Chapter 7

Search Algorithms for Graphical Models; AND/OR spaces

As discussed before, 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 already) are good at exploiting the independencies captured by the underlying graphical model and in avoiding redundant computation. They provide worst case time guarantee, as they are time 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 separator width and, therefore, not practical for models with large treewidth.

Traditional search-based algorithms (*e.g.*, depth-first branch-and-bound, best-first search) traverse the model's search space where each path represents a partial or full solution. Such search trees were just discussed in towards the end of Chapter **??** when we briefly looked at the tradeoff between hybrids conditioning and inference. Note that the linear structure of search spaces does not retain the independencies represented in the underlying graphical models and, therefore, algorithms exploring such traditional search-based algorithms may not be effective. On the other hand, the space requirements of search-based algorithms may be much less severe than those of inference-based algorithms and they can accommodate a wide spectrum of space-bounded algorithms. In addition, search methods require only an implicit, generative, specification of the functional relationship

(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-based algorithms are the only choice available for models with large treewidth and with implicit representation.

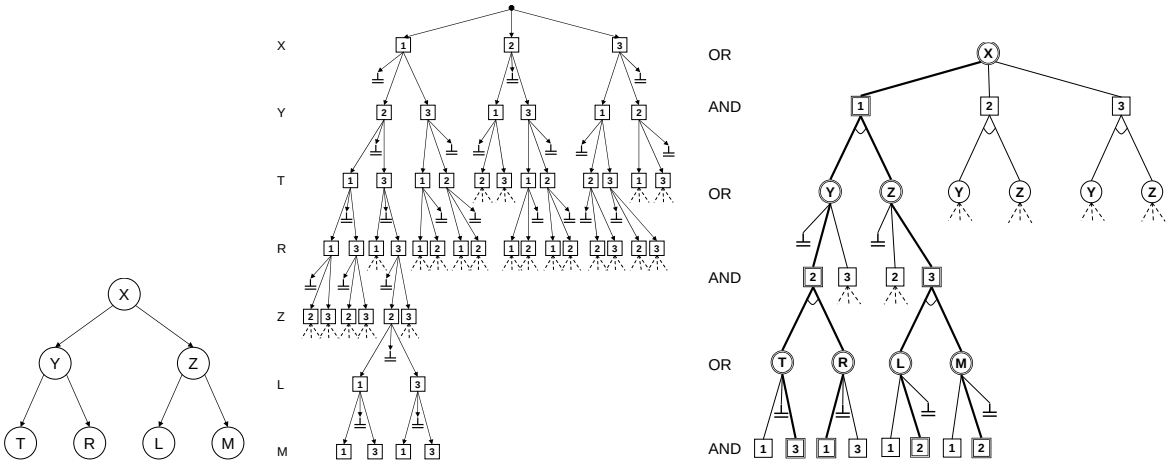
In this chapter we will show that an AND/OR search spaces, originally developed for heuristic search [45], can be used to encode some of the information in the graphical models. We will demonstrate how the independencies captured by the graphical model may be used to yield AND/OR search trees that are exponentially smaller than the standard search tree (that can be thought of as an OR tree). Specifically, we show that the size of the AND/OR search tree is bounded exponentially by the depth of a spanning pseudo tree over the graphical model. We then show that the search tree contains significant redundancy. However, if we recognize the search graph underlying the graphical model additional saving can be results. Indeed we show that the size of the AND/OR search graph is bounded exponentially by the treewidth,

7.1 AND/OR Search Trees

We will present the AND/OR search space for a *graphical model* starting with an example of a constraint network.

Example 7.1.1 Consider the simple tree graphical model (*i.e.*, the primal graph is a tree) in Figure 7.1(a), over domains $\{1, 2, 3\}$, which represents a graph-coloring problem. Namely, each node should be assigned a value such that adjacent nodes have different values. 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 need 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, $\langle X, 2\rangle$ and $\langle X, 3\rangle$. Applying the decomposition along the tree (in Figure 7.1(a) yields the AND/OR search tree in Figure 7.1(c).

In the AND/OR space a full assignment to all the variables is not a path but a subtree. For comparison, the traditional *OR* search tree is depicted in Figure 7.1(b). Clearly, the size of the AND/OR search space is smaller than that of the regular OR space. The OR search space has $3 \cdot 2^7$ nodes while the AND/OR has $3 \cdot 2^5$ (compare 7.1(b) with 7.1(c)). If k is the domain size, a balanced binary tree with n nodes has an OR search tree of



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

Figure 7.1: OR vs. AND/OR search trees; note the connector for AND arcs

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 $O(n^2)$. □

The AND/OR space is not restricted to tree graphical models. It only has to be guided by a *backbone* tree which spans the original primal graph of the graphical model in a particular way. We will define the AND/OR search space relative to a depth-first search tree (DFS tree) of the primal graph first, and will generalize to a broader class of backbone spanning trees subsequently.

