Compiling Constraint Networks into AND/OR Multi-Valued Decision Diagrams (AOMDDs)

Robert Mateescu and Rina Dechter

Donald Bren School of Information and Computer Science
University of California, Irvine, CA 92697-3425
{mateescu,dechter}@ics.uci.edu

Abstract. Inspired by AND/OR search spaces for graphical models recently introduced, we propose to augment Ordered Decision Diagrams with AND nodes, in order to capture function decomposition structure. This yields *AND/OR multi-valued decision diagram* (AOMDD) which compiles a constraint network into a canonical form that supports polynomial time queries such as solution counting, solution enumeration or equivalence of constraint networks. We provide a compilation algorithm based on Variable Elimination for assembling an AOMDD for a constraint network starting from the AOMDDs for its constraints. The algorithm uses the APPLY operator which combines two AOMDDs by a given operation. This guarantees the complexity upper bound for the compilation time and the size of the AOMDD to be exponential in the treewidth of the constraint graph, rather than pathwidth as is known for ordered binary decision diagrams (OBDDs).

1 Introduction

The work presented in this paper is based on two existing frameworks: (1) AND/OR search spaces for graphical models and (2) decision diagrams (DD). AND/OR search spaces [1–3] have proven to be a unifying framework for various classes of search algorithms for graphical models. The main characteristic is the exploitation of independencies between variables during search, which can provide exponential speedups over traditional search methods that can be viewed as traversing an OR structure. The AND nodes capture problem decomposition into **independent subproblems**, and the OR nodes represent branching according to variable values. Backjumping schemes for constraint satisfaction and satisfiability can be shown to explore the AND/OR space automatically if only one solution is sought. However, for counting and other enumeration tasks a deliberate exploration of the AND/OR space is beneficial [4].

Decision diagrams are widely used in many areas of research, especially in software and hardware verification [5, 6]. A BDD represents a Boolean function by a directed acyclic graph with two sink nodes (labeled 0 and 1), and every internal node is labeled with a variable and has exactly two children: *low* for 0 and *high* for 1. If isomorphic nodes were not merged, on one extreme we would have the full search *tree*, also called Shannon tree, which is the usual full tree explored by backtracking algorithm. The tree can be ordered if we impose that variables be encountered in the same order along every branch. It can then be compressed by merging isomorphic nodes (i.e., with the same label and identical children), and by eliminating redundant nodes (i.e., whose *low* and

high children are identical). The result is the celebrated *reduced ordered binary decision diagram*, or OBDD for short, introduced by Bryant [7]. However, the underlying structure is OR, because the initial Shannon tree is an OR tree. If AND/OR search trees are reduced by node merging and redundant nodes elimination we get a compact search graph that can be viewed as a BDD representation augmented with AND nodes.

In this paper we combine the two ideas, in order to create a decision diagram that has an AND/OR structure, thus exploiting problem decomposition. As a detail, the number of values is also increased from two to any constant, but this is less significant for the algorithms. Our proposal is closely related to two earlier research lines within the BDD literature. The first is the work on Disjoint Support Decompositions (DSD) investigated within the area of design automation [8], that were proposed recently as enhancements for BDDs aimed at exploiting function decomposition [9]. The second is the work on BDDs trees [10]. Another related proposal is the recent work by Fargier and Vilarem [11] on compiling CSPs into tree-driven automata.

A decision diagram offers a compilation of a problem. It typically requires an extended offline effort in order to be able to support polynomial (in its size) or constant time online queries. In the context of constraint networks, it could be used to represent the whole set of solutions, to give the solutions count or solution enumeration and to test satisfiability or equivalence of constraint networks. The benefit of moving from OR structure to AND/OR is a lower complexity of the algorithms and size of the compiled structure. It typically moves from being bounded exponentially in *pathwidth* pw^*, which is characteristic to chain decompositions or linear structures, to being exponentially bounded in *treewidth* w^*, which is characteristic of tree structures (it always holds that $w^* \leq pw^*$ and $pw^* \leq w^* \cdot \log n$).

Our contribution consists of: (1) we formally describe the AND/OR multi-valued decision diagram (AOMDD) and prove that it is a canonical representation of a constraint network; (2) we describe the APPLY operator that combines two AOMDDs by an operation and prove its complexity to be linear in the output. We show that the output of apply is bounded by the product of the sizes of the inputs. (3) we give a scheduling of building the AOMDD of a constraint network starting with the AOMDDs of its constraints. It is based on an ordering of variables, which gives rise to a pseudo tree (or bucket tree) according to the execution of Variable Elimination algorithm. This gives the complexity guarantees in terms of the *induced width* along that ordering (equal to the treewidth of the corresponding decomposition).

The structure of the paper is as follows: Sect. 2 provides preliminary definitions, a description of Variable Elimination and AND/OR search spaces; Sect. 3 describes the AOMDD, its graphical representation and properties, and demonstrates its compilation by Variable Elimination; Sect. 5 presents the APPLY operation; Sect. 6 discusses extensions to probabilistic models; Sect. 7 presents related work and Sect. 8 concludes.

2 Preliminaries

A constraint network and its associated graph are defined in the usual way:

Definition 1 (constraint network). *A* constraint network *is a 3-tuple* $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$, *where:* $\mathbf{X} = \{X_1, \ldots, X_n\}$ *is a set of variables;* $\mathbf{D} = \{D_1, \ldots, D_n\}$ *is the set of*

their finite domains of values, with cardinalities $k_i = |D_i|$ and $k = \max_{i=1}^n k_i$; $\mathbf{C} = \{C_1, \dots, C_r\}$ is a set of constraints over subsets of $\mathbf{X}$. Each constraint is defined as $C = (S_i, R_i)$, where S_i is the set of variables on which the constraint is defined, called its scope, and R_i is the relation defined on S_i.

