Uncovering Functional Dependencies in MDD-Compiled Product Catalogues

Tarik Hadžić and **Barry O'Sullivan**
Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{t.hadzic|b.osullivan}@4c.ucc.ie

Abstract

Product catalogues are usually represented as tables. They are regularly updated with new variants and, over time, various forms of inconsistencies or undesirable properties can be introduced, especially when global changes are made. We argue that by compiling product catalogues into decision diagrams we can support a number of high-level queries for detecting and checking for various forms of inconsistency, as well as verifying other properties relevant for user interaction. In particular, a class of functional dependency-based inconsistencies can be detected efficiently. An example is verifying properties such as: *"each identical configuration of a product has the same price"*. This paper presents a number of algorithmic advances. We illustrate the usefulness of our approach by evaluating these high-level properties over real-world publicly available product catalogues.

1 Introduction

Uncovering functional dependencies is an important problem in many artificial intelligence (AI) domains. Many AI datasets are represented in tabular form, defined in terms of a set of attributes (columns). A large dataset might be represented in a database table or spreadsheet and determining that one or more attributes is functionally determined by values of other attributes can be of critical importance, e.g. in analyzing the effect of chemical compounds on cancerogenity or studying the shopping habits of the customers. Functional dependencies are most commonly used in the process of designing relational database schema.

We suggest that uncovering functional dependencies can also be useful in an online product configuration system context. For example, various forms of *catalogue consistencies* can be expressed as functional dependencies. For example, checking whether "each identical configuration of a product has the same price" reduces to verifying that the price attribute is functionally determined by the remaining attributes. This could be relevant for product catalogues that are regularly updated with new variants, or preprocessed for various purposes. Furthermore, functional dependencies can help to reason about the length of user interaction with respect to the design of the user interface. If a small subset of attributes functionally determines all the others, then an interface that encourages a user to first assign attributes from such a subset, might help reduce the total number of interaction steps.

In this paper we study the problem of identifying and testing functional dependencies amongst subsets of the attributes X that define a catalogue. Specifically, we consider dependencies of the form $Y \rightarrow x$ for a subset of variables $Y \subset X$ and variable $x \in X$, which hold if and only if for any two products p_1, p_2 from the catalogue, whenever p_1 and p_2 agree on attributes Y, i.e. $p_1[Y] \equiv p_2[Y]$, they also agree on attribute x, i.e. $p_1[x] \equiv p_2[x]$.

A number of approaches have been suggested in the database community for uncovering functional dependencies [Huhtala *et al.*, 1999]. However, all of them find dependencies by operating over an explicit representation of the catalogue, and state-of-the art approaches take linear time, $\mathcal{O}(T)$, in the number of products, T, in the catalogue [Huhtala *et al.*, 1999; Schlimmer, 1993]. Other approaches incur even more complexity by using sorting, in $\mathcal{O}(T \cdot log(T))$ time, or explicit comparison of all tuples, in $\mathcal{O}(T^2)$ time. Note that the number of tuples can grow exponentially with the number of attributes, however most real world catalogues are very sparse and contain only a tiny proportion of all possible products.

The main contribution of this paper is an approach to uncovering functional dependencies when a dataset is compressed into a *multi-valued decision diagram* (MDD) rather than as an extensionally represented table, the standard in relational database theory. MDDs are directed acyclic graphs that, for a sorted list of attributes, use prefix and suffix sharing to compactly represent the data. Each product is encoded as a path, and every edge on the path encodes an attribute-value pair (variable assignment), $x_i = a$. While the size of an MDD is, in the worst-case, linear in the number of products, they can often be *exponentially* smaller. We propose a number of algorithms for uncovering functional dependencies; these algorithms have time complexity that is quadratic in the size of the MDD. In worst case, when there is no compression, we get $\mathcal{O}(T^2)$ running time. However, whenever the MDD is small we guarantee a sublinear-time algorithm for testing $Y \rightarrow x$.

We implemented the algorithms we have introduced in this paper and applied them to several publicly available product catalogues. As a result, we have uncovered several properties, some of which indicate "bugs" in the datasets which have previously gone undetected.

The remainder of this paper is organised as follows. Section 2 presents the necessary technical background required throughout the paper. We show how functional dependencies can be uncovered in decision diagram representations of product catalogues in Section 3. A variety of specialised functional dependencies, called subset-induced dependencies are presented in Section 4. We define the notion of approximate dependencies in Section 5, which are inexact forms of standard functional dependencies. We report on a number of experiments in Section 6. Section 7 presents a number of concluding remarks and outlines directions for future work.

2 Background

In this section we provide the necessary background on preliminary concepts and introduce our notational conventions.

2.1 Solution Sets

We are given a set of variables (product attributes) $X = \{x_1, \ldots, x_n\}$ defined over finite domains $D_1, \ldots, D_n$ of possible values of the attributes of a product. We are also given a set of solutions (available products) $Sol \subseteq D_1 \times \ldots, D_n$, which can be defined either explicitly, e.g. as a set of items in a product catalogue, or implicitly, e.g. as the set of solutions to a constraint satisfaction problem $\langle X, D, F =_{\text{def}} \{c_1, \ldots, c_m\}\rangle$ defining which products can be manufactured in a factory. The latter approach is common when defining a catalogue of configurable products. Consider for example the following implicit representation of a T-shirt catalogue.

