The Impact of AND/OR Search Spaces on Constraint Satisfaction and Counting

Rina Dechter and Robert Mateescu

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

Abstract. The contribution of this paper is in vieweing search for constraint processing in the context of AND/OR search spaces and in demonstrating the impact of this view on solutions counting. In a companion paper we introduce the AND/OR search space idea for probabilistic reasoning. In contrast to the traditional (OR) search space view, the AND/OR search tree displays some of the independencies present in the graphical model explicitly and may sometimes reduce the search space exponentially. Familiar parameters such as the depth of a spanning tree, tree-width and path-width are shown to play a key role in characterizing the effect of AND/OR search graphs vs the traditional OR search graphs. Empirical evaluation focusing on counting demonstrates the spectrum of search and inference within the AND/OR search spaces.

1 Introduction

Graphical models such as constraint networks [1], cost networks, Bayes networks [2], Markov random fields and influence diagrams [3] are knowledge representation languages that capture independencies in the knowledge and allow both concise representation and efficient graph-based algorithms for query processing. Algorithms for processing graphical models are of two types: *inference-based* and *search-based*. The latter class typically traverses the problem's search space, where each path represents a partial or a full solution and can be accomplished in linear space and exponential time. Inference algorithms provide the best worst-case guarantees, they are time exponential in the tree-width of the graph model. However, they also require space exponential in the tree-width. The virtue of search algorithms is that they can accommodate a spectrum of bounded memory algorithms, from linear space to tree-width bounded space.

In contrast to inference algorithms which exploit the independencies in the underlying graphical model effectively (e.g. variable elimination, tree-clustering), pure search is at risk of loosing this information because it is hidden in the linear structure of the search space graph. Advanced search algorithms developed for constraint satisfaction, and more recently for probabilistic reasoning can be viewed as attempting to overcome this difficulty.

The primary contribution of this paper is in viewing search for constraint processing in the context of *AND/OR search spaces* rather than *OR spaces*. We demonstrate how the AND/OR principle can exploit independencies in the graph model to yield exponentially smaller search spaces. In the companion paper we introduced the AND/OR idea

for reasoning with probabilistic networks [4]. Here we focus on constraint processing with its traditional tasks of deciding consistency, finding a solution and counting the solutions. In particular, we present and analyze counting algorithms when formulated as searching an AND/OR search tree or graph rather than searching their OR counterparts and provide initial empirical evaluation along the full spectrum of space and time. Specifically, we compare counting algorithms on the AND/OR search space when pruning is accomplished by two forward-checking strategies and show how their performance is affected by different levels of caching and how it is compared to bucket-elimination, as a function of problem tightness.

Following some preliminaries (section 2) we present an overview of the notion of AND/OR search tree and graphs specialized for constraint processing (sections 3 and 4). Subsequently we discuss unique aspects to constraint problems (section 5), describe counting algorithms formulated over the AND/OR space (section 6) and demonstrate a spectrum of these algorithms empirically (section 7). Section 8 provides discussion and related work.

2 Preliminaries and Background

A **Reasoning graphical-model** is a triplet $\mathcal{R} = (X, D, F)$ where X a set of variables $X = \{X_1, ..., X_n\}$, D is their respective domains of values $D = \{D_1, ..., D_n\}$ and F is a set of real-valued functions $F = \{F_1, ..., F_t\}$. Each function F_i is defined over a subset of variables S_i, called its scope, $S_i \subseteq X$. The *primal graph* of a reasoning problem, $G_{\mathcal{R}}$, has a node for each variable, and any two variables appearing in the same function's scope are connected.

In **constraint networks**, denoted $\mathcal{R} = (X, D, C)$, the functions F are constraints C. Each constraint is a pair $C_i = (S_i, R_i)$, where $S_i \subseteq X$ is the scope of the relation R_i defined over S_i, denoting the allowed combination of values. A solution is an assignment of values to each variable that does not violate any constraint. The primary queries over constraint network is to determine if the network is consistent, and if it is, to find one or all solutions. A related task is to compute the number of solutions.

Induced-graphs, induced width and path-width. 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* is the number of the neighbors that precede it in the ordering. The *width of an ordering* 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: nodes are processed from last to first; when node X is processed, all its preceding neighbors are connected. The *induced width of a graph*, $w*$, is the minimal induced width over all its orderings. The set of maximal cliques (also called clusters) in the induced graph provide a tree-decomposition of the graph. The tree-width is the maximal number of variables in a cluster of an optimal cluster-tree decomposition of the graph [5]. It is well known that the induced-width of a graph is identical to its *tree-width*. The path-width of a graph is the smallest induced-width along a chain-like decomposition. The optimal path-width is denoted pw^*. For various relationships see [5, 6].

AND/OR search spaces. An AND/OR state space representation of a general state space problem formulation is defined by a 4-tuple (S, O, S_g, s_0): S is a set of states,

which can be OR states or AND states (the OR states represent alternative ways for solving the problem while the AND states often represent problem decomposition into subproblems, all of which need to be solved); O is a set of operators (an OR operator transforms an OR state into another state, and an AND operator transforms an AND state into a set of states); there is a set of goal states $S_g \subseteq S$ and a start node s_0. Example problem domains modeled by AND/OR graphs are two-player games, parsing sentences, Tower of Hanoi.

The AND/OR states space model induces an explicit AND/OR search *graph*. Each node is a state and its child nodes are those obtained by applicable AND or OR operators. The search graph includes a *start* node. The terminal nodes (having no child nodes) are marked as Solved (S), or Unsolved (U).

A **solution subgraph** of an AND/OR search graph G is a subtree which: (1) contains the start node s_0; (2) if n in the subtree is an OR node then it contains one of its child nodes in G and if n is an AND nodes it contains all its children in G; (3) all its terminal nodes are "Solved" (S). The primary tasks defined over an AND/OR graph is to determine the value of the root node, and if it is solved, to find a solution subtree with optimal cost if a cost is defined.