Definition 2 (constraint graph). *The* constraint graph *of a constraint network is an undirected graph, $G = (\mathbf{X}, E)$, that has variables as its vertices and an edge connecting any two variables that appear in the scope (set of arguments) of the same constraint.*

A pseudo tree resembles the tree rearrangements introduced in [12]:

Definition 3 (pseudo tree). *A* pseudo tree *of a graph $G = (\mathbf{X}, E)$ is a rooted tree $\mathcal{T}$ having the same set of nodes $\mathbf{X}$, such that every arc in E is a backarc in $\mathcal{T}$ (i.e., it connects nodes on the same path from root).*

Definition 4 (induced graph, induced width, treewidth, pathwidth). *An* ordered graph *is a pair (G, d), where G is an undirected graph, and $d = (X_1, ..., X_n)$ is an ordering of the nodes. The* width of a node *in an ordered graph is the number of neighbors that precede it in the ordering. The* width of an ordering d*, denoted $w(d)$, is the maximum width over all nodes. The* induced width of an ordered graph*, $w^*(d)$, is the width of the induced ordered graph obtained as follows: for each node, from last to first in d, its preceding neighbors are connected in a clique. The* induced width of a graph*, w^*, is the minimal induced width over all orderings. The induced width is also equal to the* treewidth *of a graph. The* pathwidth pw^* *of a graph is the treewidth over the restricted class of orderings that correspond to chain decompositions.*

2.1 Variable Elimination (VE)

Variable elimination (**VE**) [13, 14] is a well known algorithm for inference in graphical models. We will describe it using the terminology from [14]. Consider a constraint network $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ and an ordering $d = (X_1, X_2, \dots, X_n)$. The ordering d dictates an elimination order for **VE**, from last to first. Each constraints from $\mathbf{C}$ is placed in the bucket of its latest variable in d. Buckets are processed from X_n to X_1 by eliminating the bucket variable (the constraints residing in the bucket are joined together, and the bucket variable is projected out) and placing the resulting constraint (also called *message*) in the bucket of its latest variable in d. After its execution, **VE** renders the network backtrack free, and a solution can be produced by assigning variables along d. **VE** can also produce the solutions count if marginalization is done by summation (rather than projection) and join is substituted with multiplication.

VE also constructs a bucket tree, by linking the bucket of each X_i to the destination bucket of its message (called the parent bucket). A node in the bucket tree typically has a *bucket variable*, a *collection of constraints*, and a *scope* (the union of the scopes of its constraints). If the nodes of the bucket tree are replaced by their respective bucket variables, it is easy to see that we obtain a pseudo tree.

Example 1. Figure 1a shows a network with four constraints. Figure1b shows the execution of Variable Elimination along $d = (A, B, E, C, D)$. The buckets are processed from D to A [1]. Figure 1c shows the bucket tree. The pseudo tree is given in Fig. 2a.

[1] This representation reverses the top down bucket processing described in earlier papers.

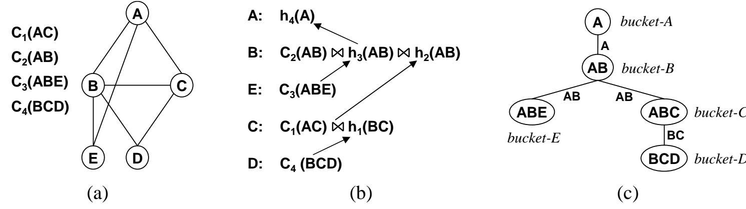


Fig. 1. Execution of Variable Elimination

2.2 AND/OR Search for Constraint Problems

The AND/OR search space is a recently introduced [1–3] unifying framework for advanced algorithmic schemes for graphical models. Its main virtue consists in exploiting independencies between variables during search, which can provide exponential speedups over traditional search methods oblivious to problem structure.

Definition 5 (AND/OR search tree of a constraint network). *Given a constraint network $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$, its constraint graph G and a pseudo tree $\mathcal{T}$ of G, the associated AND/OR search tree has alternating levels of OR and AND nodes. The OR nodes are labeled X_i and correspond to variables. The AND nodes are labeled $\langle X_i, x_i \rangle$ and correspond to value assignments. The structure of the AND/OR search tree is based on $\mathcal{T}$. The root is an OR node labeled with the root of $\mathcal{T}$. The children of an OR node X_i are AND nodes labeled with assignments $\langle X_i, x_i \rangle$ that are consistent with the assignments along the path from the root. The children of an AND node $\langle X_i, x_i \rangle$ are OR nodes labeled with the children of variable X_i in the pseudo tree $\mathcal{T}$. The leaves of AND nodes are labeled with "1". There is a one to one correspondence between solution subtrees of the AND/OR search graph and solutions of the constraint network [1].*

The AND/OR search tree can be traversed by a depth first search algorithm, thus using linear space. It was already shown [12, 15, 16, 1] that:

Theorem 1. *Given a constraint network $\mathcal{R}$ and a pseudo tree $\mathcal{T}$ of depth m, the size of the AND/OR search tree based on $\mathcal{T}$ is $O(n\ k^m)$, where k bounds the domains of variables. A constraint network of treewidth w^* has a pseudo tree of depth at most $w^* \log n$, therefore it has an AND/OR search tree of size $O(n\ k^{w^* \log n})$.*

The AND/OR search tree may contain nodes that root identical conditioned subproblems. These nodes are said to be *unifiable*. When unifiable nodes are merged, the search space becomes a graph. Its size becomes smaller at the expense of using additional memory by the search algorithm. The depth first search algorithm can therefore be modified to cache previously computed results, and retrieve them when the same nodes are encountered again. The notion of unifiable nodes is defined formally next.

