer-compressed MDDs* (lcMDDs). The same technique can be readily applied over more succinct MDD variants, such as *interval decision diagrams* (IDDs) [9], resulting in *layer-compressed IDDs* (lcIDDs). The resulting structures are not decision diagrams in the standard sense, but fall within a wider category of graphical representations of knowledge in the form of directed acyclic graphs [2, 11]. Our experimental evaluation demonstrates that the resulting structures can be significantly smaller for a set of real-world and artificial instances.

The *layer-compressed* variants of MDDs and IDDs can be seen as one of the many representation forms investigated in the knowledge compilation area. Until recently, the bulk of research considered only structures representing Boolean functions [2]. Only recently, multi-valued generalizations of these structures, e.g. *multi-state directed acyclic graphs* (MDAGs) [11], have been considered in their own right. Namely, it was recognized that certain queries become cumbersome to implement over Boolean encodings of non-binary variables, and MDAGs can become significantly smaller for large variable domains.

The rest of the paper is organized as follows. In Section 2 we present the MDD background. In Section 3 we introduce our approach to layer compression. In Section 4 we present algorithms that achieve layer compression, and perform interactive configuration over compressed representation. In Section 5 we discuss compression over IDDs. In Section 6 we experimentally evaluate the performance of our scheme, and finally, in Section 7 we conclude and outline future work.

2 Background

A configuration model can be conveniently represented as a *constraint satisfaction problem* $CSP = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, which is defined over variables $\mathcal{X} = \{x_1, \dots, x_n\}$ with finite domains D_1, \dots, D_n . Each constraint $c \in \mathcal{C}$ is defined over a subset of variables $\text{scope}(c) \subseteq \mathcal{X}$, and defines the set of assignments to the variables in the $\text{scope}(c)$ allowed by the constraint. The solution space of $Sol(CSP)$ denotes the set of all solutions to the CSP CSP . For illustration

purposes we will consider a CSP in Example 1.

Example 1. Consider a CSP specifying which sets of components that can be assembled into a functioning computer. The CSP is on three variables x_1, x_2 and x_3 specifying the choice of motherboard (*mb*), processor (*cpu*) and graphics card (*gpu*) respectively. The domains are:

$$\begin{aligned} D_1 &= \{mb_1, mb_2, mb_3\} \\ D_2 &= \{cpu_1, \dots, cpu_5\} \\ D_3 &= \{gpu_1, gpu_2, gpu_3\} \end{aligned}$$

and the constraints of the CSP are:

$$\begin{aligned} c1: \quad &x_1 = mb_1 \implies x_2 \neq cpu_4 \wedge x_2 \neq cpu_5 \\ c2: \quad &x_1 = mb_3 \implies x_2 \neq cpu_4 \\ c3: \quad &x_2 = cpu_5 \implies x_1 = mb_3 \\ c4: \quad &x_2 \neq cpu_3 \\ c5: \quad &x_3 = gpu_2 \implies x_1 = mb_1 \wedge x_2 = cpu_4 \end{aligned}$$

In the reminder, values in variable domains are for simplicity identified with natural numbers. Hence, we will take $D_1 = \{1, 2, 3\}$, $D_2 = \{1, \dots, 5\}$, $D_3 = \{1, 2, 3\}$. \diamond

Definition 1 (MDD). A multi valued decision diagram is a rooted directed acyclic graph $G = (V, E)$ where every node u is labeled with a variable x_i and every edge e , originating from a node labeled x_i , is labeled with a value $a_i \in D_i$. No node may have more than one outgoing edge with the same label. The decision diagram contains a special terminal node $\mathbf{1}$, that has no outgoing edges. The terminal node has to be reachable by every other node in V . On any path from the root to the terminal $\mathbf{1}$ each node label x_i can appear at most once.

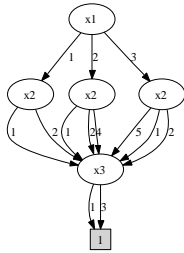


Figure 1. An MDD representing the CSP from Example 1. Each edge labeled with j , outgoing from a node labeled with x_i , corresponds to an assignment $x_i = j$. Each path represents a CSP solution.

The root of the MDD is designated as r . The mapping $var(u) \in \{1, \dots, n + 1\}$ denotes the index of a variable x_i labeling the non-terminal node u , and by definition $var(\mathbf{1}) = n + 1$.

