Compressing Configuration Data for Memory Limited Devices

Esben Rune Hansen and **Peter Tiedemann**
IT University of Copenhagen
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark
{esben, petert }@itu.dk

Abstract

The paper introduces a new type of compression for decision diagram data structures, such as BDDs, MDDs and AOMDDs. The compression takes advantage of repeated substructures within the decision diagram in order to lessen redundancy beyond what is possible using simple subfunction sharing. The resulting compressed data structure allows traversal of the original decision diagram with no significant overhead. Specifically it allows the efficient computation of valid domains, that is, the assignments for each encoded variable that can participate in a solution, which is critical when the decision diagram is used to support an interactive configurator. We relate these results to applications for interactively configurable memory limited devices and give empirical results on the amount of saved space for a wide variety of instances.

Introduction

Interactive Configuration is a special application of Constraint Satisfaction techniques. The idea is to provide a user with interactive assistance in assigning values to variables in order to obtain a customized solution to the Constraint Satisfaction problem in question. Interactive configuration has found many uses in customizing complex services and products during the sales process but also in configuring complex products during installation or maintainance. One technique for implementing an interactive configurator is to utilize Reduced Ordered Binary Decision Diagrams (ROBDDs, denoted BDDs from here on) (Bryant 1986), or similar data structures, to *compile* all valid solutions to the configuration rules in an off-line phase. It is then possible to assist the user by, at all times, displaying exactly the variable assignments that can lead to a valid configuration while guaranteeing a time complexity that is polynomial in the size of the data structure. The process of computing the choices for each variable that can lead to a valid configuration is called *valid domains computation*. There is a trend towards smart devices where knowledge and rules about how a product may be configured is embedded into the product itself. The product configuration strategy described above can be employed in order to achieve products with embedded configuration data as long as the device is capable of storing a BDD or similar structure, representing the valid configuration state of the device. If this is achieved, the product can be efficiently configured with a guaranteed response time. However, a significant amount of memory is required in order to store decision diagrams. While adding extra memory to a data center or server is typically feasible, even adding a small fraction of additional memory to a cheap mass-produced device can incur an unacceptable increase in cost. It is therefore relevant to consider strategies that allow the configuration data to be made more succint without compromising the ability to perform configuration operations efficiently. In this paper we introduce a new compression technique for decision diagrams that can be applied in the off-line phase after the decision diagram has been constructed. Using this technique the memory required from the configurable devices can be significantly reduced without any significant effect on the response time.

Related Work

The use of the BDD data structure for interactive configuration was introduced in (Hadzic *et al.* 2004; Subbarayan *et al.* 2004). A very successful approach to reducing the space required for storing the configuration data using BDDs is to use decomposition techniques as presented in (Subbarayan 2005). The technique relies on viewing the initial rules as a set of small BDD constraints in a Constraint Satisfaction Problem. These constraints form a dual constraint graph (Rossi, van Beek, & Walsh 2006) with each node corresponding to a single constraint and each edge being labeled with shared variables. A join-tree is an acyclic dual constraint graph with the property that any two constraints sharing a variable x are connected with a path where each edge has x in its label. Given such a join-tree the corresponding CSP can be made consistent by simply applying directional arc-consistency along the join-tree. The idea is now to compile subsets of the constraints into new (larger) constraints in a manner such as to achieve a join-tree. It is very simple to perform valid domains computation as the valid domains are simply the union of the valid domains for each BDD constraint in the join-tree. However, while the decomposition approach can significantly reduce the space required, the complexity of applying restrictions to the data structure, such as assigning a variable, are no longer polynomial in the size of the representation, or even polynomial in the size of the corresponding monolithic representation using a single decision diagram.