Definition 6 (isomorphism, minimal AND/OR graph). *Two AND/OR search graphs G and G' are* isomorphic *if there exists a one to one mapping σ from the vertices of G to the vertices of G' such that for any vertex v, if $\sigma(v) = v'$, then v and v' root identical subgraphs relative to σ. The* minimal AND/OR graph *is such that all the isomorphic subgraphs are merged. Isomorphic nodes are also called unifiable.*

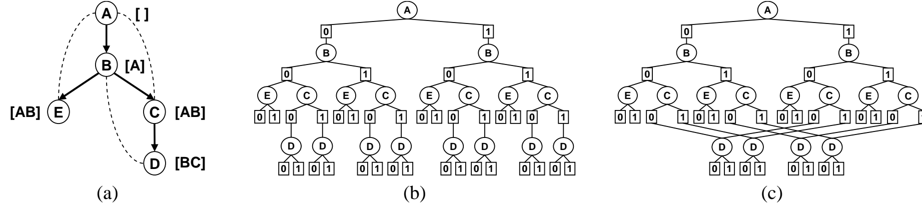


Fig. 2. AND/OR search space

Some unifiable nodes can be identified based on their *contexts*. We can define graph based contexts for both OR nodes and AND nodes, just by expressing the set of ancestor variables in $\mathcal{T}$ that completely determine a conditioned subproblem. However, it can be shown that using caching based on OR contexts makes caching based on AND contexts redundant and vice versa, so we will only use *OR caching*. Any value assignment to the context of X separates the subproblem below X from the rest of the network.

Definition 7 (OR context). *Given a pseudo tree $\mathcal{T}$ of an AND/OR search space, $context(X) = [X_1 \ldots X_p]$ is the set of ancestors of X in $\mathcal{T}$, ordered descendingly, that are connected in the primal graph to X or to descendants of X.*

Definition 8. *The* context minimal *AND/OR graph is obtained from the AND/OR search tree by merging all the context unifiable OR nodes.*

It was already shown that [15, 1]:

Theorem 2. *Given a constraint network $\mathcal{R}$, its primal graph G and a pseudo tree $\mathcal{T}$, the size of the context minimal AND/OR search graph based on $\mathcal{T}$ is $O(n\ k^{w^*_{\mathcal{T}}(G)})$, where $w^*_{\mathcal{T}}(G)$ is the induced width of G over the depth first traversal of $\mathcal{T}$, and k bounds the domain size.*

Example 2. Figure 2 shows AND/OR search spaces for the constraint network from Fig. 1, assuming universal relations (no constraints) and binary valued variables. When constraints are tighter, some of the paths in these graphs do not exists. Figure 2a shows the pseudo tree derived from ordering $d = (A, B, E, C, D)$, having the same structure as the bucket tree for this ordering. The (OR) context of each node appears in square brackets, and the dotted arcs are backarcs. Notice that the context of a node is identical to the message scope from its bucket in Fig. 1. Figure 2b shows the AND/OR search tree, and 2c shows the context minimal AND/OR graph.

3 AND/OR Multi-Valued Decision Diagrams (AOMDDs)

The *context minimal* AND/OR graph, described in section 2.2 offers an effective way of identifying unifiable nodes during the execution of the search algorithm. However, merging based on context is not complete, i.e. there may still be unifiable nodes in the search graph that do not have identical contexts. Moreover, some of the nodes in the context minimal AND/OR graph may be redundant, for example when the set of

solutions rooted at variable X_i is not dependant on the specific value assigned to X_i (this situation is not detectable based on context).

As overviewed earlier, in [1] we defined the complete *minimal AND/OR graph* which is an AND/OR graph whose all unifiable nodes are merged and we also proved canonicity [4]. Here we propose to augment the minimal AND/OR search graph with removing redundant variables as is common in OBDD representation as well as adopt some notational conventions common in this community. This yields a data structure that we call AND/OR BDDs, that exploits decomposition by using AND nodes. We present this extension over multi-valued variables yielding AND/OR MDD or AOMDD.

3.1 AND/OR Relation Graphs and Canonical AND/OR Decision Diagrams

We first define the AND/OR constraint function graph, which is an AND/OR data structure that defines a relation relative to a tree structure over a set of variables.

Definition 9 (AND/OR constraint function graph). *Given a set of variables* $\mathbf{X} = \{X_1, ..., X_n\}$*, domain of values* $\{D_1, ..., D_n\}$ *and a tree* $\mathcal{T}$ *over* $\mathbf{X}$ *as its nodes, an* AND/OR constraint function graph $\mathcal{G}$ *is a rooted, directed, labeled acyclic graph, having alternating levels of OR and AND nodes and two special terminal nodes labeled "0" and "1". The OR nodes are labeled by variables from* $\mathbf{X}$ *and the AND nodes are labeled by value assignments from respective the domains. There are arcs from nodes to their child nodes defined as follows:*

- *A nonterminal OR vertex* v *is labeled as* $l(v) = X_i$*,* $X_i \in \mathbf{X}$ *and any of its child AND nodes* u *is labeled* $l(u) = \langle X_i, x_i \rangle$*, when* x_i *is a value of* X_i*. An OR node can also have a single child node which is the terminal "0" node.*
- *An AND node* u *labeled* $l(u) = \langle X_i, x_i \rangle$ *has OR child nodes. If an OR child node* w *of AND node* u*, is labeled* $l(w) = Y$*, then* Y *must be a descendant of* X *in* $\mathcal{T}$*. If any two variables* Z *and* Y *label two OR child nodes of* u *where* $l(u) = \langle X, x \rangle$*, then* Z *and* Y *must be on different paths from* X *down to the leaves in* $\mathcal{T}$*. An AND node* u *can also have as a single child node the special node "1".*

The AND/OR constraint function graph defines a relation over the set of variables that are mentioned in $\mathcal{T}$ which can be obtained from the set of all *solution subtrees* of $\mathcal{G}$. A solution subtree of $\mathcal{G}$ contains its root node and if it contains an OR node, then it must contain one of its child nodes, and if it contains an AND node, it must contain all its child nodes. Its only leaf nodes are labeled "1". A solution tree defines a partial assignment that includes all the assignment labeling AND nodes in the solution tree. We say that this partial assignment is generated by $\mathcal{G}$.

Definition 10 (the relation defined by a constraint function graph). *Given an AND/OR constraint function graph* $\mathcal{G}$ *over variables* $\mathbf{X} = \{X_1, ..., X_n\}$ *having domain of values* $\{D_1, ..., D_n\}$ *and a tree* $\mathcal{T}$*, the relation* $rel(\mathcal{G})$ *includes all and only full assignments that extend partial assignments generated by* $\mathcal{G}$*.*

Similar to the case of OBDDs, AND/OR constraint function graphs can be reduced into canonical form by removing *isomorphism* and *redundancy*. The notion of isomorphism was defined earlier for AND/OR graphs. In order to capture the notion of redundancy, it is useful to group every OR node together with its AND children into a *meta-node*. The underlying graph is still an AND/OR graph.