An MDD is *ordered* if the variables labeling nodes in a path from the root to the terminal are in a same order. For a natural ordering x_1, \dots, x_n , for every edge (u, a, u') , it holds that $var(u) < var(u')$. In the rest of the paper, we always assume a fixed, lexicographical variable ordering $x_1 < \dots < x_n$.

An ordered MDD can be seen as organized in *node layers*, V_1, \dots, V_n , where $V_i = \{u \in V \mid var(u) = i\}$. It is useful to consider also *edge layers*, $E_i, i = 1, \dots, n$ of edges originating in V_i , i.e. where

$$E_i = \{(u, a, u') \mid var(u) = i\}.$$

In general MDD edges can skip variable layers but for the purpose of this paper we will consider MDDs where skipping of variable layers is not allowed, i.e. where for each $(u, a, u') \in E$, $var(u') = var(u) + 1$. In this case, our edge layers E_1, \dots, E_n are

$$E_i = \{(u, a, u') \mid var(u) = i, var(u') = i + 1\}.$$

Every path p_r , from the root r to the terminal $\mathbf{1}$ contains exactly n edges, $p_r = ((u_1, a_1, u_2), \dots, (u_n, a_n, \mathbf{1}))$, where $u_1 = r$. It represents a solution

$$Sol((u_1, a_1, u_2), \dots, (u_n, a_n, \mathbf{1})) = \{a_1\} \times \dots \times \{a_n\}.$$

In the same way, a path $p_{u_i} = ((u_i, a_i, u_{i+1}), \dots, (u_n, a_n, \mathbf{1}))$ rooted in a node $u_i \in V_i$ encodes a solution $Sol(p_{u_i}) = \{a_i\} \times \dots \times \{a_n\}$. The set of solutions associated with a node u , denoted as $Sol(u)$, is the union of solutions associated with every path rooted at u and ending in $\mathbf{1}$, denoted as $p_u : u \rightsquigarrow \mathbf{1}$:

$$Sol(u) = \bigcup_{p_u: u \rightsquigarrow \mathbf{1}} Sol(p_u).$$

The set of solutions represented by the MDD G is therefore $Sol(r)$, and we say that MDD represents the CSP CSP if $Sol(r) = Sol(CSP)$. An MDD representing CSP from Example 1 is shown in Figure 1.

The critical property that keeps MDDs compact is *merging of isomorphic nodes*. Two nodes u and u' , labeled with the same variable x_i , are *isomorphic* if $Sol(u) = Sol(u')$. They are *merged* by removing one of the nodes, say u , and redirecting all edges with endpoints in u to u' . By this operation, the same solution space is represented with less nodes. Isomorphic nodes can be efficiently detected by traversing a decision diagram in a bottom-up fashion and searching for nodes u and u' labeled with the same variable x_i and having identical child nodes for every outgoing edge.

Another reduction that can be applied on MDDs as well is *removal of redundant nodes*. However, since this operation usually yields an insignificant reduction in the size of MDDs [4], and it unnecessarily complicates the presentation or algorithmic concepts, we will only consider MDDs

where redundant nodes are not removed. All results presented in this paper equally apply to MDDs where redundant nodes are removed. In the rest of this paper we will by the term MDD denote an MDD where all isomorphic nodes are merged and where redundant nodes are not removed.

3 Layer Compression Over MDDs

The primary mechanism through which MDDs achieve significant space savings – merging isomorphic nodes – is essentially reusing the same path endings of different solutions. A similar phenomenon of value repetitions within a single variable domain among various solutions could also be exploited. While this might not give as significant savings for small domains - we argue that the difference in size could become substantial for large domains. We therefore suggest a mechanism for exploiting value repetitions by introducing *pseudo nodes* within the MDD layers as illustrated in the following example.

