

AND/OR Multi-Valued Decision Diagrams for Constraint Networks

Robert Mateescu^{1,*} and Rina Dechter²

¹ Electrical Engineering Department, California Institute of Technology
Pasadena, CA 91125
mateescu@paradise.caltech.edu

² Donald Bren School of Information and Computer Science, University of California, Irvine
Irvine, CA 92697
dechter@ics.uci.edu

Abstract. The paper is an overview of a recently developed compilation data structure for graphical models, with specific application to constraint networks. The AND/OR Multi-Valued Decision Diagram (AOMDD) augments well known decision diagrams (OBDDs, MDDs) with AND nodes, in order to capture function decomposition structure. The AOMDD is based on a pseudo tree of the network, rather than a linear ordering of its variables. The AOMDD of a constraint network is a canonical form given a pseudo tree. We describe two main approaches for compiling the AOMDD of a constraint network. The first is a top down, search-based procedure, that works by applying reduction rules to the trace of the memory intensive AND/OR search algorithm. The second is a bottom up, inference-based procedure, that uses a Bucket Elimination schedule. For both algorithms, the compilation time and the size of the AOMDD are, in the worst case, exponential in the *treewidth* of the constraint graph, rather than *pathwidth* as is known for ordered binary decision diagrams (OBDDs).

1 Introduction

The paper is an overview of AND/OR Multi-Valued Decision Diagrams (AOMDDs) as a compiled data structure for constraint networks. We present here an extension of the work in [1], while still maintaining the focus on constraint networks. AOMDDs for weighted graphical models and for constraint optimization were presented in [2, 3].

The AOMDD is based on two existing frameworks: (1) AND/OR search spaces for graphical models; (2) decision diagrams. AND/OR search spaces [4–6] have proven to be a unifying framework for various classes of search algorithms for graphical models. The main novelty 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.

Decision diagrams are widely used in many areas of research, especially in software and hardware verification [7, 8]. A BDD represents a Boolean function by a directed

* This work was done while at the University of California, Irvine.

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, we would have the full search *tree* explored by the backtracking algorithm. The tree is ordered if variables are 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 [9]. However, the underlying structure is OR (i.e., linear structure rather than 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.

This paper shows how to combine the two ideas, creating 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. In the context of constraint networks, decision diagrams can be used to represent the whole set of solutions, facilitating solutions count, solution enumeration and queries on equivalence of constraint networks. The benefit of moving from OR structure to AND/OR is in 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 $pw^* \leq w^* \cdot \log n$). In both cases, the compiled structure achieved in practice is often far smaller than what the bounds suggest.

A decision diagram offers a compilation of a propositional knowledge-base. A multi-valued AND/OR decision diagram extends compilation to general graphical models. The *knowledge compilation* approach has become an important research direction in automated reasoning in the past decades [10–12]. Typically, a knowledge representation language is compiled into a compact data structure on which various queries can be answered quickly. Accordingly, the computational effort can be divided between an *offline* and an *online* phase where most of the work is pushed offline. Compilation can also be used to generate compact building blocks to be used by online algorithms multiple times. Macro-operators compiled during or prior to search can be viewed in this light [13], while in graphical models the building blocks are the functions whose compact, compiled, representation can be used effectively across many tasks.

As one example, consider product configuration tasks and imagine a user that chooses sequential options to configure a product. In a naive system, the user would be allowed to choose any valid option at the current level based only on the initial constraints, until either the product is configured, or else, when a dead-end is encountered, the system would backtrack to some previous state and continue from there. This would in fact be a search through the space of possible partial configurations. Needless to say, it would be very unpractical, and would offer the user no guarantee of finishing in a limited time. A system based on compilation would actually build the *backtrack-free* search space in the offline phase, and represent it as compactly as possible. In the online phase, only valid partial configurations (i.e., that can be extended to a full valid

configuration) are allowed, and depending on the query type, response time guarantees can be offered in terms of the size of the compiled structure.

Numerous other examples, such as diagnosis and planning problems can be formulated as graphical models, and for which compilation would be useful. Compilation in diagnosis can facilitate fast detection of possible faults or explanations for some unusual behavior. In planning, compilation would allow swift adjustments according to changes in the environment. Probabilistic models are one of the most used types of graphical models, and the basic query is to compute conditional probabilities of some variables given the evidence. A compact compilation of a probabilistic model would allow fast response for any change in the evidence along time. Formal verification is another example where compilation is heavily used to compare equivalence of circuit design, or to check the behavior of a circuit. *Binary Decision Diagram* (BDD) [9] are arguably the most widely known and used compiled structure.

Our AOMDD proposal is related to two earlier research lines within the BDD literature. The first is the work on Disjoint Support Decompositions (DSD) [14], investigated within the area of design automation [15], that were proposed as enhancements for BDDs aimed at exploiting function decomposition. The second is the work on BDD trees [16]. Another related proposal is the recent work by Fargier and Vilarem [17] on compiling CSPs into tree-driven automata. We will comment more on the relationship between these work and AOMDD in the related work section.

The structure of the paper is as follows. Section 2 provides the preliminaries. Section 3 gives an overview of AND/OR search spaces. Section 4 introduces the AOMDD and Section 5 shows that it is a canonical form for constraint networks. Section 6 describes a search based algorithm for compiling the AOMDD. Section 7 presents a compilation algorithm based on a Bucket Elimination schedule and the APPLY operation. Section 8 presents related work and Section 9 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: $\mathbf{X} = \{X_1, \dots, X_n\}$ is a set of variables; $\mathbf{D} = \{D_1, \dots, D_n\}$ 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 (or primal 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 [18]:

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).