3 AND/OR Search Tree for Constraint Networks

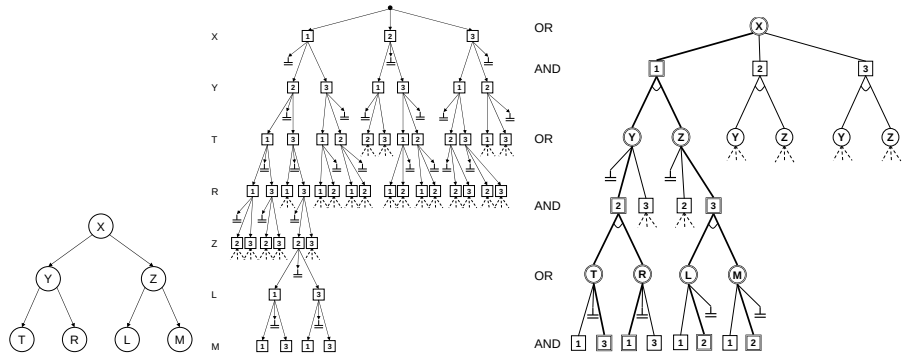
Consider a constraint network $\mathcal{R} = (X, D, C)$ and its primal graph G. Let T be a DFS spanning tree of its primal graph rooted at X_0. For each node Y, $ch(Y)$ is the set of its children in T. We will now define the AND/OR search tree of $\mathcal{R}$ relative to T.

Definition 1 (AND/OR search tree based on DFS tree). *Given a constraint network $\mathcal{R}$ and its DFS spanning tree T rooted at X_0, the AND/OR search tree of $\mathcal{R}$ based on T, denoted $S_T(\mathcal{R})$ (or just S_T when $\mathcal{R}$ is unambiguous) is defined as follows. The nodes in the search tree are either OR nodes (e.g., X, Y), denoting variables, or AND nodes denoting variable-value assignment pairs (e.g., $\langle X, v\rangle$). The path of $\langle X, v\rangle$ is the path from the initial state (the root X_0) to $\langle X, v\rangle$, which corresponds to a* consistent *partial value assignments to all the variables along the path. The successors of a node in the AND/OR search tree are defined as follows:*

- *The successor-nodes of an OR node X are all its possible value assignments $\{\langle X, v\rangle \mid v \in D_X\}$ that are consistent with the path to $\langle X, v\rangle$. (The path alternates OR and AND nodes like ($X_0, \langle X_0, v_0\rangle, X_1, \langle X_1, v_1\rangle, ..., X_i, \langle X_i, v_i\rangle ...$)*
- *The successor-nodes of an AND node $\langle X, v\rangle$ are all child nodes, $ch(X)$, in T.*

The consistency of a partial path is determined by considering all the relevant constraints whose scopes are contained in the path.

Definition 2 (labeling of AND/OR nodes). *Given an AND/OR tree $S_T(\mathcal{R})$, a terminal AND node (no child nodes in T) is always labeled "Solved" or "1". A terminal OR node is always "Unsolved" or "0" (no consistent value assignments). The labeling of internal nodes is defined recursively and is dependent on the specific task and the specific graphical-model. For the task of finding a solution an OR internal node is*



(a) A constraint tree (b) OR search tree (c) AND/OR search tree with one of its solution subtrees

Fig. 1. OR vs. AND/OR search trees; note the connector for AND arcs

labeled "1" iff one of its successor nodes is "1" and an AND internal node is labeled "1" iff all its successor nodes are labeled "1". For counting the number of solutions, an OR node is the sum of the values of its successors and an AND node is labeled by their product.

The virtue of an AND/OR search tree representation is that its size may be far smaller than the traditional OR tree representation.

Example 1. Consider the graph coloring problem in Figure 1(a), over domains $\{1, 2, 3\}$. Its OR search tree along the DFS ordering (X, Y, T, R, Z, L, M) is given in Figure 1(b) and its AND/OR search tree based on the DFS tree T rooted at X, is given in Figure 1(c). A solution subtree is highlighted in Figure 1(c). We see that the size of the traditional OR search tree is $O(2^7)$, while the size of the AND/OR search tree is $O(4 \cdot 2^3)$ (we ignore the OR levels when counting nodes because they incur at most a constant factor and we believe a non-naive implementation does not need to express OR nodes explicitly). Notice that if we add one constraint between X and R to this problem, we can still use the same DFS tree T yielding a similar structure AND/OR search tree, except that some values of variable R are no longer consistent with all their predecessors on the partial path. For example, in that case, $\langle R, 1\rangle$ in Figure 1c will not be present under the subtree of $\langle X, 1\rangle$.

Legal trees. The construction of AND/OR search trees can use as its basis not just a DFS spanning tree but a larger collection of spanning trees that we call *legal trees*. This generalization accommodates many more trees and can therefore yield better AND/OR search spaces.

Definition 3 (legal tree of a graph). *Given an undirected graph $G = (V, E)$, a directed rooted tree $T = (V, E')$ defined on all its nodes is* legal *if any arc of G which is not included in E' is a back-arc, namely it connects a node to an ancestor in T. The arcs in E' may not all be included in E. Given a legal tree T of G, the* extended graph *of G relative to T is defined as $G^T = (V, E \cup E')$.*

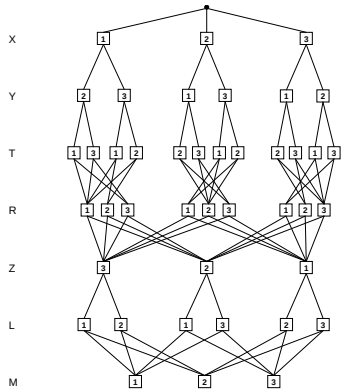


Fig. 2. Condensed OR graph for the tree problem in Figure1a

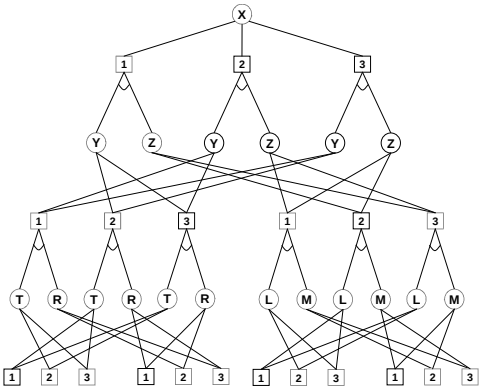


Fig. 3. The AND/OR search graph of the tree graphical-model in Figure 1a

Clearly, any DFS tree and any chain are legal trees. Searching the OR space corresponds to searching a chain that is a special legal tree. It is easy to see that searching an AND/OR tree is exponential in the depth of the legal tree. Finding a legal or a DFS tree of minimal depth is known to be NP-complete. However the problem was studied, and various greedy heuristics are available. The following relationship between the induced-width and the depth of legal trees is well known [7, 1]. Given a tree-decomposition of a primal graph G having n nodes, whose tree-width is w^*, there exists a legal tree T of G whose depth, m, satisfies: $m \leq w^* \cdot \log n$. In summary (for more details on the impact of legal trees see [4]),