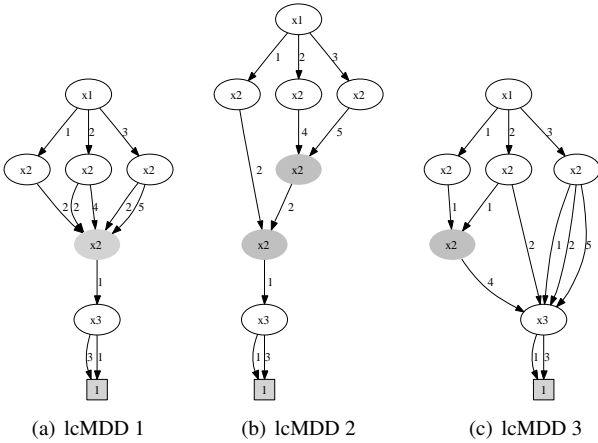


Figure 2. Three different layer compressions of the MDD in Figure 1. The gray nodes are pseudonodes

Consider the second edge-layer of the MDD in Figure 1, that is, $E_2 = \{(u, a, u') \mid \text{var}(u) = 2, \text{var}(u') = 3\}$. There are three labels, 1, 2, and 4, that occur more than once among the edges. The basic compression step is as follows: instead of having three edges, each labeled with eg. 1, we can introduce a pseudo-node, labeled with x_2 , having a single outgoing edge labeled with 1. The result is shown in Figure 2(a). The path of values involving pseudo-nodes changes its semantics. The left-most path of values for example, (1, 4, 1, 1) represents a solution subspace: $\{1\} \times \{4, 1\} \times \{1\}$. The pseudo-node indicates a set-union between the values. Further reduction can be achieved by exploiting the repetition of value 2, by introducing a new pseudo-node. This results in a struc-

ture shown in Figure 2(b). The right-most path of values (3, 5, 2, 1, 3) in Figure 2(b) is associated with the solution space: $\{3\} \times \{5, 2, 1\} \times \{3\}$. A less successful layer compression of the MDD in Figure 1 is shown in Figure 2(c). By introducing a pseudo-node labeled 4, the creation of pseudo-nodes labeled 1 and 2 are made impossible,

Each time we identify k edges pointing to the same node and labeled with the same label a , by introducing a pseudo-node with an outgoing edge labeled with a , we remove the previous k edges. This saves $k - 1$ edges and introduces an extra node. Memory requirements of pseudo-nodes can be ignored in further considerations. namely, level information need not to be stored since pseudo-nodes are always located in the same layer as their parents. The number of outgoing edges need not to be stored since a pseudo-node always has exactly one outgoing edge. All that needs to be stored is the edge pointing to the child of the pseudo-node and the label of this edge, which is already counted as the edge cost.

The compression technique just described results in a *Layer Compressed MDD* (lcMDD). It is a directed acyclic graph, that resembles a standard MDD, except for the fact that each path from root to terminal can have several nodes labeled with the same variable. The variable ordering is retained, that is, in a path $p((u_1, a_1, u_2) \dots, (u_k, a_k, \mathbf{1}))$ it holds $\text{var}(u_i) \leq \text{var}(u_{i+1})$. Nodes u_i that are preceded by a node with strictly smaller variable label $\text{var}(u_{i-1}) < \text{var}(u_i)$ are called *boundary-nodes* (and they correspond to standard MDD nodes), while all other nodes, preceded by the nodes with the same variable label, $\text{var}(u_{i-1}) = \text{var}(u_i)$ are *pseudo-nodes*.

Each path $p((u_i, a_i, u_{i+1}), \dots, (u_n, a_n, \mathbf{1}))$, where $\text{var}(u_j) = j$ contains at least $n - i + 1$ edges and it can be sliced into $n - i + 1$ sub-paths p_i, \dots, p_n , where each p_j consist of nodes labeled with x_j . Let A_j denote the union of all values in sub-path p_j . Then the solution space of such a path is:

$$\text{Sol}(p) = A_i \times \dots \times A_n.$$

As it was the case for an MDD we say that an lcMDD represents a CSP \mathcal{CSP} , if $\text{Sol}(r) = \text{Sol}(\mathcal{CSP})$, where r is the root in the lcMDD.

4 The Algorithms

The procedure described in the previous section, compressing an MDD into a lcMDD through introduction of pseudo-nodes, is presented in Figure 3. In the algorithm formulation V and E are assumed to be globally available, and $D(u)$ denotes the set of labels on outgoing edges of a node u , i.e. $D(u) = \{a \mid (u, a, v) \in E\}$.