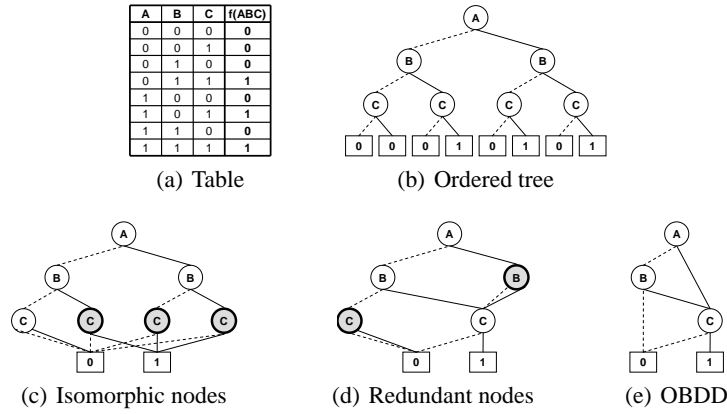


Fig. 1. Boolean function representation and reduction rules

Definition 4 (induced graph, induced width, treewidth, pathwidth). An ordered graph is a pair (G, d) , where G is an undirected graph, and $d = (X_1, \dots, X_n)$ 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 Binary Decision Diagrams Review

Decision diagrams are widely used in many areas of research to represent decision processes. In particular, they can be used to represent functions. Due to the fundamental importance of Boolean functions, a lot of effort has been dedicated to the study of *Binary Decision Diagrams* (BDDs), which are extensively used in software and hardware verification [7, 8]. Bryant [9] introduced the *Ordered Binary Decision Diagram* (OBDD). The order of variables along any path of an OBDD is the same. OBDDs provide a compact representation and efficient operations (the *apply* procedure, that combines two OBDDs by an operation is at most quadratic in the sizes of the input diagrams).

Example 1. Figure 1(a) shows a table representation of a Boolean function. A binary tree representation is shown in Figure 1(b). The internal round nodes represent the variables, the solid edges are the 1 (or high) value, and the dotted edges are the 0 (or low) value. The leaf square nodes show the value of the function for each assignment along a path. The tree is ordered, because variables appear in the same order along each path.

There are two reduction rules that transform a decision diagram into an equivalent one: (1) *isomorphism*: merge nodes that have the same label and the same children; (2)

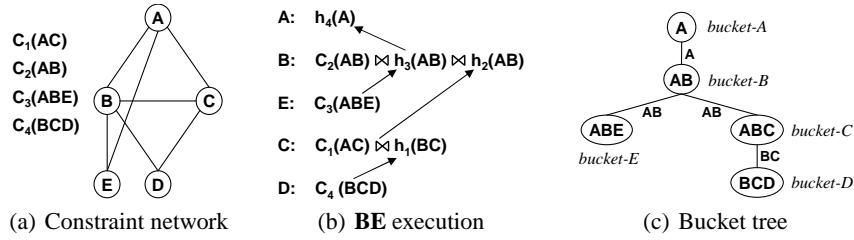


Fig. 2. Bucket Elimination

redundancy: eliminate nodes whose low and high edges point to the same node, and connect parent of removed node directly to child of removed node. Applying the two reduction rules exhaustively yields a *reduced* OBDD, sometimes denoted rOBDD. We will just use OBDD and assume that it is completely reduced.

Example 2. Figure 1(c) shows the binary tree from 1(b) after the isomorphic terminal nodes have been merged. The highlighted nodes, labeled with C, are isomorphic, and Figure 1(d) shows the result after they are merged. Now, the highlighted nodes labeled with C and B are redundant, and removing them gives the OBDD in Figure 1(e).

2.2 Bucket Elimination Review

Bucket Elimination (**BE**) [19] is a well known variable elimination algorithm for inference in graphical models. An ordering $d = (X_1, X_2, \dots, X_n)$ of the variables guides the execution of **BE**. Each variable is associated with a bucket. Each constraint from **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, **BE** renders the network backtrack free, and a solution can be produced by assigning variables along d . **BE** can also produce the solutions count if marginalization is done by summation (rather than projection) over the functional representation of the constraints, and join is substituted by multiplication.

BE 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 of the constraint graph.

Example 3. Figure 2(a) shows a network with four constraints. Figure 2(b) shows the execution of Bucket Elimination along $d = (A, B, E, C, D)$. The buckets are processed from D to A ³. Figure 2(c) shows the bucket tree. The pseudo tree corresponding to the order d is given in Fig. 3(a).

³ Figure 2 reverses the top down bucket processing described in earlier papers.

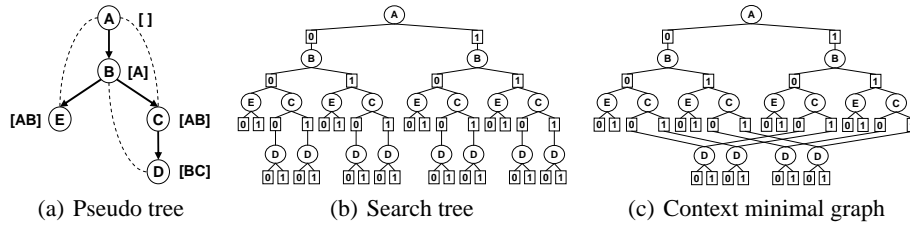


Fig. 3. AND/OR search tree

3 Overview of AND/OR Search Space for Constraint Networks

The AND/OR search space is a recently introduced [4–6] 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. Since AND/OR Multi-Valued Decision Diagrams are based on AND/OR search spaces, we provide a comprehensive overview for completeness sake.

3.1 AND/OR Search Tree

The AND/OR search tree is guided by a pseudo tree of the constraint graph. The idea is to exploit the problem decomposition into independent subproblems during search. Assigning a value to a variable (conditioning) is equivalent in graph terms to removing that variable (and its incident edges) from the constraint graph. A partial assignment can therefore lead to the decomposition of the residual constraint graph into independent components, each of which can be searched (or solved) separately. The pseudo tree captures precisely all these decompositions, given an order of variable instantiation.

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 [4].*

Example 4. Figure 3 shows an example of an AND/OR search tree for the constraint network given in Figure 2(a), assuming all tuples are consistent, and variables are binary valued. When some tuples are inconsistent, some of the paths in the tree do not exist. Figure 3(a) gives the pseudo tree that guides the search, from top to bottom, as indicated

by the arrows. The dotted arcs are backarcs from the primal graph. Figure 3(b) shows the AND/OR search tree, with the alternating levels of OR (circle) and AND (square) nodes, and having the structure indicated by the pseudo tree.