Another related result is the introduction of *differential BDDs* in (Anuchitanukul, Manna, & Uribe 1995). In this work the main reduction in the number of nodes is achieved by replacing the usual OBDD labeling scheme based on the variables by a labeling scheme based on the *distance* from the node to its parent(s). The result of their new labeling scheme is that the number of nodes that will be removed by the standard reduction algorithm in (Bryant 1986) are increased. Since they are relying on the recursive bottom-up reduction, structures will only be merged if they are labeled completely in the same way from the terminals and upwards. Hence even very large sub-graphs that are almost labeled in the same way will be not be merged if they disagree on the labeling of a few nodes. We make use of a top-down labeling and compression which are more time-consuming, but makes it possible for us to merge structures that do not agree on all labels.

Preliminaries

In this paper we consider a configuration problem ***CP***(X, D, F), where $X = \{x_1, \ldots, x_N\}$ is the set of variables, F the set of constraints and $D = \{D_1, \ldots, D_N\}$ is the multi-set of variable domains, such that the domain of a variable x_i is D_i. A *single assignment* α is a pair (x_i, a) where $x_i \in X$ and $a \in D_i$. The assignment α is said to have support, iff there exists a solution to CP where x_i is assigned a. If a single assignment (x_i, a), where $a \in D_i$, has support, a is said to be in the valid domain for x_i. A *partial assignment* ρ is a set of single assignments to distinct variables, and a *complete assignment* is an assignment that assigns all variables in X.

Interactive Configuration One successful approach to storing configuration data for use in interactive configuration is to compute a succint representation of all valid solutions to the configuration problem. If the resulting data structure is sufficiently succint and it is possible to efficiently compute the valid domains based on it, we can support an interactive configurator. One data structure that has found frequent use for this problem is the OBDD as defined below:

Definition 1 (Ordered Binary Decision Diagram). *An ordered binary decision diagram (OBDD) on n binary variables $X_{bin} = \{x_1^{bin}, \ldots, x_n^{bin}\}$ is a layered directed acyclic graph $G(V, E)$ with $n + 1$ layers (some of which may be empty) and exactly one root. We use $d(u)$ to denote the layer in which the node u resides. In addition the following properties must be satisfied:*

- *There are exactly two nodes in layer $n + 1$. These nodes have no outgoing edges and are denoted the 1-terminal and the 0-terminal*
- *All nodes in layer 1 to n have exactly two outgoing edges, denoted the* low *and* high *edge respectively. We use $low(u)$ and $high(u)$ to denote the end-point of the low and high edge of u respectively.*
- *For any edge $(u, v) \in E$ it is the case that $d(u) < d(v)$*

We use E_{low} and E_{high} to denote the set of low and high edges respectively. An edge (u, v) such that $d(u)+1 < d(v)$ is called a long edge and is said to skip layer $d(u) + 1$ to $d(v) - 1$.

Definition 2 (Reduced OBDD). *An OBDD is called* reduced *iff for any two distinct nodes u, v it holds that $low(u) \neq low(v) \vee high(u) \neq high(v)$ and further that $high(u) \neq low(u)$ for all nodes u.*

Definition 3 (Solution to an OBDD). *A complete assignment ρ_{bin} to X_{bin} is a solution to an OBDD $G(V, E)$ iff there exists a path P from the root in G to the 1-terminal such that for every assignment $(x_i, b) \in \rho_{bin}$, where $b \in \{low, high\}$, there exists an edge (u, v) in P such that one of the following holds:*

- *$d(u) < i < d(v)$*
- *$d(u) = i$ and $(u, v) \in E_b$*

As an OBDD only allows binary variables, additional steps must be taken in order to encode solutions to problems containing variables with domains of size larger than 2. In order to define such a solution space with domains of size $|D_i| > 2$ we use $\lceil \lg |D_i| \rceil$ binary variables, and the constraints are modified accordingly. Let $X_{bin}(i) \subseteq X_{bin}$ denote the ordered set of binary variables used to encode the domain variable x_i. For each complete assignment ρ_{bin} to X_{bin} that is a solution to the OBDD, the corresponding assignment to each domain variable x_i is simply the bit-string obtained by concatenating the assignments in ρ_{bin} to $X_{bin}(i)$, interpreting low as 0 and $high$ as 1.