Evaluating the condition in line 1 of the EXPANDNODE algorithm as well as iterating over the parent nodes in lines 6 and 8, requires access to information that can be computed

COMPRESSLAYERS

```

1  for each boundary node  $v$ 
2      do EXPANDNODE( $v$ )

EXPANDNODE( $v$ )
1  while there exists label  $a$  for which
     $|\{u \mid (u, a, v) \in E, |D(u)| > 1\}| > 1$ 
2      do pick such an  $a$ 
3      create a new pseudonode  $v'$ 
4      add  $v'$  to  $V$ 
5      add  $(v', a, v)$  to  $E$ 
6      for each  $(u, a, v) \in E$ 
7          do remove  $(u, a, v)$  from  $E$ 
8          for each  $(u, a', v) \in E$ 
9              do replace  $(u, a', v)$ 
                by  $(u, a', v')$  in  $E$ 
10     EXPANDNODE( $v'$ )

```

Figure 3. Every pair of edges that shares labels and end-points can be merged by introducing a pseudo-node. EXPANDNODE considers a single endpoint and introduces pseudo-nodes if it is possible. In Line 1-2 such a label a is picked if it exists. In Line 3-5 a pseudo-node v' is created and made a parent of v by creating an edge labeled a . In Line 6-9 all parents of v that are connected by an edge labeled a have their edge labeled a removed and all other edges with endpoints in v redirected to the pseudo-node v' . The criteria for selecting a , $|D(u)| > 1$ ensures that such edges *do exist*. The existence of such edge is necessary in order to connect the boundary node to the pseudo-node that is going to be created. In order to reduce these edges further EXPANDNODE is called recursively on the new pseudo-node.

in a linear time traversal through the MDD. If we compress the MDD in a layer by layer fashion we only need to traverse a single layer at a time, hence in total only a single traversal of the MDD is required in order to gather the information needed for the entire compression.

The algorithm additionally requires a linear number of operations in the amount of edges saved. Every time a pseudo-node is created, k edges are removed, one extra edge is introduced, and a number of remaining outgoing edges of the k parent nodes are redirected to the pseudo-node (line 9). In worst case, each parent will have all of its $|D_i| - 1$ edges moved, leading to $O(k \cdot |D_i|)$ operations per $k - 1$ edges saved. If K_i denotes the total number of edges saved in the i -th layer, the worst time complexity is $O(\sum_{i=1}^n |K_i| \cdot |D_i|)$. In total, if E denotes the edges in the initial MDD, the total running time is:

$$O\left(|E| + \sum_{i=1}^n |K_i| \cdot |D_i|\right)$$

In this work we do not investigate the effects of different label orderings. However, it should be noted that the quality of compression depends on an order in which we choose labels to compress (line 2 of EXPANDNODE algorithm). As we already mentioned, by compressing the MDD of Figure 1 using the label 4 first, we would result in the lcMDD of Figure 2(c). This structure cannot be compressed further and has more edges than the one shown in Figure 2(b). This is an issue for future work.

4.1 Configuration Queries Over lcMDDs

An lcMDD is a more succinct representation than the MDD. However, we can expect that the more succinct the structure, the more expensive it becomes to execute various queries and transformations on top of it [2].

Which queries and transformations we should care about, when selecting the appropriate target compilation structure, depends on our application domain. In our case, we are concerned with interactive configuration and two operations are of particular importance: *restricting* lcMDD with an assignment and *calculating valid domains* from lcMDD.

A user assigns a value to a variable, in each interaction step. In response, lcMDD should be transformed to reflect this assignment. The algorithms RESTRICT and TRAVERSEUP in Figure 4 show how to implement this transformation with a runtime linear in the size of the edge-layer E_i for an assignment to the variable x_i .

In response to a user assignment, after restricting the corresponding lcMDD, the configurator should *calculate valid domains*, $VD_i \subseteq D_i$, that correspond to those and only those values of remaining unassigned variables that are guaranteed to be part of at least one remaining solution. The algorithm VALIDDOMAINS implements this functionality in linear time by simply scanning all the edges, hence calculating the valid domain of x_i can be done in $O(E_i)$ time.

The above demonstrates that lcMDDs are well suited to support most important interactive configuration tasks. However, we do not take the position that lcMDDs should be preferred in general to MDDs, as there could be other queries and transformations whose implementation is more cumbersome and execution less efficient.