Example 1 (Representing a T-Shirt Catalogue)**.** *We are interested in selecting a T-shirt which is defined by three attributes: the color (black, white, red, or blue), the size (small, medium, or large) and the print ("Men In Black" - MIB or "Save The Whales" - STW). There are two rules that define the set of valid combinations: if we choose the MIB print then the color black has to be chosen as well, and if we choose the small size then the STW print (including a big picture of a whale) cannot be selected as the picture of a whale does not fit on the small shirt. The implicit representation* (X, D, F) *of the T-shirt example consists of variables* $X = \{x_1, x_2, x_3\}$ *representing color, size and print. Variable domains are* $D_1 = \{0, 1, 2, 3\}$ $(black, white, red, blue)$, $D_2 = \{0, 1, 2\}$ $(small, medium, large)$, *and* $D_3 = \{0, 1\}$ (MIB, STW). *The two rules translate to* $F = \{f_1, f_2\}$, *where* f_1 *is* $x_3 = 0 \Rightarrow x_1 = 0$ $(MIB \Rightarrow black)$ *and* f_2 *is* $(x_2 = 0 \Rightarrow x_3 \neq 1)$ $(small \Rightarrow not\ STW)$. ♢

The same solution space of the T-shirt example (satisfying the configuration rules F) can be also given explicitly as a set of items in a catalogue, as shown in Table 1.

Table 1: Solution set for the T-shirt example.

color	**size**	**print**
black	*small*	*MIB*
black	*medium*	*MIB*
black	*medium*	*STW*
black	*large*	*MIB*
black	*large*	*STW*
white	*medium*	*STW*
white	*large*	*STW*
red	*medium*	*STW*
red	*large*	*STW*
blue	*medium*	*STW*
blue	*large*	*STW*

2.2 Decision Diagrams

Decision diagrams are compressed representations of solution sets $Sol \subseteq D_1 \times \ldots \times D_n$. Formally, decision diagrams are a family of rooted directed acyclic graphs (DAGs) where each node u is labeled with a variable x_i and each of its outgoing edges e are labeled with a value $a \in D_i$. The decision diagram contains one or more *terminal* nodes, each labeled with a constant and having no outgoing edges. The most well known member of this family is the *binary decision diagram* (BDD) [Bryant, 1986] which is used as a compressed representation of Boolean functions in many areas, such as verification, model checking, VLSI design [Meinel and Theobald, 1998; Wegener, 2000; Drechsler, 2001], etc. In this paper we will primarily operate with the following variant, called *multi-valued decision diagrams*:

Definition 1 (Multi-valued Decision Diagram)**.** *An MDD* M *is a rooted directed acyclic graph* (V, E), *where* V *is a set of vertices containing the special terminal vertex* $\mathbf{1}$ *and a root* $r \in V$. *Further,* $var : V \rightarrow \{1, \ldots, n+1\}$ *is a labeling of all nodes with a variable index such that* $var(\mathbf{1}) = n + 1$. *Each edge* $e \in E$ *is denoted with a triple* (u, u', a) *of its start node* u, *its end node* u' *and an associated value* a.

We work only with *ordered* MDDs. A total ordering $<$ of the variables is assumed such that for all edges (u, u', a) $var(u) < var(u')$. For convenience we assume that the variables in X are ordered according to their indices. Ordered MDDs can be considered as being arranged in n *layers* of vertices, each layer being labeled with the same variable index. We will denote as V_i the set of all nodes labeled with x_i, $V_i = \{u \in V \mid var(u) = i\}$. Similarly, we will denote with E_i the set of all edges originating in V_i, i.e. $E_i = \{e(u, u', a) \in E \mid var(u) = i\}$. Unless otherwise specified, we assume that on each path from the root to the terminal, every variable labels exactly one node. An MDD encodes a CSP solution set $Sol \subseteq D_1 \times \ldots \times D_n$, defined over variables $\{x_1, \ldots, x_n\}$. To check whether an assignment $\mathbf{a} = (a_1, \ldots, a_n) \in D_1 \times \ldots \times D_n$ is in Sol we traverse M from the root, and at every node u labeled with variable x_i, we follow an edge labeled with a_i. If there is no such edge then $\mathbf{a}$ is not a solution $\mathbf{a} \notin Sol$. Otherwise, if such a traversal eventually ends in terminal $\mathbf{1}$ then $\mathbf{a} \in Sol$. We will denote with $p : u_1 \rightsquigarrow u_2$ any path in MDD from u_1 to u_2. Also, edges between u and u' will be sometimes denoted as $e : u \rightarrow u'$. A value a of an edge $e(u, u', a)$ will be sometimes denoted as $v(e)$, while a partial assignment associated with path p will be denoted as $v(p)$. We will use $Ch[u]$ to denote the set of all outgoing (children) edges of node u. Every path corresponds to a unique assignment. Hence, the set of all solutions represented by the MDD is $Sol = \{v(p) \mid p : r \rightsquigarrow \mathbf{1}\}$. In fact, every node $u \in V_i$ can be associated with two subsets of solutions. $Sol(u) = \{v(p) \mid p : u \rightsquigarrow \mathbf{1}\} \subseteq D_i \times \ldots \times D_n$, and $Sol(r, u) = \{v(p) \mid p : r \rightsquigarrow u\} \subseteq D_1 \times \ldots \times D_{i-1}$.

Consider the MDD in Figure 1. It represents directly the solution set of T-shirt catalogue from Table 1. For each node we indicate a unique identifier u_i. All nodes in the same layer correspond to the same variable. Node u_1 is the root node. Nodes in the first, second and third layer are $V_1 = \{u_1\}$, $V_2 = \{u_2, u_3, u_4, u_5\}$ and $V_3 = \{u_6, u_7, \ldots, u_{14}\}$ respectively. For each edge e we indicate a value $v(e)$. The set of outgoing edges from, for example, node u_2 is $Ch[u_2] = \{(u_2, u_6, 0), (u_2, u_7, 1), (u_2, u_8, 2)\}$. The solution set associated with, for example, node u_3 is the set of partial assignments $Sol(u_3) = \{(1, 1), (2, 1)\}$. There are in total eleven paths from u_1 to $\mathbf{1}$, corresponding directly to the eleven products in Table 1.