Compressing BDDs

BDDs give a compact but explicit representation of a boolean function. The compression achieved by a BDD is possible mainly because identical subfunctions are only represented once, that is, nodes representing identical solution space are merged. Unfortunately, if two subfunctions are closely related but disagree on one or more variables that are placed low in the BDD, the merging of identical subfunctions is of little use. An example of this can be seen Figure 2, where the subfunctions rooted by the nodes labeled 'A' are identical except on the last variable. In some cases (including Figure 2) this can be rectified by choosing a better variable ordering, placing the variables in question earlier in the ordering. However, moving a variable in this way is not always possible without introducing new redundancies due to the dependencies between variables. In this paper we therefore seek to identify repeated substructures that are embedded within the BDD and compress them. By 'embedded structures' we refer to identical substructures repeated in the BDD because they do not remain identical all the way to the terminals as defined below.

Definition 4 (embedded structure). *An embedded structure in a BDD is a set of rooted disjoint DAGs $S = \{G^1 = (V^1, E^1, r^1), \ldots, G^k = (V^k, E^k, r^k)\}$, for which there exists a labeling l of the nodes such that every pair of distinct nodes in $G^i \in S$ are labeled differently. Further for every path $\pi_i = v_1^i, \ldots, v_c^i$ in G^i where $v_1^i = r^i$ there exist for* *some $j \neq i$ a path $\pi_j = v_1^j, \ldots v_c^j$ in G^j where $v_1^j = r^j$ such that:*

- $l(v_k^i) = l(v_k^j)$ *for all* $1 \leq k \leq c$
- $d(r^i) - d(v_k^i) = d(r^j) - d(v_k^j)$ *for all* $1 \leq k \leq c$
- $(v_k^i, v_{k+1}^i) \in E_{low} \iff (v_i^j, v_{k+1}^j) \in E_{low}$ *for all* $1 \leq k < c$

Given an embedded structure in a BDD we define:

- Internal nodes V_S: The set of nodes contained in S, that is $V_S = \{v \in V^i \mid 1 \leq i \leq k\}$.
- Internal edges E_S: The set of edges contained in S, that is $E_S = \{e \in E^i \mid 1 \leq i \leq k\}$.
- Incoming edges E_I: The set of non-internal edges with endpoint in V_S, that is $E_I = \{(u,v) \in E \setminus E_S \mid v \in V_S\}$.
- Outgoing edges E_O: The set of non-internal edges originating from V_S, that is $E_O = \{(u,v) \in E \setminus E_S \mid u \in V_S\}$.
- The ith *component*: $G^i = (V^i, E^i, r^i)$.

Suppose that we are given a BDD with nodes labeled by l and a set of roots $r^1, \ldots, r^k$ that unambiguously defines an embedded structure. For simplicity we assume for now that all components are rooted in the same layer. We further assume that every node that is not contained in the embedded structure as well as the terminals all have distinct labels that differs from the labels used in the embedded structure. We note that V_S, E_I, E_S and E_O are unambiguously defined by the specified roots and the labeling of V.

In order to use the supplied embedded structure to obtain a more compact representation of the BDD containing the embedded structure, we create an auxiliary structure $G' = (V', E')$ based on the embedded structure and then subsequently replace the entire embedded structure by this more compact auxiliary structure. All nodes in the embedded structure with the same label *lb* will be represented by a unique node $v'_{lb} \in V'$. We define the function $\mu : V \to V'$ that maps every node v that is part of the embedded structure to the node $v'_{l(v)} \in V'$, and maps all other nodes to themselves.

For every edge $(u,v) \in E$ where $u \in V^i$ we add an edge $(\mu(u), \mu(v))$ with the *out-mark* i to E', in case v is in V' but in a different component than u, the added edge is also given an *in-mark* indicating the component of v (we denote such an edge as a *transit edge*). For every edge (u,v) where $v \in V^i$ and $u \notin V'$ we add the edge $(u, \mu(v))$ with an in-mark i to E'. In the following we will use $mark_{in}(e)$ and $mark_{out}(e)$ to denote the in and out marks of an edge e respectively.