The AND/OR search tree can be traversed by a depth first search algorithm, thus using linear space. It was already shown [18, 20, 21, 4, 6] 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 expresses the set of all possible assignments to the problem variables (all solutions). The difference from the traditional OR search space is that a solution is no longer a path from root to a leaf, but rather a subtree, defined as follows:

Definition 6 (solution subtree). *A solution subtree of an AND/OR search tree contains the root node. For every OR nodes it contains one of its child nodes and for each of its AND nodes it contains all its child nodes, and all its leaf nodes are consistent.*

3.2 AND/OR Search Graph

The AND/OR search tree may contain nodes that root identical conditioned (on the current path assignment) 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 7 (minimal AND/OR graph, isomorphism). *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 (that root isomorphic subgraphs) are also said to be unifiable.*

Theorem 2 ([6]). *The minimal AND/OR search graph of a constraint network \mathcal{R} relative to a pseudo-tree \mathcal{T} is unique.*

Note that the definition of minimality in [6] is based only on isomorphism reduction. We extend it by also eliminating the redundant nodes. The previous theorem only shows that given an AND/OR graph, the merge operator has a fixed point, which is the minimal AND/OR graph. It can be shown that the AOMDD is a canonical representation (given a pseudo tree), namely that any two equivalent constraint networks can be represented by the same unique AOMDD, and the AOMDD is minimal in terms of number of nodes.

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 8 (OR context). *Given a pseudo tree \mathcal{T} of an AND/OR search space, $\text{context}(X) = [X_1 \dots X_p]$ 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 9 (context unifiable OR nodes). *Given an AND/OR search graph, two OR nodes n_1 and n_2 are context unifiable if they have the same variable label X and the assignments of their contexts is identical. Namely, if π_1 is the partial assignment of variables along the path to n_1 , and π_2 is the partial assignment of variables along the path to n_2 , then their restriction to the context of X is the same: $\pi_1|_{\text{context}(X)} = \pi_2|_{\text{context}(X)}$.*

The depth first search algorithm that traverses the AND/OR search tree, can be modified to traverse a graph, if enough memory is available. We could allocate a cache table for each variable X , the scope of the table being $\text{context}(X)$. The size of the cache table for X is therefore the product of the domains of variables in its context. For each variable X , and for each possible assignment to its context, the corresponding conditioned subproblem is solved only once and the computed value is saved in the cache table, and whenever the same context assignment is encountered again, the value of the subproblem is retrieved from the cache table. Such an algorithm traverses what is called the *context minimal AND/OR graph*.

Definition 10 (context minimal AND/OR graph). *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 [20, 4, 6]:

Theorem 3. *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} , and therefore the size of its minimal AND/OR search graph, 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 5. We refer again to Figure 3. Figure 3(a) shows the pseudo tree, where the (OR) context of each node appears in square brackets. Notice that the context of a node is identical to the message scope from its bucket in Fig. 2. Figure 3(c) shows the context minimal AND/OR graph.

4 AND/OR Multi-Valued Decision Diagram (AOMDD)

The *context minimal AND/OR graph* (Definition 10) offers an effective way of identifying some unifiable nodes during the execution of the search algorithm. Namely, context unifiable nodes are discovered based only on their paths from the root, without actually

solving their corresponding subproblems. However, merging based on context is not complete, which means that there may still exist 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 does not depend on the specific value assigned to X_i (this situation is not detectable based on context). This is sometimes termed as “interchangeable values” or “symmetrical values”.

We propose to augment the minimal AND/OR search graph with removing redundant variables as is common in OBDD representation, as well as to adopt notational conventions common in this community. This yields a data structure that we call AND/OR BDD, that exploits decomposition by using AND nodes. We present the extension over multi-valued variables yielding AND/OR MDD or AOMDD. Subsequently we present two algorithms for compiling the canonical AOMDD of a constraint network: the first is search based, and uses the memory intensive AND/OR graph search to generate the context minimal AND/OR graph, and then reduces it bottom up by applying reduction rules; the second is inference based, and uses a Bucket Elimination schedule to combine the AOMDDs of initial functions by APPLY operations (similar to the *apply* for OBDDs). Both approaches have the same worst case complexity as the AND/OR graph search with context based caching, and also the same complexity as Bucket Elimination, namely time and space exponential in the treewidth of the problem, $O(n k^{w^*})$.

4.1 From AND/OR Search Graphs to Decision Diagrams

We will now show how we can process an AND/OR search graph by reduction rules similar to the case of OBDDs, in order to obtain a representation of minimal size. In the case of OBDDs, a node is labeled with a variable name, for example A , and the *low* (dotted line) and *high* (solid line) outgoing arcs capture the restriction of the function to the assignments $A = 0$ or $A = 1$. To determine the value of the function, one needs to follow either one or the other (but not both) of the outgoing arcs from A . The straightforward extension of OBDDs to multi-valued variables (multi-valued decision diagrams, or MDDs) was presented in [22].

We generalize the OBDD and MDD representations by allowing each outgoing arc to be an AND arc. An AND arc connects a node to a set of nodes, and captures the decomposition of the problem into independent components.

We define the AND/OR Decision Diagram representation based on AND/OR search graphs. We find it useful to introduce the *meta-node* data structure, which defines small portions of any AND/OR graph, based on an OR node and its AND children:

Definition 11 (meta-node). *A meta-node u in an AND/OR search graph of a constraint network consists of an OR node labeled X (therefore $\text{var}(u) = X$) and its k AND children labeled x_1, \dots, x_k that correspond to the value assignments of X . Each AND node labeled x_i stores a list of pointers to child meta-nodes, denoted by $u.\text{children}_i$.*

We also define two special meta-nodes, that will play the role of the terminal nodes in OBDDs. The terminal meta-node **0** indicates the inconsistent assignments, while the terminal meta-node **1** indicates the consistent ones.

Any AND/OR search graph can now be viewed as a diagram of meta-nodes, simply by grouping OR nodes with their AND children, and adding the terminal meta-nodes appropriately.