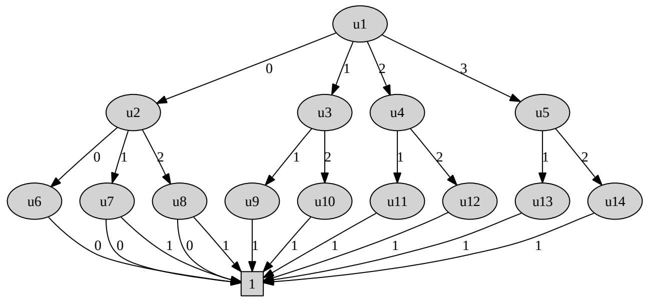
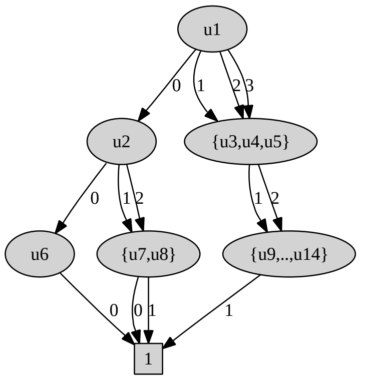


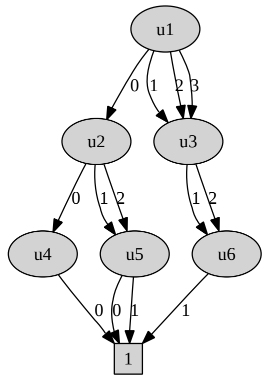
Figure 1: An MDD for the T-shirt catalogue.

The MDD from Figure 1 is as large as explicit representation in Table 1 - the number of edges is equal to the number of attribute-value pairs. However, the critical benefit of decision diagrams is that they can become exponentially smaller than the size of solution set they encode by *merging isomorphic subgraphs*. Two nodes u_1, u_2 are isomorphic if they encode the same solution set $Sol(u_1) = Sol(u_2)$. In Figure 2 we show equivalent merged MDDs for the T-shirt solution set. For the sake of clarity, we first indicate how the nodes have been merged in Figure 2(a) by using the same unique node identifiers from Figure 1. For example, nodes $\{u_3, u_4, u_5\}$ have been merged since they have the same solution sets $\{(1, 1), (2, 1)\}$. The same merged MDD, with new unique node identifiers, is shown in Figure 2(b). The utility of compressing product catalogues has already been demonstrated in [Nicholson *et al.*, 2006]. In this paper, unless emphasized otherwise, by MDD we always assume an ordered merged MDD.

On the Size and Construction of Merged MDDs. Given a variable ordering there is a unique merged MDD for a given solution set. The size of the MDD depends critically on the ordering, and could vary exponentially. The merged MDD representation of a solution set Sol with T entries defined over n attributes, $|Sol| = T$, can have at most $T \cdot n$ edges. However, data often exhibits sharing as illustrated in the T-shirt example, and a merged MDD might be exponentially



(a) A merged MDD with old unique identifiers from Figure 1 indicating how they were aggregated.



(b) A merged MDD with renamed unique node identifiers.

Figure 2: Merged MDDs for the T-Shirt example. Both graphs indicate the same structure. In Figure 2(a), for the sake of clarity, we indicate how the nodes from the uncompressed MDD in Figure 1 have been aggregated to get the MDD in Figure 2(b).

smaller. In the above example, we reduced the number of edges from 33 to 13. The effect of merging is best illustrated by considering two extreme cases. An MDD for solution set $D_1 \times \ldots \times D_n$ contains only n internal nodes and $\sum_{i=1}^{n} |D_i|$ edges while there are exponentially more solutions $T = |D_1| \cdot \ldots \cdot |D_n|$. In contrast, an MDD where every two edges at each layer are labeled with a different value is guaranteed to provide no sharing of nodes, and it would contain $T \cdot n$ nodes.

An MDD can always be constructed for a given catalogue in linear time $\mathcal{O}(T)$. We start with a direct representation, such as the T-shirt MDD in Figure 1, and in a single bottom-up pass detect and merge those nodes that root identical solution spaces. However, MDDs can be constructed also from implicit description in the form of constraint satisfaction problem. We first create an MDD M_i for each constraint c_i, and then use pairwise conjunctions to construct $M_1 \wedge \ldots \wedge M_m$. Each conjunction of two MDDs can be performed in quadratic time and space. The size of an MDD can grow exponentially in the number of variables, but in practice, for many interesting constraints the size is surprisingly small.

2.3 Functional Dependencies

Given solution set Sol defined over variables $X = \{x_1, \ldots, x_n\}$, and given solution $\mathbf{a} \in Sol$, $\mathbf{a} = (a_1, \ldots, a_n)$, we define the projection of solution $\mathbf{a}$ on variable x_i, denoted as $\mathbf{a}[x_i]$, to be the value of the i-th coordinate in the tuple, $\mathbf{a}[x_i] =_{\text{def}} a_i$. Similarly, we define projection of $\mathbf{a}$ onto a subset of variables $Y \subseteq X$, denoted as $\mathbf{a}[Y]$, as a tuple of values corresponding to variables in Y. Finally, for a given set of solutions $S \subseteq Sol$, we define the projection on subset of variables Y, denoted as $S[Y]$, as a collection of all projected tuples, $S[Y] =_{\text{def}} \{\mathbf{a}[Y] \mid \mathbf{a} \in S\}$.

