CHAPTER 6

AND/OR Search Spaces and Algorithms for Graphical Models

In this chapter we start the discussion of the second type of reasoning algorithms, those that are based on the *conditioning* step, namely, on assigning a single value to a variable. To recall, algorithms for processing graphical models fall into two general types: inference-based and search-based. Inference-based algorithms (*e.g.*, variable-elimination, tree-clustering discussed earlier) are good at exploiting the independencies displayed by the underlying graphical model and in avoiding redundant computation. They have worst case time guarantee which is exponential in the treewidth of the graph. Unfortunately, any method that is time-exponential in the treewidth is also space exponential in the treewidth or in the related separator-width parameter and therefore, not feasible for models that have large treewidths.

Traditional search algorithms (*e.g.*, depth-first branch-and-bound, best-first search) traverse the model's search space where each path represents a partial or a full solution. For example, we can compute expression 6.1 for $P(G=0, D=1)$ of the network of Figure 5.8(a) by traversing the search-tree in Figure 6.1 along an ordering, from first variable to last variable.

$$\begin{aligned} P(D=1, G=0)) &= \sum_{a,c,b,f,d=1,g=0} P(g=0|f)P(f|b,c)P(d=1|a,b)P(c|a)P(b|a)P(a) \\ &= \sum_{a} P(a) \sum_{c} P(c|a) \sum_{b} P(b|a) \sum_{f} P(f|b,c)P(d=1|b,a)P(g=0|f), \end{aligned} \tag{6.1}$$

Specifically, the arcs of each path are weighted by numerical values extracted from the CPT's of the problem that correspond to the variable assignments along the path. The bottom path shows explicitly the functions at each arc and the leaves provide the probabilities conditioned on the evidence as is shown. In this traditional search tree, every complete path expresses a *solution*, namely a full assignment to the variables and the product of its weight gives the probability of this *solution*.

The search tree can be traversed by depth-first search accumulating the appropriate sums of probabilities (details will be given shortly). It can also be *searched* to find the assignment having the highest probability, thus solving the *mpe* task.

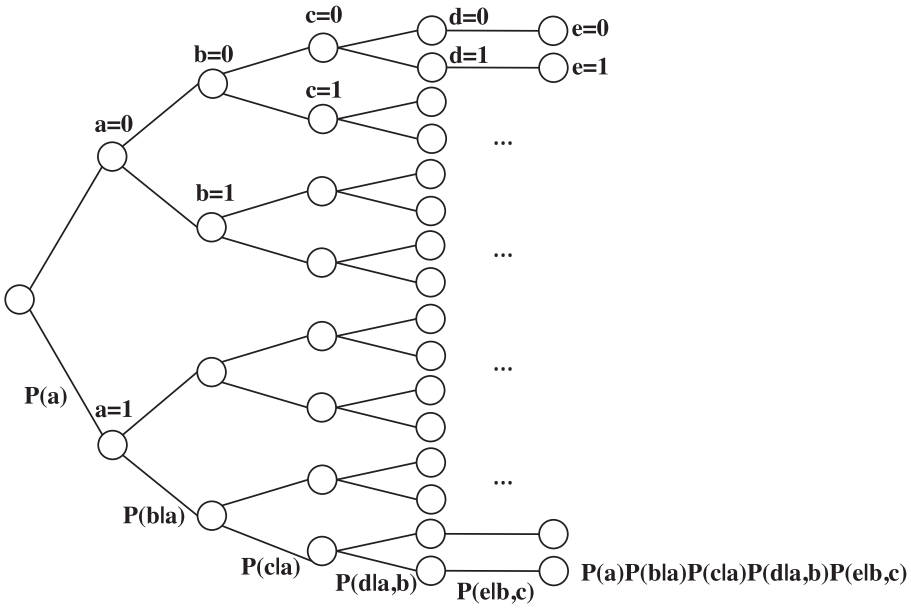


Figure 6.1: Probability tree for computing P(d=1,g=0).

Notice that the structure of search spaces does not retain the independencies represented in the underlying graphical model and may lead to inferior schemes compared with inference algorithms. The size of this search tree is $O(k^n)$ when k bounds the domain size and n is the number of variables. On the other hand, the memory requirements of search algorithms may be less severe than those of inference algorithms; if we use *DFS* traversal it can be accomplished with linear memory. Furthermore, search requires only an implicit, generative, specification of the functions (given in a procedural or functional form) while inference schemes often rely on an explicit tabular representation over the (discrete) variables. For these reasons search algorithms are the only choice available for models with large treewidth, large domains, and with implicit representation.

In this chapter we will show that it is beneficial to depart from the standard linear search space in favor of AND/OR search spaces, originally introduced in the context of heuristic search [Nillson, 1980], primarily because they encode some of the structural information in the graphical models. In particular, AND/OR search spaces can capture the independencies in the graphical model to yield AND/OR search trees that are exponentially smaller than the standard search tree, which we call *OR* tree. We will provide analysis of the size of the AND/OR search tree and show that it is bounded exponentially by the height of some tree that spans the graphical model. Subsequently, we show that the search tree may contain significant redundancy that when identified, can be removed yielding AND/OR search graphs. This additional savings can reduce the size of the AND/OR search space

further to the point that it can be guaranteed to be no larger than exponentially in the graphical model treewidth,

6.1 AND/OR SEARCH TREES

We will present and contrast the concepts of *OR* vs. *AND/OR* search spaces of *graphical models* starting with an example of a constraint network.

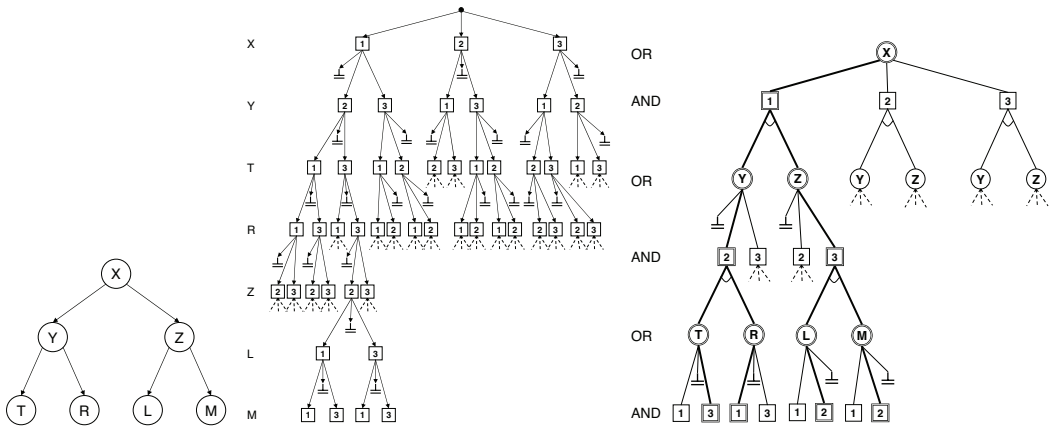
Example 6.1 Consider the simple tree graphical model (*i.e.*, whose primal graph is a tree) in Figure 6.2a, over domains of variables $\{1, 2, 3\}$, which represents a graph-coloring problem. Namely, each node should be assigned a value such that adjacent nodes have different values. The common way to solve this problem is to consider all partial and full solutions to the variables by traversing the problem's search tree, in which each partial path is an assignment to a subset of the variables, and the solutions are paths of length n when n is the number of variables. The problem depicted in Figure 6.2a yields the OR search tree in Figure 6.2b.

Notice, however, that once variable X is assigned the value 1, the search space it roots can be decomposed into two independent subproblems, one that is rooted at Y and one that is rooted at Z, both of which can to be solved independently. Indeed, given $X = 1$, the two search subspaces do not interact. The same decomposition can be associated with the other assignments to X, $(X = 2)$ and $(X = 3)$. Applying the decomposition along the tree (in Figure 6.2a) yields the AND/OR search tree in Figure 6.2c. The AND nodes denote problem-decomposition. They indicate that child nodes of an AND nodes can be solved independently. Indeed, in the AND/OR space, a full assignment to all the variables is not a path but a subtree. Comparing the size of the traditional *OR* search tree in Figure 6.2b against the size of the AND/OR search tree, the latter is clearly smaller. The OR search space has $3 \cdot 2^7$ nodes while the AND/OR one has $3 \cdot 2^5$.

More generally, if k is the domain size, a balanced binary tree graphical model (e.g., a graph coloring problem) with n nodes has an OR search tree of size $O(k^n)$. The AND/OR search tree, whose underlying tree graphical model has depth $O(\log_2 n)$, has size $O((2k)^{\log_2 n}) = O(n \cdot k^{\log_2 n}) = O(n^{1+\log_2 k})$. When $k = 2$, this becomes $O(n^2)$ instead of 2^n.

The AND/OR space is not restricted to tree graphical models as in the above example. As we show, it only has to be guided by a tree spanning the primal graph of the model that obeys some conditions to be defined in the next subsection. We define the AND/OR search space relative to a guiding spanning tree of the primal graph.

Definition 6.2 AND/OR search tree. Given a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \bigotimes \rangle$, its primal graph G and a guiding spanning tree $\mathcal{T}$ of G, the associated AND/OR search tree, denoted $S_{\mathcal{T}}(\mathcal{M})$, has alternating levels of AND and OR nodes. The OR nodes are labeled X_i and correspond to the variables. The AND nodes are labeled $\langle X_i, x_i \rangle$ (or simply x_i) and correspond to the value assignments of the variables. The structure of the AND/OR search tree is based on the underlying spanning tree $\mathcal{T}$. Its root is an OR node labeled by the root of $\mathcal{T}$.



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

Figure 6.2: OR vs. AND/OR search trees; note the connector for AND arcs.

A path from the root of the $S_{\mathcal{T}}(\mathcal{M})$ to a node n is denoted by $path(n)$. If n is labeled X_i or x_i the path will be denoted $path(n = X_i)$ or $path(n = x_i)$, respectively. The assignment sequence along $path(n)$, denoted $val(path(n))$ is the tuple of values assigned to the variables along the path. That is, the sequence of AND nodes along $path(n)$:

$$\begin{aligned} val(path(n = X_i)) &= \{\langle X_1, x_1\rangle, \langle X_2, x_2\rangle, \ldots, \langle X_{i-1}, x_{i-1}\rangle\} = \mathbf{x}_{(1..i-1)}, \\ val(path(n = x_i)) &= \{\langle X_1, x_1\rangle, \langle X_2, x_2\rangle, \ldots, \langle X_i, x_i\rangle\} = \mathbf{x}_{(1..i)}. \end{aligned}$$

The set of variables associated with OR nodes along path $path(n)$ is denoted by $var(path(n))$: $var(path(n = X_i)) = \{X_1, \ldots, X_{i-1}\}$, $var(path(n = x_i)) = \{X_1, \ldots, X_i\}$. The parent-child relationship between nodes in the search space are defined as follows.

1. An OR node, n, labeled by X_i has a child AND node, m, labeled $\langle X_i, x_i\rangle$ iff $\langle X_i, x_i\rangle$ is consistent with the assignment $val(path(n))$. Consistency is defined relative to the constraints when we have a constraint problem, or relative to the flat constraints extracted from the zeros in the CPT tables otherwise.

2. An AND node m, labeled $\langle X_i, x_i\rangle$ has a child OR node r labeled Y, iff Y is a child of X in the guiding spanning tree $\mathcal{T}$. Each OR arc emanating from an OR to an AND node is associated with a weight to be defined shortly (see Definition 6.8).

A solution in an AND/OR space is a subtree rather than a path.

Definition 6.3 Solution subtree. A *solution subtree* of an AND/OR search tree contains the root node. For every OR node, if it is in the solution tree then the solution contains one of its child nodes

and for each of its included AND nodes the solution contains all its child nodes, and all its leaf nodes are consistent.

Example 6.4 In the example of Figure 6.2a, $\mathcal{T}$ is the tree rooted at X, and accordingly the root OR node of the AND/OR tree in 6.2c is X. Its child nodes which are AND nodes, are labeled $\langle X, 1\rangle, \langle X, 2\rangle, \langle X, 3\rangle$ (only the values are noted in the figure). From each of these AND nodes emanate two OR nodes, Y and Z, since these are the child nodes of X in the guiding tree of Figure 6.2a. The descendants of Y along the path from the root, $\langle X, 1\rangle$, are $\langle Y, 2\rangle$ and $\langle Y, 3\rangle$ only, since $\langle Y, 1\rangle$ is inconsistent with $\langle X, 1\rangle$. In the next level, from each node $\langle Y, y\rangle$ emanate OR nodes labeled T and R and from $\langle Z, z\rangle$ emanate nodes labeled L and M as dictated by the guiding tree. In Figure 6.2c a solution tree is highlighted.

As noted, if the graphical model is not a tree it can be guided by some legal spanning tree of the graph. For example, as we will show in Section 6.1.2, a depth-first-search (DFS) spanning tree of the graph is a useful and legal guiding tree. The notion of a DFS spanning tree is defined for undirected graphs.