It is now easy to see when a variable is redundant with respect to the outcome of the function based on the current partial assignment. Intuitively, any assignment to a redundant variable should lead to the same set of solutions.

Definition 12 (redundant meta-node). *Given an AND/OR search graph \mathcal{G} represented with meta-nodes, a meta-node u with $\text{var}(u) = X$ and $|D(X)| = k$ is redundant iff $u.\text{children}_1 = \dots = u.\text{children}_k$.*

An AND/OR graph \mathcal{G} , that contains a redundant meta-node u , can be transformed into an equivalent graph \mathcal{G}' by replacing any incoming arc into u with its common list of children $u.\text{children}_1$ (joined in an AND arc), and then removing u and its outgoing arcs from \mathcal{G} .

The notion of isomorphism is extended naturally from AND/OR graphs to meta-nodes.

Definition 13 (isomorphic meta-nodes). *Given an AND/OR search graph \mathcal{G} represented with meta-nodes, two meta-nodes u and v having $\text{var}(u) = \text{var}(v) = X$ and $|D(X)| = k$ are isomorphic iff $u.\text{children}_i = v.\text{children}_i, \forall i \in \{1, \dots, k\}$.*

Naturally, the AND/OR graph obtained by merging isomorphic meta-nodes is equivalent to the original one. We can now define the AND/OR Multi-Valued Decision Diagram:

Definition 14 (AOMDD). *An AND/OR Multi-Valued Decision Diagram (AOMDD) is a weighted AND/OR search graph that is completely reduced by isomorphic merging and redundancy removal, namely:*

- (1) *it contains no isomorphic meta-nodes; and*
- (2) *it contains no redundant meta-nodes.*

5 AOMDDs for Constraint Networks Are Canonical Forms

It is well known that OBDDs are canonical representations of Boolean functions given an ordering of the variables [9], and this property extends to MDDs [22]. In the case of AOBDDs and AOMDDs, the canonicity is with respect to a pseudo tree, following the transition from total orders (that correspond to a linear ordering) to partial orders (that correspond to a pseudo tree ordering).

Proposition 1. *Let f be a function, not always zero, defined by a constraint network over \mathbf{X} . Given a partition $\{\mathbf{X}^1, \dots, \mathbf{X}^m\}$ of the set of variables \mathbf{X} (namely, $\mathbf{X}^i \cap \mathbf{X}^j = \emptyset, \forall i \neq j$, and $\mathbf{X} = \cup_{i=1}^m \mathbf{X}^i$), if $f = f_1 \otimes \dots \otimes f_m$ and $f = g_1 \otimes \dots \otimes g_m$, such that $\text{scope}(f_i) = \text{scope}(g_i) = \mathbf{X}^i$ for all $i \in \{1, \dots, m\}$, then $f_i = g_i$ for all $i \in \{1, \dots, m\}$. Namely, if f can be decomposed over the given partition, then the decomposition is unique.*

Based on the previous proposition, it can be shown that AOMDDs for constraint networks are canonical representations given a pseudo tree.

Theorem 4 (AOMDDs are canonical for a given pseudo tree). *Given a constraint network, and a pseudo tree \mathcal{T} of its constraint graph, there is a unique (up to isomorphism) AOMDD that represents it, and it has the minimal number of meta-nodes.*

The proof, omitted for space reasons, is by structural induction over the depth of the pseudo tree.

A constraint network is defined by its relations (or functions). There exist equivalent constraint networks that are defined by different sets of functions, even having different scope signatures. However, equivalent constraint networks define the same function, and we can ask if the AOMDD of different equivalent constraint networks is the same. The following theorem can be derived immediately from Theorem 4.

Theorem 5. *Two equivalent constraint networks that admit the same pseudo tree \mathcal{T} have the same AOMDD based on \mathcal{T} .*

6 Using AND/OR Search to Generate AOMDDs

In Section 4.1 we described how we can transform an AND/OR graph into an AOMDD by applying reduction rules. In Section 6.1 we describe the explicit algorithm that takes as input a constraint network, performs AND/OR search with context-based caching to obtain the context minimal AND/OR graph, and in Section 6.2 we give the procedure that applies the reduction rules bottom up to obtain the AOMDD. The reduction procedure can actually be incorporated in the search algorithm, but we present it separately for clarity.

6.1 Algorithm AND/OR-SEARCH-AOMDD

Algorithm 1, called AND/OR-SEARCH-AOMDD, compiles a constraint network into an AOMDD. A memory intensive (with context-based caching) AND/OR search is used to create the context minimal AND/OR graph (see Definition 10). The input to AND/OR-SEARCH-AOMDD is a constraint network \mathcal{R} and a pseudo tree \mathcal{T} , that also defines the OR-context of each variable.

Each variable X_i has an associated cache table, whose scope is the context of X_i in \mathcal{T} . This ensures that the trace of the search is the context minimal AND/OR graph. A list denoted by L^{X_i} (see line 34), is used for each variable X_i to save pointers to meta-nodes labeled with X_i . These lists are used by the procedure that performs the bottom up reduction, per layers of the AND/OR graph (one layer contains all the nodes labeled with one given variable). The fringe of the search is maintained on a stack called OPEN. The current node (either OR or AND node) is denoted by n , its parent by p , and the current path by π_n . The children of the current node are denoted by $successors(n)$. For each node n , the Boolean attribute $consistent(n)$ indicates if the current path can be extended to a solution. This information is useful for pruning the search space.