We can now remove all nodes and edges from the embedded structure and still traverse the BDD using the new nodes and edges of G', the only change being that the components visited must be tracked by remembering the most recent in-mark m_i encountered. Faced with a node with more than two outgoing edges, the edge e such that $mark_{out}(e) = m_i$ should be followed.

The above step can remove a significant number of nodes from the BDD, but their edges remain and carries more information than before. In order to compress the edges as well, every node in the auxiliary structure is given an unmarked *default* high and low edge. The default high and low edge of a node points to the most frequent end-point among the marked high and low edge respectively. Hence if a node in the auxiliary structure corresponds to a set of nodes in the embedded structure each having a low child with the same label, all necessary information on the location of the low children is contained in the single default low edge. For all edges that disagree with the default edge we keep the edge and its out-mark. We denote the latter type of edges as *extended edges* and denote all other edges in E as *default edges*. For each node u we denote by $low_{ext}(u)$ and $high_{ext}(u)$ the extended low and high edges originating from u respectively. In order to traverse the compressed BDD, we as before follow the edge e such that $mark_{out}(e) = m_i$, if it exists, and otherwise follow the default edge. An example showing how an embedded structure is compressed is shown in Figure 2 and pseudo-code is given in Figure 1.

COMPRESSEMBEDDING(G, L, l, cp)

1 $V' \leftarrow cld \leftarrow low_{ext} \leftarrow high_{ext} \leftarrow \emptyset$
2 **for** each $lb \in L$
3   **do** make new node v_{lb}^S
4   **add** v_{lb}^S **to** V'
5 **for** each $lb \in L$
6   **do for** each $w \in V'$ **where** $l(w) = lb$
7   **do add** $(w, l(low(w)))$ **to** *cld*
8   let l_{max} be the label occurring most often in *cld*
9         $low(v_{lb}^S) \leftarrow v_{l_{max}}^S$
10   **for** each $(w, lb') \in cld$ **where** $lb' \neq l_{max}$
11   **do add** $\mu(low(w))$ **to** $low_{ext}(v_{lb}^S)$
12                 $mark_{out}((v_{lb}^S, \mu(low(w)))) \leftarrow cp(w)$
13   **for** each $(w, lb') \in cld$
14   **do if** $w, low(w) \in V' \wedge cp(w) \neq cp(low(w))$
15   **then** $e \leftarrow (v_{lb}^S, \mu(low(w)))$
16                     $mark_{in}(e) \leftarrow cp(low(w))$
17   **for** each $u \in V \setminus V'$ for which $l(low(u)) = lb$
18   **do** *component* $\leftarrow cp(low(u))$
19                 $low(u) \leftarrow \mu(low(u))$
20                 $mark_{in}((u, low(u)) \leftarrow$ *component*
21         $cld \leftarrow \emptyset$
22 (Line 5-21 repeated with low replaced by $high$)
23 Remove nodes v such that $l(v) \in L$ and all incident edges

Figure 1: Above, G is the original BDD, L the set of labels in the embedded structure, l the node labeling and *cp* maps $v \in V^i$ to i. Line 6-9 sets the default edge, while line 10-12 adds extended edges for those edges that disagree with the default edge. Line 13-16 set in-marks on transit edges. Line 17-20 redirects incoming edges and sets in-marks according to the component they pointed to previously. Line 23 removes the embedded structure from the BDD.

