Column Oriented Compilation of Variant Tables

Albert Haag[1]

Abstract. The objective of this work is to improve variant table evaluation in a configurator by compiling/compressing the tables individually to reduce both processing time and space. The main result is a proposed simple heuristic for decomposing the variant table into subtables and a representation linking these subtables in a directed acyclic graph (*DAG*). The size of the compression obtained by this heuristic for examples used in [2, 10] is comparable to that achieved there. However, a formal analysis of complexity has not yet been completed. A prototype implemented in Java exists. Objectives in designing it were to keep it completely decoupled from any particular configurator, while using little machinery in order to keep software maintenance costs low. Testing both on abstract examples and on tables that resemble real customer data is ongoing and looks promising. Non-atomic table cells (such as real intervals, or value sets) are supported. My approach to negative variant tables [8] has been incorporated into the implementation.

1 Preamble

Following the usage of the *SAP Variant Configurator - SAP VC* [4] I term a table that lists all valid combinations of properties of a product as a *variant table*. One use of a variant table in configuration is as a table constraint. However, variant tables and their maintenance by modelers entail some special considerations that have implications beyond what is formally captured by that concept:

- The product properties and the values referred to have a business meaning outside of configuration. Value domains for a product property are naturally maintained in a defined sort order
- Variant tables may be stored in a database table outside the model. A data definition in a database management system (*DBMS*) may exist that defines database keys etc.
- Tables will often be *relational*, i.e., table cells will be atomic values (*strings* or *numbers*), but non-atomic entries may occur [2]
- Customers have the tendency to maintain large wide tables, i.e., normalization is often sacrificed in favor of fewer tables. In such cases, compression techniques seem particularly advantageous
- A variant tables can have its own individual update cycle. The overall model should not need to be compiled or otherwise processed each time a change is made to a table

The relevant functionality a configurator has to provide regarding variant tables is to

- restrict domains via contraint propagation (general arc consistency *GAC* (see [3])), treating the variant table as a constraint relation.
- query the table using a key defined in the DBMS
- iterate over the current solution set of the table given domain restrictions for the associated product properties

I assume that an existing (legacy) configurator has already implemented this functionality. This will include means for efficiently testing membership in a domain (*memberp*), testing if domains intersect (*intersectsp*), and for calculating the intersection of domains (*intersection*).

2 Introduction and Notation

In [8] I look at tables that list excluded combinations of product properties. I term these *negative variant tables*. The techniques and the associated prototypical implementation I present here cover that approach as well. Here, except in Section 7, I limit the exposition to positive tables.

For simplicity, I take a positive variant table $\mathcal{T}$ to be given as a relational table of values (atoms). I relax the assumption about $\mathcal{T}$ being relational later in Section 3.3. If $\mathcal{T}$ has k columns and r rows it is an $r \times k$ array of values: $\mathcal{T} = \left(a_{ij}\right); i = 1 \ldots r; j = 1 \ldots k$. k is the *arity* of $\mathcal{T}$. Each column is mapped to a product property such as *Color*, *Size*, ... The product properties are denoted by $v_j : j = 1 \ldots k$.

Following notation in [8], I define the column domains $\pi_j(\mathcal{T})$ as the set of all values occurring in the j-th column of T:[3]

$$\pi_j(\mathcal{T}) := \bigcup_{i=1}^{r} \{a_{ij} \in \mathcal{T}\} \tag{1}$$

I define s_j as the number of elements in $\pi_j(\mathcal{T})$:

$$s_j := |\pi_j(\mathcal{T})| \tag{2}$$

I call

$$\pi(\mathcal{T}) := \pi_1(\mathcal{T}) \times \ldots \times \pi_k(\mathcal{T})$$

the global domain tuple for $v_1, \ldots, v_k$ and I denote a run-time domain restriction for the product property v_j as $R_j \subseteq \pi_j(\mathcal{T})$.

For ease of notation, I refer to any subset of $\pi(\mathcal{T})$ that is a Cartesian product as a *c-tuple*. Both $\pi(\mathcal{T})$ itself and the tuple of run-time restrictions $R := R_1 \times \ldots \times R_k$ are c-tuples.

$\mathcal{T}$ can be seen as a set of value tuples and, as such, $\mathcal{T} \subseteq \pi(\mathcal{T})$, but $\mathcal{T}$ is not necessarily a c-tuple. In the special case that $\mathcal{T}$ itself is a c-tuple[4], i.e., $\mathcal{T} = \pi_1(\mathcal{T}) \times \ldots \times \pi_j(\mathcal{T})$, then

$$s := \sum_{j=1}^{k} s_j \tag{3}$$

[1] SAP SE, Germany, email: albert.haag@t-online.de

[2] The SAP VC allows real-valued intervals and and sets of values in table cells. Such a table cannot be transparently stored as a relation table in a DBMS

[3] $\pi_j(\mathcal{T})$ can be seen as the *projection* of $\mathcal{T}$ for the j-th column

[4] In this case, it holds that for any given c- tuple (run-time restriction) R

$$\pi(\mathcal{T} \cap R) = \mathcal{T} \cap R$$

Juha Tiihonen, Andreas Falkner, and Tomas Axling, Editors
Proceedings of the 17th International Configuration Workshop
September 10-11, 2015, Vienna, Austria

values suffice to represent the $N := (\prod_{j=1}^{k} s_j)$ tuples in $\mathcal{T}$. In this case, the c-tuple $\pi(\mathcal{T})$ is a compressed way of representing the array a_{ij}. This observation is central to attempts to compress a given table into a disjoint union[5] of as few as possible c-tuples. It has been utilized in various other work. Notably, I cite [10, 6] in this context.

When $\mathcal{T}$ is viewed as a constraint relation, $v_1 \dots v_k$ are just the constraint variables, and each value $x \in \pi_j(\mathcal{T})$ maps to a proposition $p(v_j, x)$ that states that v_j can be consistently assigned to x: $p(v_j, x) \models (v_j = x)$. In this case, a row (tuple) r_i in $\mathcal{T}$ directly maps to a conjunction of such propositions: $r_i = (a_{i1}, \dots, a_{ik}) \models \tau_i := p(v_1, a_{i1}) \wedge \dots \wedge p(v_k, a_{ik})$, and $\mathcal{T}$ itself represents the disjunction: $\mathcal{T} \models \bigvee_{i=1}^{r} \tau_i$.