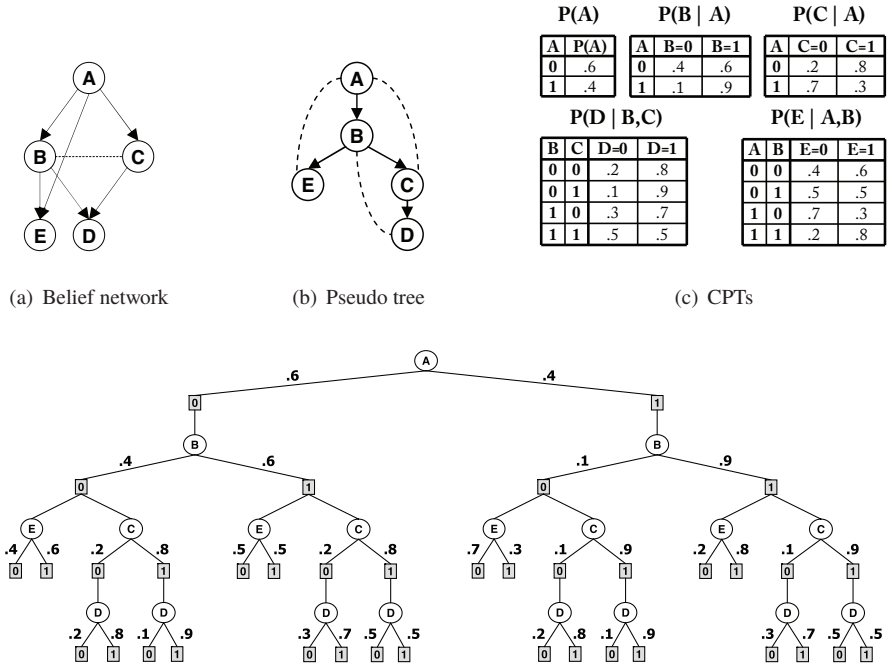
Definition 6.5 DFS spanning tree. Given a graph $G = (V, E)$ and given a node X_1, a *DFS* tree $\mathcal{T}$ of G is generated by applying a depth-first-search traversal over the graph, yielding an ordering $d = (X_1, \ldots, X_n)$. The *DFS spanning tree* $\mathcal{T}$ of G is defined as the tree rooted at the first node, X_1, and which includes only the traversed (by DFS) arcs of G. Namely, $\mathcal{T} = (V, E')$, where $E' = \{(X_i, X_j) \mid X_j \ traversed \ from \ X_i \ by \ DFS \ traversal\}$.

Example 6.6 Consider the probabilistic network given in Figure 6.3a whose undirected primal graph is obtained by including the broken arcs and removing the arrows. A guiding tree which in this case is a DFS spanning-tree of the graph is given in part (b). The dashed arcs are part of the graph but not the spanning-tree arcs. The AND/OR search tree associated with this guiding tree is given in part (d) of the figure. The weights on the arcs will be explained next.

6.1.1 WEIGHTS OF OR-AND ARCS

The arcs in AND/OR trees are associated with weights defined based on the graphical model's functions and the combination operator. The simplest case is that of constraint networks.

Definition 6.7 arc weights for constraint networks. In an AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ of a constraint network $\mathcal{R}$, each terminal node is assumed to have a single, dummy, outgoing arc. The outgoing arc of a terminal AND node always has the weight "1" (namely it is consistent). An outgoing arc of a terminal OR node has weight "0", (there is no consistent value assignments if an OR



(a) Belief network (b) Pseudo tree (c) CPTs

(d) Labeled AND/OR tree

Figure 6.3: Labeled AND/OR search tree for belief networks.

node is a leaf). The weight of any internal OR to AND arc is "1." The arcs from AND to OR nodes have no weight.

We next define arc weights for any general graphical model using the notion of buckets of functions. The concept is simple even if the formal definition may look complex. When considering an arc (n, m) having labels (X_i, x_i) (X_i labels n and x_i labels m), we identify all the functions over variable X_i that are fully instantiated in path(n) once X_i is assigned. We then associate each function with its valuation given the current value-assignment along the path to n. The products of all these *function values* is the weight of the arc. The following definition identify those functions of X_i we want to consider in the product. Weights are assigned only on arcs connecting an OR node to an AND node.

Definition 6.8 OR-to-AND weights, buckets relative to a tree. Given an AND/OR tree $S_{\mathcal{T}}(\mathcal{M})$, of a graphical model $\mathcal{M}$ along a guiding spanning tree $\mathcal{T}$, the weight $w_{(n,m)}(X_i, x_i)$ of arc (n, m) is the *combination* (e.g., product) of all the functions in X_i's bucket relative to $\mathcal{T}$, denoted $B_{\mathcal{T}}(X_i)$, which are assigned by their values along $path(m)$. $B_{\mathcal{T}}(X_i)$ include all functions f having X_i in their scopes and whose $scope(f) \subseteq path_{\mathcal{T}}(X_i)$. Formally, $w_{(n,m)}(X_i, x_i) =$

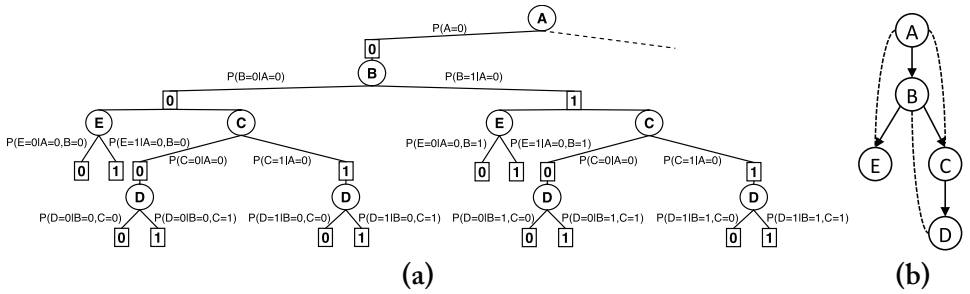


Figure 6.4: Arc weights for probabilistic networks.

$\bigotimes_{f \in B_{\mathcal{T}}(X_i)} f(val(path(m)))$. If the set of functions is empty the weight is the constant 1 (or the identity relative to the combination operator).

Definition 6.9 Weight of a solution subtree. Given a weighted AND/OR tree $S_{\mathcal{T}}(\mathcal{M})$, of a graphical model $\mathcal{M}$, the weight of a subtree t is $w(t) = \bigotimes_{e \in arcs(t)} w(e)$, where $arcs(t)$ is the set of arcs in subtree t.

Example 6.10 Figure 6.4b shows a guiding DFS tree of the Bayesian network in Figure 6.3 and in 6.4a a portion of the AND/OR search tree with the appropriate weights on the arcs expressed symbolically. The bucket of variable E contains the function $P(E|A,B)$, and the bucket of D contains function $P(D|B,C)$. We see indeed that the weights from nodes labeled E and from any of its AND value assignments include only the instantiated function $P(E|A,B)$. The evaluated weights along this pseudo-tree are depicted in Figure 6.3d.

6.1.2 PSEUDO TREES

We have mentioned that a DFS spanning tree is a legal guiding tree for the AND/OR search. This is indeed the case because child nodes branching reflect problem decomposition. However, there is a more general class of spanning trees, called *pseudo-trees*, which can be considered. In order to guide a proper decomposition for a graphical model, such trees need to obey the back-arc property.

Definition 6.11 Pseudo tree, extended graph. Given an undirected graph $G = (V, E)$, a directed rooted tree $\mathcal{T} = (V, E')$ defined on all its nodes is a *pseudo tree* if any arc in E which is not in E' is a back-arc in $\mathcal{T}$, namely, it connects a node in $\mathcal{T}$ to an ancestor in $\mathcal{T}$. The arcs in E' may not all be included in E. Given a pseudo tree $\mathcal{T}$ of G, the *extended graph* of G relative to $\mathcal{T}$ includes also the arcs in E' that are not in E. That is, the extended graph is defined as $G^{\mathcal{T}} = (V, E \cup E')$.

Clearly, a DFS-tree is a pseudo-tree with the additional restriction that all its arcs are in included in the original graph. The use of a larger class of pseudo trees has the potential of yielding smaller depth guiding trees which are highly desirable, as we show in the next example.

Example 6.12 Consider the graph G displayed in Figure 6.5a. Ordering $d_1 = (1, 2, 3, 4, 7, 5, 6)$ is a DFS ordering of a DFS spanning tree $\mathcal{T}_1$ having depth of 3 (Figure 6.5b). The tree $\mathcal{T}_2$ in Figure 6.5c is a pseudo tree and has a tree depth of 2 only. The two tree-arcs (1,3) and (1,5) are not in G. The tree $\mathcal{T}_3$ in Figure 6.5d, is a chain. The extended graphs $G^{\mathcal{T}_1}$, $G^{\mathcal{T}_2}$ and $G^{\mathcal{T}_3}$ are presented in Figure 6.5b, c, d when we ignore directionality and include the broken arcs.

Figure 6.6 shows the AND/OR search trees along the pseudo trees $\mathcal{T}_1$ and $\mathcal{T}_2$ in Figure 6.5. The domains of the variables are $\{a, b, c\}$ and there is no pruning due to hard constraints. We see that the AND/OR search tree based on $\mathcal{T}_2$ is smaller because $\mathcal{T}_2$ has a smaller height than $\mathcal{T}_1$. In fact, the number of AND nodes in the top AND/OR search tree has 237 AND nodes, while the bottom has only 129 AND nodes. The weights are not specified here.

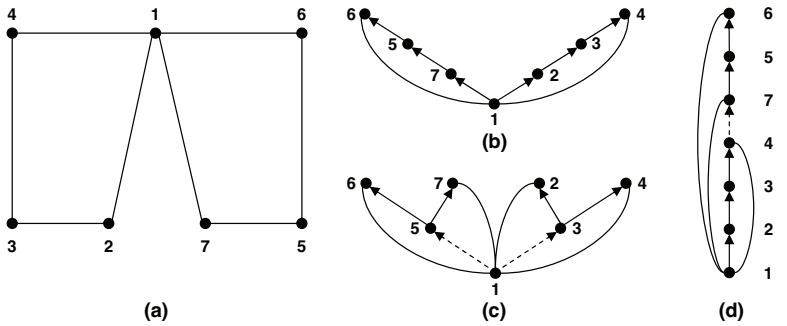


Figure 6.5: (a) A graph; (b) a DFS tree $\mathcal{T}_1$; (c) a pseudo tree $\mathcal{T}_2$; and (d) a chain pseudo tree $\mathcal{T}_3$.

6.1.3 PROPERTIES OF AND/OR SEARCH TREES

Any pseudo tree $\mathcal{T}$ of a graph G has the property that the arcs of G which are not in $\mathcal{T}$ are back arcs. Namely, they connect a node to one of its ancestors in the guiding tree. This property implies that each scope of a function in F will be fully assigned on some path in $\mathcal{T}$, a property that is essential for the ability of the AND/OR search space to consider all the functions in the model and supports correct computation. In fact, the AND/OR search tree can be viewed as an alternative representation of the graphical model.

Theorem 6.13 Correctness. *Given a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} = \{f_1, ..., f_r\}, \bigotimes \rangle$ having a primal graph G and a guiding pseudo-tree $\mathcal{T}$ of G and its associated weighted AND/OR search tree $S_{\mathcal{T}}(\mathcal{M})$ then (1) there is a one-to-one correspondence between solution subtrees of $S_{\mathcal{T}}(\mathcal{M})$ and*

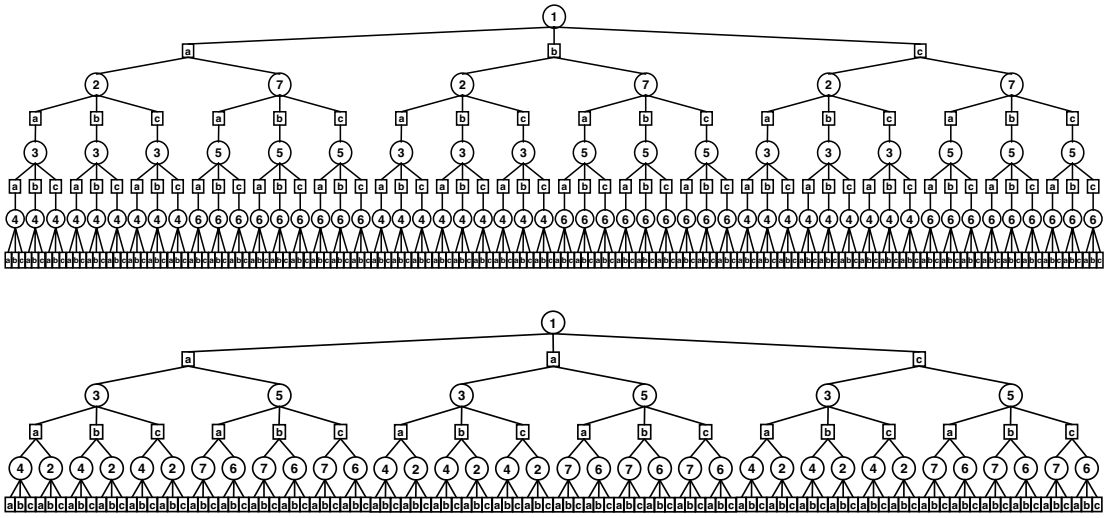


Figure 6.6: AND/OR search tree along pseudo trees $\mathcal{T}_1$ and $\mathcal{T}_2$.

solutions of $\mathcal{M}$; (2) the weight of any solution tree equals the cost of the full solution assignment it denotes; namely, if t is a solution tree of $S_{\mathcal{T}}(\mathcal{M})$ then $F(val(t)) = w(t)$, where $val(t)$ is the full solution defined by tree t. (See Appendix for a proof.)