Theorem 1. *Given a graphical model $\mathcal{R}$ and a legal tree T, its AND/OR search tree $S_T(\mathcal{R})$ is sound and complete (contains all and only solutions) and its size is $O(n \cdot \exp(m))$ where m is the legal tree's depth. A graphical model that has a tree-width w^* has an AND/OR search tree whose size is $O(\exp(w^* \cdot \log n))$.*

4 AND/OR Search Graphs

It is often the case that certain states in the search tree can be merged because the subtree they root are identical. Any two such nodes are called *unifiable*, and when merged, transform the search tree into a search graph. For example, in Figure 1(c), the search trees below any appearance of $\langle Y, 1\rangle$ are identical, so all nodes $\langle Y, 1\rangle$ are unifiable.

4.1 Minimal AND/OR Search Graphs

A partial path in the AND/OR search-tree S_T $(\langle X_1, a_1\rangle, \langle X_2, a_2\rangle, ..., \langle X_i, a_i\rangle)$ is abbreviated to $(\bar{X}, \bar{a}_i)$, where $\bar{X}$ is the sequence of variables and $\bar{a}$ is their corresponding sequence of value assignments.

Definition 4 (legal transformation). *Given two partial paths over the same set of variables, $s_1 = (\bar{X}_i, \bar{a}_i)$, $s_2 = (\bar{X}_i, \bar{b}_i)$ where $a_i = b_i = v$, we say that s_1 and s_2 are*

unifiable *at* $\langle X_i, v\rangle$ *(can be merged) iff the search subgraphs rooted at* s_1 *and* s_2 *are identical. The* Merge *operator over search graphs,* $Merge_{(s_1,s_2)}$ *transforms* S_T *into a graph* S'_T *by merging* s_1 *with* s_2.

It can be shown that given an AND/OR search graph, its closure under the merge operator yields a unique fixed point, called the minimal AND/OR search graph.

Definition 5 (minimal AND/OR search graph). *The minimal AND/OR search graph relative to* T *is the closure under* merge *of the AND/OR search tree* S_T.

The above definition is applicable, via the legal-chain definition, to the traditional OR search tree as well. However, in many cases we will not be able to reach the compression we see in the AND/OR search graph, because of the linear structure imposed by the OR search tree.

Example 2. The smallest OR search graph of the problem in Figure 1(a) is given in Figure 2 along the DFS order $d = (X, Y, T, R, Z, L, M)$. The smallest AND/OR graph of the same problem along some DFS tree is given in Figure 3. We see that some variable-value pairs must be repeated in Figure 2 while in an AND/OR case they appear just once. For example, the subgraph below the paths $\langle X, 1\rangle, \langle Y, 2\rangle$ and $\langle X, 3\rangle, \langle Y, 2\rangle$ in Figure 2 cannot be merged.

4.2 Rules for Merging Nodes

Given a graphical model $\mathcal{R} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and a legal tree T, there could be many AND/OR graphs relative to T that are equivalent to the AND/OR search tree S_T, each obtained by some sequence of merging. The following rules provide an efficient way for generating such graphs without creating the whole search tree first.

It uses a variant definition of *induced-width of a legal tree of G* which is instrumental for characterizing OR graphs vs. AND/OR *graphs*. We denote by $d_{dfs}(T)$ a DFS ordering of a tree T.

Definition 6 (induced-width of a legal tree). *Given* G^T*, an extended graph of* G *relative to* T*, the* induced width of G relative to legal tree T, $w_T(G)$ *is the induced-width of* G^T *along* $d_{dfs}(T)$.

We can show that, 1. The minimal induced-width of G relative to all legal trees is identical to the induced-width (tree-width) of G. 2. The induced-width of a legal chain d is identical to its path-width $pw(d)$ along d.

Next we provide a general generative rule for unifying nodes in the AND/OR search graph and will use the generalized width parameter to characterize its effect. Given an induced graph of G^T, denoted G^{*T}, each variable and its parent set is a clique. We associate each variable with its *parents* and *parent-separators*:

Definition 7 (parents, parent-separators). *Given the induced-graph,* G^{*T}*, the parents of* X *denoted* ps_X*, are its earlier neighbors in the induced-graph. Its parent-separators,* psa_X *are its parents that are also neighbors of future variables in* T.

Note that for every node except those latest in the clique, the parent-separators are identical to the parents. For nodes latest in cliques, the parent-separators are the separators between cliques.

In G^{*T}, for every node X_i, the parent-separators of X_i separate in T its ancestors on the path from the root, and all its descendents in G^T.