Given the definition of s in (3), there are s distinct propositions associated with $\mathcal{T}$, one for each value in each $\pi_j(\mathcal{T})$. Hence, given any value assignment to these s propositions, $\mathcal{T}$, seen as a logical expression implementing the constraint relation, will evaluate to 1 ($\top$/true) or 0 ($\bot$/false), and $\mathcal{T}$ defines a Boolean function:

$$\mathcal{F} : 2^{\pi_1(\mathcal{T}) \times \dots \pi_k(\mathcal{T})} \rightarrow \{0, 1\} \tag{4}$$

$\mathcal{F}$ can be represented by a BDD (*Ordered Binary Decision Diagram*) or one of its cousins [12]. BDDs have the enticing property that finding the right ordering of their Boolean variables (the propositions $p(v_j, x)$) can lead to a very compact representation. Furthermore, this representation can potentially be found by existing agnostic optimization algorithms. The complexity of this optimization is high, and heuristics are employed in practice. The construction of an optimal BDD is not suitable for configuration run-time, but must be performed in advance. Hence, this approach is referred to as a *compilation approach*. For configuration, this has been pursued in [9]. The approach using *multi-valued decision diagrams* (*MDD*) [2] is related to the BDD approach. *Zero-Suppressed decision diagrams* (*ZDDs*) [12] are another flavor of *BDD*, which I refer to again below.

From a database point of view, approaches based on compression that allow fast reading but slow writing have been developed, among them column-oriented databases [14]. The SAP HANA database supports column orientation as well. My work, here, is not directly based on database techniques, although thinking about column-orientation was the trigger for the heuristics detailed in Section 4[6].

Both the BDD approaches and the database approaches entail a maintenance-time transformation into a compact representation that facilitates run-time evaluation, and both strive for a compact representation (*"hard to write, easy to read"*). This would also apply to various approaches at identifying c-tuple subsets of $\mathcal{T}$ whether with the explicit notion of achieving compression [10] or of simply speeding up constraint propagation (GAC) algorithms [6].

Thus, all these approaches could be termed as compression or compilation or as both. Indeed, I conjecture that the ultimately achievable results to be more or less identical, up to differences forced by the respective formalism, which may be unfavorable in some circumstances. For example, my experiences suggest that a BDD may be less suitable than a ZDD for compiling a single table constraint, because in the latter propositions need only be represented where they occur in positive form (see [12]). The approach I follow here is motivated by looking at ways of decomposing a table into disjoint subtables based on a particular heuristic.

In Section 3, I introduce the basic approach to decomposing a table. In Section 4, I discuss the heuristic. My running example is a (single) variant table listing all variants of a t-shirt. The example is taken and adapted further from [2]. The t-shirt has three properties *Color* (v_1), *Size* (v_2), and *Print* (v_3) with global domains:

- $\pi_1(\mathcal{T}) := \{Black, Blue, Red, White\}$
- $\pi_2(\mathcal{T}) := \{Large, Medium, Small\}$
- $\pi_3(\mathcal{T}) := \{MIB(\text{Men in Black}), STW(\text{Save the Whales})\}$

Of the 24 possible t-shirts only 11 are valid due to constraints that state that MIB implies $black$ and STW implies $\neg small$ as depicted in table (1). In Section 5, I extend this example to be slightly more complex.

I have implemented a prototype in Java that meets the functionality requirements listed above. Here, I refer to it as the *VDD prototype*. It functions standalone, independent of any particular configurator. A feature of this implementation is that it can be selectively applied to some tables, while processing others with the existing means of the configurator. Results obtained using this prototype validate the approach (see Section 6). In Section 7, I comment on results for *double negation* of variant tables, an approach I develop in [8]. Real run-time performance evaluations have not been done, but in Section 8 I discuss what results I have. I close this paper with an outlook and conclusions (Section 9).

Finally, a disclaimer: While the motivation for this work lies in my past at SAP and is based on insights and experiences with the product configurators there [4, 7], all work on this paper was performed privately during the last two years after transition into partial retirement. The implementation is neither endorsed by SAP nor does it reflect ongoing SAP development.

3 Decomposition

3.1 Column Oriented Decomposition

Let s be defined as in (3). Given a table $\mathcal{T}$ of arity k with r rows $r_i = (a_{i1}, \dots, a_{ik})$ and selecting one of the s propositions $p(v_j, x)$ associated with $\mathcal{T}$, then $\mathcal{T}$ can be decomposed into those rows in which x occurs in column j and those where it doesn't. Define $\mathcal{L}(\mathcal{T}, j, x)$ as the sub-table of $\mathcal{T}$ consisting of those rows that do not reference $p(v_j, x)$:

$$\mathcal{L}(\mathcal{T}, j, x) := \{r_i = (a_{i1}, \dots, a_{ik}) \in \mathcal{T} \quad | \quad a_{ij} \neq x\} \tag{5}$$

$\mathcal{L}(\mathcal{T}, j, x)$ has the same arity k as $\mathcal{T}$. In the complementary sub-table $\mathcal{T} \setminus \mathcal{L}(\mathcal{T}, j, x)$ all values in the j-th column are equal to x by construction. Define $\mathcal{R}(\mathcal{T}, j, x)$ as the sub-table of arity $(k-1)$ obtained by removing the j-th column from $\mathcal{T} \setminus \mathcal{L}(\mathcal{T}, j, x)$:

$$\mathcal{R}(\mathcal{T}, j, x) := \{(a_{ih}) \subset (\mathcal{T} \setminus \mathcal{L}(\mathcal{T}, j, x)) \quad | \quad h \neq j\} \tag{6}$$

Given a table $\mathcal{T}$ and a proposition $p(v_j, x)$ associated with $\mathcal{T}$, then I call $\mathcal{L}(\mathcal{T}, j, x)$ defined in (5) the *left sub-table* of $\mathcal{T}$ and $\mathcal{R}(\mathcal{T}, j, x)$ defined in (6) the *right sub-table* of $\mathcal{T}$. Either $\mathcal{L}(\mathcal{T}, j, x)$ and/or $\mathcal{R}(\mathcal{T}, j, x)$ may be empty.