As already mentioned, the virtue of an AND/OR search tree representation is that its size can be far smaller than the traditional OR search tree. The size of an AND/OR search tree depends on its depth, also called height, of its pseudo-tree $\mathcal{T}$. Therefore, pseudo trees of smaller height should be preferred. An AND/OR search tree becomes an OR search tree when its pseudo tree is a chain.

Theorem 6.14 Size of AND/OR search tree. *Given a graphical model $\mathcal{M}$, with domains size bounded by k, having a pseudo tree $\mathcal{T}$ whose height is h and having l leaves, the size of its AND/OR search tree $S_{\mathcal{T}}(\mathcal{M})$ is $O(l \cdot k^h)$ and therefore also $O(nk^h)$ and $O((bk)^h)$ when b bounds the branching degree of $\mathcal{T}$ and n bounds the number of nodes in the pseudo-tree. The size of its OR search tree along any ordering is $O(k^n)$ and these bounds are tight. (See Appendix for proof.)*

We can give a more refined bound on the search space size by spelling out the height h_i of each leaf L_i in $\mathcal{T}$ as follows. Given a guiding spanning $\mathcal{T}$ having $L = \{L_1, \ldots, L_l\}$ leaves of a model $\mathcal{M}$, where the depth of leaf L_i is h_i and k bounds the domain sizes, the size of its full AND/OR search tree $S_{\mathcal{T}}(\mathcal{M})$ is $O(\sum_{k=1}^{l} k^{h_i+1})$. Using also the domain sizes for each variable yields an even more accurate expression of the search tree size: $|S_{\mathcal{T}}(\mathcal{M})| = O(\sum_{L_k \in L} \prod_{\{X_j | X_j \in path_{\mathcal{T}}(L_k)\}} |D(X_j)|)$.

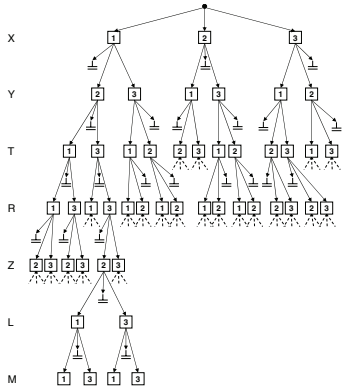


Figure 6.7: OR search tree for the tree problem in Figure 6.2a.

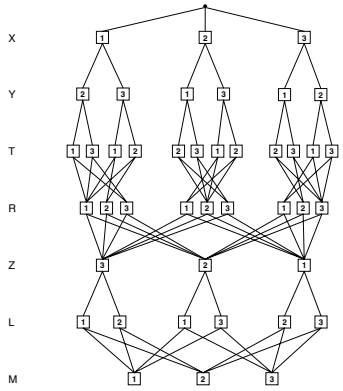


Figure 6.8: The minimal OR search graph of the tree graphical model in Figure 6.2a.

6.2 AND/OR SEARCH GRAPHS

It is often the case that a search space that is a tree can become a graph if nodes that root identical search subspaces, or correspond to identical subproblems, are identified. Any two such nodes can be *merged*, yielding a graph and thus reducing the size of the search space.

Example 6.15 Consider again the graph in Figure 6.2a and its AND/OR search tree in Figure 6.2c depicted again in Figure 6.9 representing a constraint network. Observe that at level 3, node $\langle Y, 1\rangle$ appears twice, (and so are $\langle Y, 2\rangle$ and $\langle Y, 3\rangle$) (not shown explicitly in the figure). Clearly however, the subtrees rooted at each of these two AND nodes are identical and they can be merged because in this tree model, any specific assignment to Y uniquely determines its rooted subtree. Indeed, the resulting merged AND/OR search graph depicted in Figure 6.10 is equivalent to the AND/OR search tree in Figure 6.9.

It may also occur that two nodes that do not root identical subtrees still correspond to equivalent subproblems. Such nodes can also be *unified*, even if their *explicit* weighted subtrees do not look identical. To discuss this issue we need the notion of *equivalent graphical models*. In general, two graphical models are equivalent if they have the same set of solutions, and if each is associated with the same *cost*. We will use the notion of *universal graphical model* to define *equivalence*. A universal graphical model represents the *solutions* of a graphical model, through a single global function over all the variables. For example, the universal model of a Bayesian network is the joint probability distribution it represents.

Definition 6.16 Universal equivalent graphical model. Given a graphical model $\mathcal{M} = \langle X, D, F, \bigotimes\rangle$ the universal equivalent model of $\mathcal{M}$ is $u(\mathcal{M}) = \langle X, D, F = \{\bigotimes_{i=1}^{r} f_i\}\rangle$.

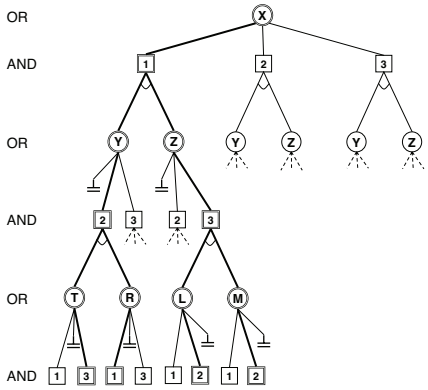


Figure 6.9: AND/OR search tree for the tree problem in Figure 6.2a.

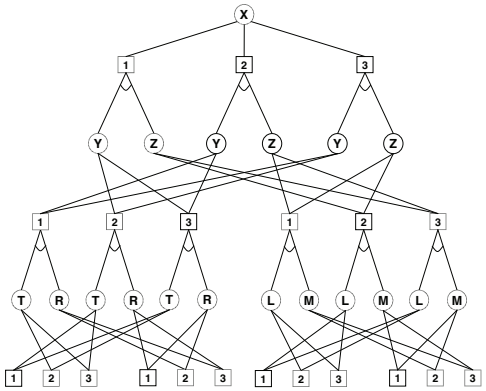


Figure 6.10: The minimal AND/OR search graph of the tree graphical model in Figure 6.2a.

We also need to define the cost of a partial solution and the notion of a graphical model conditioned on a partial assignment. Informally, a graphical model conditioned on a particular partial assignment is obtained by assigning the appropriate values to all the relevant variables in the function (to all the conditioning set) and modifying the output functions appropriately.

Definition 6.17 Cost of an assignment, conditional model. Given a graphical model $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \bigotimes \rangle$:

1. the cost of a full assignment $\mathbf{x} = (x_1, ..., x_n)$ is defined by $c(\mathbf{x}) = \bigotimes_{f \in F} f(x_f)$. The *cost of a partial assignment* $\mathbf{y}$, over $\mathbf{Y} \subseteq \mathbf{X}$ is the combination of all the functions whose scopes are included in $\mathbf{Y}$ (denoted F_Y) evaluated at the assigned values. Namely, $c(y) = \bigotimes_{f \in F_\mathbf{Y}} f(y_f)$.
2. the graphical model conditioned on $Y = y$ is $\mathcal{M}|_y = \langle X - Y, D|_{X-Y}, F|_y, \bigotimes \rangle$, where $F|_y = \{f|_{Y=y}, f \in F\}$.

6.2.1 GENERATING COMPACT AND/OR SEARCH SPACES

We will next define the merge operator. It transforms AND/OR search trees into an equivalent AND/OR graphs.

Definition 6.18 Merge. Assume a given weighted AND/OR search graph $S'_\mathcal{T}$ of a graphical model $\mathcal{M}$ and assume two paths $path(n_1)$ and $path(n_2)$ ending by AND nodes at level i having the same label x_i. Nodes n_1 and n_2 can be *merged* iff the weighted search subgraphs rooted at n_1 and n_2 are identical. The *merge* operator, $merge(n_1, n_2)$, redirects all the arcs going into n_2 into n_1 and

removes n_2 and its subgraph. When we merge AND nodes only we call the operation AND-merge. The same reasoning can be applied to OR nodes, and we call the operation OR-merge.

Proposition 6.19 Merge-minimal AND/OR graphs *Given a weighted AND/OR search graph $\mathcal{G}_T$ guided by a pseudo tree $\mathcal{T}$: The* merge *operator has a unique fix point, called the* ***merge-minimal*** *AND/OR search graph. (See proof in the Appendix).*

When $\mathcal{T}$ is a chain pseudo tree, the above definitions are applicable to the traditional OR search tree as well. However, we may not be able to reach the same compression as in some AND/OR cases, because of the linear structure imposed by the OR search tree.

Example 6.20 The smallest OR search graph of the graph-coloring problem in Figure 6.2a (depicted again in Figure 6.7) is given in Figure 6.8 along the DFS order X, Y, T, R, Z, L, M. The smallest AND/OR graph of the same problem along the DFS tree is given in Figure 6.10. We see that some variable-value pairs (AND nodes) must be repeated in Figure 6.8 while in the AND/OR case (Figure 6.10) they appear just once. In particular, the subgraph below the paths $(\langle X,1\rangle, \langle Y,2\rangle)$ and $(\langle X,3\rangle, \langle Y,2\rangle)$ in the OR tree cannot be merged at $\langle Y,2\rangle$. You can now compare all the four search space representations side by side in Figures 6.7–6.10.

6.2.2 BUILDING CONTEXT-MINIMAL AND/OR SEARCH GRAPHS

The merging rule seems to be quite operational; we can generate the AND/OR search tree and then recursively merge identical subtrees going from leaves to root. This, however, requires generating the whole search tree first, which would still be costly. It turns out that for some nodes it is possible to recognize that they can be merged by using graph properties only, namely based on their *contexts*. The context of a variable is the set of its ancestor variables in the pseudo tree $\mathcal{T}$ that completely determine the conditioned subproblems below it.

We have already seen in Figure 6.2a that at level 3, node $\langle Y,1\rangle$ appears twice (and so are $\langle Y,2\rangle$ and $\langle Y,3\rangle$). Clearly we can see that Y uniquely determines its rooted subtree. In this case Y is its own context and the AND/OR search graph in Figure 6.10 is equivalent to the AND/OR search tree in Figure 6.7. In general, an AND/OR search graph of a graphical model that is a tree can be obtained by merging all AND nodes having the same label $\langle X,x\rangle$. That is, every variable is its own context. The resulting equivalent AND/OR graph has search space size of $O(nk)$.

The general idea of a context is to identify a minimal set of ancestor variables, along the path from the root to the node in the pseudo tree, such that when assigned the same values they yield the same conditioned subproblem, regardless of value assigned to the other ancestors. To derive a general merging scheme we define the induced-width of a pseudo-tree.

Definition 6.21 Induced width of a pseudo tree. The induced width of G relative to a pseudo tree $\mathcal{T}$, is the maximum width of its *induced pseudo tree* obtained by recursively connecting the

parents of each node, going from leaves to root along each branch. In that process we consider both the extended arcs in the pseudo tree and those in the graphical model.

Definition 6.22 Parents, parents-separators. Given a primal graph G and a pseudo tree $\mathcal{T}$ of its graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \bigotimes \rangle$, the *parents* of an OR node X_i, denoted by pa_i or pa_{X_i}, are the ancestors of X_i that are connected in G to X_i or to descendants of X_i. The *parent-separators* of X_i (or of $\langle X_i, x_i \rangle$), denoted by pas_{X_i}, are formed by X_i and its ancestors that have connections in G to descendants of X_i.

It follows from these definitions that the parents of X_i, pa_{X_i} separate in the primal graph G, the ancestors of X_i in $\mathcal{T}$, from X_i and its descendants. Similarly, the parents-separators of X_i, pas_{X_i}, separate the ancestors of X_i from its descendants. It is also easy to see that each variable X_i and its parents pa_{X_i} form a clique in the induced pseudo-graph. The following proposition establishes the relationship between pa_{X_i} and pas_{X_i}. We use both in order to characterize two types of merging: AND merge and OR merge. The following claim follows directly from Definitions 6.22. It is easy to see the following.