Theorem 2. *Given G^{*T}, let $s_1 = (\bar{a}_i, \langle X_{i+1}, v\rangle)$ and $s_2 = (\bar{b}_i, \langle X_{i+1}, v\rangle)$ be two partial paths of assignments in its AND/OR search tree S_T, ending with the same assignment variable $\langle X_{i+1}, v\rangle$. If $s_1[psa_{i+1}] = s_2[psa_{i+1}]$ then the AND/OR search subtrees rooted at s_1 and s_2 are identical and s_1 and s_2 can be merged at $\langle X_{i+1}, v\rangle$.*

Definition 8 (context). *For every state s_i, $s[psa_i]$ is called the context of s_i when psa_i is the parent-separators set of X_i relative to the legal tree T.*

Theorem 3. *Given G, a legal tree T and its induced width $w = w_T(G)$, the size of the AND/OR search graph based on T obtained when every two nodes in S_T having the same context are merged is $O(n \cdot k^w)$, when k bounds the domain size.*

Thus, the minimal AND/OR search graph of G relative to T is $O(n \cdot k^w)$ where $w = w_T(G)$. Since $\min_T\{w_T(G)\}$ equals w^* and since $\min_{T \in chain}\{w_T(G)\}$ equals pw^* we get,

Corollary 1. *The minimal AND/OR search graph is bounded exponentially by the primal graph's tree-width while the OR minimal search graph is bounded exponentially by its path-width.*

It is well known [6] that for any graph $w^* \leq pw^* \leq w^* \cdot \log n$. It is also easy to place m^* (the minimal depth legal tree) yielding $w^* \leq pw^* \leq m^* \leq w^* \cdot \log n$.

In our companion paper [4] we showed that for some graphical models the difference between the tree-width and path-width can be substantial. In fact for balanced trees the tree-width is 1 while the path-width is $\log n$ when n is the number of variables yielding a substantial difference between OR and AND/OR search graphs.

5 Unique Aspects to Constraints

5.1 Pruning Inconsistent Subtrees

Most advanced constraint processing algorithms incorporate no-good learning during search, or use variable-elimination algorithms such as *adaptive-consistency* [8], generating all relevant no-goods, prior to search. Such schemes can be viewed as traversing a *pruned* AND/OR search tree. We next define the *backtrack-free* AND/OR search tree.

Definition 9 (backtrack-free AND/OR search tree). *Given an AND/OR search tree $S_T(\mathcal{R})$ whose internal nodes are labeled by $0/1$, the backtrack-free AND/OR search tree of $\mathcal{R}$ based on T, denoted $BF_T(\mathcal{R})$, is obtained by pruning all subtrees labeled "0" from $S_T(\mathcal{R})$.*

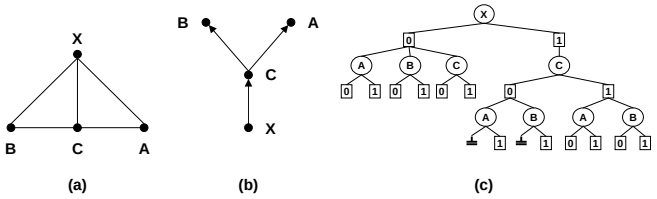


Fig. 4. (a) A constraint graph; (b) a spanning tree; (c) a dynamic AND/OR tree

Clearly, if we traverse the AND/OR backtrack-free search space we can find a solution after traversing a single solution subgraph of the backtrack-free search graph. It is well known that if we apply variable-elimination algorithms such as *adaptive-consistency* in a reversed order of d then we can find a solution with no dead-end.

Definition 10 (directional extension). *Let $\mathcal{R}$ be a constraint problem and let d be a DFS traversal ordering of a legal tree of its primal graph, then we denote by $E_d(\mathcal{R})$ the constraint problem compiled by Adaptive-consistency in the reverse order of d.*

Proposition 1. *The AND/OR search tree $S_T(E_d(\mathcal{R}))$ when d is a DFS ordering of T, is identical to the backtrack-free AND/OR search tree of $\mathcal{R}$ based on T. Namely $S_T(E_d(\mathcal{R})) = BF_T(\mathcal{R})$.*

Proposition 1 emphasizes the significance of no-good learning [9] for deciding inconsistency or for finding a single solution. If the search-space is backtrack-free its size is irrelevant for the consistency query. For counting however, pruning inconsistent subtrees yields partial help only.

5.2 Using Dynamic Variable Ordering

The AND/OR search tree we defined used a fixed variable ordering. It is known that exploring the search space in a dynamic variable ordering is highly beneficial. AND/OR search trees for graphical models can be modified to allow dynamic variable ordering. A dynamic AND/OR tree that allows varied variable ordering has to satisfy that for every subtree in the search tree rooted in $\langle X, v\rangle$, there are no *active* constraints (or arcs) in the subproblem conditioned on the current path to $\langle X, v\rangle$ that connect different branches of the tree.

Example 3. Consider the propositional formula $X \rightarrow A \vee C$ and $X \rightarrow B \vee C$. The constraint graph is given in Figure 4(a) and a dfs tree in Figure 4(b). However, the constraint subproblem conditioned on $\langle X, 0\rangle$, has no real constraint between A, B, C, so the effective spanning tree below $\langle X, 0\rangle$ is $\{\langle X, 0\rangle \rightarrow A, \langle X, 0\rangle \rightarrow B, \langle X, 0\rangle \rightarrow C\}$, yielding the AND/OR search tree in Figure 4(c). Note that while there is an arc between A and C in the constraint graph, the arc is *not active* when X is assigned the value 0.

Clearly, the constraint graph conditioned on any partial assignment can only be sparser than the original graph and therefore may yield a smaller AND/OR search tree than with fixed ordering.

5.3 Merging and Pruning: Two Orthogonal Concepts

Notice that the notion of minimality vs that of pruning of inconsistent subtrees (yielding the backtrack-free search space) are orthogonal. When we merge two subtrees whose root value is "0" (unsolved) via their context, we can still keep around all the subtree of the merged node roots. On the other hand, recording a no-good, implies that we prune the subtree below that node and summarize the value of this subtree by "0". Recording no-goods, namely the parent-context of an unsolved node, can be used as a new constraint which can affect and prune the search tree itself.

Therefore, we can have a minimal search graph that is NOT backtrack-free as well as a search tree that is backtrack-free. When the search space is backtrack-free (no dead-end nodes) and if we seek a single solution, the size of the minimal AND/OR search graph and its being OR vs AND/OR are both irrelevant. It will, however, affect a traversal algorithm that counts all solutions or compute an optimal solution. This means that for deciding consistency or for finding the first solution, once we record no-goods, backjumping becomes irrelevant, as was often observed [10].