For a solution set Sol, defined over variables $X = \{x_1, \ldots, x_n\}$ we say that a variable x_i is *functionally deter-*

mined by a subset of variables $Y \subseteq X$, denoted as $Y \rightarrow x_i$, if for any two solutions $\mathbf{a}_1, \mathbf{a}_2 \in Sol$, whenever $\mathbf{a}_1$ and $\mathbf{a}_2$ agree on variables Y ($\mathbf{a}_1[Y] = \mathbf{a}_2[Y]$), they also agree on variable x_i ($\mathbf{a}_1[x_i] = \mathbf{a}_2[x_i]$). Formally:

$$Y \rightarrow x_i \Leftrightarrow_{\text{def}} \forall_{\mathbf{a_1},\mathbf{a_2} \in Sol}\, \mathbf{a}_1[Y] = \mathbf{a}_2[Y] \Rightarrow \mathbf{a}_1[x_i] = \mathbf{a}_2[x_i].$$

A number of approaches are known in the database community for uncovering all minimal functional dependencies. A core operation that is executed repeatedly is testing for atomic functional dependencies of the form $Y \rightarrow x_i$. State-of-the art approaches for testing atomic dependencies first cluster data in equivalence classes with respect to the value of x_i, and then make multiple linear iterations through the dataset. This incurs linear complexity in the number of solutions $\mathcal{O}(T)$ [Huhtala *et al.*, 1999; Schlimmer, 1993]. Other approaches incur even more complexity, using sorting $\mathcal{O}(T \cdot log(T))$ or explicit comparison of all tuples $\mathcal{O}(T^2)$.

Application to Recommendation and Configuration

Detecting functional dependencies has applications in many areas. For example, uncovering that an attribute is functionally determined by values of other attributes can be of critical importance in analyzing the effect of chemical compounds on cancerogenity, studying the shopping habits of customers, etc. In this paper however, we suggest that uncovering functional dependencies can also be useful in a recommendation and interactive configuration context.

Firstly, various forms of *catalogue consistencies* can be expressed as functional dependencies. If product catalogues are frequently updated by the addition, removal or change of its items, over time various forms of inconsistencies or undesirable properties might be introduced. In particular, if care is not taken, a change of pricing policy might result in the addition of an item to the dataset that is already present in the catalogue but differs in price. Furthermore, catalogue datasets are often transformed or preprocessed for various forms of analysis or communication. Such transformations might involve the removal of "redundant" attributes, such as textual descriptions. If care is not taken, non-redundant attributes might be removed as well, thus influencing the soundness of the results of the analysis performed over the processed dataset. Thus, analyzing functional dependencies can help detect possible inconsistencies. As an illustration, checking whether "each identical configuration of a product has the same price" reduces to verifying that the price attribute is functionally determined by the remaining attributes.

Secondly, functional dependencies can help us reason about the length of user interaction with respect to the design of the user interface. If a subset of attributes $Y \subset X$ functionally determines all the other variables, $Y \rightarrow X$, then a user is guaranteed to completely specify the product as soon as all the variables Y are assigned. Hence, an interface in which a user is encouraged to first assign variables Y, might help reduce the total number of interaction steps. This could be an important addition to recent efforts towards the formal analysis of user navigation. In [Felfernig, 2006] the author used a formal model of the recommender process, based on finite state automata, to support automatic debugging of faulty models of recommender user interfaces. In [Mahmood and Ricci, 2007] the authors presented a recommender system that autonomously learns an adaptive interaction strategy, using a formal model of user interaction based on Markov decision processes. In [Hadzic and O'Sullivan, 2008] the authors introduced critique graphs as a formalism for analyzing various aspects of interaction in conversational recommender systems. In particular reachability of products through critiquing was discussed.

3 Functional Dependencies in MDDs

The main contribution of this paper is an approach to uncovering functional dependencies when a product (solution) set is compressed into a *multi-valued decision diagram* (MDD). As noted earlier, MDDs are, in the worst-case, linear in the size of the catalogue $\mathcal{O}(T)$, but they can often be exponentially smaller. Since the algorithms we suggest for uncovering functional dependencies, in this and the following sections, have quadratic complexity in the size of the MDD, whenever the MDD is sufficiently small we guarantee a sublinear-time algorithm for performing atomic dependency tests $Y \rightarrow x$.

We will first discuss how to detect *directional dependencies*, which respect variable ordering and are particularly easy to detect. We will then discuss uncovering *general dependencies* of the form $X \setminus \{x_i\} \rightarrow x_i$.

3.1 Directional Dependencies

Directional dependencies $\{x_1, \ldots, x_{i-1}\} \rightarrow x_i$ state that a variable x_i is determined by the subset of all variables preceding it in the variable ordering of the MDD. This is a particularly easy to detect class of dependencies as shown in the following proposition.

Proposition 1. $\{x_1, \ldots, x_{i-1}\} \rightarrow x_i$ *iff for all* $u \in V_i$, u *has only one outgoing edge* $|Out(u)| = 1$.

Proof. Let $p : r \rightsquigarrow u$ be a path from root to u, $e_1 : u \rightarrow u_1$ and $e_2 : u \rightarrow u_2$ be two outgoing edges and $p_1 : u_1 \rightsquigarrow \mathbf{1}$ and $p_2 : u_2 \rightsquigarrow \mathbf{1}$ be paths from u_1 and u_2 to terminal $\mathbf{1}$ respectively. Then paths (p, e_1, p_1) and (p, e_2, p_2) represent two solutions with identical assignments to variables $\{x_1, \ldots, x_{i-1}\}$ and two different assignments to x_i variable, $v(e_1) \neq v(e_2)$. □

Proposition 1 provides us with a simple test for checking whether $\{x_1, \ldots, x_{i-1}\} \rightarrow x_i$. It suffices that all nodes in V_i have exactly one outgoing edge. This can be easily checked by verifying that the number of nodes and outgoing edges is the same, $|V_i| = |E_i|$. The set of all variables implied by variables preceding in the order are given by:

$$Imp_{\prec} = \{x_i \mid |V_i| = |E_i|\}.$$

3.2 General Dependencies

Variables determined by subsets of variables preceding in the order, $Imp_{\prec}$, do not account for all implied variables. If variable x_i is implied by any subset $Y \subseteq X \setminus \{x_i\}$ then it will be also implied by $X \setminus \{x_i\}$. Therefore, the set of all implied variables Imp is the set of all x_i such that $X \setminus \{x_i\} \rightarrow x_i$. To detect such variables in an MDD, we will use the following proposition.