Algorithm 1: AND/OR SEARCH - AOMDD

```
input   :  $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ ; pseudo tree  $\mathcal{T}$  rooted at  $X_1$ ; parents  $pa_i$  (OR-context) for every variable  $X_i$ .
output  : AOMDD of  $\mathcal{R}$ .
1 forall  $X_i \in \mathbf{X}$  do
2   | Initialize context-based cache table  $Cache_{X_i}(pa_i)$  with null entries
3 Create new OR node  $\tau$ , labeled with  $X_i$ ;  $consistent(\tau) \leftarrow true$ ; push  $\tau$  on top of OPEN
4 while OPEN  $\neq \phi$  do
5   |  $n \leftarrow top(OPEN)$ ; remove  $n$  from OPEN // Forward
6   |  $successors(n) \leftarrow \phi$ 
7   | if  $n$  is an OR node labeled with  $X_i$  then // OR-expand
8     | if  $Cache_{X_i}(asgn(\pi_n)[pa_i]) \neq null$  then
9       | Connect parent of  $n$  to  $Cache_{X_i}(asgn(\pi_n)[pa_i])$  // Use the cached pointer
10    | else
11      | forall  $x_i \in D_i$  do
12        | Create new AND node  $\tau$ , labeled with  $\langle X_i, x_i \rangle$ 
13        | if  $\langle X_i, x_i \rangle$  is consistent with  $\pi_n$  then // Constraint Propagation
14          |  $consistent(\tau) \leftarrow true$ 
15          | add  $\tau$  to  $successors(n)$ 
16        | else
17          |  $consistent(\tau) \leftarrow false$ 
18          | make terminal  $\mathbf{0}$  the only child of  $\tau$ 
19    | if  $n$  is an AND node labeled with  $\langle X_i, x_i \rangle$  then // AND-expand
20      | if  $children_{\mathcal{T}}(X_i) == \phi$  then
21        | make terminal  $\mathbf{1}$  the only child of  $n$ 
22      | else
23        | forall  $Y \in children_{\mathcal{T}}(X_i)$  do
24          | Create new OR node  $\tau$ , labeled with  $Y$ 
25          |  $consistent(\tau) \leftarrow false$ 
26          | add  $\tau$  to  $successors(n)$ 
27    | Add  $successors(n)$  to top of OPEN
28    | while  $successors(n) == \phi$  do // Backtrack
29      | let  $p$  be the parent of  $n$ 
30      | if  $n$  is an OR node labeled with  $X_i$  then
31        | if  $X_i == X_1$  then // Search is complete
32          | Call BottomUpReduction procedure // begin reduction to AOMDD
33          |  $Cache(asgn(\pi_n)[pa_i]) \leftarrow n$  // Save in cache
34          | Add meta-node of  $n$  to the list  $L^{X_i}$ 
35          |  $consistent(p) \leftarrow consistent(p) \wedge consistent(n)$ 
36          | if  $consistent(p) == false$  then // Check if  $p$  is dead-end
37            | remove  $successors(p)$  from OPEN
38            |  $successors(p) \leftarrow \phi$ 
39      | if  $n$  is an AND node labeled with  $\langle X_i, x_i \rangle$  then
40        |  $consistent(p) \leftarrow consistent(p) \vee consistent(n)$ ;
41      | remove  $n$  from  $successors(p)$ 
42      |  $n \leftarrow p$ 
```

The algorithm is based on two mutually recursive steps: **Forward** (beginning at line 5) and **Backtrack** (beginning at line 28), which call each other (or themselves) until the search terminates. In the forward phase, the AND/OR graph is expanded top down. The two types of nodes, AND and OR, are treated differently according to their semantics.

Before an OR node is expanded, the cache table of its variable is checked (line 8). If the entry is not null, a link is created to the already existing OR node that roots the graph equivalent to the current subproblem. Otherwise, the OR node is expanded by

generating its AND descendants. Each value x_i of X_i is checked for consistency (line 13). Any level of constraint propagation can be performed in this step (e.g., look ahead, arc consistency, path consistency, i-consistency etc.). The computational overhead can increase, in the hope of pruning the search space more aggressively. We should note that constraint propagation is not crucial for the algorithm, and the complexity guarantees are maintained even without it. The consistent AND nodes are added to the list of successors of n (line 15), while the inconsistent ones are linked to the terminal **0** meta-node (line 18).

An AND node n labeled with $\langle X_i, x_i \rangle$ is expanded (line 19) based on the structure of the pseudo tree. If X_i is a leaf in \mathcal{T} , then n is linked to the terminal **1** meta-node (line 21). Otherwise, an OR node is created for each child of X_i in \mathcal{T} (line 23).

The forward step continues as long as the current node is not a dead-end and still has unevaluated successors. The backtrack phase is triggered when a node has an empty set of successors (line 28). Note that, as each successor is processed, it is removed from the set of successors in line 41. When the backtrack reaches the root (line 31), the search is complete, the context minimal AND/OR graph has been generated, and the Procedure `BOTTOMUPREDUCTION` is called.

When the backtrack step processes an OR node (line 30), it saves a pointer to it in cache, and also adds a pointer to the corresponding meta-node to the list L^{X_i} . The *consistent* attribute of the AND parent p is updated by conjunction with *consistent*(n). If the AND parent p becomes inconsistent, it is not necessary to check its remaining OR successors (line 37). When the backtrack step processes an AND node (line 39), the *consistent* attribute of the OR parent p is updated by disjunction with *consistent*(n).

The AND/OR search algorithm usually maintains a value for each node, corresponding to a task that is solved (e.g., counting solutions, or cost of the optimal solution). We did not include values in our description because an AOMDD is just an equivalent representation of the original constraint network \mathcal{R} . Any task over \mathcal{R} can be solved by a traversal of the AOMDD. It is however up to the user to include more information in the meta-nodes (e.g., number of solutions for a subproblem).