The variant table of the 11 variants of the t-shirt example is shown in Table 1. It illustrates a decomposition of this table table based on the proposition $p(v_2, Medium)$. $\mathcal{L}(\mathcal{T}, 2, Medium)$ is in boldface, and $\mathcal{R}(\mathcal{T}, 2, Medium)$ is underlined. (The *value block* of cells $\{a_{i2} \in \mathcal{T} \quad | \quad a_{i2} = Medium\}$ is in italics.)

[5] I exclusively use the term *disjoint union* to refer to a union of disjoint sets [5]. I denote the disjoint union of two sets A and B by $A \uplus B$ which implies that $A \cap B = \emptyset$

[6] It would be interesting to investigate whether a column-oriented database could in itself be beneficially employed in this context. This was proposed by colleague at SAP some time ago, but has not been followed up on

Table 1. Basic decomposition of a table

Color	Size	Print
Black	**Small**	**MIB**
Black	*Medium*	MIB
Black	**Large**	**MIB**
Black	*Medium*	STW
Black	**Large**	**STW**
White	*Medium*	STW
White	**Large**	**STW**
Red	*Medium*	STW
Red	**Large**	**STW**
Blue	*Medium*	STW
Blue	**Large**	**STW**
Blue	**Small**	**STW**

The decomposition process can be continued until only empty subtables remain. The question, which propostion (value block) to decompose on next at each non-empty subtable will depend a suitable heuristic (see Section 4).

3.2 Variant Decision Diagram - *VDD*

A variant table can be represented as a decomposition tree. The root represents the entire table. It is labeled with a first chosen proposition $p(v_{j_1}, x_1)$. It has two children. One, termed the *left child*, represents the left sub-table $\mathcal{L}(\mathcal{T}, j_1, x_1)$. The other, termed the *right child*, represents the right sub-table $\mathcal{R}(\mathcal{T}, j_1, x_1)$. Each of these children can in turn have children if it can be decomposed further. An empty left child is labeled by a special symbol $\bot$. An empty right child is labeled by a special symbol $\top$. (All leaves of a decomposition tree are labeled either by $\bot$ or $\top$.) Figure 1 shows a graphic depiction[7]. It also shows the further decomposition of $\mathcal{R}(\mathcal{T}, j_1, x_1)$, here indicating that it has two empty children.

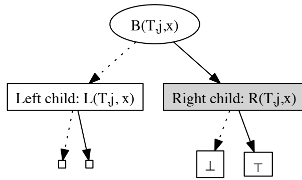

Figure 1. Basic scheme of a decomposition

Identical subtables may arise at different points in the decomposition tree. A goal of minimal representation is to represent these multiple occurrences only once by transforming the decomposition tree into a directed acyclic graph (*DAG*), which I call a *VDD* or *variant decision diagram*. All leaves can be identified with one of two predefined nodes also labeled $\bot$ and $\top$. Subsequently, all nodes labeled by the same proposition $p(v_j, x)$ that have identical children are represented by re-using one shared node. This reduction can be accomplished by an algorithm in the spirit of *Algorithm R* in [12].

[7] For simplicity in creating the graph, the label of the root node is given as $B(T, j, x)$ for $p(v_{j_1}, x_1)$

Figure 2 shows an entire VDD[8].[9]

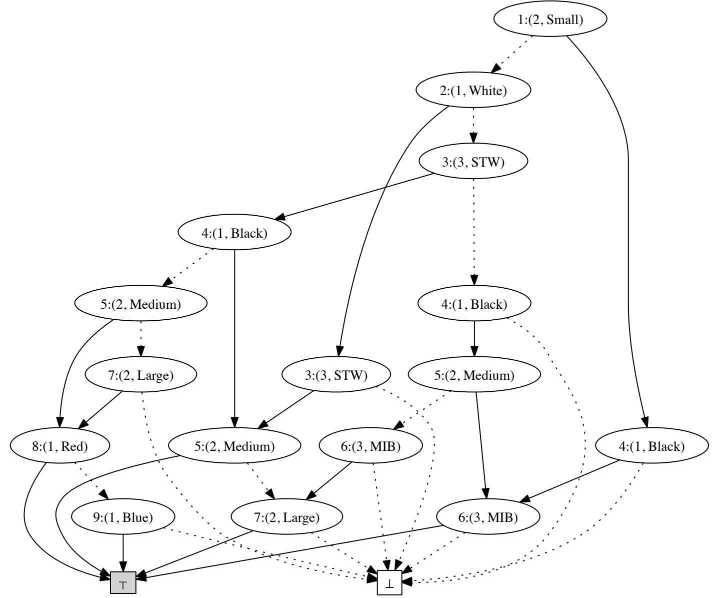

Figure 2. VDD of t-shirt using heuristic $h1$ (Section 4)

In Figure 2 each node is labeled in the form $\langle p : (j, val)\rangle$, where (j, val) is the column/value pair that denotes the proposition $p(v_j, x)$, and p is a unique identifier for the node/proposition. The identifiers are contiguously numbered. Thus, the set of propositions for $\mathcal{T}$ can be seen as totally ordered according to this numbering. The ordering underlying the graph in Figure 2 is

$$p(v_2, Small), p(v_1, x)White, p(v_3, x)STW, p(v_1, Black),$$
$$p(v_2, Medium), p(v_3, MIB), p(v_2, Large), p(v_1, Red),$$
$$p(v_1, Blue)$$

Given that such a total ordering can be identified in the decomposition, the resulting VDD may be seen as a ZDD (*zero-suppressed decision diagram*) [12]. The gist of the algorithms for evaluation of BDDs and their cousins given in [12], such as *Algorithm C* and *Algorithm B* would apply. However, I have not made any verbatim use of these so far.[10]

VDDs have certain special characteristics beyond ZDDs. A terminal *HI*-link always leads to $\top$, and a terminal *LO*-link always leads to $\bot$. This and further characteristics that are ensured by the heuristic $h2$ given in Algorithm 1 allow certain short-cuts in the implementation (see Sections 4 and 6).