Proposition 2. *$X \setminus \{x_i\} \not\to x_i$ if and only if there is a node $u \in V_i$ with two outgoing edges $e_1 : u \to u_1$, $e_2 : u \to u_2$ such that $v(e_1) \neq v(e_2)$ and $Sol(u_1) \cap Sol(u_2) \neq \emptyset$.*

Proof. If $X \setminus \{x_i\} \not\to x_i$ then there are two solutions $\mathbf{a} = (a_1, \ldots, a_n), \mathbf{a}' = (a'_1, \ldots, a'_n)$ differing only in the i-th coordinate, $a_i \neq a'_i$. Let $p_\mathbf{a}$ and $p_{\mathbf{a}'}$ be the paths encoding these solutions. These paths must be of the form: $p_\mathbf{a} = (p, e_1, p_1)$, $p_{\mathbf{a}'}(p, e_2, p_2)$, where p is a unique path encoding $(a_1, \ldots, a_{i-1})$. Path p ends in a node $u \in V_i$. Since $a_i \neq a'_i$, $v(e_1) \neq v(e_2)$ and since $v(p_1) = v(p_2) = (a_{i+1}, \ldots, a_n)$ it follows $Sol(u_1) \cap Sol(u_2) \supseteq \{(a_{i+1}, \ldots, a_n)\}$.

On the other hand, if there is $u \in V_i$ with two outgoing edges e_1, e_2, such that $v(e_1) \neq v(e_2)$ and $Sol(u_1) \cap Sol(u_2) \neq \emptyset$ then we can choose two paths $p_1 : u_1 \rightsquigarrow \mathbf{1}$, $p_2 : u_2 \rightsquigarrow \mathbf{1}$ such that $v(p_1) = v(p_2)$. It suffices to choose any path from root to u, $p : r \rightsquigarrow u$ to construct paths $p_\mathbf{a} = (p, e_1, p_1)$, $p_{\mathbf{a}'}(p, e_2, p_2)$ which encode solutions differing only at the i-th coordinate and thus proving $X \setminus \{x_i\} \not\to x_i$. □

Assume that for each pair of nodes in the same layer u_1, u_2 we have precomputed Boolean indicators $D[u_1, u_2]$

$$D[u_1, u_2] = 1 \Leftrightarrow Sol(u_1) \cap Sol(u_2) \neq \emptyset.$$

Whenever we encounter a pair of nodes (u_1, u_2) such that $D[u_1, u_2] = 1$, we are guaranteed that there are at least two paths $p_1 : u_1 \rightsquigarrow \mathbf{1}$ and $p_2 : u_2 \rightsquigarrow \mathbf{1}$ encoding the same solution $v(p_1) = v(p_2)$. Given such labels, we can compute all functionally determined variables using Algorithm 1. In each layer we check for all pairs of edges with the same parent $e_1(u, u_1, a_1)$,$e_2(u, u_2, a_2)$ whether $D[u_1, u_2] = 1$. As soon as such edges are found we have proven that x_i is not implied and we may proceed to the next layer. The algorithm runs in $\mathcal{O}(\sum_{i=1}^n |V_i| \cdot |D_i|^2)$ steps, since for each node in each layer $u \in V_i$, we compare in worst case all pairs of its children edges, and there are at most $|D_i| \times (|D_i| - 1)/2$ such pairs. The space complexity is $\mathcal{O}(\sum_{i=1}^n |V_i|^2)$ since we have to store Boolean indicators $D[u_1, u_2]$ for each pair $(u_1, u_2) \in V_i^2$.

Algorithm 1: Compute functionally determined variables.

```
Data: MDD $M(V, E)$
$Imp = X$;
foreach $i = 1, \ldots, n$ do
    foreach $u \in V_i,\ |Ch(u)| > 1$ do
        foreach $e_1 : u \to u_1, e_2 : u \to u_2$ do
            if $v(e_1) \neq v(e_2) \wedge D[u_1, u_2] = 1$ then
                $Imp \leftarrow Imp \setminus \{x_i\}$;
                go to next layer;
return $Imp$;
```

Compatibility pairs $D[u_1, u_2]$ can be computed in quadratic time and space using a dynamic programming scheme from Algorithm 2. We first initialize $D[u_1, u_2] = 0$ for all pairs of nodes, except for the terminal $\mathbf{1}$, setting $D[\mathbf{1}, \mathbf{1}] = 1$. We then, in a bottom-up manner, traverse the MDD. The recursive relationship used for dynamic programming is based on observing that $D[u_1, u_2] = 1$ iff there are two outgoing edges $e_1 : u_1 \to u'_1, e_2 : u_2 \to u'_2$ such that $v(e_1) = v(e_2) \wedge D[u'_1, u'_2] = 1$. The algorithm runs in $\mathcal{O}(\sum_{i=1}^n |E_i|^2)$ time since in each layer E_i each pair of edges is compared at most once. The algorithm takes $\Theta(\sum_i |V_i|^2)$ space, since we introduce a compatibility indicator for each pair of nodes in the layer.