6.2 Reducing the Context Minimal AND/OR Graph to an AOMDD

Procedure `BottomUpReduction` processes the variables bottom up relative to the pseudo tree \mathcal{T} . We use the depth first traversal ordering of \mathcal{T} (line 1), but any other bottom up ordering is as good. The outer *for* loop (starting at line 2) goes through each level of the context minimal AND/OR graph (where a level contains all the OR and AND nodes labeled with the same variable, in other words it contains all the meta-nodes of that variable). For efficiency, and to ensure the complexity guarantees, a hash table, initially empty, is used for each level. The inner *for* loop (starting at line 4) goes through all the meta-nodes of a level, that are also saved (or pointers to them are saved) in the list L^{X_i} . For each new meta-node n in the list L^{X_i} , in line 5 the hash table H is checked to verify if a node isomorphic with n already exists. If the hash table H already contains a node p corresponding to the hash key $(X_i, n.children_1, \dots, n.children_{k_i})$, then p and n are isomorphic and should be merged. Otherwise, if the new meta-node n

Procedure BottomUpReduction

```

input      : A constraint network  $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ ; a pseudo tree  $\mathcal{T}$  of the primal graph, rooted at  $X_1$ ;
              Context minimal AND/OR graph, and lists  $L^{X_i}$  of meta-nodes for each level  $X_i$ .
output    : AOMDD of  $\mathcal{R}$ .
1 Let  $d = \{X_1, \dots, X_n\}$  be the depth first traversal ordering of  $\mathcal{T}$ 
2 for  $i \leftarrow n$  down to 1 do
3   Let  $H$  be a hash table, initially empty
4   forall meta-nodes  $n$  in  $L^{X_i}$  do
5     if  $H(X_i, n.children_1, \dots, n.children_{k_i})$  returns a meta-node  $p$  then
6       merge  $n$  with  $p$  in the AND/OR graph
7     else if  $n$  is redundant then
8       eliminate  $n$  from the AND/OR graph
9     else
10      hash  $n$  into the table  $H$ :
11       $H(X_i, n.children_1, \dots, n.children_{k_i}) \leftarrow n$ 
12 return reduced AND/OR graph

```

is redundant, then it is eliminated from the AND/OR graph. If none of the previous two conditions is met, then the new meta-node n is hashed into the table H .

Proposition 2. *The output of Procedure BottomUpReduction is the AOMDD of \mathcal{R} along the pseudo tree \mathcal{T} , namely the resulting AND/OR graph is completely reduced.*

Note that we explicated Procedure BottomUpReduction separately only for clarity. In practice, it can actually be included in Algorithm AND/OR-SEARCH-AOMDD, and the reduction rules can be applied whenever the search backtracks. We can maintain a hash table for each variable, during the AND/OR search, to store pointers to meta-nodes. When the search backtracks out of an OR node, it can already check the redundancy of that meta-node, and also look up in the hash table to check for isomorphism. Therefore, the reduction of the AND/OR graph can be done during the AND/OR search, and the output will be the AOMDD of \mathcal{R} .

From Theorem 3 and Proposition 2 we can conclude:

Theorem 6. *Given a constraint network \mathcal{R} and a pseudo tree \mathcal{T} of its constraint graph G , the AOMDD of \mathcal{R} corresponding to \mathcal{T} has size bounded by $O(n k^{w_{\mathcal{T}}^*(G)})$ and it can be computed by Algorithm AND/OR-SEARCH-AOMDD in time $O(n k^{w_{\mathcal{T}}^*(G)})$, 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.*

7 Using Bucket Elimination to Generate AOMDDs

In this section we propose to use a Bucket Elimination (**BE**) type algorithm to guide the compilation of a constraint network into an AOMDD. The idea is to express the constraints as AOMDDs, and then combine them via the APPLY operator by following a **BE** schedule. The APPLY is a procedure very similar to that from OBDDs [9], but it is adapted to AND/OR search graphs. It takes as input two constraints represented as AOMDDs based on the same pseudo tree, and outputs their join, also represented as an AOMDD based on the same pseudo tree.

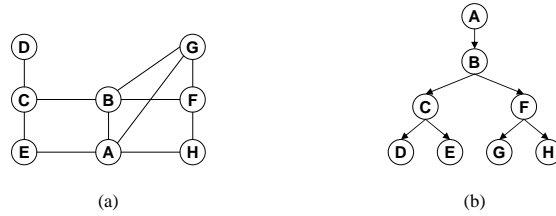


Fig. 4. (a) Constraint graph for $\mathbf{C} = \{C_1, \dots, C_9\}$, where $C_1 = F \vee H$, $C_2 = A \vee \neg H$, $C_3 = A \oplus B \oplus G$, $C_4 = F \vee G$, $C_5 = B \vee F$, $C_6 = A \vee E$, $C_7 = C \vee E$, $C_8 = C \oplus D$, $C_9 = B \vee C$; (b) Pseudo tree (bucket tree) for ordering $d = (A, B, C, D, E, F, G, H)$

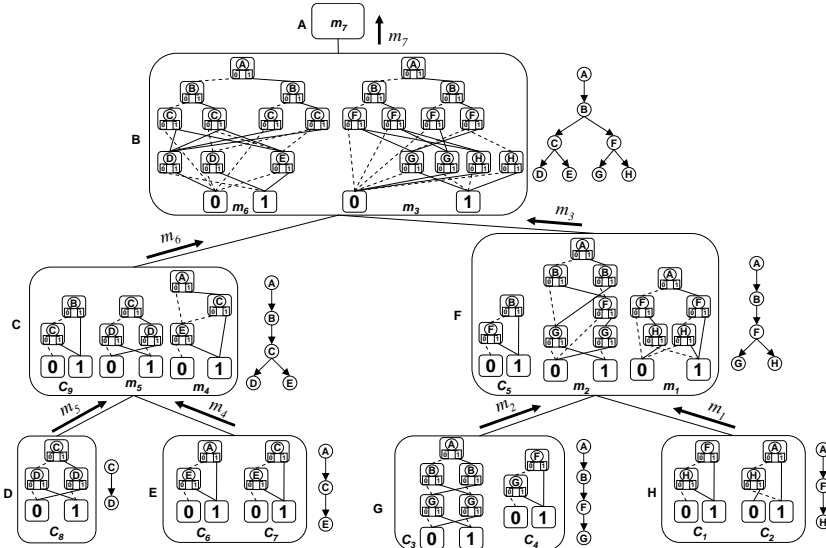


Fig. 5. Execution of VE with AOMDDs

Example 6. Consider the constraint network defined by $\mathbf{X} = \{A, B, \dots, H\}$, $D_A = \dots = D_H = \{0, 1\}$ and the constraints (where \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 4(a). 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. 4(b). Figure 5 shows the execution of **BE** 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 . The scope of any original constraint always appears on a path from root to a leaf in the pseudo tree. Therefore, each *original* constraint is represented by an AOMDD based on a chain. (i.e. there is no branching into independent components at any point). The chain is just the scope of the constraint, ordered according to d . For bi-valued variables, the original constraints are represented by OBDDs, for multiple-valued variables they are MDDs. Note that we depict meta-