[8] By conventions established in [12], I term a link to the left child as a *LO*-link, drawn with a dotted line, preferably to the left, and a link to the right child as a *HI*-link, drawn with a filled line, preferably to the *right*. A *LO*-link is followed when the proposition in the node label is disbelieved. A *HI*-link is followed when it is believed. The terminal nodes $\bot$ and $\top$ are called *sinks*

[9] The amount of compression achieved in figure 2 is not overwhelming. The heuristic $h2$ does better (see figure 3)

[10] Both heuristics $h1$ and $h2$ in Section 4 are designed to guarantee such an ordering, but this is doesn't seem essential to the VDD approach in general

3.3 Set-labeled Nodes

There is an additional reduction of a VDD I have implemented as an option. This is most easily described using the concept of an *l-chain*. Define the subgraph composed of a node and all its descendent nodes that can be reached from it following only *LO*-links as the l-chain of the node[11]. Nodes in an l-chain that pertain to the same column and have the same right child can be joined into a single node. Let $p(v_{j_1}, x_1), \ldots, p(v_h, x_h)$ be the propositions in the labels of members in an l-chain that can be joined. The resulting combined node is labeled with the disjunction of these propositions: $P := p(v_{j_1}, x_1) \vee \ldots \vee p(v_h, x_h)$. This disjunction is represented, for short, by the (non-atomic) set of values $X := \bigcup_{i=1}^{h} x_h$ occurring in P. In the sequel, I refer to a node by

$$\nu(j, X) \tag{7}$$

where j is the column index of the referenced product property v_j, and X is the set of values represented by the node[12]. In case the node label is a single atomic value $\{x\}$, I also denote this by $\nu(j, x)$.

Figure 3 is a graph of the t-shirt using set-labeled nodes[13]. It is not a reduction of Figure 2, but rather a reduction of the graph in Figure 4 (see Section 4).

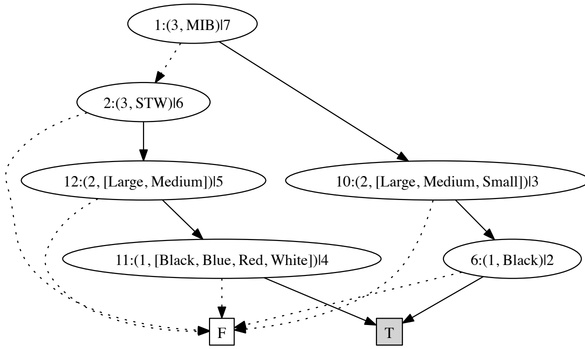

Figure 3. VDD of t-shirt with set-labeled nodes

By inspection, the complexity of the graph in Figure 3 is comparable to that obtained for the merged MDD in [2] (Figure 2 (b) there).

This further reduction is important from the viewpoint of compression, as each path from the root of the graph to a sink $\top$ can be seen as a c-tuple in the solution of $\mathcal{T}$, and a set-labeled node reduces the number of such c-tuples. It is also important from the viewpoint of run-time performance, if set intersection (*intersectsp*) is more efficient than multiple membership tests. A VDD using set-labeled nodes should result in similar c-tuples as the approach in [10] (depending of course on the heuristic). A key difference is that VDD paths may share nodes (c-tuples sharing common tails)[14], whereas this is not the case for a set representation of c-tuples. Hence, a VDD is a more compact representation.

Also, there may be external sources for set-labeled nodes if the maintained variant table is not relational. The SAP VC allows modeling variant tables with cells that contain real-valued intervals as well as a condensed form, where a cell may contain a set of values. Such cells can be directly represented as set-labeled nodes.

I close this section in noting that in [10] a Boolean function as in (4) is also used to construct a decomposition of a table into disjoint union of Cartesian products (c-tuples). There the resulting decomposition into c-tuples is the goal. Here the VDD is the goal, as I base the evaluation on it (see Section 8).

4 Heuristics

For the exposition in this section, I assume $\mathcal{T}$ to be in relational form with arity k.

The graph in Figure 2 is derived using on an initial heuristic $h1$, which I tried. $h1$ is based on trying to minimize splitting value blocks during decomposition. A value block for a proposition $p(v_j, x)$ is the rows in a table $\mathcal{T}$ that reference that proposition. Decomposing $\mathcal{T}$ on some other proposition $p(v_h, y)$ will split $p(v_j, x)$, if it has rows in both $\mathcal{L}(\mathcal{T}, j, y)$ and $\mathcal{R}(\mathcal{T}, j, y)$. Subsequently, both of these children need to be decomposed by $p(v_j, x)$, whereas a single decomposition would have handled $p(v_j, x)$ for $\mathcal{T}$ at its root level. It is assumed that keeping large value blocks intact is good, and the order of decomposition decisions should incur as little damage to these value blocks as possible. In order to apply this heuristic, all value blocks, their sizes, and the row indices that pertain to each one are initially calculated and stored. I do not go into further detail on this here.

The current implementation relies on characteristics of a VDD ensured by the decomposition heuristic $h2$ given in Algorithm 1. Recall that the total number of propositions is s, defined by (3), and that the number of propositions that pertain to the j-th column is s_j, defined by (2).

Algorithm 1 *(Heuristic $h2$)*
First, define $\boldsymbol{P}(\mathcal{T})$ as an ordered set of all s propositions $p(v_j, x)$ by

1. *Sorting the k columns of $\mathcal{T}$ by s_j, ascending (largest values last). $p(v_j, x)$, below, refers to the j-th column with respect to this ordering of the columns*
2. *Within each column, sort the values by their business order (any defined order the implementation can readily implement). x_{pj} refers to the p-th value in the j-th column domain $\pi_j(\mathcal{T})$.*

Then,

1. *Make a root node of the VDD for the first proposition in the first column: $p(v_1, x_{11})$*
2. *While nodes with a non-terminal subtable remain: split them always using the first proposition (value block) in their first column*
3. *Optionally, collect members of an l-chain with the same child nodes into an aggregated set-labeled node. I refer to this variant of the heuristic as $h2^*$*
4. *Reduce the nodes by unifying equivalent nodes as discussed in Section 3.2*

Figure 4 shows the graph of Table 1 produced using Algorithm 1 without set-labeled nodes ($h2$). Figure 3 shows the same graph produced with set-labeled nodes ($h2^*$).