Algorithm 2: Compute Boolean indicators.

```
Data: MDD $M(V, E)$
$D[\cdot, \cdot] = 0, D[\mathbf{1}, \mathbf{1}] = 1$;
foreach $i = n, \ldots, 1$ do
    foreach $(u_1, u_2) \in V_i \times V_i$ do
        if $u_1 = u_2$ then
            $D[u_1, u_2] = 1$;
            continue;
        foreach $e_1 : u_1 \to u'_1, e_2 : u_2 \to u'_2$ do
            if $v(e_1) = v(e_2) \wedge D[u'_1, u'_2] = 1$ then
                $D[u_1, u_2] = 1$;
                break;
return $D$;
```

4 Subset-Induced Dependencies

Given a subset of variables $Y \subseteq X$ we may want to compute the set of all variables implied by Y:

$$Imp_Y = \{x_i \in X \setminus Y \mid Y \to x_i\}.$$

This could help us to evaluate an overall impact of a subset of variables. We could exploit this information in a number of different settings. In particular, if we find Y such that $Imp_Y = X \setminus Y$, then regardless of how we assign variables Y, we would completely specify entire solution. This could be important for increasing the usability of user interaction since, if a user is assigning only variables Y we guarantee that the number of user interactions before completely specifying a solution is at most $|Y|$. Furthermore, such an information could help organize the visual layout of the variables in a user interface. If a variable x_i is determined by variables Y, by displaying x_i closer to Y in the user interface, a user would faster evaluate implications of his assignments to Y variables.

By definition, a variable x_i is not implied by Y iff there are two solutions $\mathbf{a_1}, \mathbf{a_2}$ such that $\mathbf{a_1}[Y] = \mathbf{a_2}[Y]$ and $a_i \neq a'_i$. Recall that $S[Y]$ denotes a projection of set S on variables Y. An observation that would help us detect such variables is provided in the following proposition.

Proposition 3. *$Y \not\to x_i$ iff there are two edges $e_1, e_2 \in E_i$, $e_1 : u_1 \to u'_1$, $e_2 : u_2 \to u'_2$, such that $v(e_1) \neq v(e_2)$ and $Sol(r, u_1)[Y] \cap Sol(r, u_2)[Y] \neq \emptyset$ and $Sol(u'_1)[Y] \cap Sol(u'_2)[Y] \neq \emptyset$.*

Assuming that for every pair of nodes u_1, u_2 we have computed Boolean indicators:

$$U_Y[u_1, u_2] = 1 \Leftrightarrow Sol(r, u_1)[Y] \cap Sol(r, u_2)[Y] \neq \emptyset$$

$$D_Y[u_1,u_2] = 1 \Leftrightarrow Sol(u_1)[Y] \cap Sol(u_2)[Y] \neq \emptyset$$

we could use an adaptation of Algorithm 1 to detect all variables implied by subset Y. The adaptation is presented in Algorithm 3.

Algorithm 3: Compute Y-implied variables.

```
Data: MDD $M(V,E)$, $Y \subset X, U_Y, D_Y$
$Imp_Y = X$;
foreach $i = 1,\dots,n$ do
    foreach $(u_1,u_2) \in V_i \times V_i$ do
        foreach $e_1 : u_1 \to u_1', e_2 : u_2 \to u_2'$ do
            if $v(e_1) \neq v(e_2) \wedge U_Y[u_1,u_2] =$
            $1 \wedge D_Y[u_1',u_2'] = 1$ then
                $Imp_Y \leftarrow Imp_Y \setminus \{x_i\}$;
                go to next layer;
return $Imp_Y$;
```

Compatibility labels U_Y, D_Y can be computed in quadratic time and space using an adaptation of Algorithm 2 which is presented in Algorithm 4. To construct a recursive relationship on which the computation is based, for a given pair of nodes $u_1, u_2 \in V_j$ we have to differentiate between two cases. If $x_j \notin Y$, then $D_Y[u_1,u_2] = 1$ iff there are two outgoing edges $e_1 : u_1 \to u_1', e_2 : u_2 \to u_2'$ such that $D_Y[u_1',u_2'] = 1$ regardless of whether $v(e_1) = v(e_2)$ or $v(e_1) \neq v(e_2)$. If $x_j \in Y$ then $D_Y[u_1,u_2] = 1$ iff in addition to $D_Y[u_1',u_2'] = 1$ it also holds $v(e_1) = v(e_2)$. Compatibility labels U_Y are computed in an analogous manner.

The algorithm runs in $\mathcal{O}(\sum_{i=1}^n |E_i|^2)$ time since for both traversals, in each layer E_i each pair of edges is compared at most once. The algorithm takes $\Theta(\sum_i |V_i|^2)$ space, since we introduce two Boolean compatibility indicators U_Y, D_Y for each pair of nodes in the layer.

Algorithm 4: Compute Y-Boolean indicators.

```
Data: MDD $M(V,E)$, $Y \subset X$
$D_Y[\cdot,\cdot] = 0$, $D_Y[\mathbf{1},\mathbf{1}] = 1$;
foreach $i = n,\dots,1$ do
    foreach $(u_1,u_2) \in V_i \times V_i$ do
        if $u_1 = u_2$ then
            $D_Y[u_1,u_2] = 1$;
            continue;
        foreach $e_1 : u_1 \to u_1', e_2 : u_2 \to u_2'$ do
            if $x_i \notin Y \wedge D_Y[u_1',u_2'] = 1$ then
                $D_Y[u_1,u_2] = 1$;
                break;
            else if
            $x_i \in Y \wedge v(e_1) = v(e_2) \wedge D_Y[u_1',u_2'] = 1$
            then
                $D_Y[u_1,u_2] = 1$;
                break;
$U_Y[\cdot,\cdot] = 0$, $U_Y[r,r] = 1$;
foreach $i = 1,\dots,n$ do
    foreach $(u_1',u_2') \in V_{i+1} \times V_{i+1}$ do
        if $u_1' = u_2'$ then
            $U_Y[u_1',u_2'] = 1$;
            continue;
        foreach $e_1 : u_1 \to u_1', e_2 : u_2 \to u_2'$ do
            if $x_i \notin Y \wedge U_Y[u_1,u_2] = 1$ then
                $U_Y[u_1',u_2'] = 1$;
                break;
            else if
            $x_i \in Y \wedge v(e_1) = v(e_2) \wedge U_Y[u_1,u_2] = 1$
            then
                $U_Y[u_1',u_2'] = 1$;
                break;
```