Another independent option is to prune full subtrees. In other words, if an AND node roots a full consistent AND/OR search subtree that expresses the cartesian product of all value below the node, it can be pruned and its internal label be summarized as "1". This is exactly what happens in OBDD representation of a CNF formula [11]. An OBDD is a minimal OR search graph that is collapsed as far as possible. A tree-obdd is a minimal AND/OR search graph that fully collapsable [12].

6 AND/OR Algorithms for Counting

6.1 Counting Over the AND/OR Search Tree

Any algorithm that traverses the AND/OR search tree in a depth-first manner is guaranteed to have time bound exponential in the depth of the legal tree of the graphical model and may operate using linear space.

igure 5 presents the basic DFS traversal of the AND/OR search tree for counting the number of solutions of a constraint network. The reader should ignore the bracketed lines. Given a legal tree T, AND-OR-COUNTING places the root of T in OPEN and starts the current solution subgraph assembled by the algorithm. The algorithm picks the top node in OPEN and expands it generating its succesors. If it is an OR node, X, it generates its value assignments that are consistent with the value assignments along its path. The generated nodes, labeled "AND", are recorded in the current solution subgraph and placed on top of OPEN. When an AND node $\langle X, v\rangle$ is expanded, its successors are $ch(X)$ in T.

The nodes in the tree are labeled by g-values. These stand for the number of solutions below that variable (or variable-value) in the constraint subproblem restricted to the variables rooted at that node in T. The terminal nodes are evaluated as either solved, (g=1) if they are AND nodes, or unsolved (g=0) if they are OR nodes. The g-values of internal nodes are evaluated in step 4. The g-value of an AND node is the product of the g-values of its successors, while the g-value of an OR node is the sum of the g-values of its successors. The algorithm terminates when the root is labeled.

procedure AND-OR-COUNTING
Input: A constraint network $\mathcal{R} = (X, D, C)$; a legal tree, T, of its constraint graph rooted at X_0; separator parents psa_i and parents pa_i for every variable X_i.
Output: The number of solutions $g(X_0)$. π denotes the current partial assignment path.

1. Initialize OPEN (OP): OPEN $\leftarrow X_0$, type(X_0) = OR; Create a search graph G, $G \leftarrow X_0$;
   Create a list called CLOSED (CL), initially empty.
2. $n \leftarrow$ first node in OPEN, move to CLOSED
3. Expand n generating all its successors as follows:
   if $type(n) = OR$, $n = X$
   succ(X) $\leftarrow \{\langle X, v\rangle | consistent(\langle X, v\rangle)\}$
   if $succ(X) = \Phi$, $g(X) = 0$; (deadend)
   [[$cache - nogood \leftarrow \pi_{pa_X}$, update constraints and go to **Propagate**]]
   else , add all $succ(X)$ to G, set pointers back to X.
   for each $\langle X, v\rangle \in succ(X)$ do, $\pi \leftarrow \pi \cup (\langle X, v\rangle)$
   [[**if** π_{psa_X} is a new context not in $OP \cup CL$, then $c = c(\langle X, v\rangle) = \pi_{psa_X}$]]
   add $\langle X, v\rangle$ to OP.
   [[**else** , there exists, $m \in OP \cup CL$ s.t $c(m) = c$, unify $\langle X, v\rangle$ with m; merge]]
   EndFor
   If $type(n) = AND$; $n = \langle X, v\rangle$
   if X is a leaf in T then, $g(\langle x, v\rangle) = 1$,
   remove $\langle X, v\rangle$ from OP
   $succ(\langle X, v\rangle) \leftarrow \{Y | Y \in ch(X) \ in \ T\}$
   put $succ(\langle X, v\rangle)$ on top of OPEN and in G
   set back-pointers.
4. **Propagate: while** you can propagate g values:
   a. For a non-terminal AND node $\langle X, v\rangle$:
   [[**if** $Y \in succ(\langle X, v\rangle)$ and $g(Y) = 0$,
   remove siblings of Y from OPEN and from G.
   set $g(\langle X, v\rangle) = 0$.]]
   else ,
   if all $succ(\langle X, v\rangle)$ are evaluated, $g(\langle X, v\rangle) = \Pi_{Y \in succ(\langle X, v\rangle)} g(Y)$
   b. For a non-terminal OR node X:
   if all succ(X) have g values, $g(X) = \sum_{\langle X, v\rangle \in succ(X)} g(\langle X, v\rangle)$
   end while
   c. **if** X_0 was evaluated, exit with $g(X_0)$
5. Go to step 2.

end procedure

Fig. 5. The counting algorithm.

We can easily modify the algorithm to find a single solution yielding algorithm AND-OR-FIND-SOLUTION, (not presented for lack of space). The main difference is that the $0/1$ g-values of internal nodes are propagated using Boolean summation and product instead of regular operators, yielding the 0/1 labeling. From Theorem 1 we can conclude that,

Theorem 4. *The complexities of* AND-OR-COUNTING *and* AND-OR-FIND-SOLUTION *are space linear and time* $O(nk^m)$*, where* m *is the depth of the legal tree of its constraint graph. If the constraint graph has a tree-decomposition with tree-width* w^**, both algorithms have time complexity* $O(n \cdot \exp(w^* \cdot \log n))$.

Obviously, the ability to terminate early with first solution makes AND-OR-FIND-SOLUTION much faster than AND-OR-COUNTING in practice.