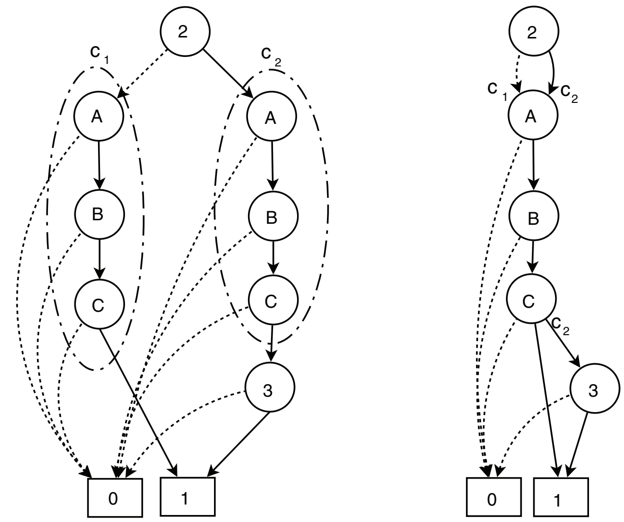

Figure 2: On the left is shown an example BDD. Dotted edges are low edges and solid edges are high edges. An embedded structure with components c_1 and c_2 is indicated. On the right is the result of running COMPRESSEMBEDDING. The outgoing edges of the root node have in-marks indicating which of the components they originally lead to. The compressed node 'C' has one extended edge with the out-mark c_2.

In summary COMPRESSEMBEDDING saves

$$\left(\sum_{1 \le i \le k} |V^i| \right) - |L|$$

nodes and saves $E_{saved} =$

$$\sum_{1 \le l_{start} \le k} \max_{l_{end} \in \mathbb{N}} \left| \{(u,v) \mid l(u) = l_{start} \wedge l(v) = l_{end}\} \right| - 1$$

edges. We add $E_S + E_O - E_{saved}$ out-marks and E_I in-marks, if we assume that all components are rooted in the same layer.

Handling components rooted in different layers

In order to handle embedded structures that consist of components rooted in different layers, we give each incoming edge an *offset-mark*, that indicates the difference between the layers of the nodes in the compressed embedded structure and the layers of the nodes in the original component, specifically, we give every incoming edge, entering a component rooted in r, the offset-mark $d(\mu(r)) - d(r)$. During a traversal we can then determine the layer of a compressed node v as being $d(v) + \textit{offset}$ where *offset* is the offset-marking of the most recently traversed incoming edge. This approach requires that we know whether or not a node is located in a compressed embedded structure. This information could for instance be specified by adding a flag to every node in the compressed BDD.

Finding and choosing embeddings

In the previous section we covered how an embedded structure can be utilized to compress a BDD, in this section we discuss how to find and choose between embedded structures. Suppose that we have a set of nodes $\{r'_1, \ldots, r'_k\}$ and we want to construct an embedded structure rooted in $r'_1, \ldots, r'_k$ that satisfies Definition 4. We start by giving an algorithm that can find all possible embedded structures and based on this present a feasible heuristic method. The basis of this algorithm is a specialized simultaneous DFS starting in the nodes $r'_1, \ldots r'_k$. In each step of the DFS, let r and p be vectors of elements drawn from $V \cup \{\emptyset\}$, such that r_i is either the node visited by the ith DFS or $\emptyset$ to indicate that the ith DFS did not visit a node in this step, while p is a similar vector of the nodes from which r was visited. Additionally define $|r|$ as the number of elements in r different from $\emptyset$. Hence, initially $r = (r'_1, \ldots r'_k)$ and $|r| = k$. We assume that initially $p = r$.

In each step the algorithm starts by replacing any nodes in r that have already been labeled by $\emptyset$. Then, for each possible length of a long edge λ (in the order determined by the permutation function π), the function $drop_\lambda(r, p)$ is used to create a vector v^λ containing the elements from r with some entries replaced by $\emptyset$. Specifically $drop_\lambda$ replaces a node r_i with $\emptyset$ if $d(r_i) - d(p_i) \neq \lambda$. Additionally $drop_\lambda$ may replace any additional number of nodes with $\emptyset$, in such a way that there are no duplicate nodes in v^λ. If at least two elements different from $\emptyset$ remains in v^λ the algorithm labels all remaining nodes in v^λ with the same timestamp and continues the DFS by visiting all high and all low children of the nodes in v^λ. The order in which the high and low children are visited is determined by the function w. The pseudo-code for this algorithm is given as DFS-LABEL in Figure 3. Intuitively the algorithm is simply a simultaneous DFS that explores subgraph isomorhisms, allowing subsets of the traversals to pause, or partition into separate searches. The important property of DFS-LABEL is that we by using it in combination with each possible choice of initial roots, $drop_\lambda$, w and π, can enumerate all embedded structures satisfying Definition 4.