4.1 Finding Minimal Dependencies

Given a subset of variables $Y_0 \subseteq X$ such that $X \setminus Y_0 \to Y_0$, it is often required to compute the set of all *minimal* subsets of variables $Y \subseteq X \setminus Y_0$ that imply Y_0. In other words, we want to compute:

$$\mathcal{Y} = \{Y \subseteq X \setminus Y_0 \mid Y \to Y_0,\ s.t.\ \nexists_{Y' \subset Y} Y' \to Y_0\}.$$

There could be an exponential number of such sets, and a number of approaches has been developed that operate on a set-containment lattice [Huhtala *et al.*, 1999] and avoid unnecessary tests $Y \to x$. For example, whenever Y determines x, $Y \to x$, all supersets of Y also determine x. A number of similar optimizations are implemented in [Huhtala *et al.*, 1999]. Extending existing approaches by incorporating our MDD-tests is an interesting direction for future research, but falls out of the scope of this paper.

5 Approximative Dependencies

We have so far discussed only *exact* dependencies, i.e. we detect only whether variable x_i is determined or not. However, a subset of variables Y might have a significant implicative influence on a variable x even though it does not imply it exactly. Therefore we are interested in computing a *degree of dependency* $Y \to x$. Let $d(Y,x) \in [0..1]$ denote such a degree. There is a number of ways to define it. In [Huhtala *et al.*, 1999] the authors use for the measure a minimal percentage of solutions that has to be removed in order for dependency to hold. We however use the following definition:

$$d(Y,x) = \frac{|Sol[Y]|}{|Sol[Y \cup \{x\}]|}$$

When $Y \to x$, then $|Sol[Y]| = |Sol[Y \cup \{x\}]|$, and $d(Y,x) = 1$. On the other hand, when x is completely undetermined by Y, then every assignment to Y variables $\mathbf{a} \in Sol[Y]$ can be combined with all values in domain for x, D_x. In that case,

$$d(Y,x) = \frac{1}{|D_x|}.$$

Hence, to each dependency statement $Y \to x$ we assign a measure

$$d(Y,x) \in [\frac{1}{|D_x|}, 1]$$

where larger values indicate larger degrees of dependency, and when $d(Y, x) = 1$, $Y \to x$ holds exactly. We can interpret this measure as follows - whenever $d(Y, x) = d$, every assignment to variables Y can on average be combined with $\frac{1}{d}$ values of x.

Note that such a statistic can be easily computed using a BDD-based representation of solution set Sol. A BDD-package *BuDDy* [Lind-Nielsen, online] supports both projection and counting operations. Counting the number of solutions in a BDD is an efficient operation - linear in the number of nodes. While computing a BDD representation of a projected solution space, such as $Sol[Y]$, can in theory increase the size of the BDD - in practice projecting out variables is almost always an efficient operation that decreases the number of nodes significantly.

6 Experimental Evaluation

We have applied our techniques to four well-known product catalogues that are frequently used in recommender system research. These are related to digital cameras, laptop computers, property lettings and travel [Nicholson *et al.*, 2006]. For the purposes of our evaluation, we analyzed the data under the same adjustments that are usually done for experimental evaluation. Firstly, all unique identifiers or textual descriptions, were removed. Secondly, all declared domain values that did not appear in at least one product, but appeared in the specification, were also removed. If some values appeared in datasets but were not declared, we added them to model specification.

A summary of the properties of the instances are reported in Table 2. For each instance, *Cameras, Laptops, Travel, Lettings* we show the number of rows in the initial explicit solution set representation, the number of solutions Sol extracted from the MDD representation, the number of variables X, and minimal, maximal and average domain size. We uncovered that three out of four datasets contain duplicate entries, since the number of solutions is smaller than the number of rows.

Table 2: Basic properties of product catalogues. For each instance we show the number of rows in the initial table representation, number of solutions Sol extracted from the MDD representation, number of variables X, and minimal, maximal and average domain size.

Instance	*Rows*	$\|Sol\|$	**X**	$\mathbf{d_{min}}$	$\mathbf{d_{max}}$	$\mathbf{d_{avg}}$
Cameras	210	210	9	5	165	40
Laptops	693	683	14	2	438	42
Travel	1470	1461	7	4	839	134
Lettings	794	751	6	2	174	45

Dependency Analysis

After compiling catalogues into MDDs, we performed a dependency analysis $X \setminus \{x_i\} \to x_i$ for each instance and each attribute. A detailed analysis of product catalogues is presented in Table 3. For each instance, we list all variables X and for each variable $x_i \in X$ we indicate its *type* (Categorical, Numerical, Boolean), *domain size* and *degree of dependence*:

$$d(X \setminus \{x_i\}, x_i) = \frac{|Sol[X \setminus \{x_i\}]|}{|Sol]|}.$$

Recall that the degree of dependence $d(Y, x)$ captures how many values of x_i can on average be combined with every assignment to variables Y. For example, if $d(Y, x) = 1/2$, then every assignment to variables Y can on average be combined with 2 values in a domain of x. If $d(Y, x) = 1$ then for every assignment to Y variables there is exactly one compatible value in domain of x and hence, x is functionally determined by Y. We can see from the table that almost all variables are functionally determined, and those that are not have a very high level of functional dependency. This is not surprising given that the price of the product is the most distinguishing attribute - behaving similarly as a unique key in a database. However, to our surprise, price is *not functionally dependent* in any of the catalogues! This indicates that in each catalogue there are identical configurations which have different prices! This is particularly emphasized for the *Lettings* catalogue, where degree of dependence is 0.563. This means that each configuration of remaining attributes on average has two different prices.