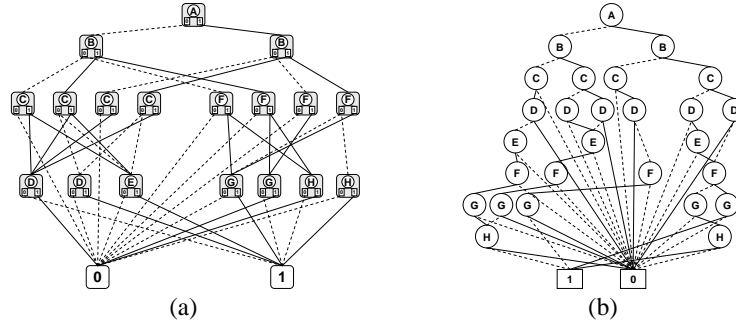


Fig. 6. (a) The final AOMDD; (b) The OBDD corresponding to d

nodes: one OR node and its two AND 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 (dotted arcs from “0” and solid arcs from “1”).

The **BE** 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 elimination of the bucket variable is 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 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 6(a). The OBDD based on the same ordering d is shown in Fig. 6(b). Notice that the AOMDD has 18 nonterminal nodes and 47 edges, while the OBDD has 27 nonterminal nodes and 54 edges.

7.1 Algorithm BE-AOMDD

Algorithm 2, called BE-AOMDD, creates the AOMDD of a constraint network by using a **BE** schedule for **APPLY** operations. Given an order d of the variables, a pseudo tree is created based on the constraint graph (this is just the bucket tree, or elimination tree of d). Each initial constraint C_i is then represented as an AOMDD, denoted by $\mathcal{G}_{C_i}^{aomdd}$, and placed in its bucket. To obtain the AOMDD of a constraint, its scope is ordered according to d , a search tree (based on a chain) that represents C_i is generated, and then reduced by Procedure **BottomUpReduction**. Then, the algorithm proceeds exactly like **BE**, with the only difference that the join of constraints (represented as AOMDDs) is realized by the **APPLY** algorithm, and variables are not eliminated but carried around to the destination bucket. The messages between buckets are initialized with the dummy AOMDD of 1, denoted by \mathcal{G}_1^{aomdd} , which is neutral for join.

Algorithm 2: BE-AOMDD

```

input   : Constraint network  $\mathcal{R} = (\mathbf{X}, \mathbf{D}, \mathbf{C})$ , where  $\mathbf{X} = \{X_1, \dots, X_n\}$ ,  $\mathbf{C} = \{C_1, \dots, C_r\}$ ;
           order  $d = (X_1, \dots, X_n)$ 
output  : AOMDD representing  $\bigwedge_{i \in \mathcal{F}} C_i$ 
1 Let  $\mathcal{T}$  be the pseudo tree (bucket tree) corresponding to  $d$ ; for  $i \leftarrow 1$  to  $r$  do // place constraints
  in buckets
2   place  $\mathcal{G}_{C_i}^{aomdd}$  in the bucket of its latest variable in  $d$ 
3 for  $i \leftarrow n$  down to 1 do // process buckets
4    $message(X_i) \leftarrow \mathcal{G}_1^{aomdd}$  // initialize with AOMDD of 1;
5   while  $bucket(X_i) \neq \emptyset$  do // combine AOMDDs in bucket of  $X_i$ 
6     pick  $\mathcal{G}_f^{aomdd}$  from  $bucket(X_i)$ ;
7      $bucket(X_i) \leftarrow bucket(X_i) \setminus \{\mathcal{G}_f^{aomdd}\}$ ;
8      $message(X_i) \leftarrow \text{APPLY}(message(X_i), \mathcal{G}_f^{aomdd})$ 
9   add  $message(X_i)$  to the bucket of the parent of  $X_i$  in  $\mathcal{T}$ 
10 return  $message(X_1)$ 

```

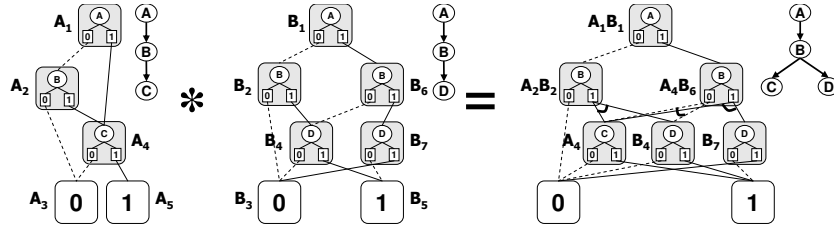


Fig. 7. Example of APPLY operation

7.2 The AOMDD APPLY Operation

The apply operator takes as input two AOMDDs representing constraints C_1 and C_2 and returns an AOMDD representing their join $C_1 \wedge C_2$. In OBDDs the *apply* operator combines two input diagrams based on the same variable ordering. Likewise, in order to combine two AOMDDs we assume that they are based on the same pseudo tree. This restriction is satisfied when we use APPLY to combine the constraints in the same bucket of the **BE** based algorithm. There are also more relaxed version of APPLY, when the pseudo trees of the two input constraints need only be *compatible*, rather than identical. Intuitively, this means that the pseudo trees generate partial orders (based on descendance relation) that are not in conflict. For space reasons, we will not present the details of the APPLY algorithm here, but refer the reader to [1]. We will just briefly describe it by an example.

Example 7. Figure 7 shows the result of combining two Boolean functions by an AND operation (or product). The input functions f and g are represented by AOMDDs based on chain pseudo trees, while the results is based on the pseudo tree that expresses the decomposition after variables A and B are instantiated. The APPLY operator performs a depth first traversal of the two input AOMDDs, and generates the resulting AOMDD based on the output pseudo tree. Similar to the case of OBDDs, a function or an AOMDD can be identified by its root meta-node. In this example the input meta-nodes have labels $(A_1, A_2, B_1, B_2, \text{etc.})$. The output meta-node labeled by A_2B_2 is

the root of a diagram that represents the function obtained by combining the functions rooted by A_2 and B_2 .