Proposition 6.23 Relations between contexts

1. *If Y is the single child of X in $\mathcal{T}$, then $pas_X = pa_Y$.*
2. *If X has children $Y_1, \ldots, Y_k$ in $\mathcal{T}$, then $pas_X = \cup_{i=1}^{k} pa_{Y_i}$.*

Theorem 6.24 Context-based merge operators. *Let $G^{\mathcal{T}^*}$ be the induced pseudo tree of $\mathcal{T}$ and let $path(n_1)$ and $path(n_2)$ be any two partial paths in an AND/OR search graph.*

1. *If n_1 and n_2 are AND nodes annotated by $\langle X_i, x_i \rangle$ and*

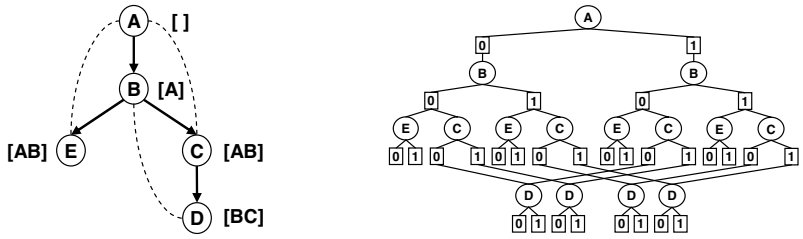
$$val(path(n_1))[pas_{X_i}] = val(path(n_2))[pas_{X_i}] \tag{6.2}$$

   then the AND/OR search subtrees rooted by n_1 and n_2 can be merged.

2. *If n_1 and n_2 are OR nodes annotated by X_i and*

$$val(path(n_1))[pa_{X_i}] = val(path(n_2))[pa_{X_i}] \tag{6.3}$$

   then the AND/OR search subtrees rooted by n_1 and n_2 can be merged.



(a) Pseudo tree (b) Context minimal graph

Figure 6.11: AND/OR search graph.

Definition 6.25 context. The $val(path(n_i))[pas_{X_i}]$ is called the **AND context** of n_i and the $val(path(n_i))[pa_{X_i}]$ is called the **OR context** of n_i.

Example 6.26 For the balanced tree in Figure 6.2a consider the *chain* pseudo tree $d = (X, Y, T, R, Z, L, M)$. Namely, the chain has arcs $\{(X, Y), (Y, T), (T, R), (R, Z), (Z, L), (L, M)\}$ and the extended graph includes also the arcs $(Z, X), (M, Z)$ and (R, Y). The AND-context of T along d is XYT (since the induced graph has the arc (T, X)), of R it is XR, for Z it is Z and for M it is M. Indeed in the first three levels of the OR search graph in Figure 6.8 there are no merged nodes. In contrast, if we consider the AND/OR ordering along the DFS tree, the context of every node is itself yielding a single appearance of each AND node having the same assignment annotation in the minimal AND/OR graph (See Figure 6.10 and contrast it with Figure 6.8).

Definition 6.27 Context minimal AND/OR search graph. The AND/OR search graph of $\mathcal{M}$ guided by a pseudo-tree $\mathcal{T}$ that is closed under context-based merge operator, (namely no more merging is possible), is called the *context minimal* AND/OR search graph and is denoted by $CM_{\mathcal{T}}(\mathcal{R})$.

We should note that we can, in general, merge nodes based both on AND and OR contexts. However, Proposition 6.23 shows that doing just one type of merging renders the other unnecessary (up to some small constant factor). In practice, we would recommend just the OR context based merging, because it has a slight (albeit by a small constant factor) space advantage.

Example 6.28 Figure 6.11a refer back to the model given in Figure 6.3a, again assuming that all assignments are valid and that variables take binary values. Figure 6.11a shows, again, the pseudo tree derived from ordering $d = (A, B, E, C, D)$. The (OR) context of each node appears in square brackets, and the broken arcs are backarcs. The context-minimal AND/OR graph appears in 6.11(b).

Since each context must appear only once in the Context-minimal graph (different appearances should be merged) the number of nodes in the context minimal AND/OR search graph cannot exceed the number of different contexts. Since, as we will show, the context's scope size is bounded by the induced width of the pseudo tree that guides it, the size of the context minimal graph can be bounded exponentially by the induced width along the pseudo-tree.

Proposition 6.29 *Given a graphical model $\mathcal{M}$, and a pseudo tree $\mathcal{T}$ having induced width w, then the size of the context minimal AND/OR search graph based on $\mathcal{T}$, $CM_{\mathcal{T}}(\mathcal{R})$, is $O(n \cdot k^w)$, when k bounds the domain size and n is the number of variables.*

Proof. For any variable, the number of its contexts is bounded by the number of possible instantiations to the variables in it context. Since the context size of each variable is bounded by its induced-width of the pseudo tree (prove as an exercise), we get the bound of $O(k^w)$. Since we have n variables, the total bound is $O(n \cdot k^w)$. □

In summary, context-based merge (AND and/or OR) offers a powerful way of trimming the size of the AND/OR search space, and therefore of bounding the truly minimal AND/OR search graph. We can generate $CM_{\mathcal{T}}$ using depth-first or breadth first traversals while figuring the converging arcs into nodes via their contexts. This way we avoid generating duplicate searches for the same contexts. All in all, the generation of the search graph is linear in its size. The AND/OR graph is exponential in w and linear in n. Based on Proposition 6.29 we can conclude the following.

Theorem 6.30 *The context minimal AND/OR search graph $CM_{\mathcal{T}}$ of a graphical model whose guiding pseudo tree has a treewidth w can be generated in time and space $O(nk^{w+1})$ and $O(nk^w)$, respectively. (Prove as a exercise.)*

6.3 FINDING GOOD PSEUDO TREES

Since the AND/OR search space, be it a tree or a graph, depends on a guiding pseudo-tree we should address the issue of finding good pseudo-tress. We will discuss two schemes for generating good pseudo-trees. One based on an induced graph along an ordering of the variables, while the other is based on hypergraph-decomposition.

6.3.1 PSEUDO TREES CREATED FROM INDUCED GRAPHS

We saw that the complexity of an AND/OR search trees is controlled by the height of the pseudo tree. It is desirable therefore to find pseudo trees having minimal height. This is yet another graph problem (in addition to finding minimal induced-width) which is known to be NP-complete but greedy algorithms and polynomial time heuristic scheme are available.

A general scheme for generating pseudo trees starts from an induced graphs along some ordering d. A pseudo-tree can then be obtained via a depth-first traversal of the induced-ordered graph

starting from the first node in d and breaking ties in favor of earlier variables in d. An alternative way for generating a pseudo-tree from an induced ordered graph is based on the observation that a bucket tree is a pseudo tree (see Definition 5.2). Summarizing:

Proposition 6.31 *Given a graphical model $\mathcal{M} =< X, D, F, \bigotimes >$ and an ordering d,*

1. *the bucket tree derived from the induced ordered graph along d of $\mathcal{M}$, $T = (X, E)$ with $E = \{(X_i, X_j)|(B_{X_i}, B_{X_j}) \in bucket - tree\}$, is a pseudo tree of $\mathcal{M}$, and*
2. *the DFS-tree generated by traversing the induced-order graph starting at the first variable of its ordering, is a pseudo tree.*

Proof. All one need to show is that all the arcs in the primal graph of $\mathcal{M}$ which are not in T are back-arcs and this is easy to verify based on the construction of DFS tree in part (1) and of a bucket-tree in part (2). (Exercise: complete the proof). □

It is interesting to note that a chain graphical model has a (non-chain) pseduo-tree of depth $\log n$, when n is the number of variables. The induced width of such a tree is $\log n$ as well. On the other hand the minimum induced width of a chain pseudo tree is 1. Therefore, on the one hand a chain can be solved in linear space and in $O(k^{\log n})$ time along its $logn$ height pseudo tree, and on the other hand it can also be solved in $O(nk^2)$ time with $O(nk)$ memory using bucket-elimination along its chain whose induced width is 1 and height is $n/2$. This example generalizes into a relationship between the treewidth and the pseudo-tree height of a graph.

Proposition 6.32 *[Bayardo and Miranker, 1996, H.L. Bodlaender and Kloks, 1991] The minimal height, h^*, of all pseudo trees of a given graph G satisfies $h^* \leq w^* \cdot \log n$, where w^* is the tree width of G.*

Proof. If there is a tree decomposition of G having a treewidth w, then we can create a pseudo tree whose height h satisfies $h \leq w \cdot \log n$ (prove as an exercise). From this it follows that $h^* \leq w^* \cdot \log n$. □

The above relationship suggests a bound on the size of AND/OR search trees of a graphical models in terms of their treewidth.

Theorem 6.33 *A graphical model that has a treewidth w^* has an AND/OR search tree whose size is $O(k^{(w^* \cdot \log n)})$, where k bounds the domain size and n is the number of variables.*

Notice, however, that even though a graph may have induced-width of w^*, the induced width of the pseudo tree created as suggested by the above theorem may be of size $w^* logn$ and not w^*.

Table 6.1: Bayesian networks repository (left); SPOT5 benchmarks (right).

Network	hypergraph		min-fill		Network	hypergraph		min-fill	
	width	depth	width	depth		width	depth	width	depth
barley	7	13	7	23	spot_5	47	152	39	204
diabetes	7	16	4	77	spot_28	108	138	79	199
link	21	40	15	53	spot_29	16	23	14	42
mildew	5	9	4	13	spot_42	36	48	33	87
munin1	12	17	12	29	spot_54	12	16	11	33
munin2	9	16	9	32	spot_404	19	26	19	42
munin3	9	15	9	30	spot_408	47	52	35	97
munin4	9	18	9	30	spot_503	11	20	9	39
water	11	16	10	15	spot_505	29	42	23	74
pigs	11	20	11	26	spot_507	70	122	59	160

Width vs. height of a given pseudo tree. Since an induced-ordered graph can be a starting point in generating a pseudo tree, the question is if the min-fill ordering heuristic which appears to be quite good for finding small induced-width is also good for finding pseudo trees with small heights (see Chapter 3). A different question is what is the relative impact of the width and the height on the actual search complexity. The AND/OR search graph is bounded exponentially by the induced-width while the AND/OR search tree is bounded exponentially by the height. We will have a glimpse into these questions by comparing with an alternative scheme for generating pseudo trees which is based on the hypergraph decompositions scheme.

6.3.2 HYPERGRAPH DECOMPOSITIONS

Definition 6.34 Hypergraph separators. Given a dual hypergraph $\mathcal{H} = (\mathbf{V}, \mathbf{E})$ of a graphical model, a *hypergraph separator decomposition* of size k by nodes S is obtained if removing S yields a hypergaph having k disconnected components. S is called a separator.

It is well known that the problem of finding the minimal size hypergraph separator is hard. However, heuristic approaches were developed over the years.[1]. Generating a pseudo tree $\mathcal{T}$ (yielding also a tree-decomposition) for $\mathcal{M}$ using hypergraph decomposition is fairly straightforward. The vertices of the hypergraph are partitioned into two balanced (roughly equal-sized) parts, denoted by $\mathcal{H}_{left}$ and $\mathcal{H}_{right}$, respectively, while minimizing the number of hyperedges across. A small number of crossing edges translates into a small number of variables shared between the two sets of functions. $\mathcal{H}_{left}$ and $\mathcal{H}_{right}$ are then each recursively partitioned in the same fashion, until they

[1] A good package `hMeTiS` is Available at: http://www-users.cs.umn.edu/karypis/metis/hmetis

contain a single vertex. The result of this process is a tree of hypergraph separators which can be shown to also be a pseudo tree of the original model where each separator corresponds to a subset of variables connected by a chain.

Table 6.1 illustrates and contrasts the induced width and height of pseudo trees obtained with the hypergraph and min-fill heuristics for 10 Bayesian networks from the Bayesian Networks Repository[2] and 10 constraint networks derived from the SPOT5 benchmarks [Bensana *et al.*, 1999]. It is generally observed that the min-fill heuristic generates lower induced width pseudo trees, while the hypergraph heuristic produces much smaller height pseudo trees. Note that it is not possible to generate a pseudo-tree that is optimal w.r.t. both the treewidth and the height (remember our earlier example of a chain).

Notice that for graphical models having a bounded treewidth w, the minimal AND/OR graph is bounded by $O(nk^w)$ while the minimal OR graph is bounded by $O(nk^{w \cdot \log n})$. We conclude this section with the following example which is particularly illustrative of the tradeoff involved.

Example 6.35 Consider the graph of a graphical model given in Figure 6.12a. We see the pseudo tree in part (b) having w=4 and h=8 and the corresponding context-minimal search graph in (c). The second pseudo-tree in part (d) has w=5, h=6 and the context-minimal graph appears in part (e).

6.4 VALUE FUNCTIONS OF REASONING PROBLEMS

As we described earlier, there are a variety of reasoning problems over graphical models (see Chapter 2). For constraint networks, the most popular tasks are to decide if the problem is consistent, to find a single solution or to count solutions. If a cost function is defined by the graphical model we may also seek an optimal solution. The primary tasks over probabilistic networks are computing beliefs (i.e., the posterior marginals given evidence), finding the probability of the evidence and finding the most likely tuple given the evidence (i.e., mpe and map queries). Each of these reasoning problems can be expressed as finding the *value* of nodes in the *weighted AND/OR search space*.

For example, for the task of finding a solution to a constraint network, the value of every node is either "1" or "0." The value "1" means that the subtree rooted at the node is consistent and "0" otherwise. Therefore, the value of the root node determines the consistency query for the full constraint network. For solutions-counting the value function of each node is the number of solutions of the subproblem rooted at that node.