Definition 11 (meta-node). *A* nonterminal meta-node v *in an AND/OR search graph consists of an OR node labeled* $var(v) = X_i$ *and its* k_i *AND children labeled* $\langle X_i, x_{i_1} \rangle$, $\ldots$, $\langle X_i, x_{i_{k_i}} \rangle$ *that correspond to its value assignments. We will sometimes abbreviate* $\langle X_i, x_{i_j} \rangle$*, by* x_{i_j}*. Each AND node labeled* x_{i_j} *points to a list of child meta-nodes,* $u.children_j$*. Examples of meta-nodes appear in Fig. 4.*

A variable is considered redundant with respect to the partial assignment of its parents, if assigning it any specific value does not change the value of any possible full assignment. Formally:

Definition 12 (redundant vertex). *Given an AND/OR constraint function graph* $\mathcal{G}$*, a node* v *is* redundant *if for all* u_1 *and* u_2 *which are AND child nodes of* v*,* u_1 *and* u_2 *have the same child meta nodes.*

If we want to remove a redundant vertex v from an AND/OR constraint function graph $\mathcal{G}$, then we make all of its parent AND nodes v_1, point directly to the common set of its grandchild meta nodes and the meta-node of v is removed from the graph.

Finally we define AOMDD:

Definition 13 (AOMDD). *An AND/OR multi-valued decision diagram (AOMDD) is an AND/OR constraint function graph that: (1) contains no redundant vertex and (2) it contains no isomorphic subgraphs.*

Analogously to OBDDs we can show that AOMDDs are a canonical representation of constraint networks, with respect to a given pseudo tree. Thus, AOMDDs can be used as a compiled representation of a constraint network.

Theorem 3 (AOMDDs are canonical). *Given a constraint network, whose constraint set is* $\mathbf{C}$*, having a constraint graph* G *and given a pseudo tree* $\mathcal{T}$ *of* G*, there is a unique (up to isomorphism) reduced AND/OR constraint function graph of* $\mathbf{C}$ *based on* $\mathcal{T}$*, and any other constraint function graph of* $\mathbf{C}$ *based on* $\mathcal{T}$ *has more vertices.*

4 Using Variable Elimination to Generate AOMDDs

In this section we propose to use a **VE** type algorithm to guide the compilation of a set of constraints into an AOMDD. Let's look at an example first.

Example 3. Consider the network defined by $\mathbf{X} = \{A, B, \ldots, H\}$, $D_A = \ldots = D_H = \{0, 1\}$ and the constraints ($\oplus$ denotes XOR): $C_1 = F \vee H$, $C_2 = A \vee \neg H$, $C_3 = A \oplus B \oplus G$, $C_4 = F \vee G$, $C_5 = B \vee F$, $C_6 = A \vee E$, $C_7 = C \vee E$, $C_8 = C \oplus D$, $C_9 = B \vee C$. The constraint graph is shown in Figure 3a. Consider the ordering $d = (A, B, C, D, E, F, G, H)$. The pseudo tree (or bucket tree) induced by d is given in Fig. 3b. Figure 4 shows the execution of **VE** with AOMDDs along ordering d. Initially, the constraints C_1 through C_9 are represented as AOMDDs and placed in the bucket of their latest variable in d. Each *original* constraint is represented by an AOMDD based on a chain. For bi-valued variables, they are OBDDs, for multiple-valued they are MDDs. Note that we depict meta-nodes: one OR node and its two AND

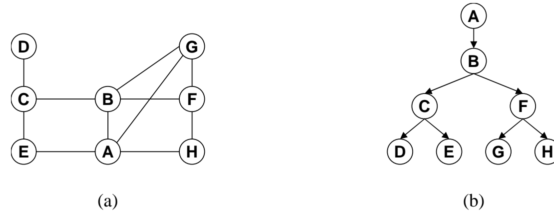


Fig. 3. (a) Constraint graph for $C = \{C_1, \ldots, C_9\}$, where $C_1 = F \vee H$, $C_2 = A \vee \neg H$, $C_3 = A \oplus B \oplus G$, $C_4 = F \vee G$, $C_5 = B \vee F$, $C_6 = A \vee E$, $C_7 = C \vee E$, $C_8 = C \oplus D$, $C_9 = B \vee C$; (b) Pseudo tree (bucket tree) for ordering $d = (A, B, C, D, E, F, G, H)$

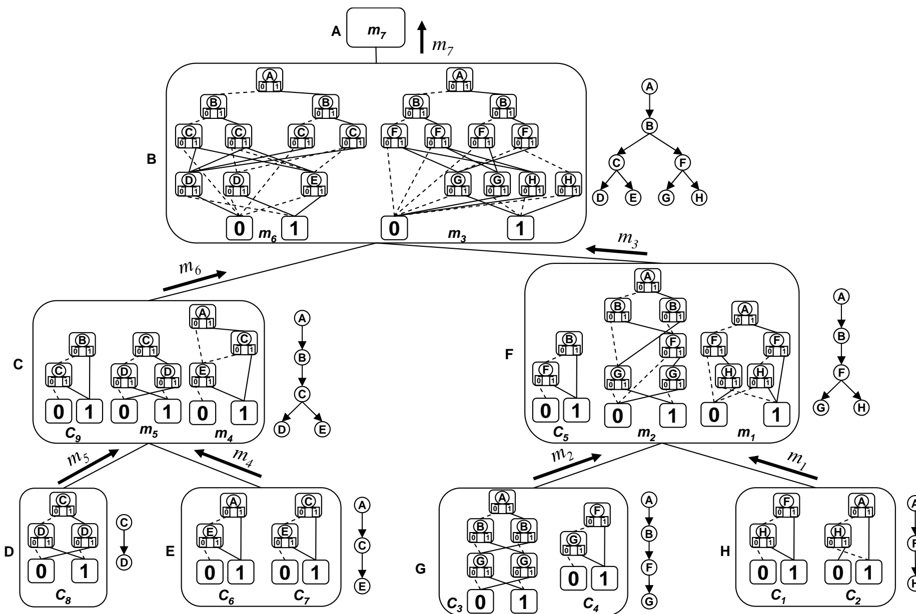
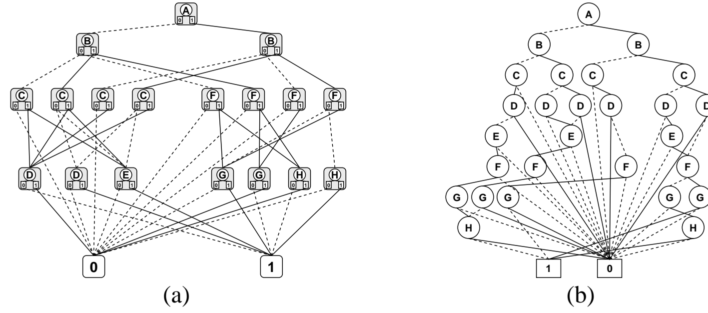