A decomposition based on Algorithm 1 has the following characteristics:

- After k *HI*-links the $\top$-sink is always reached. (This is trivially true for all VDDs, as each row consists of k elements)

[11] A maximal l-chain would be one for a node which does not itself occur as the left child of any other node

[12] For the exposition, here, X represents a finite disjunction of propositions. In the implemented prototype it can also be an interval with continuous bounds

[13] The new set-labeled nodes are assigned a uniquely identifying node number outside the range used for numbering the propositions

[14] Figure 5 has shared nodes

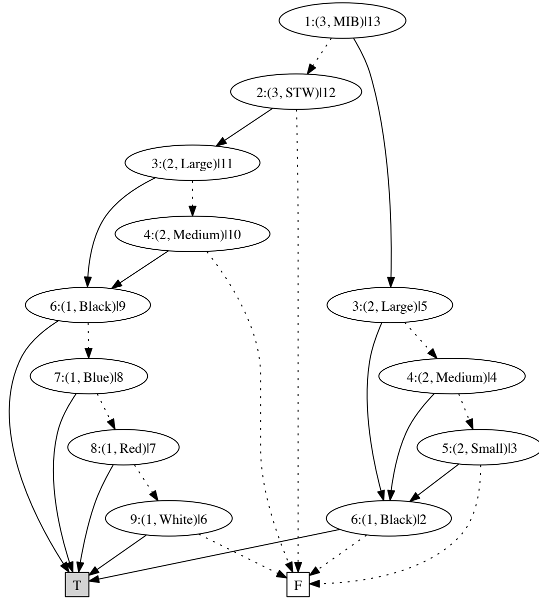

Figure 4. Basic VDD of t-shirt using algorithm 1

- All nodes in an l-chain (i.e., linked via *LO*-links) will always pertain to the same column. This follows from the fact that the columns of any non-empty left subtable of a table $\mathcal{T}$ are the same as the columns of $\mathcal{T}$. The heuristic says to always choose from the first column
- A node pertaining to the jth column is always $(j-1)$ *HI*-links distant from the root node. This follows by iterating the argument that if a table $\mathcal{T}$ is decomposed by a proposition $p(v_1, x)$ referencing its first column, then the right sub-table $\mathcal{T}$ has the second column of $\mathcal{T}$ as its first column (by construction))

Note that for BDDs an optimal ordering of $\boldsymbol{P}(\mathcal{T})$ is important to achieve a minimal graph. Thus, the search space for finding this is $s!$ (s factorial). In Algorithm 1 only the order of the columns is important. It is not important how the values are ordered within the column. To see this, note that the proposition $p(v_1, x_{11})$ used in decomposition *slices* $\mathcal{T}$ horizontally[15]. The slices obtained overall with respect to all values x_{p1} in the first column are the same, regardless of the order of the values in a column. Thus, the search space for an optimal column order is merely $k!$. As Algorithm 1 indicates, I am currently only exploring one ordering of columns, supposing that it will dominate the others. However, this still needs to be verified empirically.

5 Example of Extended T-shirt Model

In [8] I extended the t-shirt by adding the colors $Yellow$ and $DarkPurple$, the sizes XL and XXL, and the print $none$ to the global domains $\pi_j(\mathcal{T})$ (given in Section 2):

$$\pi_1(\mathcal{T}) = \{Black, Red, White, Blue, Yellow, DarkPurple\}$$
$$\pi_2(\mathcal{T}) = \{Large, Medium, Small, XL, XXL\}$$
$$\pi_3(\mathcal{T}) = \{MIB, STW, none\}$$

[15] The terminology is inspired by [6]

The new values combine with everything, except that no rows are added for combinations of MIB (print) and $Yellow$ (color). The table of all variants then has 73 rows (as opposed to the 11 of table 1)[16]. The graph is given in Figure 5

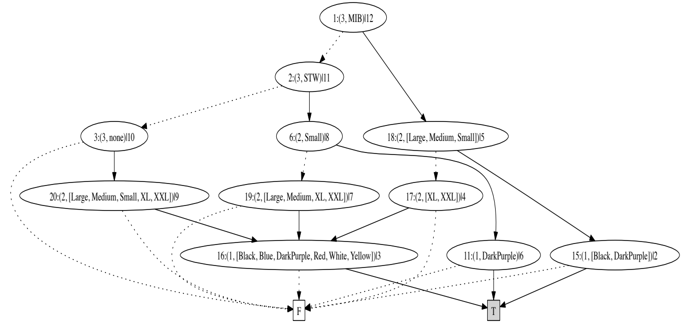

Figure 5. VDD of extended t-shirt with set-labeled nodes

Now 11 nodes are needed instead of the 6 nodes in Figure 3. The table is decomposed into 5 c-tuples. The node labeled $\langle 16 : (1, [Black, Blue, DarkPurple, Red, White, Yellow])\rangle$ is shared by three parents.

6 Empirical Compression Results

A prototype I have implemented in Java was tested for functional correctness against small exemplary tables such as the t-shirt model given in Table 1. This set of exemplary tables also contains some negative variant tables to test the approach in [8]. I refer to the implementation as the *VDD prototype*. It was further tested against 238 relational variant tables taken from three product models used at SAP in configurator maintenance issues. Since this data is proprietary, I give only summary statistics on the results. Testing on publicly available models, such as the Renault model [1] is a next step for future work.

It proved possible to successfully compile all 238 tables in this test base with all of the following three approaches:

- the heuristic $h1$ used to obtain the graph in Figure 2
- the heuristic $h2$ in Algorithm 1 - without merging nodes to set-labeled nodes
- the heuristic $h2^*$ in Algorithm 1 - with merging nodes to set-labeled nodes

Table 2 gives statistics on the table size and complexity. It lists the *minimal*, *maximal*, and *average* values, as well as the values for the four quartiles Q_1, Q_2, Q_3, Q_4, for each of the following parameters: k (*arity*), r (*number of rows*), s (*number of propositions*), and N (*number of cells* - $k * r$).