Given a graphical model and a specification by an AND/OR search space of the model. The submodel associated with a node n in the search space, is the submodel conditioned on all the value assignments along the path from the root node to n.

Definition 6.36 Value function for consistency and counting. Given AND/OR tree $S_{\mathcal{T}}(\mathcal{R})$ of a constraint network. The value of a node (AND or OR) for *deciding consistency* is "1" if it roots a

[2] Available at: http://www.cs.huji.ac.il/labs/compbio/Repository

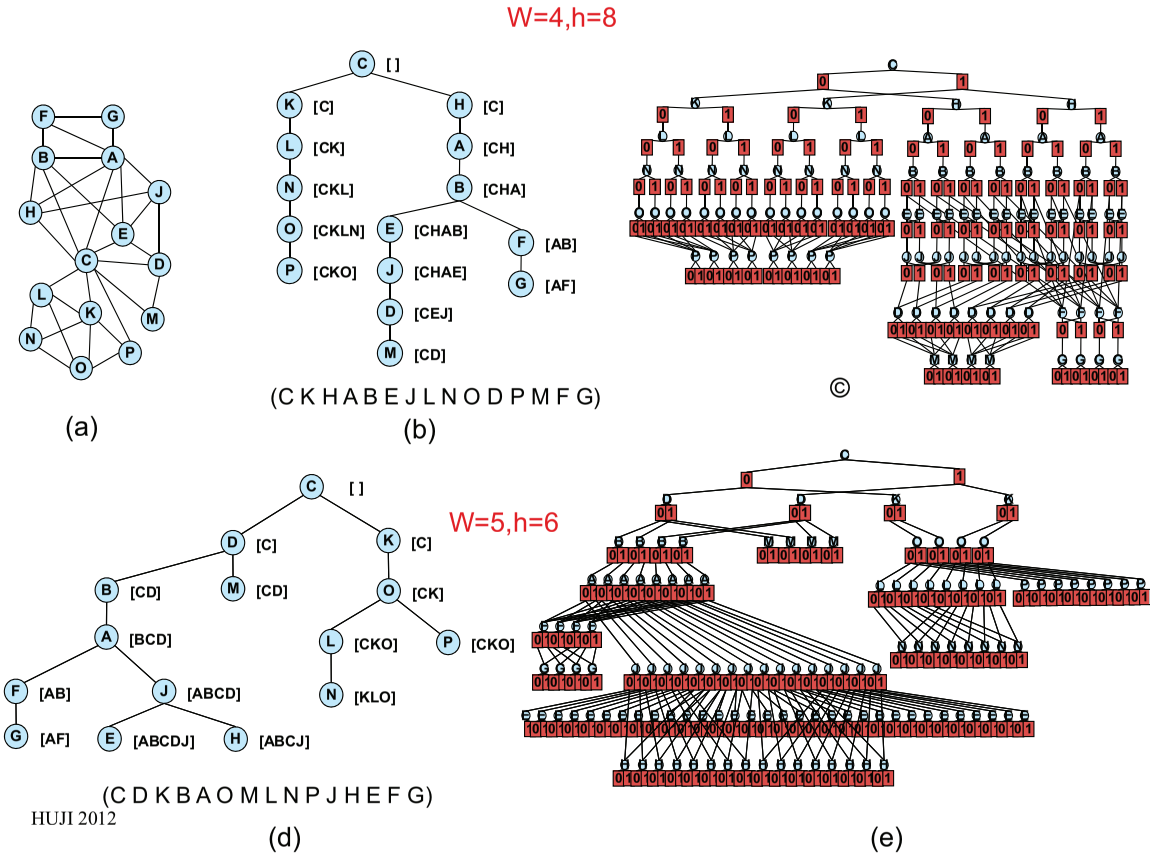


Figure 6.12: A graphical model; (a) one pseudo-tree; (b) its context-minimal search graph; (c) a second pseudo-tree; (d) its corresponding context-minimal AND/OR search graph; and (e) its corresponding context-minimal AND/OR search graph

consistent subproblem and "0" otherwise. The value of a node (AND or OR) for *counting solutions* is the number of solutions in the subproblem it roots which is the number of solutions in its subtree.

The value of nodes in the search graph can be expressed as a function of the values of their child nodes, thus allowing a recursive value computation from leaves to root.

Proposition 6.37 Recursive value computation for constraint queries *Consider the following.*

1. *For the consistency task the value of AND leaves is "1" and the value of OR leaves is "0" (they are inconsistent). An internal OR node is labeled "1" if one of its successor nodes is "1" and an internal node has value "1" iff all its child OR nodes have value "1."*
2. *The counting values of leaf AND nodes are "1" and of leaf OR nodes are "0." The counting value of an internal OR node is the sum of the counting-values of all its child nodes. The counting-value of an internal AND node is the product of the counting-values of all its child nodes. (Exercise: prove the proposition.)*

We now move to probabilistic queries. We can generalize to any graphical model and to any query. We provide the recursive definition of values and then prove that it is correct, namely, that it has the intended meaning. Remember that the label of an arc $(X_i, \langle X_i, x_i\rangle)$ along path $path(n = x_i)$ is defined as $w(X_i, \langle X_i, x_i\rangle) = \prod_{f\in B(X_i)} f(val(path(n = x_i)))$, where $B(X_i)$ are the functions in its bucket (see Definition 6.8.)

Definition 6.38 Recursive value computation for a general reasoning problems. The value function of a reasoning problem $\mathcal{M} = \langle X, D, F, \bigotimes \Downarrow\rangle$ is defined as follows: the value of leaf AND nodes is "1" and of leaf OR nodes is "0." The value of an internal OR node is obtained by *combining* the value of each AND child node with the weight (see Definition 6.8) on its incoming arc and then *marginalizing* over all AND children. The value of an AND node is the combination of the values of its OR children. Formally, if $children(n)$ denotes the children of node n in the AND/OR search graph, then[3]:

$$v(n) = \bigotimes_{n'\in children(n)} v(n'), \qquad \text{if } n = \langle X, x\rangle \text{ is an AND node,}$$
$$v(n) = \Downarrow_{n'\in children(n)} (w_{(n,n')} \bigotimes v(n')), \qquad \text{if } n = X \text{ is an OR node.}$$

Given a reasoning task, the value of the root node is the answer to the query as stated next

Proposition 6.39 *Let $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \bigotimes \Downarrow\rangle$ be a graphical model reasoning task and let $n = X_1$ be the root node in an AND/OR search graph $S'_{\mathcal{T}}(\mathcal{M})$. Then the value of X_1 defined recursively by Definition 6.38 obeys that $v(X_1) = \Downarrow_X \bigotimes_{f\in F} f$. (For a formal proof see [Dechter and Mateescu, 2007b].)*

[3] We abuse notations here as $\bigotimes$ is defined between matrices or tables and here we have scalars

For probabilistic network when the combination is a product and the marginalization is a sum, the value of the root node is the probability of evidence. If we use the max marginalization operator, the value of the root is the mpe cost.

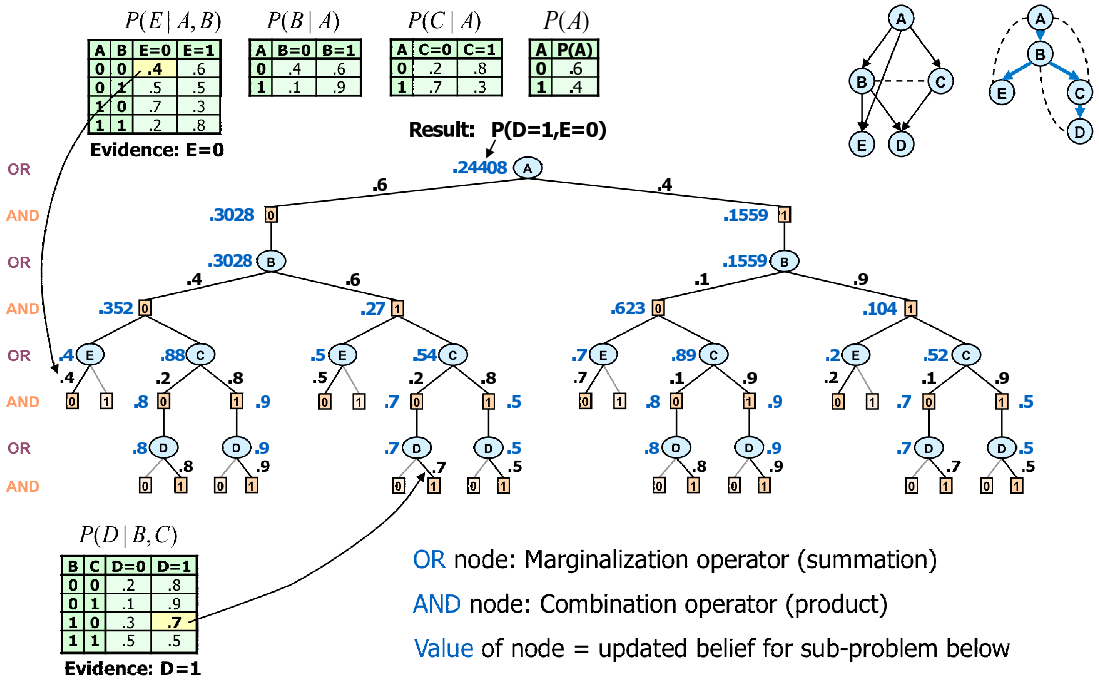
Example 6.40 Consider our AND/OR tree example of the probabilistic Bayesian network and assume that we have to find the probability of evidence P(D=1,E=0). The weighted AND/OR tree was depicted in 6.3. In Figure 6.13a we show also the value of each node for the query. Starting at leaf bodes for E and D we see that their values is "0" for the non-evidence value and "1" otherwise, indicated through faded arcs. Therefore the value of OR nodes D is as dictated by the appropriate weights. We see that when we go from leaves to root, the value of an OR node is the sum value of their child nodes, each multiplied by the arc-weight. For example, the value $v(n) = 0.88$ in the left part of the tree is obtained from the values of its child nodes, (0.8 and 0.9), each multiplied by their respective weights (0.2,0.8) yielding $v(n) = 0.2 \cdot 0.8 + 0.8 \cdot 0.9 = 0.88$. The value of AND nodes is the product of value of their OR child nodes. In part (b) we see the value associated with nodes in the AND/OR graph. In this case merging occur only in OR nodes labeled D, and the value of each node is computed in the same way.

6.4.1 SEARCHING AND/OR TREE (AOT) AND AND/OR GRAPH (AOG)

Search algorithms that traverse the AND/OR search space can compute the value of the root node yielding the answer to the query. In this section we present a typical depth-first algorithms that traverse AND/OR trees and graphs. We use *solution counting* as an example for a constraint query and the probability of evidence as an example for a probabilistic query. The application of these ideas for combinatorial optimization tasks, such as MPE is straightforward (at least in its brute-force manner). Effective and more sophisticated schemes (e.g., branch and bound or best-first search) were developed [Marinescu and Dechter, 2005, 2009a,b, Otten and Dechter, 2012].

Example 6.41 Looking again at Figure 6.13 we can see how value computation can be accomplished when we traverse the search space in a depth-first manner, using only linear space. If instead, the AND/OR graph is searched, we need to cache the results of subproblem's value computation in order to avoid redundant computation. This can be accomplished by caching using the node context. For example, in Figure 6.13 we keep a table for node D index by its context consisting of variables B and C. For each path to an OR node labeled D like (A=0, B=0, C=1, D=1), once we discovered that the value below this path is 0.8, we keep this value in the cache table indexed by the pair $(B = 0, C = 1)$. Likewise, for each assignment to these two variables the solution is cached and retrieved when the same context is encountered (see Figure 6.13b).