Fig. 4. Execution of VE with AOMDDs

children, that appear inside each gray node. The dotted edge corresponds to the 0 value (the *low* edge in OBDDs), the solid edge to the 1 value (the *high* edge). We have some redundancy in our notation, keeping both AND value nodes and arc-types (doted arcs from "0" and solid arcs from "1").

The **VE** scheduling is used to process the buckets in reverse order of d. A bucket is processed by *joining* all the AOMDDs inside it, using the *apply* operator. However, the step of eliminating the bucket variable will be omitted because we want to generate the full AOMDD. In our example, the messages $m_1 = C_1 \bowtie C_2$ and $m_2 = C_3 \bowtie C_4$ are still based on chains, so they are still OBDDs. Note that they still contain the variables H and G, which have not been eliminated. However, the message $m_3 = C_5 \bowtie m_1 \bowtie m_2$ is not an OBDD anymore. We can see that it follows the structure of the pseudo



(a) (b)

Fig. 5. (a) The final AOMDD; (b) The OBDD corresponding to d

tree, where F has two children, G and H. Some of the nodes corresponding to F have two outgoing edges for value 1.

The processing continues in the same manner The final output of the algorithm, which coincides with m_7, is shown in Figure 5a. The OBDD based on the same ordering d is shown in Fig. 5b. Notice that the AOMDD has 18 nonterminal nodes and 47 edges, while the OBDD has 27 nonterminal nodes and 54 edges.

4.1 Algorithm VE-AOMDD

Given an ordering d there is a unique pseudo tree $\mathcal{T}_d$ (or bucket tree) corresponding to it Each constraint C_i is compiled into an AOMDD that is compatible with $\mathcal{T}$ and placed into the appropriate bucket. The buckets are processed from last variable to first as usual. Each bucket contains AOMDDs that are either initial constraints or AOMDDs received from previously processed buckets. The scope of all the variables that are mentioned in a bucket include *relevant* variables, i.e. the ones whose buckets were not yet processed (note that they are identical to the OR context), and *superfluous* variables, the ones whose buckets had been proceessed. The number of relevant variables is bounded by the induced width (because so is the OR context). It is easy to see that all the AOMDDs in a bucket only have in common variables which are relevant, and which reside on the top chain portion of the bucket pseudo tree. The superfluous variables appear in disjoint branches of the bucket pseudo tree.

Consequently combining any two AOMDDs in a bucket amounts to using the regular MDD (OBDD) *apply* operator on their respective common parts that are simple MDDs. The rest of the branches can be attached at the end of the combine operation. Thus, the complexity of processing a bucket of AOMDDs is like the complexity of pairwise MDD *apply* over constraints restricted to scopes bounded by the induced width.

Proposition 1. *The complexity of processing a bucket by* ***VE-AOMDD*** *is exponential in the number of relevant variables, therefore it is exponential in the induced width.*

In summary, to create the compiled AOMDD, we can use only regular MDD *apply* operators on the common (separator) portion in each bucket. In the next section however we will define a more general AOMDD *apply* operator that can combine any

two AOMDDs whose pseudo trees are compatible (to be defined). This can be useful when the two input AOMDDs are created completely independently and we want to still exploit their given structure.

5 The AOMDD APPLY Operation

We will now describe how to combine two general AOMDDs. We focus on combination by join, as usual in constraint processing, but other operations can be equally easily applied. The apply operator takes as an input two AOMDDs representing constraints C_1 and C_2 and returns an AOMDD representing $C_1 \bowtie C_2$, respectively.

In traditional OBDDs the combining *apply* operator assumes that the two input OBDDs have compatible variable ordering. Likewise, in order to combine two AOMDDs, we assume that their backbone pseudo trees are *compatible*. Namely, there should be a pseudo tree in which both can be embedded. In general a pseudo tree induces a strict partial order between the variables where a parent node always precedes its child nodes.

Definition 14 (compatible pseudo tree). *A strict partial order $d_1 = (X, <_1)$ over a set X is consistent with a partial order $d_2 = (Y, <_2)$ over a set Y, if for all $x_1, x_2 \in X \cap Y$ and $x_1 <_2 x_2$ then $x_1 <_1 x_2$. Two partial orders d_1 and d_2 are compatible iff there exists a partial order d that is consistent with both. Two pseudo trees are compatible iff the partial orders induced via the parent-child relationship, are compatible.*

For simplicity, we focus on a more restricted notion of compatibility, which is sufficient when using a VE like schedule for the *apply* operator to combine the input AOMDDs. It is easy to extend this operator to the more general notion of compatibility.