6.2 Counting Over the AND/OR Search Graphs

Any algorithm traversing the AND/OR search graph needs to record nodes during search so merging would be possible. To do that the algorithm associates a context

N=20, K=3, C=20, S=4, 20 instances, w*=9, h=14															
t		10%		20%		30%		40%		50%		60%		70%	
# solutions		0		0		0		49		3,842		126,957		2,856,064	
Time (seconds)															
	BE	0.10110		0.10155		0.10115		0.10025		**0.10000**		**0.08970**		**0.08805**	
i=0	A/O FC	0.00650		0.01250		**0.02450**		**0.06555**		0.22940		1.09355		5.81740	
	A/O RFC	**0.00350**		**0.01005**		0.02555		0.07660		0.27490		1.33295		6.94850	
	OR FC	0.00505		0.01200		0.02755		0.08670		0.52620		5.49720		65.68775	
	OR RFC	0.00400		0.01255		0.02800		0.09870		0.56040		5.72635		67.94275	
i=3	A/O FC	0.00550		0.01210		0.02555		**0.06410**		0.22925		1.09505		5.79485	
	A/O RFC	**0.00300**		0.01305		0.02550		0.07810		0.27850		1.33705		6.90190	
	OR FC	0.00555		0.01250		0.02750		0.08765		0.52405		5.48500		65.83190	
	OR RFC	0.00400		**0.01000**		0.02810		0.09820		0.56400		5.72880		67.98520	
i=6	A/O FC	0.00500		0.01250		**0.02405**		**0.06455**		0.21370		0.91375		4.33875	
	A/O RFC	0.00500		0.01100		0.02750		0.07555		0.25930		1.09625		5.08375	
	OR FC	0.00450		0.01250		0.02960		0.08860		0.49920		4.66985		49.77530	
	OR RFC	**0.00300**		0.01050		0.03200		0.09805		0.53625		4.87520		51.24910	
i=9	A/O FC	0.00455		0.01155		0.02500		**0.06405**		0.17240		0.48865		1.22135	
	A/O RFC	0.00450		**0.00950**		0.02600		0.07310		0.20530		0.58830		1.46265	
	OR FC	0.00550		0.01355		0.02950		0.08160		0.40010		2.98980		23.39555	
	OR RFC	0.00450		0.01150		0.03020		0.09415		0.43620		3.15515		24.25300	
Number of expanded nodes (# n) / Number of dead-ends (# d)															
		# n	# d	# n	# d	# n	# d	# n	# d	# n	# d	# n	# d	# n	# d
i=0	A/O FC	225	453	518	1032	1192	2330	3552	6579	16003	24402	106651	119059	735153	553820
	A/O RFC	154	311	387	771	1052	2056	3407	6307	15737	23987	106617	118989	735153	553820
	OR FC	225	453	519	1040	1203	2408	3810	7476	28079	44634	414463	448055	6533674	4499159
	OR RFC	154	311	387	777	1062	2126	3664	7183	27801	44078	414428	447986	6533674	4499159
i=3	A/O FC	225	453	518	1032	1192	2330	3552	6579	16003	24402	106651	119059	735153	553820
	A/O RFC	154	311	387	771	1052	2056	3407	6307	15737	23987	106617	118989	735153	553820
	OR FC	225	453	519	1040	1203	2408	3810	7476	28079	44634	414463	448055	6533674	4499159
	OR RFC	154	311	387	777	1062	2126	3664	7183	27801	44078	414428	447986	6533674	4499159
i=6	A/O FC	224	451	512	1021	1162	2285	3306	6269	12765	21129	70273	88589	436554	368111
	A/O RFC	154	311	384	765	1028	2019	3175	6012	12562	20776	70238	88519	436554	368111
	OR FC	225	453	519	1040	1203	2408	3764	7418	24700	41194	294525	349350	3931078	3068920
	OR RFC	154	311	387	777	1062	2126	3618	7124	24422	40638	294491	349281	3931078	3068920
i=9	A/O FC	224	449	499	978	1093	2112	2883	5288	8873	14193	28038	33210	**79946**	**60144**
	A/O RFC	153	308	371	722	962	1857	2761	5063	8705	13899	28003	33141	79946	60144
	OR FC	225	453	518	1032	1192	2333	3604	6874	18729	30992	166912	203854	**1516976**	**1259120**
	OR RFC	154	311	387	771	1052	2058	3461	6597	18457	30477	166877	203784	1516976	1259120

Table 1. AND/OR search vs. OR search vs. Bucket Elimination

with each AND node and whenever a new AND node is generated, its context is compared against the list of contexts for the same variable in the same level. If it was already generated, only pointers will be established appropriately.

Algorithm AND-OR-GRAPH-COUNTING is presented in Figure 5. For this version the reader should include all the bracketed lines. In step 3, the algorithm expands the next node in OPEN (OP). If this is an OR node that has no consistent successors, it is identified as dead-end and assigned $g = 0$. A no-good is recorded and the set of constraints are modified to include this new constraint. This step will cause pruning of the search tree. Otherwise, for each consistent, value v of X the algorithm computes the context of $\langle X, v\rangle$ denoted $c(\langle X, v\rangle)$ and check it against recorded contexts.

Theorem 5. *The complexity of algorithm* AND-OR-GRAPH-COUNTING *is time and space exponential in the induced width of the legal tree, which is identical to the tree-width. For OR space, the complexity is exponential in the path-width.*

N=40, K=3, C=50, S=3, 20 instances, w*=13, h=20													
t		10%		20%		30%		40%		50%		60%	
# solutions		0		0		0		0		46582		147898575	
Time (seconds)													
	BE	8.674		8.714		8.889		8.709		8.531		**8.637**	
i=0	A/O FC	**0.011**		**0.030**		**0.110**		**0.454**		**3.129**		32.931	
	OR FC	**0.009**		0.031		0.113		0.511		14.615		9737.823	
i=3	A/O FC	0.011		0.031		0.111		**0.453**		**3.103**		31.277	
	OR FC	0.009		0.030		0.112		0.509		14.474		9027.365	
i=6	A/O FC	0.011		0.029		0.110		0.454		**3.006**		25.140	
	OR FC	0.010		0.032		0.113		0.508		13.842		7293.472	
i=9	A/O FC	0.010		0.030		0.114		0.453		**2.895**		21.558	
	OR FC	0.010		0.031		0.111		0.509		12.336		5809.917	
i=13	A/O FC	0.011		0.030		0.111		0.457		**2.605**		**11.974**	
	OR FC	0.010		0.032		0.123		0.494		8.703		1170.203	
Number of expanded nodes (# n) / Number of dead-ends (# d)													
		# n	# d	# n	# d	# n	# d	# n	# d	# n	# d	# n	# d
i=0	A/O FC	78	159	265	533	999	1994	4735	9229	60163	101135	1601674	1711947
	OR FC	78	159	265	533	1000	2003	4947	9897	273547	407350	384120807	324545908
i=3	A/O FC	78	159	265	533	986	1990	4525	9166	46763	98413	689154	1625075
	OR FC	78	159	265	533	1000	2003	4947	9897	224739	399210	228667363	287701079
i=6	A/O FC	78	159	265	533	981	1971	4467	8991	41876	85583	487320	917612
	OR FC	78	159	265	533	1000	2003	4947	9897	185422	329754	141610990	208159068
i=9	A/O FC	78	159	265	533	981	1958	4451	8866	37314	70337	362024	580325
	OR FC	78	159	265	533	1000	2003	4947	9897	147329	270446	102316417	135655353
i=13	A/O FC	78	159	265	533	981	1955	4415	8533	30610	50228	**170827**	**181157**
	OR FC	78	159	265	533	999	1994	4761	9283	99923	176630	**16210028**	**20018823**