Algorithm 2, presents the basic depth-first traversal of the AND/OR search tree or search graph for counting the number of solutions of a constraint network, AO-COUNTING, (or for probability of evidence for belief networks, AO-BELIEF-UPDATING). As noted, the context based caching is done

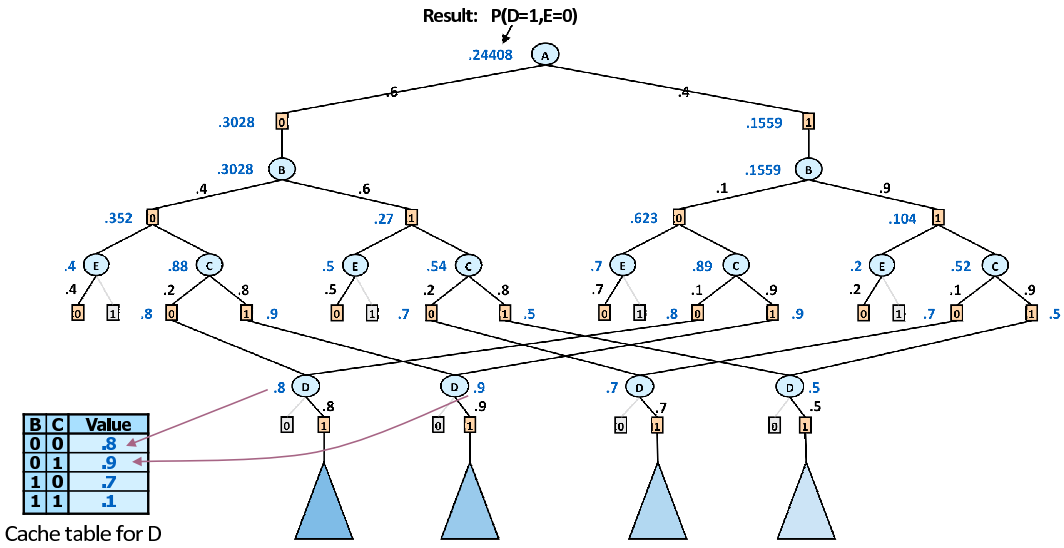


(a) Labeled AND/OR tree

Figure 6.13: Labeled AND/OR search tree and graphs for belief networks.

using tables. For each variable X_i, a table is reserved in memory for each possible assignment to its parent set pa_i which is its context. Initially each entry has a predefined value, in our case "-1." The fringe of the search is maintained on a stack called OPEN. The current node is denoted by n, its parent by p, and the current path by $path(n)$. The children of the current node are denoted by $successors(\mathtt{n})$. If caching is set to "false" the algorithm searches the AND/OR tree and we will refer to it as AOT.

The algorithm is based on two mutually recursive steps: EXPAND and PROPAGATE, which call each other (or themselves) until the search terminates. Before expanding an OR node, its cache table is checked (line 5). If the same context was encountered before, it is retrieved from cache, and $successors(\mathtt{n})$ is set to the empty set, which will trigger the PROPAGATE step. If a node is not



(b) Labeled AND/OR graph

Figure 6.13: Labeled AND/OR search tree and graphs for belief networks.

found in cache, it is expanded in the usual way, depending on whether it is an AND or OR node (lines 9–16). The only difference between counting and belief updating is line 11 vs. line 12. For counting, the value of a consistent AND node is initialized to 1 (line 11), while for belief updating, it is initialized to the bucket value for the current assignment (line 12). As long as the current node is not a dead-end and still has unevaluated successors, one of its successors is chosen (which is also the top node on `OPEN`), and the expansion step is repeated.

The bottom up propagation of values is triggered when a node has an empty set of successors (note that as each successor is evaluated, it is removed from the set of successors in line 30). This means that all its children have been evaluated, and its final value can now be computed. If the current node is the root, then the search terminates with its value (line 19). If it is an OR node, its value is saved in cache before propagating it up (line 21). If `n` is OR, then its parent `p` is AND and `p` updates its value by multiplication with the value of `n` (line 23). If the newly updated value of `p` is 0 (line 24), then `p` is a dead-end, and none of its other successors needs to be evaluated. An AND node `n` propagates its value to its parent `p` in a similar way, only by summation (line 29). Finally, the current node `n` is set to its parent `p` (line 31), because `n` was completely evaluated. The search continues either with a propagation step (if conditions are met) or with an expansion step.

Algorithm 2: AO-COUNTING / AO-BELIEF-UPDATING.

Input: A constraint network $\mathcal{R} = \langle X, D, C \rangle$, or a belief network $\mathcal{B} = \langle X, D, P \rangle$;
a pseudo tree $\mathcal{T}$ rooted at X_1; parents pa_i (OR-context) for every variable X_i;
`caching` set to $true$ or $false$.

Output: The number of solutions, or the updated belief, $v(X_1)$.

if `caching` $== true$ **then** `// Initialize cache tables`

1 Initialize cache tables with entries of "-1"

2 $v(X_1) \leftarrow 0$; `OPEN` $\leftarrow \{X_1\}$ `// Initialize the stack OPEN`

3 **while** `OPEN` $\neq \Phi$ **do**

4 $\mathtt{n} \leftarrow top(\mathtt{OPEN})$; remove $\mathtt{n}$ from `OPEN`

5 **if** `caching` $== true$ **and** $\mathtt{n}$ *is OR, labeled* X_i **and** $Cache(val(path(n))[pa_{X_i}]) \neq -1$ **then** `// In cache`

6 $v(\mathtt{n}) \leftarrow Cache(val(path(n))[pa_{X_i}])$ `// Retrieve value`

7 $successors(\mathtt{n}) \leftarrow \phi$ `// No need to expand below`

8 **else** `// EXPAND`

9 **if** $\mathtt{n}$ *is an OR node labeled* X_i **then** `// OR-expand`

10 $successors(\mathtt{n}) \leftarrow \{\langle X_i, x_i \rangle \mid \langle X_i, x_i \rangle$ is consistent with $path(n)\}$

11 $v(\langle X_i, x_i \rangle) \leftarrow 1$, for all $\langle X_i, x_i \rangle \in successors(\mathtt{n})$

12 $v(\langle X_i, x_i \rangle) \leftarrow \prod_{f \in B_{\mathcal{T}}(X_i)} f(val(path(n))[pa_{X_i}])$, for all $\langle X_i, x_i \rangle \in successors(\mathtt{n})$
`// AO-BU`

13 **if** $\mathtt{n}$ *is an AND node labeled* $\langle X_i, x_i \rangle$ **then** `// AND-expand`

14 $successors(\mathtt{n}) \leftarrow children_{\mathcal{T}}(X_i)$

15 $v(X_i) \leftarrow 0$ for all $X_i \in successors(\mathtt{n})$

16 Add $successors(\mathtt{n})$ to top of `OPEN`

17 **while** $successors(\mathtt{n}) == \Phi$ **do** `// PROPAGATE`

18 **if** $\mathtt{n}$ *is an OR node labeled* X_i **then**

19 **if** $X_i == X_1$ **then** `// Search is complete`

20 **return** $v(\mathtt{n})$

21 **if** `caching` $== true$ **then**

22 $Cache(val(path(n))[pa_{X_i}]) \leftarrow v(\mathtt{n})$ `// Save in cache`

23 $v(\mathtt{p}) \leftarrow v(\mathtt{p}) * v(\mathtt{c})$

24 **if** $v(\mathtt{p}) == 0$ **then** `// Check if p is dead-end`

25 remove $successors(\mathtt{p})$ from `OPEN`

26 $successors(\mathtt{p}) \leftarrow \Phi$

27 **if** $\mathtt{n}$ *is an AND node labeled* $\langle X_i, x_i \rangle$ **then**

28 let $\mathtt{p}$ be the parent of $\mathtt{n}$

29 $v(\mathtt{p}) \leftarrow v(\mathtt{p}) + v(\mathtt{n})$;

30 remove $\mathtt{n}$ from $successors(\mathtt{p})$

31 $\mathtt{n} \leftarrow \mathtt{p}$

6.5 GENERAL AND-OR SEARCH - AO(I)

General AND/OR algorithms for evaluating the value of a root node for any reasoning problem using tree or graph AND/OR search are identical to the above algorithms when product is replaced by the combination operator and summation is replaced by the marginalization operator. We can view the AND/OR tree algorithm (which we will denote AOT) and the AND/OR graph algorithm (denoted AOG) as two extreme cases in a parameterized collection of algorithms that trade space for time via

a controlling parameter i. We denote this class of algorithms as $AO(i)$ where i determines the size of contexts that the algorithm caches. Algorithm $AO(i)$ records nodes whose context size is i or smaller (the test in line 21 needs to be a bit more elaborate and check if the context size is smaller than i). Thus, AO(0) is identical to AOT, while $AO(w)$ is identical to AOG, where w is the induced width of the used pseudo-tree. For any intermediate i we get an intermediate level of caching, which is space exponential in i and whose execution time will increase as i decreases. Some elaboration follows.

6.5.1 COMPLEXITY

From Theorem 6.33 we can clearly conclude the following.

Theorem 6.42 *For any reasoning problem, algorithm* AOT *runs in linear space and in $O(nk^h)$ time, when h is the height of the guiding pseudo tree and k is the maximum domain size. If the primal graph has a tree decomposition with treewidth w, then there exists a pseudo tree $\mathcal{T}$ for which AOT is $O(nk^{w \cdot \log n})$. (Exercise: provide a proof).*

Obviously, for constraint satisfaction the algorithm would terminate early with a first solution, and would potentially be much faster than for the rest of the queries. Based on Theorem 6.29 we can derive a complexity bound when searching the AND/OR context-minimal search graph.

Theorem 6.43 *For any reasoning problem, the complexity of algorithm* AOG *(i.e., algorithm 2 for the flag caching=true) is time and space $O(nk^w)$ where w is the induced width of the guiding pseudo tree and k is the maximum domain size.*

Dead Caches. The space complexity of AOG can often be far tighter than exponential in the treewidth. One reason is related to the space complexity of tree decomposition schemes which, as we know, can operate in space exponential in the size of the cluster separators only, rather than be exponential in the cluster size. We will use the term *dead caches* to address this issue. Intuitively, a node that has only one incoming arc in the search tree will be traversed only once by DFS search, and therefore its value does not need to be remembered, as it will never be used again. Luckily, such nodes can be recognized based only on their context.

Definition 6.44 Dead cache. [Darwiche, 2001a] If X is the parent of Y in pseudo tree $\mathcal{T}$, and $context(X) \subset context(Y)$, then $context(Y)$ is a *dead cache*.

We know that a pseudo-tree is also a bucket-tree. That is, given a pseudo-tree we can generate a bucket-tree by associating a cluster for each variable X_i and its parents pa_{X_i} in the induced-graph. Following the pseudo-tree structure, some of the clusters may not be maximal, and these are

precisely the ones that correspond to dead caches. The parents pa_{X_i} that are not dead caches are those separators between maximal clusters in the bucket tree associated with the pseudo-tree.

Example 6.45 Consider the graphical models and the pseudo tree in Figure 6.12a. The context in the left branch (C, CK, CKL, $CKLN$) are all dead-caches. The only one which is not a dead cache is CKO, the context of P. As you can see, there are converging arcs into P only along this branch. Indeed if we describe the clusters of the corresponding bucket-tree, we would have just two maximal clusters: $CKLNO$ and $PCKO$ whose separator is CKO, which is the context of P. (Exercise: Determine the dead caches for the pseudo-tree in Figure 6.12d).

We can conclude the following.

Proposition 6.46 *If dead caches are not recorded, the space complexity of AOG can be reduced to being exponential in the separator's size only, while still being time exponential in the induced-width . (Prove as an exercise.)*

Finding a single solution. We can easily modify the algorithm to find a single solution. The main difference is that the $0/1$ v values of internal nodes are propagated using *Boolean* summation and product instead of regular operators. If there is a solution, the algorithm terminates as soon as the value of the root node is updated to 1. The solution subtree can be generated by following the pointers of the latest solution subtree. This, of course, is a very naive way of computing a single consistent solution as would be discussed in the context of mixed networks in Section 6.6.

Finding posterior marginals. To find posterior marginal of the root variable, we only need to keep the computation at the root of the search graph and normalize the result. However, if we want to find the belief (i.e., posterior marginal) for each variable we would need to make a more significant adaptation of the search scheme.

Optimization tasks. General AND/OR algorithms for evaluating the value of a root node for any reasoning problem using tree or graph AND/OR search spaces are identical to the above algorithms when product is replaced by the appropriate combination operator (i.e., product or summation) and marginalization by summation is replaced by the appropriate marginalization operator. For optimization (e.g., mpe) all we need is to change line 30 of the algorithm from summation to maximization. Namely, we should have $v(p) \leftarrow max\{v(p), v(n)\}$. Clearly, this will yield a base-line scheme that can be advanced significantly using heuristic search ideas. To compute the marginal map query we will use marginalization by sum or max based on the variable identity to which the marginalization operator is applied. In this case the pseudo-tree is *is constrained* to have all the max variables be a connected subtree that contain the root node.

Depth-first vs Best-first searches This could be the right place to make a clear distinction between searching the AND/OR space depth-first of best-first for optimization tasks. Best-first cannot exploit dead-caches, but must cache all nodes in the explicated graph. For this reason DFS can have far better memory utilization even when both scheme search an AND/OR graph.

6.6 AND/OR SEARCH ALGORITHMS FOR MIXED NETWORKS

We will consider now AND/OR search schemes in the context of mixed graphical models (see definition 2.25 in Section 2.6). To refresh, the mixed network is defined by a pair of a Bayesian network and a constraint network. This pair expresses a probability distribution over all the variables which is conditioned on the requirement that all the assignments having nonzero probability satisfy all the constraints. The constraint network may be specified explicitly as such, or can be extracted from the probabilistic network as those partial tuples whose probability is zero (see Definition 2.25 and [Darwiche, 2009] chapter 13).