There is one table with arity one. This is used as a technique of dynamically defining a global domain for a product property in a variant table, rather than doing this directly in the declaration of the product property in the model. This technique has the disadvantage that it is more difficult to ensure translation of all relevant texts associated with a value in multi-lingual deployments of the configuration solution. It may, however, be used to dynamically restrict a large,

[16] I elaborate on the derivation of this example in [8]

Table 2. Size statistics on 238 SAP VC variant tables

Range	k	r	s	N
Minimum	1	1	2	2
Q_1	2	14	17	42
Q_2	3	56.5	46	176
Q_3	5	137.5	93.75	635.5
Q_4	16	21372	998	213720
Average	4.29	238.53	79.04	1590.51
Maximum	16	21372	998	213720

pre-defined, enterprise-wide global domain to the requirements of a particular product. General modeling considerations and experiences with the SAP VC are elaborated in [4].

The largest arity is 16. The associated table has only 76 rows. The largest table with 21372 rows has arity 10. The table with the largest number of propositions (998) has arity six and 469 rows.

Table 3 gives statistics on the achieved compression for the three compression techniques. (I discuss *double negation* separately, in Section 7.) The variant tables are partitioned into the four quartiles for s (*number of distinct node labels*[17]). These are denoted by Qs_1, Qs_2, Qs_3, and Qs_4. Averages are given for each of these four partitions for the following parameters: s, N (*number of cells in variant table*), n (*number of nodes*), $reduct$ (*reduction*: $(N - n)$), and t (*compilation time* in milli-seconds). Explicit results are also given for the table with largest number of cells (*Max* N), largest number of propositions (*Max* s), and largest arity (*Max* k), as well as the overall average.

Table 3. Average compression on 238 SAP VC variant tables

Range	N	*Heur*	s	n	$reduct$	t (msec)
Qs_1	42	$h1$	17	20	15.75	4
		$h2$	17	18	19.5	0
		$h2^*$	8	8	28.75	0
Qs_2	176	$h1$	46	73.5	95.5	17.5
		$h2$	46	65	102.5	1
		$h2^*$	17	19.5	155	1
Qs_3	635.5	$h1$	93.75	184	418.5	112.75
		$h2$	93.75	154.75	489.5	3
		$h2^*$	53.75	74.5	558.5	5
Qs_4	213720	$h1$	998	3756	213329	1192988
		$h2$	998	2728	213289	598
		$h2^*$	988	2381	213575	659
Average	1590.51	$h1$	79.04	178.95	1411.57	5475.59
		$h2$	79.04	149.95	1440.56	9.77
		$h2^*$	50.32	83.92	1506.59	12.43
Max N	213720	$h1$	152	391	54793	1192988
		$h2$	152	431	213289	598
		$h2^*$	70	145	213575	659
Max s	2814	$h1$	998	998	868	478
		$h2$	998	998	868	86
		$h2^*$	130	130	1736	91
Max k	1216	$h1$	181	885	331	874
		$h2$	181	653	563	5
		$h2^*$	182	649	567	5

The compilation times of $h2$ are drastically better for large tables than those for $h1$. This is because the information needed by $h1$ has some components that are non-linear to process, whereas $h2$ does not. Not surprisingly, using merged nodes further reduces both the number of nodes (n) in the VDD as well as the number of distinct labels of the nodes (s). The times are obtained on my Apple Mac mini with 2.5 GHz Intel Core i5 and 8GB memory. Times on other development PCs (both Windows and MacBook) are comparable. The time to compile the largest table with $h1$ is almost 20 minutes, but it takes less than one second using $h2$ with and without merging for the same table. Thus, compiling these tables into VDDs with $h2$ would almost be feasible at run-time.

Heuristic $h2$ strictly dominates $h1$ with respect to achieved compression (smaller number of nodes) for 143 of the 238 tables[18]. For 71 tables the same compression was achieved. For 24 tables $h1$ strictly dominates $h2$. Table 4 compares the advantages/disadvantages of $h2$ over $h1$. The three cases are labeled "$h2 > h1$" ($h2$ strictly dominates $h1$), "$h2 = h1$" (indifference), and "$h2 < h1$" ($h2$ is strictly dominated by $h1$). Table 4 lists averages for the following parameters: Tab, s, N , n, ΔR , and Δt. Tab is the number of tables that pertain to that case. The parameters s, N , and n are as defined above for Table 3. ΔR is the weighted difference in reduction: $\Delta R = Tab * (reduct_{h2} - reduct_{h1})$. It is positive where $h2$ has the advantage. Δt is the weighted difference in compile time: $\Delta t = Tab * (t_{h2} - t_{h1})$ in milli-seconds. It is negative where $h2$ has the advantage. The last row gives the averages per table $\Delta R/238$ and $\Delta t/238$ over all rows.

Table 4. Averages for dominated heuristics

Dominance	Tab	N	s	n	ΔR	Δt
$h2 > h1$	143	867.06	85.03	222.59	7702	-563065
$h2 = h1$	71	114.76	54.83	56.00	0	-1208
$h2 < h1$	24	10266.88	114.92	282.58	-992	-1206770
Average					28.19	-7446.43

The largest table is one where *h1* strictly dominates *h2*. However, the compile time using *h1* is almost 20 min. That for *h2* is 0.6 sec. Overall, $h2$ seems to prevail over $h1$. The average gain in reduction over all 238 tables is 28.19. The average gain in compilation time is 7446.43 (msec). The large gain in the latter is due to the non-linear performance of $h1$, which makes it grossly uncompetitive for large tables. Further experiments with other column orderings and with other heuristics remains a topic of future work.

7 Excursion: Negation

In [8] I describe *double negation* of a positive table as one approach to compression that is completely independent of the VDD mechanism. Being able to negate a table, i.e. calculate the complement with respect to a given domain restrictions is central in that approach.