Table 2. AND/OR search vs. OR search vs. Bucket Elimination

7 Empirical Demonstration

We present here an empirical evaluation of the counting algorithm. We ran two different versions of it, using forward checking (FC) and relational forward checking (RFC) as the constraint propagation methods. This was done by defining the *consistent* function in step 3 of the algorithm accordingly. RFC is a bit more costly computationally, but its search space is at most as big as that of FC. For the smaller problems, the algorithms were ran on AND/OR and OR search spaces, resulting in a total of four algorithms:A/O FC, A/O RFC, OR FC, OR RFC. We only mention that the basic version of the counting algorithm for which *consistent* function only checks the existing constraints, but does no propagation, was in general too slow so we do not include those results here. For each of the above four algorithms, we tried different levels of caching, controlled by an *i-bound*, from 0 up to as much as our computer memory permitted (this equals $w*$ for the smaller problems). The i-bound is the maximum context size that can be cached. We also compared against bucket elimination (BE) in some cases, where space was available. We report average measures over 20 instances: *time* (in seconds), *number of expanded nodes (#n), number of deadends (#d)* and *number of solutions (# sol)*. Also, $w*$ is the induced width and h is the height of the legal tree.

The constraint networks were generated randomly uniformly given a number of input parameters: N - number of variables; K - number of values per variable; C - number of constraints; S - the scope size of the constraints; t - the tightness (percentage of allowed tuples per constraint).

N=40, K=2, C=40, S=3, 20 instances, w*=10, h=17											
t		10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
# sol		0	0	0	0	0	13,533	2,414,724	190,430,000	21,549,650,000	1,099,511,627,776
Time											
A/O FC	i=0	0.000	0.001	0.002	0.005	0.011	0.065	0.289	1.931	7.979	30.094
	i=3	0.001	0.002	0.002	0.003	0.008	0.060	0.253	1.525	6.062	22.340
	i=6	0.001	0.001	0.004	0.003	0.009	0.052	0.182	0.883	2.873	8.847
	i=10	**0.000**	**0.001**	**0.003**	**0.004**	**0.010**	**0.038**	**0.110**	**0.343**	**0.587**	**0.985**
OR FC	i=10	**0.000**	**0.003**	**0.003**	**0.004**	**0.012**	**0.671**	**24.912**	**1009.025**	-	-
Number of nodes											
A/O FC	i=0	11	17	32	55	166	3078	22273	204562	988136	4145934
	i=3	11	17	32	55	155	1503	8747	57778	236466	870866
	i=6	11	17	32	55	148	975	4292	24542	95394	298236
	i=10	**11**	**17**	**32**	**55**	**135**	**746**	**2365**	**8646**	**15050**	**25717**
OR FC	i=10	**11**	**17**	**32**	**55**	**166**	**14049**	**635331**	**25078186**	-	-
Number of deadends											
A/O FC	i=0	13	19	34	57	162	1978	10298	57678	134324	0
	i=3	13	19	34	57	159	1662	8569	45336	92263	0
	i=6	13	19	34	56	149	974	3721	13655	19257	0
	i=10	**13**	**19**	**34**	**55**	**125**	**533**	**1312**	**2313**	**1887**	**0**
OR FC	i=10	**13**	**19**	**34**	**57**	**164**	**9693**	**299138**	**11541863**	-	-

Table 3. The impact of caching (algorithms A/O FC and OR FC)

Tables 1 and 2 show an ample comparison of the algorithms on moderate size problems which allowed bucket elimination to run. The bolded time numbers show the best values in each column. The most important thing to note is the vast superiority of AND/OR space over the traditional OR space. Only for the very tight problems ($t = 10\% - 40\%$), which are also inconsistent, the two search spaces seem to be comparable. The picture is clearer if we look at the number of expanded nodes and number of deadends. When the problems are loose and have a large number of solutions AND/OR algorithms are orders of magnitudes better (see *#n, #d* bolded figures for *i=9* in Table 1, and for *i=13* in Table 2, where A/O FC explores a space two orders of magnitude smaller than that of OR FC, resulting in a time two orders of magnitude smaller). In Table 1 we also see the impact of more constraint propagation. The RFC algorithms always explore a smaller space than the FC, but this comes with an overhead cost, and may not always be faster. For BE we only report time, which is not sensitive to the tightness of the problem, so we see that for tight networks search can be faster than BE.

Caching doesn't seem to play a big role in this first set of problems. Especially, for inconsistent networks, caching doesn't improve performance. This is probably because the type of networks we generate turn out to be fairly easy for forward checking, so even without caching the no-goods of the inconsistent networks, forward checking is able to easily detect them.

Table 3 gives a better idea of why caching is useful. First, let's look at the A/O FC entries. This is again a smaller problem for which A/O FC could be run even for $t = 100\%$. When problems become loose, caching is essential to speed up the search. Of course, caching no-goods which are hard to detect by propagation would also improve the search. We show here again that AND/OR space can yield exponential improvement over OR space. OR FC is shown for *i=10* (see bolded figures). For *t=90%* and *t=100%* OR FC would take too long. For *t=80%* even the linear space A/O FC is 3 orders of magnitude faster than OR FC with $i = 10$ caching.