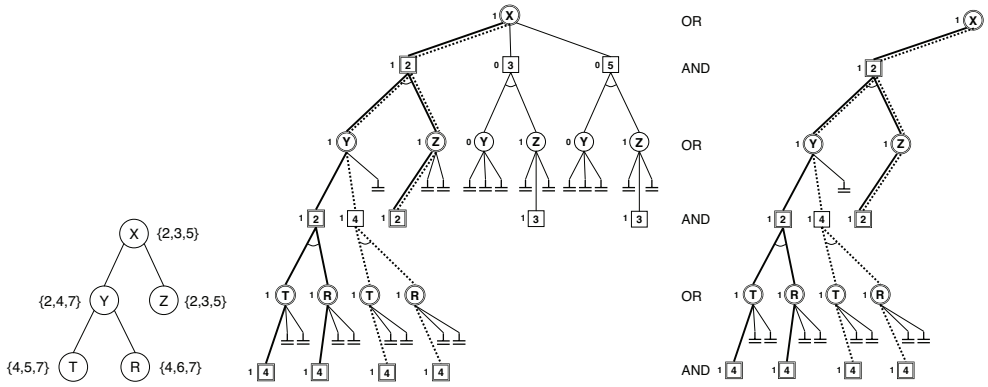
All advanced constraint processing algorithms [Dechter, 2003], either incorporating no-good learning and constraint propagation during search, or using variable elimination algorithms such as *adaptive-consistency* and *directional resolution*, can be applied to AND/OR search for mixed networks. In this section we will touch on these methods briefly because they need a considerable space to be treated adequately. Our aim is to point to the main principles of constraint processing that can have impact in the context of AND/OR search and refer the reader to the literature for full details (see [Dechter, 2003]).

Overall, the virtue of having the mixed network view is that the constraint portion can be processed by a wide range of constraint processing techniques, both statically before search, or dynamically during search [Dechter, 2003]. Recall the concept of *backtrack-free* (see Chapter 3).

Definition 6.47 Backtrack-free AND/OR search tree. Given graphical model $\mathcal{M}$ and given an AND/OR search tree $S_{\mathcal{T}}(\mathcal{M})$, the *backtrack-free AND/OR search tree* of $\mathcal{M}$ w.r.t. $\mathcal{T}$ is obtained by pruning from $S_{\mathcal{T}}(\mathcal{M})$ all inconsistent subtrees, namely all nodes that root no consistent partial solution (have a value 0).

Example 6.48 Consider 5 variables X, Y, Z, T, R over domains $\{2, 3, 5\}$, where the constraints are: X divides Y and Z, and Y divides T and R. The constraint graph and the AND/OR search tree relative to the guiding DFS tree rooted at X, are given in Figure 6.14a,b. In 6.14b we present the $S_{\mathcal{T}}(\mathcal{R})$ search space whose nodes' consistency status are already evaluated as having value "1" if consistent and "0" otherwise. We also highlight two solutions subtrees; one depicted by solid lines and one by dotted lines. Part (c) presents the backtrack-free tree where all nodes that do not root a consistent solution are pruned.

If we traverse the backtrack-free AND/OR search tree we can find a solution subtree without encountering any dead-ends. Some constraint networks specifications yield a backtrack-free search



(a) A constraint tree (b) Search tree (c) Backtrack-free search tree

Figure 6.14: AND/OR search tree and backtrack-free tree.

space. Others can be made backtrack-free by massaging their representation using *constraint propagation* algorithms. In particular, the variable-elimination algorithms *adaptive-consistency* described in Chapter 3, and directional resolution, compiles a constraint specification (resp., a Boolean CNF formula) into one that has a backtrack-free search space along some orderings. We remind now the definition of the directional extension (see also Chapter 3).

Definition 6.49 Directional extension [Dechter and Pearl, 1987, Rish and Dechter, 2000]. Let $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \bowtie \rangle$ be a constraint problem and let d be a DFS ordering of a pseudo tree, then the directional extension $E_d(\mathcal{R})$ denotes the constraint network (resp., the CNF formula) compiled by adaptive-consistency (resp., directional resolution) in reversed order of d.

Example 6.50 In Example 6.48, if we apply adaptive-consistency in reverse order of $d = (X, Y, T, R, Z)$, the algorithm will remove the values $\{3, 5\}$ from the domains of both X and Z yielding a tighter directional extension network $\mathcal{R}'$. As noted before, the AND/OR search tree of $\mathcal{R}'$ in Figure 6.14c is backtrack-free.

Proposition 6.51 *Given a constraint network $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \bowtie \rangle$, and a pseudo-tree $\mathcal{T}$, the AND/OR search tree of the graphical model compiled into a directional extension $E_d(\mathcal{R})$ when d is a DFS ordering of $\mathcal{T}$, coincides with the backtrack-free AND/OR search tree of $\mathcal{R}$ based on $\mathcal{T}$. (See Appendix for proof.)*

Proposition 6.51 emphasizes the significance of no-good learning for deciding inconsistency or for finding a single solution. These techniques are known as clause learning in SAT solvers [Jr.

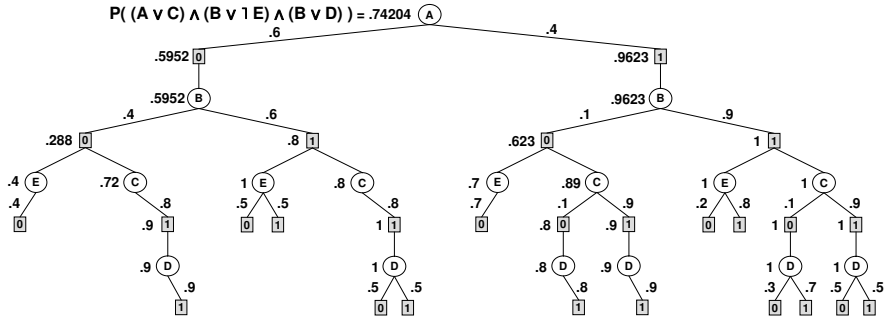


Figure 6.15: Mixed network defined by the query $\varphi = (A \vee C) \wedge (B \vee \neg E) \wedge (B \vee D)$.

and Schrag, 1997] and are currently used in most advanced solvers [Marques-Silva and Sakalla, 1999]. Namely, when we apply no-good learning we explore a pruned search space whose many inconsistent subtrees are removed, but we do so more gradually, during search, then when applying the full variable-elimination compilation, before search.

We will now give more details on applying constraint techniques while searching the AND/OR search space for processing queries over mixed networks. The mixed network can be processed by tightening the constraint network only. Namely we can process the deterministic information separately (e.g., by enforcing some consistency level [Dechter and Mateescu, 2007b]).

6.6.1 AND-OR-CPE ALGORITHM

Algorithm AND-OR-CPE for the constraint probabilistic evaluation query (CPE) (Definition 2.27) is given in Algorithm 3. The input is a mixed network, a pseudo tree $\mathcal{T}$ of the mixed graph and the context of each variable. The output is the probability that a random tuple generated from the belief network distribution is consistent (satisfies the constraint portion). As common with other queries, AND-OR-CPE traverses the AND/OR search tree or graph corresponding to $\mathcal{T}$ in a DFS manner and each node maintains a value v which accumulates the computation in its own subtree. As before we have a recursive computation. OR nodes accumulate the summation of the product between each child's value and its OR-to-AND weight, while AND nodes accumulate the product of their children's values. The context-based caching is done using table data structures as described earlier.

Example 6.52 We refer back to the example in Figure 6.3. Consider a constraint network that is defined by the CNF formula $\varphi = (A \vee C) \wedge (B \vee \neg E) \wedge (B \vee D)$. The trace of algorithm AND-OR-CPE without caching is given in Figure 6.15. Notice that the clause $(A \vee C)$ is not satisfied if $A = 0$ and $C = 0$, therefore the paths that contain this assignment cannot be part of a solution of the mixed network. The value of each node is shown to its left (the leaf nodes assume a dummy value of 1, not shown in the figure). The value of the root node is the probability of φ. Notice the similarity

Algorithm 3: AND-OR-CPE.

Input: A mixed network $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$ that expresses $P_{\mathcal{B}}$ and a constraint network $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \bowtie \rangle$; a pseudo tree $\mathcal{T}$ of the moral mixed graph, rooted at X_1; parents pa_i (OR-context) for every variable X_i; `caching` set to $true$ or $false$.

Output: The probability $P(\bar{x} \in \rho(\mathcal{R}))$ that a tuple satisfies the constraint query.

if `caching` $== true$ **then** `// Initialize cache tables`

1 Initialize cache tables with entries of "-1"

2 $v(X_1) \leftarrow 0$; `OPEN` $\leftarrow \{X_1\}$ `// Initialize the stack OPEN`

3 **while** `OPEN` $\neq \Phi$ **do**

4 $\mathtt{n} \leftarrow top(\mathtt{OPEN})$; remove $\mathtt{n}$ from `OPEN`

5 **if** `caching` $== true$ **and** $\mathtt{n}$ *is OR, labeled* X_i **and** $Cache(val(path(n))[pa_{X_i}]) \neq -1$ **then** `// If in cache`

6 $v(\mathtt{n}) \leftarrow Cache(val(path(n))[pa_{X_i}])$ `// Retrieve value`

7 $successors(\mathtt{n}) \leftarrow \Phi$ `// No need to expand below`

8 **else** `// Expand search (forward)`

9 **if** $\mathtt{n}$ *is an OR node labeled* X_i **then** `// OR-expand`

10 $successors(\mathtt{n}) \leftarrow$ `ConstraintPropagation`$(\langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle, val(path(n)))$ `//` **CONSTRAINT PROPAGATION**

11 $v(\langle X_i, x_i \rangle) \leftarrow \prod_{f \in B_{\mathcal{T}}(X_i)} f(val(path(n))[pa_{X_i}])$, for all $\langle X_i, x_i \rangle \in successors(\mathtt{n})$

12 **if** $\mathtt{n}$ *is an AND node labeled* $\langle X_i, x_i \rangle$ **then** `// AND-expand`

13 $successors(\mathtt{n}) \leftarrow children_{\mathcal{T}}(X_i)$

14 $v(X_i) \leftarrow 0$ for all $X_i \in successors(\mathtt{n})$

15 Add $successors(\mathtt{n})$ to top of `OPEN`

16 **while** $successors(\mathtt{n}) == \Phi$ **do** `// Update values (backtrack)`

17 **if** $\mathtt{n}$ *is an OR node labeled* X_i **then**

18 **if** $X_i == X_1$ **then** `// Search is complete`

19 **return** $v(\mathtt{n})$

20 **if** `caching` $== true$ **then**

21 $Cache(val(path(n))[pa_i]) \leftarrow v(\mathtt{n})$ `// Save in cache`

22 let $\mathtt{p}$ be the parent of $\mathtt{n}$

23 $v(\mathtt{p}) \leftarrow v(\mathtt{p}) * v(\mathtt{n})$

24 **if** $v(\mathtt{p}) == 0$ **then** `// Check if p is dead-end`

25 remove $successors(\mathtt{p})$ from `OPEN`

26 $successors(\mathtt{p}) \leftarrow \Phi$

27 **if** $\mathtt{n}$ *is an AND node labeled* $\langle X_i, x_i \rangle$ **then**

28 let $\mathtt{p}$ be the parent of $\mathtt{n}$

29 $v(\mathtt{p}) \leftarrow v(\mathtt{p}) + v(\mathtt{n})$;

30 remove $\mathtt{n}$ from $successors(\mathtt{p})$

31 $\mathtt{n} \leftarrow \mathtt{p}$

between Figure 6.15 and Figure 6.13. Indeed, in 6.13 we seek the probability of evidence which can be modeled as the CNF formula having unit clauses $D \wedge \neg E$.

Procedure `ConstraintPropagation`$(\mathcal{R}, \mathbf{x}_1^i)$.

Input: A constraint network $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$; a partial assignment path $\bar{x}_i$ to variable X_i.

Output: reduced domain D_i of X_i; reduced domains of future variables; newly inferred constraints.
This is a generic procedure that performs the desired level of constraint propagation, for example forward checking, unit propagation, arc consistency over the constraint network $\mathcal{R}$ and conditioned on $\mathbf{x}_1^i$.

return *reduced domain of* X_i

6.6.2 CONSTRAINT PROPAGATION IN AND-OR-CPE

We next discuss the use of constraint propagation during search. This methods are used in any constraint or SAT/CSP (see Chapters 5 and 6 in [Dechter, 2003]). In general, constraint propagation helps to discover what variable and what value to not instantiate in order to avoid dead-ends as much as possible. This is done with a bounded level of computation. The incorporation of these methods on top of AND/OR search for computation of the value of the root is straightforward. For illustration, we will only consider static variable ordering based on a pseudo tree, and will focus on the impact of constraint propagation on domain-value order of assignments to the variables.

In algorithm AND/OR-CPE, line 10 contains a call to the generic `ConstraintPropagation` procedure consulting only the constraint subnetwork $\mathcal{R}$, conditioned on the current partial assignment. The constraint propagation is relative to the current set of constraints, the given path that defines the current partial assignment, and the newly inferred constraints, if any, that were learned during search. `ConstraintPropagation` which requires polynomial time, may discover that some domain value-assignments to the variables cannot be extended to a full solution. These assignments are marked as dead-ends and removed from the current domain of the variable. All the remaining domain values remain feasible and are returned by the procedure as possible candidates to extend the search frontier. Clearly, not all those assignments are guaranteed to lead to a solution.