My implementation of the VDD-prototype supports the approach in [8], and supports negation of a VDD. A BDD can be negated by switching the sinks $\bot$ and $\top$. This doesn't work for a VDD (and for a ZDD in general). Algorithm 2 gives the spirit of my implementation for negating a VDD produced using Algorithm 1 for a variant table $\mathcal{T}$ of arity k against a domain restriction tuple R. It produces a negated VDD that has set-labeled nodes.

[17] For VDDs without merged nodes this is just the number of propositions. For VDDs with merged nodes this is a different number, because labels for disjunctions of propositions are added, but not all original propositions still explicitly appear as node labels

[18] As merging could potentially also be done in conjunction with heuristic $h1$, it is not a fair comparison to compare the number of nodes achieved with $h1$ against that achieved with $h2^*$

Algorithm 2 *(Negation)*
Start with the root node of $\mathcal{V}$. Negate it as described below

- *If ν is a non-negated terminal node, i.e., ν references the last column index k, replace it with a node $\overline{\nu}$ that is assigned the complementary label $\overline{LC}(\nu) := \pi_j(\mathcal{T}) \setminus LC(\nu)$, where $LC(\nu)$ is defined as the union of all values/sets in the* l-chain[19] *of ν*
- *If ν is a non-negated non-terminal node that references column index $j < k$, negate it by doing the following in order:*
  - *negate each right child of each node in its l-chain in turn*
  - *add a node $\nu_\perp$ to the end of the l-chain of ν , where $\nu_\perp$ represents the c-tuple $\overline{LC}(\nu) \times R_{j+1} \times \ldots \times R_k$. $\nu_\perp$ itself is labeled with $\overline{LC}(\nu)$. Its right child is a node labeled R_{j+1} which has a right child $R_{j+2} \ldots$. All* LO*-links of added nodes point to $\perp$*

Prune any unneeded nodes that have an empty set as a label or have a right child that is pruned, suitably rerouting a link to a pruned node to the left child of the pruned node

The VDD prototype actually does the pruning of empty nodes on the fly. If the root node itself is pruned, the entire resulting VDD is empty after negation.

The fact that double negation of a positive table needs to yield the same solution set as the original table (see [8]) provides a straightforward possibility to test for the correctness of this approach to negation.

In order to avoid complexity issues with very large complements, I so far applied double negation only in those cases where the number of tuples in the complement was smaller or equal to the number of tuples in the original table. 57 of the 238 SAP VC variant tables that are the basis for the results I presented in Section 6 proved amenable to double negation in this sense. Of these, 18 did not yield a smaller VDD than that obtained with heuristic $h2^*$. For the remaining 39 the maximal gain was 4 nodes, the average gain was 1.89 nodes.

Run-time performance tests have yet to be made, but these results raise the question, whether double negation will add value over direct compression. However, the concept of double negation is independent of the VDD concept, and could be applied on its own without using VDDs. Also, it remains to be seen, if the test on whether constraint propagation can be gainfully applied, given in [8], proves valuable.

8 Evaluation of a VDD

Given a run-time restriction R, a VDD $\mathcal{V}$, and a node $\nu(j, X)$ in $\mathcal{V}$ (using the notation in (7)). $\nu(j, X)$ can be marked as *out* if $X \cap R_j = \emptyset$. The admissible solutions of $\mathcal{V}$ are all paths from the root node to the sink $\top$ that do not contain any node marked *out*. I denote this set as $\mathcal{V} \cap R$, for short. If there are no such paths, then R is inconsistent given $\mathcal{V}$.

I do not go into further detail on how to determine $\mathcal{V} \cap R$. This follows the spirit of known algorithms for directed acyclic graphs. For example, see [12] for an exposition in the context of BDDs/ZDDs.

Concerning the general complexity of the calculations:

- Let s_j be the number of distinct node labels pertaining to the j-th column of $\mathcal{V}$ (as in (2), but modified to allow for set-valued labels). s_j node-labels must be intersected with R_j for each column index j to determine admissibility of all occurring node labels. Given that the domains are naturally ordered, binary search can be used to speed this up. The VDD prototype also imposes an ordering on the node labels to facilitate this
- Let n be the number of nodes in $\mathcal{V}$. The question of which nodes have admissible paths to $\top$, is related to the problem of counting the admissible paths. After determining which node labels are admissible, this has the remaining complexity of $O(n)$ (c.f., *Algorithm C* in [12])

[19] Defined in Section 3.3, the *l-chain* of a node is the sub-graph consisting of the node and all nodes reachable from it via *LO*-links, but excluding the sink $\perp$. All nodes in an l-chain reference the same column index j

In the case that $\mathcal{V}$ is a VDD without set-valued nodes, i.e., all node labels are of the form $X = \{x\}$, $\mathcal{V} \cap R$ is just the solution set of $\mathcal{T} \cap R$, where $\mathcal{T}$ is the variant table encoded by $\mathcal{V}$. But, if $\mathcal{V}$ has non-atomic set-valued nodes[20] $\mathcal{T} \cap R \subseteq \mathcal{V} \cap R$. Here, the evaluation comes at the slight additional cost of determining the solution set by additionally intersecting $\mathcal{V} \cap R$ with R.[21] The intersection of two c-tuples is easy to calculate. Thus, this additional cost is offset, because determining the admissibility of each node is now faster (a smaller number of *intersectsp* tests hopefully offsets the otherwise greater number of *memberp* tests).

Real run-time measurements have not yet been performed. However, in the beginning, in trying to determine whether the VDD approach is worthwile, I did an experimental integration with the (Java-based) SAP IPC configurator (see [4]) with the product models encompassing the 238 tables mentioned in Section 6. The software configuration I used was completely non-standard, so any results are not objectively meaningful, but they did encourage me to pursue this approach. The expectations on performance gains might be roughly oriented on the inverse of the compression ratio N/n (*total number of table cells/number of nodes*). For heuristics $h1/h2/h2^*$ the averages for N/n are 2.2/2.4/3.9, respectively. But, this does not account for losses due to the overhead of needing more complex set operations. In any case, real run-time measurements need to be performed as a future step.