However, while we could use this method to obtain all possible embeddings it should be clear that this is not feasible. Just considering embedded structures with two roots, we are in the worst case able to produce a number of embeddings that are exponential in the height of the BDD. Further there are $|V|^2/2!$ different pairs to choose from, $|V|^3/3!$ triplets etc. Hence even if there was not an exponential number of embedded structures for each set of roots, it would be infeasible to consider them all.

Implementation

Given the infeasibility of working with all possible embeddings we constrain our-selves to considering a small subset of all possible embeddings in our implementation. First off, we only consider sets of roots on the same level. Additionally we only consider the weight function by which the shortest edge is visited first. Finally we do not let the simultaneous DFS partition into multiple separate searches. Instead the layer in which the first $r_i \neq \emptyset$ resides is used to decide for which nodes the search continues, that is, all nodes in r in a different layer are replaced by $\emptyset$. Note that the embedding that will be found by this approach is com-

```
DFS-LABEL(r, p)
1  replace all labeled nodes in r by ∅
2  for i ← 0 to n
3       do λ ← π(i)
4          v^λ ← drop_λ(r, p)
5          if |v| ≥ 2
6            then c^1 ← c^2 ← (∅, ..., ∅)
7                 for each v_i^λ ≠ ∅
8                     do time ← time + 1
9                        l(v_i^λ) ← time
10                       cp(v_i^λ) ← i
11                       c_i^1 ← high(v_i^λ)
12                       c_i^2 ← low(v_i^λ)
13                if w(r, high) < w(r, low)
14                  then exchange c^1 ↔ c^2
15                DFS-LABEL(c^1, v^λ)
16                DFS-LABEL(c^2, v^λ)
```

Figure 3: Line 1 exclude the labeled nodes from the current search. Line 4 invokes $drop_\lambda$, storing in v^λ a subset of the nodes in r for which the simultaneous DFS should continue. In line 8-9 the selected nodes are given identical labels, marking them for merging by COMPRESSEMBEDDING. Line 11-12 builds the vectors of the high and low children of the visited nodes. In line 13, the function w is used to determine whether to visit the high or low children first. Line 15-16 continues the search for the selected nodes.

pletely determined by the roots.

Our implemented algorithm considers all ordered pairs consisting of nodes from the same layer. Triplets (v_1, v_2, v_3) are only considered if we find that (v_1, v_2) and (v_2, v_3) both roots worthwhile embedded structures, quadruples (v_1, v_2, v_3, v_4) are only considered if (v_1, v_2, v_3) and (v_2, v_3, v_4) are considered and are worthwhile and so forth.

Each embedded structure S discovered in this way is analyzed to compute the potential saving of compressing it. If this saving is above a certain threshold S is added to a queue, otherwise it is discarded. Even so, there will frequently not be sufficient internal memory to store the roots and saving of all embedded structures that can yield savings larger than the threshold. If the queue grows too large the embedded structures yielding the poorest savings are removed from the queue to make room.

Given that we have computed some selection of embedded structures, we place them in a max-priority queue using the saving of each as priority. We then simply compress them one by one until no embedded structure yielding a positive saving is left. Each time we dequeue an embedded structure S from the priority queue we reanalyze the saving of compressing it. This is necessary as the compression of other embedded structures might have decreased the saving that can be obtained from S. If the recalculated saving is smaller than the embedded structure at the top of the queue we dequeue the top element and enqueue the reanalyzed structure with its updated saving as priority. Otherwise we compress the BDD based on S.

Terminal suppression

In most BDDs it is natural to expect a very large number of long edges leading to terminals. We therefore apply a very simple method for saving additional space, by simply not storing the edge to the terminal for nodes with exactly one terminal edge. Instead all nodes are given a mark indicating whether they have two normal edges, a high edge to the 1-terminal, etc. While this mark incurs an additional space cost for all nodes, it fits well with our other compression scheme which will remove a large part of the nodes in the BDD. Edges to terminals that have an out-mark are not suppressed in this manner.