After analyzing more closely the variable specification and original and processed datasets, we discovered that a *Street* attribute of the *Lettings* catalogue was not declared in the dataset specification. Therefore, the preprocessing step ignored the corresponding column. Hence, a number of lettings with identical specifications and in the same region were treated as identical even though they were offered in different streets of the same region. For the same reason, the number of duplicate entries in Table 2 of the *Lettings* catalogue was disproportionably large.

7 Conclusions

In this paper we presented an approach to computing functional dependencies over an MDD representation of a product catalogue. We focused on verifying catalogue consistencies as an application relevant within an online configuration/recommendation context. Using functional dependencies as an analytic tool we discovered that a set of publicly available product catalogues exhibits specific characteristics that have not been noted to-date; some of these characteristics can be regarded as bugs in the catalogue definition or in the preprocessing step of the catalogues. This warrants caution in the future investigations in the area of web-based configuration and recommender systems that rely on preprocessed forms of product catalogues. The fact that the datasets might violate some consistency criteria should be taken into account when drawing conclusions from experimental evaluations.

We also identified that functional dependencies can be used to formally reason about the length of user interaction. This topic fits within recent research about recommender systems [Felfernig, 2006; Mahmood and Ricci, 2007; Hadzic and O'Sullivan, 2008] but was not the focus of this paper. Instead, it would be pursued in future work. In addition, in the future we plan to further investigate the utility

Table 3: A dependency analysis of product catalogues *Cameras, Laptops, Travel* and *Lettings*. For each instance, we list variables X and for each variable we indicate its *type* , *domain size* and *degree of dependence*. Variable type *Boolean** indicates that beside *yes* or *no* values, a value *unknown* is also permitted.

Instance	**Variable**	**Type**	**Domain size**	**Degree of dependence**
Cameras	Format	Categorical	5	1.0
	Storage type	Categorical	7	1.0
	Storage amount	Numerical	12	1.0
	Manufacturer	Categorical	15	1.0
	Optical zoom	Numerical	21	1.0
	Digital zoom	Numerical	22	1.0
	Resolution	Numerical	31	1.0
	Weight	Numerical	87	1.0
	Price	Numerical	165	0.966
Laptops	Microphone	Boolean	2	0.995
	Speakers	Boolean	2	0.998
	DVD	Boolean*	2	1.0
	Floppy	Boolean	2	1.0
	Modem	Boolean*	3	1.0
	CDROM	Boolean*	3	1.0
	Pointing device	Categorical	3	1.0
	RAM	Numerical	9	1.0
	Screen size	Numerical	9	1.0
	Processor type	Categorical	10	1.0
	Processor speed	Numerical	15	0.998
	HD size	Numerical	28	0.998
	Weight	Numerical	64	1.0
	Price	Numerical	438	0.855
Travel	Transport	Categorical	4	0.999
	Accommodation	Categorical	6	0.998
	Holiday type	Categorical	8	0.964
	Duration	Numerical	9	0.998
	Number of persons	Numerical	11	0.991
	Region	Categorical	65	0.971
	Price	Numerical	839	0.809
Lettings	Type	Categorical	2	0.993
	Furnished	Boolean	2	0.968
	Baths	Numerical	6	0.981
	Beds	Numerical	8	0.955
	Area	Categorical	80	0.650
	Price	Numerical	174	0.563

of decision diagrams for recommending general configurable products.

Acknowledgements

Hadzic is supported by a Post-doctoral Research Fellowship from the Irish Research Council for Science, Engineering and Technology. O'Sullivan is supported by Science Foundation Ireland (Grant Number 05/IN/I886). We would like to thank anonymous reviewers for their useful comments.

References

[Bryant, 1986] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 1986.

[Drechsler, 2001] Rolf Drechsler. Binary decision diagrams in theory and practice. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(2):112–136, May 2001.

[Felfernig, 2006] Alexander Felfernig. Diagnosing faulty transitions in recommender user interface descriptions. In *Advances in Applied Artificial Intelligence*, pages 869–878. Springer-Verlag, 2006.

[Hadzic and O'Sullivan, 2008] Tarik Hadzic and Barry O'Sullivan. Critique graphs for catalogue navigation. In *RecSys '08: Proceedings of the 2008 ACM conference on Recommender systems*, pages 115–122, New York, NY, USA, 2008. ACM.

[Huhtala *et al.*, 1999] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, March 1999.

[Lind-Nielsen, online] J. Lind-Nielsen. BuDDy - A Binary Decision Diagram Package. `http://sourceforge.net/projects/buddy`, online.

[Mahmood and Ricci, 2007] Tariq Mahmood and Francesco Ricci. Learning and adaptivity in interactive recommender systems. In *ICEC '07: Proceedings of the ninth international conference on Electronic commerce*, pages 75–84, New York, NY, USA, 2007. ACM.

[Meinel and Theobald, 1998] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer, 1998.

[Nicholson *et al.*, 2006] Ross Nicholson, Derek Bridge, and Nic Wilson. Decision diagrams: Fast and flexible support for case retrieval and recommendation. In *Proceedings of Eighth European Conference on Case-Based Reasoning (ECCBR 2006)*, 2006.

[Schlimmer, 1993] Jeffrey C. Schlimmer. Efficiently inducing determinations: A complete and systematic search algorithm that uses optimal pruning. In *ICML*, pages 284–290, 1993.

[Wegener, 2000] Ingo Wegener. *Branching Programs and Binary Decision Diagrams*. Society for Industrial and Applied Mathematics (SIAM), 2000.