We note however, that many MDD operations relying on iteration over children nodes, e.g. by executing:

```
for each  $(u, a, u')$  do some action on  $u'$ 
```

can be simulated efficiently over lcMDDs, by performing a DFS traversal initiated in each boundary node, in a similar

```

RESTRICT( $x_i, a^*$ )
1  for each  $v \in V_{boundary}^{i+1}$ 
2    do  $P \leftarrow \text{TRAVERSEUP}(i, v, a^*, false)$ 
3    for  $u \in P$ 
4    do add  $(u, a^*, v)$  to  $E$ 

```

```

TRAVERSEUP( $i, v, a^*, found$ )
1   $P \leftarrow \emptyset$ 
2  if  $v \in V_{boundary}^i$  and  $found = true$ 
3    then  $P \leftarrow \{v\}$ 
4  if  $v \notin V_{boundary}^i$ 
5    then for each  $(u, a, v)$ 
6      if  $a = a^*$ 
7        then  $found = true$ 
8       $P_u \leftarrow \text{TRAVERSEUP}(i, u, a^*, found)$ 
9       $P \leftarrow P \cup P_u$ 
10     remove  $(u, a, v)$  from  $E$ 
11  return  $P$ 

```

Figure 4. Restricting the lcMDD with assignment $x_i = a^*$. From each end node v , at the boundary of the $i + 1$ -st layer (denoted $V_{boundary}^{i+1}$), the algorithm traverses upwards through the i -th layer and records all parent nodes P at the boundary of the i -th layer to which there is a path from v involving value a^* on at least one edge. For each such parent u , the algorithm adds an edge (u, a^*, v) . Traversal is performed in TRAVERSEUP function. All encountered edges are deleted as they cannot be traversed for different end points $v \in V_{boundary}^{i+1}$. Boolean variable $found$ indicates if the label a^* has already been encountered. Since every edge is traversed exactly once, the overall runtime complexity is $O(|E_i|)$.

fashion as in TRAVERSEUP algorithm. For that reason, for example, both the *shortest path* and *solution count* operations can be performed efficiently over lcMDDs.

5 Layer Compression Over IDD

The problem that was the main motivation for this work, unnecessarily large MDDs due to large variable domains, could be addressed through techniques other than the introduction of pseudo-nodes. In particular, if between two vertices u and v there is a number of edges labeled with consecutive labels, e.g. $1, \dots, k$, instead of having k edges labeled with single values, we could have a single edge, labeled with the *interval* $[1, k]$. By performing this transformation we end up with *Interval Decision Diagrams* [9] (IDDs). An IDD is an extension of an MDD where every edge is labeled by an interval instead of a single value as in

```

VALIDDOMAINS( $x_i$ )
1   $VD_i \leftarrow \emptyset$ 
2  for each  $(u, a, u') \in E_i$ 
3    do  $VD_i \leftarrow VD_i \cup \{a\}$ 
4  return  $VD_i$ 

```

Figure 5. The algorithm computes valid domains for a variable x_i by traversing all edges in the i -th layer. The complexity is linear.

the usual MDD.

Since IDD exploits repetitiveness of values just as adaptive MDDs, we need to compare against IDD to verify if there is a benefit in our technique beyond interval compression. In particular, it would be interesting to find out if layer compression through pseudo-nodes could yield any savings *after* all consecutively labeled edges have been compacted into intervals. The same procedure COMPRESADAPTIVEENCODING could be applied by treating each interval $[a, b]$ as a single value. Also note that VALIDDOMAINS and RESTRICT would need only minor modifications in order to accommodate the fact that each edge can correspond to more than one value. We will denote the result of the layer compression of an IDD as a *Layer Compressed Interval Decision Diagram* (lcIDD).

Example 2. Consider the graphs in Figure 6. Figure 6(a) shows an MDD containing six edges and Figure 6(b) the corresponding IDD. The layer compression can reduce the number of edges further by merging the two edges labeled by the interval $[1, 2]$. The result shown in Figure 6(c) contains only three edges.

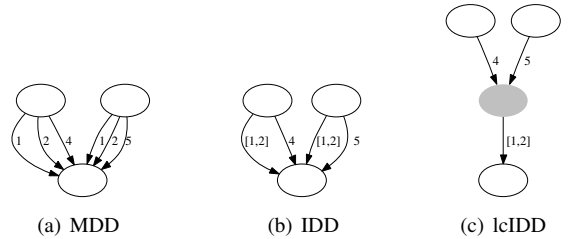


Figure 6. Three nodes from Figure 1 and how their edges will be represented in an MDD, an IDD, and an lcIDD.

6 Experiments

We implemented and tested our algorithm on artificially constructed and real-world instances.