Valid domains and restrictions

In order to utilize the compressed BDD for interactive configuration we need to efficiently support the computation of valid domains. Recall that given a configuration problem *CP* on some variables $X = \{x_1, \ldots, x_N\}$, and a partial assignment ρ to X, the valid domain for x_i is the set of values a such that there exists a solution to *CP*, consistent with ρ, where x_i is assigned a.

We will not describe here how to calculate valid domains but instead refer to (Hadzic 2006). For our purpose the applied restrictions are stored outside the compressed BDD and are only used to determine which combinations of edges that may be traversed. We first note that any traversal of the original BDD taking time t can be trivially performed in time $O(t)$ in the compressed BDD, hence the traversal needed in computing the valid domains will not be significantly slower.

The algorithm used for calculating the valid domains needs to mark every node in the original BDD. These markings cannot be compressed along with the node they are placed on, and hence needs to be replicated in the compressed BDD. This implies that the space usage for these markings will be proportional to the number of nodes in the original BDD. By making a slight modification of the algorithm described in (Hadzic 2006), where we calculate the valid domains in a bottom-up fashion, it is possible to achieve an algorithm that only requires three bits per node in the original BDD. This does not affect the worst-case running time which remains $O(n + \sum_{1 \le i \le N} |V(x_i)||D_i|)$.

Experiments

In this section we apply our compression scheme to a wide variety of BDD instances. Nearly all of them have been compiled by either ConfigIT Product Modeler (2007) or CLab (2007), encoding domains of size d using $\lceil \lg d \rceil$ binary variables. All instances but the 10-queen problem and the 9-bit multiplier are available online from either CLib (2007) or (Hadzic 2005). Below we quickly summarize the origin and nature of each of the included instances.

Product Configuration We have included four instances constructed to provide interactive configuration of customizable products. The instance 'Bike' is constructed based on configuration options for a bike shop, 'PC1' and 'PC2' is based on the same PC configurator but with different variable orderings and 'Big-PC' represent a more complicated PC configurator. Finally the instance 'Renault' is an representation of the valid configuration options for the Renault Megane family of cars.

Power Supply Restoration The Power Supply Restoration (PSR) problem involves manipulating a power grid in order to return power to the consumers in the grid. We have included two such instances, for further information please see (Hadzic & Andersen 2005).

Other Finally we have tested some assorted BDDs, one representing an 9-bit multiplier ('9-bit Mult.'), one representing the solutions to the 10-Queen problem ('10-queen') and one being a small fault tree representation ('Chinese').

Calculating saved space

For each of the tested instances in our experiments we identify a number of embedded structures in the tested instance and use a greedy approach to choose which structures to compress as described previously. While we calculate the number of nodes removed, labels added, space used for markings etc, the precise saving will depend on the exact implementation of the original BDD data structure, in particular the number of bits used for each node.

Instead of making assumptions on the original data structure, we make the following sceptic estimate on the costs. The cost of an edge is 1 unit, the cost of a node with its edges is 2 units, the cost of marking an extended node as extended and the cost of an in-mark or out-mark is $1/2$ a unit. The cost of marking an edge for terminal suppression is $3/8$ units. The space used for the markings required to support valid domains computation is $|V|/8$ units in total.

Results

The results of our tests are shown in Figure 4. For most instances our compression technique removes at least half the nodes in the original BDD. In most instances this node reduction is achieved while only adding a small number of extended edges and markings, resulting in a high saving. As can be seen from PC1/2, a poor variable ordering means that we can compress more, though the better ordered BDD remains significantly compressed. The worst result is achieved with the 9-bit multiplier for which the estimate is 3%, due to the high number of extended edges and markings that were added to the compressed structure. We note however that our estimate of the cost of labeling an edge is a bit exaggerated so in practice the saving would be larger. Additionally we believe the result for the multiplier would have been significantly improved if we had allowed embedded structures rooted in different layers in our implementation. Overall, the results indicate a consistent and significant saving in space.