N=60, **K=3**, C=80, S=3, 20 instances, **w*=19**, h=28						
t	10%	20%	30%	40%	50%	60%
# sol	0	0	0	0	2161	2156438278
Time (seconds)						
i=12	0.042	0.186	1.229	14.499	279.051	2296.488
Number of nodes						
i=12	155	643	4181	50544	1091748	14874689
Number of deadends						
i=12	314	1289	8365	100760	2118274	25996533

Table 4. A/O FC, N=60, K=3

N=100, **K=2**, C=130, S=3, 20 instances, **w*=32**, h=43							
t	10%	20%	30%	40%	50%	60%	70%
# sol	0	0	0	0	0	0	0
Time (seconds)							
i=20	0.069	0.096	0.193	0.725	3.572	27.680	677.045
Number of nodes							
i=20	70	96	406	832	4264	35353	1139860
Number of deadends							
i=20	72	98	204	834	4266	34793	1043692

Table 5. A/O FC, N=100, K=2

Finally, Tables 4 and 5 show examples of large networks for which BE was infeasible. Traditional OR space search would also not be possible. We ran only A/O FC with the maximum cache size possible for our machine. This show that AND/OR search is more flexible, being able to solve problems of much larger size than inference algorithms or OR search.

8 Conclusions, Discussion and Related Work

The primary contribution of this paper is in viewing search for constraint processing in the context of AND/OR search spaces rather than OR spaces and in demonstrating the impact of this view on counting solutions. The paper first overviews the notion of AND/OR search space (which was introduced for graphical models in general [4]) for constraint networks. It describes the AND/OR search tree showing that its size can be bounded exponentially by the depth of its legal tree implying exponential saving for any linear space algorithms traversing the AND/OR graph. It describes the minimal AND/OR search graph, showing that it is exponential in the tree-width while the size of the minimal OR graph is exponential in the path-width. Since for some graphs the difference between the path-width and tree-width is substantial (e.g., balanced trees) the AND/OR representation implies exponential time and space savings for all algorithms that cache goods and no-goods.

The paper then shows how counting algorithms can be affected when formulated as searching AND/OR search trees and graphs rather than searching their OR counterparts. We present and analyze counting algorithms and provide initial empirical evaluation along the full spectrum of space and time. We compared counting algorithms on the AND/OR search space when pruning is accomplished by two forward-checking strategies and showed how their performance is affected by different levels of caching and how it is compared to bucket-elimination, as a function of problem tightness. These results show that AND/OR search space is always better than the traditional OR space, often yielding exponential improvements. Compared to inference based algorithms (bucket elimination), AND/OR search is more flexible and able to adapt to the amount of space available. All the existing constraint propagation techniques are readily available for AND/OR search. Coupling this with the possibility of caching makes AND/OR search a very powerful scheme.

Related work. Algorithm backjumping [13], graph-based or conflict-based for constraint satisfaction was designed to overcome the redundancy imposed by the OR struc-

ture of the search tree. It can be shown that graph-based backjumping mimics the exploration of an AND/OR search tree. Indeed, it was shown that the depth of a DFS-tree or a legal-tree [14, 7] plays an important role in bounding backjumping complexity. Also, recent algorithms in probabilistic reasoning such as recursive-conditioning [15] can operate in linear space and can be viewed as searching AND/OR search trees and graphs. Memory-intensive algorithms can be viewed as searching the AND/OR search graph, such as recent work [16] which performs search guided by a tree-decomposition for constraint satisfaction and optimization. A similar approach was introduced recently in [17] both for belief updating and counting models of a CNF formula. The notion of minimal OR search graphs is similar to the known concept of *Ordered Binary Decision Diagrams (OBDD)* in the literature of hardware and software design and verification[11]. It is well known that the size of OBDDs is bounded exponentially by the *path-width* of the CNF's interaction graph. Our notion of minimal AND/OR search graphs, if applied to CNFs, resembles *tree OBDDS* developed subsequently [12].

References

1. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers (2003)
2. Pearl, J.: Probabilistic Reasoning in Intelligent Systems. Morgan Kaufmann (1988)
3. Howard, R.A., Matheson, J.E.: Influence diagrams. (1984)
4. Dechter, R.: AND/OR search spaces for graphical models. In: Submitted, UAI. (2004)
5. Arnborg, S.A.: Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. BIT **25** (1985) 2–23
6. Bodlaender, H., Gilbert, J.R.: Approximating treewidth, pathwidth and minimum elimination tree-height. In: Technical report RUU-CS-91-1, Utrecht University. (1991)
7. Bayardo, R., Miranker, D.: A complexity analysis of space-bound learning algorithms for the constraint satisfaction problem. In: AAAI'96: Proceedings of the Thirteenth National Conference on Artificial Intelligence. (1996) 298–304
8. Dechter, R., Pearl, J.: Network-based heuristics for constraint satisfaction problems. Artificial Intelligence **34** (1987) 1–38
9. Dechter, R.: Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. Artificial Intelligence **41** (1990) 273–312
10. Frost, D.H.: Algorithms and heuristics for constraint satisfaction problems. Technical report, Ph.D. thesis, Information and Computer Science, University of California, Irvine, California (1997)
11. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic (1993)
12. McMillan, K.L.: Hierarchical representation of discrete functions with application to model checking. In: Computer Aided Verification, 6th International conference, David L. Dill ed. (1994) 41–54
13. Gaschnig, J.: Performance measurement and analysis of search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University (1979)
14. Freuder, E.C., Quinn, M.J.: The use of lineal spanning trees to represent constraint satisfaction problems. Technical Report 87-41, University of New Hampshire, Durham (1987)
15. Darwiche, A.: Recursive conditioning. In: Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI99). (1999)
16. Terrioux, C., Jegou, P.: Hybrid backtracking bounded by tree-decomposition of constraint networks. In: Artificial Intelligence. (2003)
17. F. Bacchus, S.D., Piassi, T.: Value elimination: Bayesian inference via backtracking search. In: Uncertainty in AI (UAI03). (2003)