Clarifying Compression Parameters. From the compression algorithm in Figure 3 we can see that the opportunities for compression at each MDD node v depend on the amount of *value sharing* in that node, i.e. the amount of values shared among edges coming from different parent nodes. The more parent nodes, and more values (edges) coming from each parent node, the more compression is likely to take place.

However, what is not clear is how the compression based on pseudo-nodes and compression based on intervals are interacting. More values coming from each parent node imply greater likelihood of large intervals and compact IDD. This would lead to IDD with very few labels from each parent, and undermine the further savings introduced by pseudo-nodes on top of IDD. On the other hand, small number of values from each parent would undermine the savings achieved by introduction of pseudo-nodes as well as intervals. This raises the question if pseudo-nodes introduced on top of IDD could ever lead to significantly smaller lcIDDs?

In order to clarify the above relationships between MDDs, lcMDDs, IDD, lcIDD we constructed a family of CSP models declared over variables $x_1, x_2, x_3 \in D$ for some fixed domain D . For each model we fix $L \subseteq D$ and introduce only one constraint

$$(x_1 = x_3) \vee (x_2 \in L).$$

Figure 7 illustrates the range of resulting MDDs, where $D = \{1, 2, 3, 4\}$ and L is any subset of D . All the edges labeled with a value from L are ending in a special node \hat{u} (denoted by a double circle). Edges originating in \hat{u} lead to $\mathbf{1}$ for all values from D . By changing the size of $L \subseteq D$ we are getting a different ratio between the number of values on each parent edge.

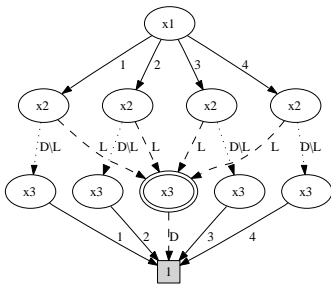


Figure 7. Edges labeled with sets $D, D \setminus L$ and L correspond to the set of edges between the start and end node that is needed to represent support of the values in the set.

In the above example, by taking $L = \{1, 2, 4\}$, introducing intervals and pseudo-nodes, we get an lcIDD shown in Figure 8. Since L contains two intervals, $[1..2]$ and $[4..4]$, the intermediate IDD will involve two edges incoming to \hat{u}

from each parent. Finally, an lcIDD will be further compressed by an introduction of a pseudo-node and an extra edge.

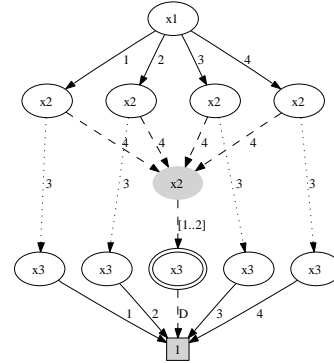


Figure 8. The lcIDD of an IDD from Figure 7 where $D = \{1, 2, 3, 4\}$ and $L = \{1, 2, 4\}$.

We can now highlight some relationships between introduction of intervals and pseudo-nodes. By changing the size of $L \subseteq D$ in the above example we get a different number of values on each parent edge of the node \hat{u} . We fix a domain D to $\{1, \dots, 40\}$. Also, the number of different parent nodes is fixed as long as $L \neq D$.

For each size $k = 1, \dots, 40$, we generated L by randomly selecting k different elements from D . We then computed the sizes of an MDD, lcMDD, IDD and lcIDD for the corresponding CSP problem. We report the results in Figure 9, where each data point is an average size over 10 random instances.

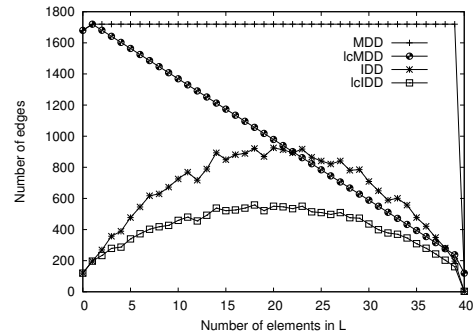


Figure 9. The size of an MDD, lcMDD, IDD and lcIDD on the CSP for $D = [1, 40]$ and L varying from \emptyset to D .

For small L , the effect of introduction of pseudo-nodes is minor, since there is only a small number of values incoming to the node \hat{u} . On the other hand, significant savings are achieved by introduction of intervals alone, since there is a lot of edges between other pairs of nodes.