Application to other data structures

While we have presented our results within the context of binary decision diagrams it should be clear that our approach applies to, for example, multi-valued decision diagrams and even AND/OR multi-valued decision diagrams with only trivial modifications, as we are simply taking advantage of repeated subgraph structures. We do note however that it is not possible to easily predict the compression performance of applying our approach to these data structures based on our current experiments.

Instance	$\lvert V\rvert$	In	Out	Ext.	Rem.	TS	Sav
Product Customization							
Bike	1514	152	174	112	761	15%	40%
PC1	16496	453	792	317	11998	33%	67%
PC2	3467	726	837	271	2625	8%	44%
Big-PC	6468	566	567	450	3170	11%	39%
Renault	455798	13580	16513	7355	267356	24%	58%
Power Supply Restoration							
1-6+22-32	20937	2497	2428	1099	12863	22%	50%
Cplx-P2	163432	8930	13540	5098	89442	22%	51%
Other							
9-bit Mult.	13918	2375	4907	3370	6195	-16%	3%
10-queen	10049	1161	1498	902	5297	24%	44%
Chinese	3590	228	427	194	2113	21%	51%

Figure 4: The table shows the result of applying our compression technique to a variety of different BDDs. The column $|V|$ contains the number of nodes in the original BDDs, *In* is the number of in-markings, *Out* is the number of out-markings, *Ext.* is the number of extended nodes and *Rem.* is the number of nodes removed by the compression step. The entry *TS* gives an estimate on the amount of space saved solely by using terminal suppression and *Sav* gives a pessimistic estimate on the space saved by combining terminal suppression with compression of embedded structures, based on our sceptical estimate. Note that for the instance *9-bit Mult* the *Sav* entry gives the saving for applying just compression of embedded structures as terminal suppression increase the amount of space required.

Conclusion

In this paper we have introduced a new type of compression for decision diagram data structures for the purpose of storing configuration data on memory limited devices. For the case of binary decision diagrams we have shown how to perform restrictions and valid domains computation on the compressed data structure. In addition we have given experimental results for a wide variety of relevant instances of configuration data and shown that the compression achieved is significant in most cases.

References

Anuchitanukul, A.; Manna, Z.; and Uribe, T. E. 1995. Differential BDDs. In *Computer Science Today*, 218–233.

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

2007. Clab: Configuration laboratory. http://www.itu.dk/people/rmj/systems/clab10.html.

2007. Clib: Configuration benchmarks library. http://www.itu.dk/research/cla/externals/clib/.

2007. Configit software. http://www.configit.dk/.

Hadzic, T., and Andersen, H. R. 2005. Interactive reconfiguration in power supply restoration. In *Proceedings of Principles and Practice of Constraint Programming - CP 2005*, 767 – 771. Springer-Verlag GmbH.

Hadzic, T.; Subbarayan, S.; Jensen, R. M.; Andersen, H. R.; Hulgaard, H.; and Moller, J. 2004. Fast backtrack-free product configuration using a precompiled solution space representation. In *Proceedings of the International Conference on Economic, Technical and Organizational aspects of Product Configuration Systems*, 131–138. DTU-tryk.

Hadzic, T. 2005. Power supply restoration instances. http://www.itu.dk/people/tarik/psr/.

Hadzic, T. 2006. Calculating valid domains for bdd-based interactive configuration. http://www.itu.dk/people/tarik/cvd/cvd.pdf.

Rossi, F.; van Beek, P.; and Walsh, T. 2006. *Handbook of Constraint Programming*.

Subbarayan, S.; Jensen, R.; Hadzic, T.; Andersen, H.; Hulgaard, H.; and ller, J. M. 2004. Comparing two implementations of a complete and backtrack-free interactive configurator. In *Proceedings of the CP-04 Workshop on CSP Techniques with Immediate Application*, 97–111.

Subbarayan, S. 2005. Integrating csp decomposition techniques and bdds for compiling configuration problems. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CP-AI-OR*. Springer.