Definition 15 (strictly compatible pseudo trees). *A pseudo tree $\mathcal{T}_1$ having the set of nodes $\mathbf{X}_1$ can be* embedded *in a pseudo tree $\mathcal{T}$ having the set of nodes $\mathbf{X}$ if $\mathbf{X}_1 \subseteq \mathbf{X}$ and $\mathcal{T}_1$ can be obtained from $\mathcal{T}$ by deleting each node in $\mathbf{X} \setminus \mathbf{X}_1$ and connecting its parent to each of its descendents. Two pseudo trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are* compatible *if there exists $\mathcal{T}$ such that both $\mathcal{T}_1$ and $\mathcal{T}_2$ can be embedded in $\mathcal{T}$.*

Algorithm 1, called APPLY, takes as input two AOMDDs for two constraints f and g defined along strictly compatible pseudo trees, and a common target pseudo tree $\mathcal{T}$. We will start by describing the intuition behind the algorithm. An AOMDD along a pseudo tree can be regarded as a union of regular MDDs, each restricted to a full path from root to a leaf in the pseudo tree. Let $\pi_{\mathcal{T}}$ be a path in $\mathcal{T}$. Based on the definition of strictly compatible pseudo trees, $\pi_{\mathcal{T}}$ has a corresponding path in $\mathcal{T}_f$ and $\mathcal{T}_g$, which are the pseudo trees for f and g. The MDDs from f and g corresponding to $\pi_{\mathcal{T}_f}$ and $\pi_{\mathcal{T}_g}$ can be combined using the regular MDD *apply*. This process can be repeated for every path $\pi_{\mathcal{T}}$. The resulting MDDs, one for each path in $\mathcal{T}$ need to be synchronized by another MDD *apply* on their common parts (on the intersection of the paths). The algorithm we propose does all this processing at once, in a depth first search traversal over the inputs. Based on our construction it is clear that the complexity of AOMDD-apply is governed by the complexity of MDD-apply.

Algorithm 1: APPLY($v_1; w_1, \ldots, w_m$)

input : AOMDDs f with nodes v_i and g with nodes w_j, based on *compatible* pseudo trees $\mathcal{T}_1, \mathcal{T}_2$ that can be embedded in $\mathcal{T}$.
$var(v_1)$ is an ancestor of all $var(w_1), \ldots, var(w_m)$ in $\mathcal{T}$.
$var(w_i)$ and $var(w_j)$ are not in ancestor-descendant relation in $\mathcal{T}$.

output : AOMDD $v_1 \bowtie (w_1 \wedge \ldots \wedge w_m)$, based on $\mathcal{T}$.

1 **if** $H_1(v_1, w_1, \ldots, w_m) \neq null$ **then return** $H_1(v_1, w_1, \ldots, w_m)$; `// is in cache`
2 **if** *(any of* $v_1, w_1, \ldots, w_m$ *is* **0***)* **then return 0**
3 **if** *(*$v_1 = \mathbf{1}$*)* **then return 1**
4 **if** *(*$m = 0$*)* **then return** v_1 `// nothing to join`
5 create new nonterminal meta-node u
6 $var(u) \leftarrow var(v_1)$ (call it X_i, with domain $D_i = \{x_1, \ldots, x_{k_i}\}$)
7 **for** $j \leftarrow 1$ **to** k_i **do**
8 $\quad u.children_j \leftarrow \phi$ `// children of the j-th AND node of` u
9 $\quad$**if** *((*$m = 1$*) and (*$var(v_1) = var(w_1) = X_i$*))* **then**
10 $\quad\quad tempChildren \leftarrow w_1.children_j$
11 $\quad$**else**
12 $\quad\quad tempChildren \leftarrow \{w_1, \ldots, w_m\}$
13 $\quad$group nodes from $v_1.children_j \cup tempChildren$ in several $\{v^1; w^1, \ldots, w^r\}$
14 $\quad$**for** *each* $\{v^1; w^1, \ldots, w^r\}$ **do**
15 $\quad\quad y \leftarrow$ APPLY($v^1; w^1, \ldots, w^r$)
16 $\quad\quad$**if** *(*$y = \mathbf{0}$*)* **then**
17 $\quad\quad\quad u.children_j \leftarrow \mathbf{0}$; break
18 $\quad\quad$**else**
19 $\quad\quad\quad u.children_j \leftarrow u.children_j \cup \{y\}$
20 $\quad$**if** *(*$u.children_1 = \ldots = u.children_{k_i}$*)* **then** `// redundancy`
21 $\quad\quad$**return** $u.children_1$
22 $\quad$**if** *(*$H_2(var(u), u.children_1, \ldots, u.children_{k_i}) \neq null$*)* **then** `// isomorphism`
23 $\quad\quad$**return** $H_2(var(u), u.children_1, \ldots, u.children_{k_i})$
24 Let $H_1(v_1, w_1, \ldots, w_m) = u$ `// add` u `to` H_1
25 Let $H_2(var(u), u.children_1, \ldots, u.children_{k_i}) = u$ `// add` u `to` H_2
26 **return** u

Theorem 4. *Let $\pi_1, ...\pi_l$ be the set of path in $\mathcal{T}$ enumerated from left to right and let $\mathcal{G}_f^i$ and $\mathcal{G}_g^i$ be the OBDDs restricted to path π_i, then the size of the output of AOBDD-apply is bounded by $\sum_i |\mathcal{G}_f^i| \cdot |\mathcal{G}_g^i| \leq n \cdot max_i |\mathcal{G}_f^i| \cdot |\mathcal{G}_g^i|$. The time complexity is also bounded by $\sum_i |\mathcal{G}_f^i| \cdot |\mathcal{G}_g^i| \leq n \cdot max_i |\mathcal{G}_f^i| \cdot |\mathcal{G}_g^i|$.*

Algorithm APPLY takes as input one node from f and a list of nodes from g. Initially, the node from f is the root of f, and the list of nodes from g is in fact just one node, the root of g. The list of nodes from g always has a special property: there is no node in it that can be the ancestor in $\mathcal{T}$ of another (we refer to the variable of the meta-node). Therefore, the list $w_1, \ldots, w_m$ from g expresses a decomposition with respect to $\mathcal{T}$, so all those nodes appear on different branches. We will employ the usual

techniques from OBDDs to make the operation efficient. First, since join can be viewed as multiplication, if one of the arguments is $\mathbf{0}$, then we can safely return $\mathbf{0}$. Second, a hash table H_1 is used to store the nodes that have already been processed, based on the nodes $(v_1, w_1, \ldots, w_r)$. Therefore, we never need to make multiple recursive calls on the same arguments. Third, a hash table H_2 is used to detect isomorphic nodes. If at the end of the recursion, before returning a value, we discover that a meta-node with the same variable and the same children had already been created, then we don't need to store it and we simply return the existing node. And fourth, if at the end of the recursion we discover we created a redundant node (all children are the same), then we don't store it, and return instead one of its identical lists of children.