Each time the size of L increases by one, the lcMDD introduces an additional pseudo-node, thereby decreasing the size of the lcMDD by $|D| - 1$ edges. That is why we see a linear dependency between the size of lcMDDs and the size of L .

Furthermore, as we increase the size of L , its elements will begin to form intervals. For very large $|L|$, all elements will be represented by very few intervals, thus leading to small IDD, but undermining further savings through pseudo-nodes. The maximal difference in size between IDDs and lcIDDs is reached when L contains roughly the half of all the elements $|L| = 20$. In that case, the introduction of intervals would still lead to sufficiently many edges so that the following introduction of pseudo-nodes could make further savings (up to 50%).

Overall, we notice that lcIDDs always perform at least as good as both IDDs and lcMDDs and even though introduction of intervals and pseudo-nodes are not orthogonal techniques, they seem to fit well together.

The $(n \times m)$ -queens problem. We evaluated performance on the $(n \times m)$ -queens problem which consists of placing n queens on an $n \times m$, $n \leq m$, chess board in such a way that no queen is able to capture any other queen. We model the problem with the variables x_1, \dots, x_n each having the domain $\{1, \dots, m\}$. Assigning $x_i = j$ corresponds to putting a queen on the coordinate (i, j) on the chess board. This generalization of the well known n -queens problem allows us to vary the level of interchangeability and tightness of the problem, which are important parameters to vary when evaluating compilation techniques. Since we have argued earlier that the representation of pseudo-nodes requires no extra space and since the number of boundary nodes is the same for (lc)MDDs and (lc)IDDs we only consider the number of edges of the different representation in our experiments.

In Figure 10 we show the size of the MDD, the IDD and the lcIDD for the range of $n \times m$ -queens instances, where $n = 3$, while varying m . Large space savings are attained by introducing interval edges only, and IDDs can be further reduced for about 50% by applying layer compression on the IDDs. We can see that the difference in the size of the three representations increases rapidly as m increases.

In Figure 11 we have made three graphs. All graphs considers the $n \times m$ -queens problem and makes a plot for each $n \in \{3, 4, 5\}$ with an increasing m on the x -axis. The three graphs show the percentage of the edges that will be saved by converting and MDD into an IDD, by converting and IDD into an lcIDD and by converting an MDD directly into an lcIDD.

The figures show the same trends as Figure 10. Using interval edges makes a saving of approx 90% of the edges. Using layer-compression on the IDD yields an additional

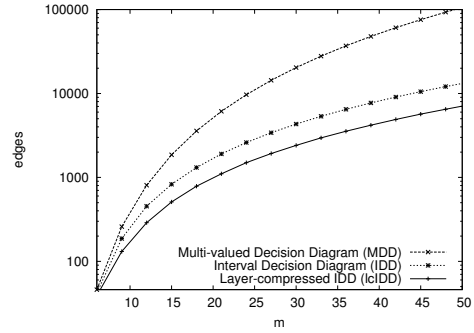


Figure 10. The number of edges in the MDD, IDD and lcMDD representation of the $n \times m$ queens problem for $n \in \{3, 4, 5\}$ and an increasing m on the x -axis.

saving of approx 50% getting a total saving of approx 95% on using lcIDD instead of MDDs. We also note that whereas the savings yielded by the intervals declines as n increases the savings yielded by layer compression increases, which makes the combined approach (lcIDD) yield almost the same compression independently on the size of n , as long as m is a couple of factors larger than n .

Real-World Product Configuration. We analyzed the performance of layer-compression on a real-world car configuration benchmark, Renault [1]. This benchmark comprises of 112 individual configuration constraints. We analyzed the impact of layer-compression on each constraint individually. In Table 1 we present results for the 30 constraints for which lcIDDs give additional space savings over IDDs. We can see that introducing interval edges can yield savings of as much as 87% (constraint 35). However, some further savings are still possible by applying layer-compression - note eg. 32% savings in comparison to the IDD in constraint 77.

Our experiments indicate that in general IDDs are to be preferred over standard MDDs. Furthermore, layer compression will, in many cases, further decrease the number of edges. Both interval edges and layer-compression performs especially well for CSPs with large domains and loose constraints.