We therefore have the freedom to employ any procedure for checking the consistency of the constraints of the mixed network. The simplest case is when no constraint propagation is used and only the initial constraints of $\mathcal{R}$ are checked for consistency. We denote this algorithm by AO-C.

We also consider two forms of constraint propagation on top of AO-C. The first algorithm AO-FC, is based on *forward checking*, which is one of the weakest forms of propagation. It propagates the effect of a domain-value assignment to each future uninstantiated variable separately, and checks consistency against the constraints whose scope would become fully instantiated by just one such future variable.

The second algorithm, referred to as AO-RFC, performs a variant of *relational forward checking*. Rather than checking only constraints whose scope becomes fully assigned, AO-RFC checks all the existing constraints by looking at their projection on the variables along the current path. If the projection is empty an inconsistency is detected. AO-RFC is computationally more expensive

than AO-FC, but yields a more pruned search space. Finaly, AO-MAC applies full arc-consistency at each candidate value of the variable.

Example 6.53 Figure 6.16a shows the belief part of a mixed network, and Figure 6.16b the constraint part. All variables have the same domain, $\{1, 2, 3, 4\}$, and the constraints express "less than" relations. Figure 6.16c shows the search space of AO-C. Figure 6.16d shows the space traversed by AO-FC. Figure 6.16e shows the space when consistency is enforced with Maintaining Arc Consistency (which enforces full arc-consistency after each new instantiation of a variable).

SAT solvers. One possibility that was explored with success (e.g., [Allen and Darwiche, 2003]) is to delegate the constraint processing to a separate off-the-shelf SAT solver. In this case, for each new variable assignment the constraint portion is packed and fed into the SAT solver. If no solution is reported, then that value is a dead-end. If a solution is found by the SAT solver, then the AND/OR search continues (remember that for some tasks we may have to traverse all the solutions of the graphical model, so the one solution found by the SAT solver does not finish the task). Since, the worst-case complexity of this level of constraint processing, at each node, is exponential in the worst-case, a common alternative is to use *unit propagation*, or *unit resolution*, as a form of bounded resolution (see Section 3 and [Rish and Dechter, 2000]).

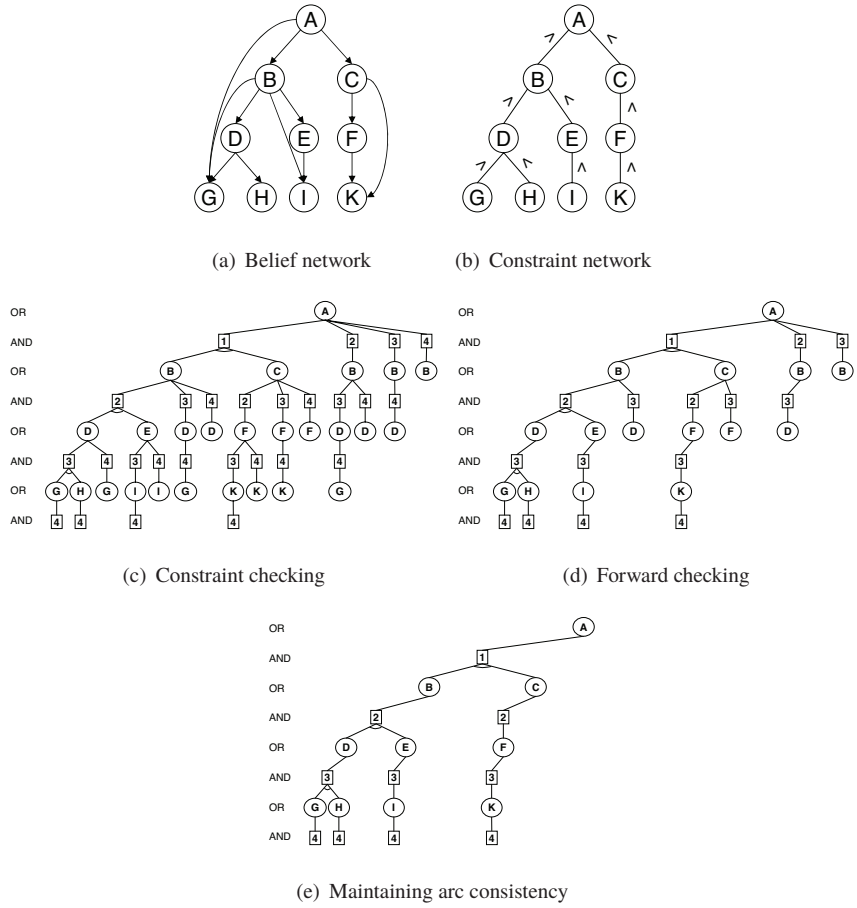
Such a hybrid use of search and a specialized efficient SAT (or constraint) solver can be very useful, and it underlines further the power that the mixed network representation has in delimiting the constraint portion from the belief network.

6.6.3 GOOD AND NOGOOD LEARNING

When a search algorithm encounters a dead-end, it can use different techniques to identify the ancestor variable assignments that caused the dead-end, which is called a *conflict-set*. It is conceivable that the same assignment of that set of ancestor variables may be encountered in the future, and they would then lead to the same dead-end. Rather than rediscovering it again, if memory allows, it is useful to record the dead-end conflict-set as a new constraint (or clause) over the ancestor variable set that is responsible for it. Recording dead-end conflict-sets is sometimes called nogood learning.

One form of nogood learning is graph-based, and it uses various techniques to identify the ancestor variables that generate the nogood. The information on conflicts is generated from the primal graph information alone. It is easy to show that AND/OR search already implements this information within the context of the nodes. Therefore, if caching is used, just saving the information about the nogoods encountered amounts to nogood learning techniques such as graph-based learning (see [Dechter, 2003]).

If deeper types of nogood learning are desirable, they need to be added on top of the AND/OR search. In such a case, a smaller set than the context of a node may be identified as a culprit assignment, and may help discover future dead-ends much earlier than when context-based caching alone is used. Needless to say, deeper learning is computationally more expensive.



(a) Belief network (b) Constraint network

(c) Constraint checking (d) Forward checking

(e) Maintaining arc consistency

Figure 6.16: Traces of AND-OR-CPE with various levels of constraint propagation.

In recent years [Beam, Kautz, Tian Sang, Bacchus and Piassi, 2004, Darwiche, 2001a, Dechter and Mateescu, 2007b], several schemes propose not only the learning of nogoods, but also that of their logical counterparts, the *goods*. Traversing the context minimal AND/OR graph and caching appropriately can be shown to implement both good and nogood graph-based learning.

6.7 SUMMARY AND BIBLIOGRAPHICAL NOTES

Chapters 6 presents search for graphical models using the concept of AND/OR search spaces rather than OR spaces. It introduced the AND/OR search tree, and showed that its size can be bounded exponentially by the depth of its pseudo tree over the graphical model. This implies exponential sav-

ings for any linear space algorithms traversing the AND/OR search tree. Specifically, if the graphical model has treewidth w^*, the height of the pseudo tree is $O(w^* \cdot \log n)$.

The AND/OR search tree can be further compacted into a graph by merging identical subtrees. We showed that the size of the minimal AND/OR search graph is exponential in the treewidth, while the size of the minimal OR search graph is exponential in the pathwidth. Since for some graphs the difference between treewidth and pathwidth is substantial, the AND/OR representation implies substantial time and space savings for memory intensive algorithms traversing the AND/OR graph. Searching the AND/OR search *graph* can be implemented by caching during search, while no-good recording is interpreted as pruning portions of the search space independent of it being a tree or a graph, an OR or an AND/OR.

The chapter is based on the work by Dechter and Mateescu [Dechter and Mateescu, 2007b]. The AND/OR search space is inspired by search advances introduced sporadically in the past three decades for constraint satisfaction and more recently for probabilistic inference and for optimization tasks. Specifically, it resembles pseudo tree rearrangement [Freuder and Quinn, 1987, Freuder, 1985], briefly introduced more than two decades ago, which was adapted subsequently for distributed constraint satisfaction [Collin, Dechter and Katz, 1991, 1999] and more recently in [Modi *et al.*, 2005], and was also shown to be related to graph-based backjumping [Dechter, 1992]. This work was extended in [Bayardo and Miranker, 1996] and more recently applied to optimization tasks [Larrosa and Sanchez, 2002]. Another version that can be viewed as exploring the AND/OR graphs was presented recently for constraint satisfaction [Terrioux and Jegou, 2003b] and for optimization [Terrioux and Jegou, 2003a]. Similar principles were introduced for probabilistic inference in algorithm Recursive Conditioning [Darwiche, 2001a] as well as in Value Elimination [F. Bacchus and Piassi, 2003a,b] and currently provide the backbones of the most advanced SAT solvers [Beam, Kautz, Tian Sang, Bacchus and Piassi, 2004].

It is known that exploring the search space in a dynamic variable ordering is highly beneficial. AND/OR search trees for graphical models can also be modified to allow dynamic variable ordering. This require a careful balancing act of the computational overhead that normally accompanies dynamic search schemes. For further information see [F. Bacchus and Piassi, 2003b, Marinescu and Dechter, 2009a].

6.8 APPENDIX: PROOFS

Proof of Theorem 6.13

(1) By definition, all the arcs of $S_{\mathcal{T}}(\mathcal{R})$ are consistent. Therefore, any solution tree of $S_{\mathcal{T}}(\mathcal{R})$ denotes a solution for $\mathcal{R}$ whose assignments are all the labels of the AND nodes in the solution tree. Also, by definition of the AND/OR tree, every solution of $\mathcal{R}$ must corresponds to a solution subtree in $S_{\mathcal{T}}(\mathcal{R})$.
(2) By construction, the set of arcs in every solution tree have weights such that each function of F contribute to one and only one weight via the combination operator. Since the total weight of the tree is derived by combination, it yields the cost of a solution. □.

Proof of Theorem 6.14

Let p be an arbitrary directed path in the pseudo tree $\mathcal{T}$ that starts with the root and ends with a leaf. This path induces an OR search subtree which is included in the AND/OR search tree $S_{\mathcal{T}}$, and its size is $O(k^h)$ when h bounds the path length. The pseudo tree $\mathcal{T}$ is covered by l such directed paths, whose lengths are bounded by h. The union of their individual search trees covers the whole AND/OR search tree $S_{\mathcal{T}}$, where every distinct full path in the AND/OR tree appears exactly once, and therefore, the size of the AND/OR search tree is bounded by $O(l \cdot k^h)$. Since $l \leq n$ and $l \leq b^h$, it concludes the proof. The bounds are tight because they are realizable for graphical models whose all full assignments are consistent. □.

Proof of Theorem 6.19

(1) All we need to show is that the *merge* operator is not dependant on the order of applying the operator. Mergeable nodes can only appear at the same level in the AND/OR graph. Looking at the initial AND/OR graph, before the merge operator is applied, we can identify all the mergeable nodes per level. We prove the proposition by showing that if two nodes are initially mergeable, then they must end up merged after the operator is applied exhaustively to the graph. This can be shown by induction over the level where the nodes appear.

Base case: If the two nodes appear at the leaf level (level 0), then it is obvious that the exhaustive merge has to merge them at some point.

Inductive step: Suppose our claim is true for nodes up to level k and two nodes n_1 and n_2 at level $k+1$ are initially identified as mergeable. This implies that, initially, their corresponding children are identified as mergeable. These children are at level k, so it follows from the inductive hypothesis that the exhaustive merge has to merge the corresponding children. This in fact implies that nodes n_1 and n_2 will root the same subgraph when the exhaustive merge ends, so they have to end up merged. Since the graph only becomes smaller by merging, based on the above the process, merging has to stop at a fix point.

(2) Analogous to (1).

(3) If the nodes can be merged, it follows that the subgraphs are identical, which implies that they define the same conditioned subproblems, and therefore the nodes can also be unified. □

Proof of Proposition 6.51.

Note that if $\mathcal{T}$ is a pseudo tree of $\mathcal{R}$ and if d is a DFS ordering of $\mathcal{T}$, then $\mathcal{T}$ is also a pseudo tree of $E_d(\mathcal{R})$ and therefore $S_{\mathcal{T}}(E_d(\mathcal{R}))$ is a faithful representation of $E_d(\mathcal{R})$. $E_d(\mathcal{R})$ is equivalent to $\mathcal{R}$, therefore $S_{\mathcal{T}}(E_d(\mathcal{R}))$ is a supergraph of $BF_{\mathcal{T}}(\mathcal{R})$. We only need to show that $S_{\mathcal{T}}(E_d(\mathcal{R}))$ does not contain any dead-ends, in other words any consistent partial assignment must be extendable to a solution of $\mathcal{R}$, This however is obvious, because Adaptive consistency makes $E_d(\mathcal{R})$ strongly directional $w^*(d)$ consistent, where $w^*(d)$ is the induced width of R along ordering d [Dechter and Pearl, 1987], a notion that is synonym with backtrack-freeness. □