The complexity of BE-AOMDD and the output size are similar to those of the search based algorithm:

Theorem 7. *The space complexity of BE-AOMDD and the size of the output AOMDD are $O(n k^{w^*})$, where n is the number of variables, k is the maximum domain size and w^* is the treewidth of the bucket tree. The time complexity is bounded by $O(r k^{w^*})$, where r is the number of initial functions.*

8 Related Work

There are various lines of related research. The formal verification literature, beginning with [9] 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 [14]. 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.

McMillan introduced the BDD trees [16], along with the operations for combining them. For circuits of bounded tree width, BDD trees have linear space upper 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.

The AND/OR structure restricted to propositional theories is very similar to deterministic decomposable negation normal form (d-DNNF) [11]. More recently, in [23], the trace of the DPLL algorithm is used to generate an OBDD, and compared with the typical formal verification approach of combining the OBDDs of the input function according to some schedule. The structures that were investigated are still OR.

McAllester [24] 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 [4, 5], Fargier and Vilarem [17] proposed the compilation of CSPs into tree-driven automata, which have many similarities to our work. 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 Bucket Elimination and cluster-tree decomposition are in principle, the same [25].

9 Conclusion

This paper gives an overview of a new compilation data structure for constraint networks. The AND/OR Multi-valued Decision Diagram (AOMDD) [1–3] emerges from the study of AND/OR search spaces for graphical models [4, 5, 26, 6] and ordered binary decision diagrams (OBDDs) [9]. Graphical models 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 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.

We presented the two main algorithmic approaches for compiling an AOMDD for constraint networks. The first is a top down procedure, that uses memory intensive AND/OR search, and applies reduction rules to the trace of the search. The second is a bottom up procedure that uses a Bucket Elimination schedule to combine the constraints via the APPLY operator.

As part of our current and future work, we are implementing, and experimenting with, the algorithms described here in order to provide an empirical evaluation.

Acknowledgments

This work was supported in part by the NSF grants IIS-0412854 and IIS-0713118.

References

1. Mateescu, R., Dechter, R.: Compiling constraint networks into AND/OR multi-valued decision diagrams (AOMDDs). In: Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP'06). (2006) 329–343
2. Mateescu, R., Dechter, R.: And/or multi-valued decision diagrams (aomdds) for weighted graphical models. In: Proceedings of the Twenty Third Conference on Uncertainty in Artificial Intelligence (UAI'07). (2007)
3. Mateescu, R., Marinescu, R., Dechter, R.: AND/OR multi-valued decision diagrams for constraint optimization. In: Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP'07). (2007) 498–513

4. Dechter, R., Mateescu, R.: Mixtures of deterministic-probabilistic networks and their AND/OR search space. In: Proceedings of the Twentieth Conference on Uncertainty in Artificial Intelligence (UAI'04). (2004) 120–129
5. Dechter, R., Mateescu, R.: The impact of AND/OR search spaces on constraint satisfaction and counting. In: Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP'04). (2004) 731–736
6. Dechter, R., Mateescu, R.: AND/OR search spaces for graphical models. *Artificial Intelligence* **171** (2007) 73–106
7. Clarke, E., Grumberg, O., Peled, D.: *Model Checking*. MIT Press (1999)
8. McMillan, K.L.: *Symbolic Model Checking*. Kluwer Academic (1993)
9. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers* **35** (1986) 677–691
10. Selman, B., Kautz, H.: Knowledge compilation and theory approximation. *Journal of the ACM* **43** (1996) 193–224
11. Darwiche, A., Marquis, P.: A knowledge compilation map. *Journal of Artificial Intelligence Research (JAIR)* **17** (2002) 229–264
12. Cadoli, M., Donini, F.M.: A survey on knowledge compilation. *AI Communications* **10** (1997) 137–150
13. Korf, R., Felner, A.: Disjoint pattern database heuristics. *Artificial Intelligence* **134** (2002) 9–22
14. Bertacco, V., Damiani, M.: The disjunctive decomposition of logic functions. In: International Conference on Computer-Aided Design (ICCAD). (1997) 78–82
15. 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
16. McMillan, K.L.: Hierarchical representation of discrete functions with application to model checking. In: *Computer Aided Verification*. (1994) 41–54
17. Fargier, H., Vilarem, M.: Compiling csp's into tree-driven automata for interactive solving. *Constraints* **9** (2004) 263–287
18. Freuder, E.C., Quinn, M.J.: Taking advantage of stable sets of variables in constraint satisfaction problems. In: Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI'85). (1985) 1076–1078
19. Dechter, R.: Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence* **113** (1999) 41–85
20. Bayardo, R., Miranker, D.: A complexity analysis of space-bound learning algorithms for the constraint satisfaction problem. In: Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96). (1996) 298–304
21. Darwiche, A.: Recursive conditioning. *Artificial Intelligence* **125** (2001) 5–41
22. Srinivasan, A., Kam, T., Malik, S., Brayton, R.K.: Algorithms for discrete function manipulation. In: International Conference on Computer-Aided Design (ICCAD). (1990) 92–95
23. Huang, J., Darwiche, A.: Dpll with a trace: From sat to knowledge compilation. In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05). (2005) 156–162
24. McAllester, D., Collins, M., Pereira, F.: Case-factor diagrams for structured probabilistic modeling. In: Proceedings of the Twentieth Conference on Uncertainty in Artificial Intelligence (UAI'04). (2004) 382–391
25. Dechter, R., Pearl, J.: Tree clustering for constraint networks. *Artificial Intelligence* **38** (1989) 353–366
26. Mateescu, R., Dechter, R.: The relationship between AND/OR search and variable elimination. In: Proceedings of the Twenty First Conference on Uncertainty in Artificial Intelligence (UAI'05). (2005) 380–387