Note that v_1 is always an ancestor of all $w_1, \ldots, w_m$ in $\mathcal{T}$. We consider a variable in $\mathcal{T}$ to be an ancestor of itself. A few self explaining checks are performed in lines 1-4. Line 2 is specific for multiplication, and needs to be changed for other operations. The algorithm creates a new meta-node u, whose variable is $var(v_1) = X_i$ - recall that $var(v_1)$ is highest (closest to root) in $\mathcal{T}$ among $v_1, w_1, \ldots, w_m$. Then, for each possible value of X_i, line 7, it starts building its list of children.

One of the important steps happens in line 13. There are two lists of meta-nodes, one from each original AOMDD f and g, and we will refer only to their variables, as they appear in $\mathcal{T}$. Each of these lists has the important property mentioned above, that its nodes are not ancestors of each other. The union of the two lists is grouped into maximal sets of nodes, such that the highest node in each set is an ancestor of all the others. It follows that the root node in each set belongs to one of the original AOMDD, say v^1 is from f, and the others, say $w^1, \ldots, w^r$ are from g. As an example, suppose $\mathcal{T}$ is the pseudo tree from Fig. 3b, and the two lists are $\{C, G, H\}$ from f and $\{E, F\}$ from g. The grouping from line 13 will create $\{C; E\}$ and $\{F; G, H\}$. Sometimes, it may be the case that a newly created group contains only one node. This means there is nothing more to join in recursive calls, so the algorithm will return, via line 4, the single node. From there on, only one of the input AOMDDs is traversed, and this is important for the complexity of APPLY, discussed below.

5.1 Complexity of APPLY

Given AOMDDs f and g, based on compatible pseudo trees $\mathcal{T}_1$ and $\mathcal{T}_2$ and the common pseudo tree $\mathcal{T}$, we define the *intersection pseudo tree* $\mathcal{T}_\cap$ as being obtained from $\mathcal{T}$ by marking all the subtrees whose nodes belong to either $\mathcal{T}_1$ or $\mathcal{T}_2$ but not to both, and removing them simultaneously (not recursively). The part of AOMDD f corresponding to the variables in $\mathcal{T}_\cap$ is denoted by $\mathcal{G}_{f_\cap}$.

Proposition 2. *The time complexity of* APPLY *and the size of the output are* $O(|\mathcal{G}_{f_\cap}| * |\mathcal{G}_{g_\cap}| + |\mathcal{G}_f| + |\mathcal{G}_g|)$.

Proof (sketch). The proof that APPLY makes an effort $O(|\mathcal{G}_{f_\cap}| * |\mathcal{G}_{g_\cap}|)$ when combining nodes from $f_\cap$ and $g_\cap$ is identical to the proof that OBDDs have complexity of the order of the product of the inputs. For the parts of f and g that don't belong to $\mathcal{G}_{f_\cap}$ and $\mathcal{G}_{g_\cap}$, APPLY generates recursive calls that have only one argument, v_1, effectively calling for a simple traversal of just one of the inputs.

6 Compiling any probabilistic model to AOMDD

As we showed in the past, the notion of AND/OR graphs is applicable to any graphical models, such as probabilistic networks, cost networks and influence diagrams. Indeed, the compiled canonical forms of *minimal AND/OR graphs* are well defined and were already introduced [1, 4]. Therefore all the ideas we introduced in this paper can be generalized, yielding a canonical representation in the style of AOMDD (i.e., by removing redundant variables), which we can term *weighted AOMDDs*. In particular, weighted AOMDD can be compiled using Variable Elimination schedule that exploits an *apply operator* in each bucket, very similar to the way we carried this task here. The apply operator by itself, combining two weighted AOMDDS should have the same flavor. Compiling graphical models into weighted AOMDDs also extends the work of [17], which defines decision diagrams for the computation of semiring valuations, from linear variable ordering into tree-based partial ordering.

7 Related Work

There are various lines of related research. The formal verification literature, beginning with [7] contains a very large number of papers dedicated to the study of BDDs. However, BDDs are in fact OR structures (the underlying pseudo tree is a chain) and do not take advantage of the problem decomposition in an explicit way. The complexity bounds for OBDDs are based on *pathwidth* rather than *treewidth*.

As noted earlier, the work on Disjoint Support Decomposition (DSD) is related to AND/OR BDDs in various ways [9]. The main common aspect is that both approaches show how structure decomposition can be exploited in a BDD-like representation. DSD is focused on Boolean functions and can exploit more refined structural information that is inherent to Boolean functions. In contrast, AND/OR BDDs assumes only the structure conveyed in the constraint graph, they are therefore more broadly applicable to any constraint expression and also to graphical models in general. They allow a simpler and higher level exposition that yields graph-based bounds on the overall size of the generated AOMDD. The full relationship between these two formalisms should be studied further.

McMillan introduced the BDD trees [10], along with the operations for combining them. For circuits of bounded tree width, BDD trees have linear upper space bound $O(|g|2^{w2^{2w}})$, where $|g|$ is the size of the circuit g (typically linear in the number of variables) and w is the treewidth. This bound hides some very large constants to claim the linear dependence on $|g|$ when w is bounded. However, McMillan maintains that when the input function is a CNF expression BDD-trees have the same bounds as AND/OR BDDs, namely they are exponential in the treewidth only.