I close this section with a remark on using a VDD as a simple database. A query with an instantiated database key is equivalent to a run-time restriction R, where the key's properties are restricted to singleton values, and the other properties are *unconstrained* (or have the domain that is established at the time of the query). The solution set of the VDD will contain only tuples consistent with the query (by construction). If, for example, the key is defined as unique in the database (and the table content is consistent with this definition), the result can contain at most the unique response (or the empty set, if no row for the key exists in the table). For queries with non-unique database keys, the solution set needs to be intersected with R, as discussed above.

9 Conclusion

Although it remains to explore other variants of Algorithm 1 with different orderings of the columns, the compression achieved with the current version (both $h2$ and $h2^*$) has been surprisingly satisfactory. The very short compile times may be of more practical advantage than a more expensive search for heuristics that provide (somewhat) better results. However, further investigation into search and heuristics of an altogether different type should to be done more completely and formally. This is future work.

[20] Either merged nodes or nodes representing continuous intervals

[21] To see this, suppose that $\mathcal{T}$ is itself a c-tuple, so that there is at most one admissible c-tuple consisting of $\mathcal{T}$ itself. Obviously R may be more restrictive

The goal of this work was not only to find a good compression technique, but also to provide a solution that can be used in conjunction with a legacy configurator to enhance the handling of variant tables, either as a whole or individually. The VDD prototype I implemented uses little machinery and adds little to software maintenance (training, size of the code base, etc.). It also conveys little risk. In the event that some tables cannot be compiled to a VDD, the legacy handling can be seamlessly kept. (This did not occur in the initial explorations using the $h1$ heuristic with the test models.) The SAP VC is a configurator with a very large user base. Thus, any change to it comes with large risk. This is the type of situation I had in mind when designing the VDD prototype.[22]

In the course of this work I have come to believe that all of the following three approaches to speed up variant table handling and to look for a compact representation yield very similar results:

- BDDs in various flavors ([9, 2])
- Compression into c-tuples and constraint slicing ([6, 10])
- Read optimized databases (such as column-oriented databases ([14])) in conjunction with a constraint propagation algorithm (e.g. the STR-algorithm [13])

The differences are in the machinery needed for the intended deployment and in the heuristics that suggest themselves. Although the representation as a VDD is central to my approach at compression and evaluation, functionally, the two important aspects in practice are that it functions as a (limited) replacement for a database, and that it performs constraint propagation. I would see as a topic of future work to both look more closely at read optimized databases and to investigate, if the VDD approach can be extended to support more complex database queries. Investigating the commonality between the three approaches (BDDs, compression, read-optimized databases) more formally could be another interesting topic. The VDD approach has elements of all three.

A note in closing: The compression algorithm in [10] is based on a very similar approach to table decomposition. I was not aware of this work until recently, so there are some unfortunate disconnects in conventions I use. I tend to follow [12], whereas in [10] left and right are used the other way around. I did adopt use of the term c-tuple. I think the column oriented view, here, is more intuitive and has resulted in more useful heuristics. Another major difference to [10], which I see, is that the I base evaluation directly on the VDD. Furthermore, the VDD supports nodes labeled with real-valued intervals, but his could also be added to [10] in a straightforward manner[23].

ACKNOWLEDGEMENTS

I would like to thank the anonymous reviewers and my daughter Laura for their constructive suggestions, on how to improve the intelligibility of this paper. I am afraid the current result may not yet meet their expectations, but I think it is a step forward from the previous version.

REFERENCES

[1] J. Amilhastre, H. Fargier, and P Marquis, 'Consistency restoration and explanations in dynamic csps application to configuration', *Artif. Intell.*, **135**(1-2), 199–234, (2002).

[2] H.R. Andersen, T. Hadzic, and D. Pisinger, 'Interactive cost configuration over decision diagrams', *J. Artif. Intell. Res. (JAIR)*, **37**, 99–139, (2010).

[3] C. Bessiere, 'Constraint propagation', in *Handbook of Constraint Programming*, eds., F. Rossi, P. van Beek, and T. Walsh, chapter 3, Elsevier, (2006).

[4] U. Blumöhr, M. Münch, and M. Ukalovic, *Variant Configuration with SAP, second edition*, SAP Press, Galileo Press, 2012.

[5] K. Ferland, *Discrete Mathematics*, Cengage Learning, 2008.

[6] Nebras Gharbi, Fred Hemery, Christophe Lecoutre, and Olivier Roussel, 'Sliced table constraints: Combining compression and tabular reduction', in *CPAIOR'14*, pp. 120–135, (2014).

[7] A. Haag, 'Chapter 27 - product configuration in sap: A retrospective', in *Knowledge-Based Configuration*, eds., Alexander Felfernig, Lothar Hotz, Claire Bagley, and Juha Tiihonen, 319 – 337, Morgan Kaufmann, Boston, (2014).

[8] A. Haag, 'An approach to arc consistency with negative variant tables', in *Proceedings of the 17th International Configuration Workshop, Vienna, Austria, September 10-11, 2015.*, pp. 81–87, (2015).

[9] Tarik Hadzic, 'A bdd-based approach to interactive configuration', in *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, ed., Mark Wallace, volume 3258 of *Lecture Notes in Computer Science*, p. 797. Springer, (2004).

[10] G. Katsirelos and T. Walsh, 'A compression algorithm for large arity extensional constraints', in *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, ed., Christian Bessiere, volume 4741 of *Lecture Notes in Computer Science*, pp. 379–393. Springer, (2007).

[11] H. Keller, *The Official ABAP Reference*, number Bd. 1 in Galileo SAP Press, Galileo Press, 2005.

[12] D.E. Knuth, *The Art of Computer Programming*, volume 4A Combinatorial Algorithms Part 1, chapter Binary Decision Diagrams, 202–280, Pearson Education, Boston, 2011.

[13] C. Lecoutre, 'STR2: optimized simple tabular reduction for table constraints', *Constraints*, **16**(4), 341–371, (2011).

[14] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik, 'C-store: A column-oriented DBMS', in *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, eds., Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, pp. 553–564. ACM, (2005).

[22] In my estimation, the effort to reimplement the current VDD functionality in SAP ABAP ([11]) for use with the SAP VC would be feasible from a cost and risk perspective

[23] Basically treating an interval syntactically like a value in compilation, but evaluating it like a set at run-time