7 Conclusions

In this paper we proposed a compression technique of multi-valued and interval decision diagrams, based on reuse of repetitive value occurrences in decision diagram layers through introduction of pseudo-nodes. An empirical study demonstrated that this technique can yield significant space savings for certain class of instances. Given the easiness of implementation and amount of savings achieved, layer

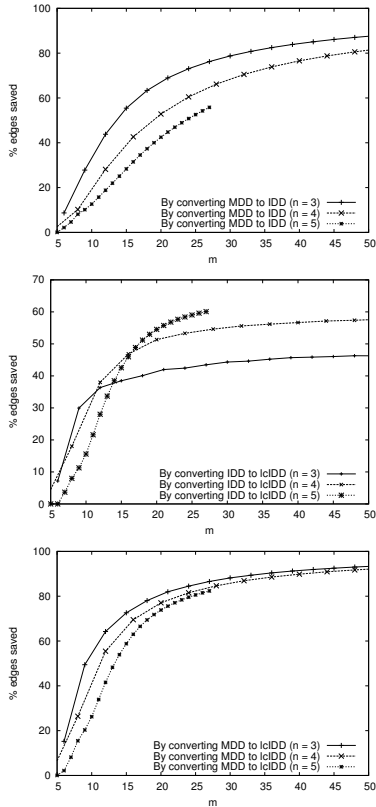


Figure 11. In both plots we consider the $n \times m$ -queens problem for $n \in \{3, 4, 5\}$ and an increasing m . Above we have plotted the percentage of space savings by replacing an MDD by an corresponding IDD. Below we have plotted the percentage of space savings by replacing an IDD by an corresponding lcIDD. Compared to the MDDs we save approximately 90% for all schemes. Compared to the IDDs, lcIDDs additionally save approximately 50% of edges.

compression should be considered as one of the techniques in a knowledge-compilation repertoire. In future, we will explore the applicability of this technique on top of other knowledge-compilation forms, and evaluate its impact in application areas where large domains occur naturally.

References

[1] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs-application to configuration. *Artificial Intelligence*, 2002.

[2] A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[3] T. Hadzic and H. R. Andersen. A BDD-based Polytime Algorithm for Cost-Bounded Interactive Configuration. In *Proceedings of AAAI’06*, 2006.

Table 1. Layer compression over table constraints in the Renault benchmark. The first two columns indicate the table number and the number of solutions. The following three columns present the number of edges for the corresponding MDD, IDD and lcIDD, respectively.

Constraint No.	Solutions	MDD	IDD	lcIDD
2	1050	111	27	19
65	984	173	107	80
68	199	97	83	60
70	394	163	116	85
74	32072	770	186	161
75	8694	449	94	84
76	1396	83	59	50
77	365	197	154	105
78	33437	1222	421	345
106	14238	199	102	86

[4] T. Hadzic, E. R. Hansen, and B. O’Sullivan. On Automata, MDDs and BDDs in Constraint Satisfaction. In *Proceedings of the ECAI 2008 Workshop on Inference Methods based on Graphical Structures of Knowledge*, July 2008. To appear.

[5] T. Hadzic and J. N. Hooker. Cost-bounded binary decision diagrams for 0-1 programming. In P. V. Hentenryck and L. A. Wolsey, editors, *Proceedings of CPAIOR 2007*, pages 84–98, 2007.

[6] T. Hadzic and B. O’Sullivan. Beyond Valid Domains in Interactive Configuration. In *Proceedings of the ECAI 2008 Workshop on Configuration*, July 2008. To appear.

[7] T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Møller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *PETO Conference*, pages 131–138. DTU-tryk, June 2004.

[8] R. Mateescu and R. Dechter. Compiling constraint networks into AND/OR multi-valued decision diagrams (AOMDDs). In F. Benhamou, editor, *CP*, volume 4204 of *Lecture Notes in Computer Science*, pages 329–343. Springer, 2006.

[9] K. Strehl and L. Thiele. Interval diagrams for efficient symbolic verification of process networks. *Computer-Aided Design of Integrated Circuits and Systems*, Aug 2000.

[10] N. R. Vempaty. Solving constraint satisfaction problems using finite state automata. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 453–458, 1992.

[11] M. Wachter and R. Haenni. Multi-state directed acyclic graphs. In Z. Kobti and D. Wu, editors, *Canadian Conference on AI*, volume 4509 of *Lecture Notes in Computer Science*, pages 464–475. Springer, 2007.