Darwiche has done much research on compilation, using insights from the AI community. The AND/OR structure restricted to propositional theories is very similar to deterministic decomposable negation normal form (d-DNNF) [18]. More recently, in [19], the trace of the DPLL algorithm is used to generate an OBDD, and compared with the bottom up approach of combining the OBDDs of the input function according to some schedule (as is typical in formal verification). The structures that are investigated

are still OR. The idea can nevertheless be extended to AND/OR search. We could run the depth first AND/OR search with caching, generating the *context minimal* AND/OR graph, which can then be processed bottom up by layers to be reduced even further by eliminating isomorphic subgraphs and redundant nodes.

McAllester [20] introduced the case factor diagrams (CFD) which subsume Markov random fields of bounded tree width and probabilistic context free grammars (PCFG). CFDs are very much related to the AND/OR graphs. The CFDs target the minimal representation, by exploiting decomposition (similar to AND nodes) but also by exploiting context sensitive information and allowing dynamic ordering of variables based on context. CFDs do not eliminate the redundant nodes, and part of the cause is that they use zero suppression. There is no claim about CFDs being a canonical form, and also there is no description of how to combine two CFDs.

More recently, independently and in parallel to our work on AND/OR graphs [1, 2], Fargier and Vilarem [11] proposed the compilation of CSPs into tree-driven automata, which have many similarities to our work. In particular, the compiled tree-automata proposed there is essentially the same as what we propose here. Their main focus is the transition from linear automata to tree automata (similar to that from OR to AND/OR), and the possible savings for tree-structured networks and hyper-trees of constraints due to decomposition. Their compilation approach is guided by a tree-decomposition while ours is guided by a variable-elimination based algorithms. And, it is well known that Variable Elimination and cluster-tree decomposition are in principle, the same [21].

We see that our work using AND/OR search graphs has a unifying quality that helps make connections among seemingly different compilation approaches.

8 Conclusion

We propose the AND/OR multi-valued decision diagram (AOMDD), which emerges from the study of AND/OR search for graphical models [1–3] and ordered binary decision diagrams (OBDDs) [7]. This data-structure can be used to compile any set of relations over multi-valued variables as well as any CNF Boolean expression.

The approach we take in this paper may seem to go against the current trend in model checking, which moves away from BDD-based algorithms into CSP/SAT based approaches. However, constraint processing algorithms that are search-based and compiled data-structures such as BDDs differ primarily by their choices of time vs memory. When we move from regular OR search space to an AND/OR search space the spectrum of algorithms available is improved for all time vs memory decisions. We believe that the AND/OR search space clarifies the available choices and helps guide the user into making an informed selection of the algorithm that would fit best the particular query asked, the specific input function and the available computational resources.

In summary, the contribution of our work is: (1) We formally describe the AOMDD and prove that it is a canonical representation of a constraint network; (2) We describe the APPLY operator that combines two AOMDDs by an operation and give its complexity bounded by the product of the sizes of the inputs; (3) We give a scheduling of building the AOMDD of a constraint network starting with the AOMDDs of its constraints. It is based on an ordering of variables, which gives rise to a pseudo tree (or bucket tree) according to the execution of Variable Elimination algorithm. This gives

the complexity guarantees in terms of the *induced width* along the ordering (equal to the treewidth of the corresponding decomposition); 4) We show how AOMDDs relate to various earlier and recent works, providing a unifying perspective for all these methods.

Acknowledgments

This work was supported in part by the NSF grant IIS-0412854. The initial part of this work was also supported by the Radcliffe fellowship 2005-2006, as part of the partner program, with Harvard undergraduate student John Cobb [22].

References

1. Dechter, R., Mateescu, R.: Mixtures of deterministic-probabilistic networks and their and/or search space. In: UAI'04. (2004) 120–129
2. Dechter, R., Mateescu, R.: The impact of and/or search spaces on constraint satisfaction and counting. In: CP'04. (2004) 731–736
3. Mateescu, R., Dechter, R.: The relationship between and/or search and variable elimination. In: UAI'05. (2005) 380–387
4. Dechter, R., R.Mateescu: And/or search spaces for graphical models. Artificial Intelligence (2006) forthcoming.
5. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
6. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic (1993)
7. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transaction on Computers **35** (1986) 677–691
8. Brayton, R., McMullen, C.: The decomposition and factorization of boolean expressions. In: ISCAS,Proceedings of the International Symposium on Circuits and Systems. (1982) 49–54
9. Bertacco, V., Damiani, M.: The disjunctive decomposition of logic functions. In: ICCAD, International Conference on Computer-Aided Design. (1997) 78–82
10. McMillan, K.L.: Hierarchical representation of discrete functions with application to model checking. In: Computer Aided Verification. (1994) 41–54
11. Fargier, H., Vilarem, M.: Compiling csps into tree-driven automata for interactive solving. Constraints **9**(4) (2004) 263–287
12. Freuder, E.C., Quinn, M.J.: Taking advantage of stable sets of variables in constraint satisfaction problems. In: IJCAI'85. (1985) 1076–1078
13. Bertele, U., Brioschi, F.: Nonserial Dynamic Programming. Academic Press (1972)
14. Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence **113** (1999) 41–85
15. Bayardo, R., Miranker, D.: A complexity analysis of space-bound learning algorithms for the constraint satisfaction problem. In: AAAI'96. (1996) 298–304
16. Darwiche, A.: Recursive conditioning. Artificial Intelligence **125**(1-2) (2001) 5–41
17. Wilson, N.: Decision diagrams for the computation of semiring valuations. In: IJCAI'05. (2005) 331–336
18. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of Artificial Intelligence Research (JAIR) **17** (2002) 229–264
19. Huang, J., Darwiche, A.: Dpll with a trace: From sat to knowledge compilation. In: IJCAI'05. (2005) 156–162
20. McAllester, D., Collins, M., Pereira, F.: Case-factor diagrams for structured probabilistic modeling. In: UAI'04. (2004) 382–391
21. Dechter, R., Pearl, J.: Tree clustering for constraint networks. Artificial Intelligence **38** (1989) 353–366
22. Cobb, J.: Joining and/or networks. In: Report, Radcliffe fellowship, Cambridge, MA. (2006)