Definition 7.1.2 (DFS spanning tree) *Given a DFS traversal ordering $d = (X_1, \ldots, X_n)$, of an undirected graph $G = (V, E)$, the* DFS spanning tree $\mathcal{T}$ *of G is defined as the tree rooted at the first node, X_1, which includes only the traversed (by DFS) arcs of G . Namely, $\mathcal{T} = (V, E')$, where $E' = \{(X_i, X_j) \mid X_j \ traversed \ from \ X_i\}$.*

Wenext define the notion of AND/OR search tree for a graphical model.

Definition 7.1.3 (AND/OR search tree) *Given a graphical model $\mathcal{R} = \langle X, D, F, \bigotimes \rangle$, its primal graph G and a backbone DFS tree $\mathcal{T}$ of G, the associated AND/OR search tree, denoted $S_{\mathcal{T}}(R)$, 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 backbone tree $\mathcal{T}$. The root of the AND/OR search tree is an OR node labeled by the root of $\mathcal{T}$. A path from the root of the search tree $S_{\mathcal{T}}(\mathcal{R})$ to a node n is denoted by π_n. If n is labeled X_i or x_i the path will be denoted $\pi_n(X_i)$ or $\pi_n(x_i)$, respectively. The assignment sequence along path π_n, denoted $asgn(\pi_n)$ is the set of value assignments associated with the sequence of AND nodes along π_n:

$$\begin{aligned} asgn(\pi_n(X_i)) &= \{\langle X_1, x_1 \rangle, \langle X_2, x_2 \rangle, \ldots, \langle X_{i-1}, x_{i-1} \rangle\}, \\ asgn(\pi_n(x_i)) &= \{\langle X_1, x_1 \rangle, \langle X_2, x_2 \rangle, \ldots, \langle X_i, x_i \rangle\}. \end{aligned}$$

The set of variables associated with OR nodes along path π_n is denoted by $var(\pi_n)$: $var(\pi_n(X_i)) = \{X_1, \ldots, X_{i-1}\}$, $var(\pi_n(x_i)) = \{X_1, \ldots, X_i\}$. The exact 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 $asgn(\pi_n)$. Consistency is defined relative to the flat constraints.*

2. *An AND node m, labeled $\langle X_i, x_i \rangle$ has a child OR node r labeled Y, iff Y is child of X in the backbone 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 7.1.8).*

Clearly, if a node n is labeled X_i (OR node) or x_i (AND node), $var(\pi_n)$ is the set of variables mentioned on the path from the root to X_i in the backbone tree, denoted by $path_{\mathcal{T}}(X_i)$[1].

A solution subtree is defined in the usual way:

Definition 7.1.4 (solution subtree) *A* solution subtree *of an AND/OR search tree contains the root node. For every OR nodes it contains one of its child nodes and for each of its AND nodes it contains all its child nodes, and all its leaf nodes are consistent.*

Example 7.1.5 In the example of Figure 7.1(a), $\mathcal{T}$ is the DFS tree which is the tree rooted at X, and accordingly the root OR node of the AND/OR tree in 7.1(c) is X. Its child nodes are labeled $\langle X, 1 \rangle, \langle X, 2 \rangle, \langle X, 3 \rangle$ (only the values are noted in the Figure),

[1]When the AND/OR tree is extended to dynamic variable orderings the set of variables along different paths may vary.

which are AND nodes. From each of these AND nodes emanate two OR nodes, Y and Z, since these are the child nodes of X in the DFS tree of (7.1(a)). 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 DFS tree. In 7.1(c) a solution tree is highlighted. □

7.1.1 Weights of OR-AND Arcs

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

Definition 7.1.6 (arc weight for constraint networks) *Given an AND/OR 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 and thus solved). An outgoing arc of a terminal OR node has weight "0", (there is no consistent value assignments). 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 graphical model using the notion of buckets of functions.

Definition 7.1.7 (buckets relative to a backbone tree) *Given a graphical model $\mathcal{R} = \langle X, D, F, \bigotimes\rangle$ and a backbone tree $\mathcal{T}$, the* bucket *of X_i relative to $\mathcal{T}$, denoted $B_{\mathcal{T}}(X_i)$, is the set of functions whose scopes contain X_i and are included in $path_{\mathcal{T}}(X_i)$, which is the set of variables from the root to X_i in $\mathcal{T}$. Namely,*

$$B_{\mathcal{T}}(X_i) = \{f \in F | X_i \in scope(f), scope(f) \subseteq path_{\mathcal{T}}(X_i)\}.$$

Definition 7.1.8 (OR-to-AND weights) *Given an AND/OR tree $S_{\mathcal{T}}(\mathcal{R})$, of a graphical model $\mathcal{R}$, the weight $w_{(n,m)}(X_i, x_i)$ of arc (n, m) where X_i labels n and x_i labels m, is the* combination *of all the functions in $B_{\mathcal{T}}(X_i)$ assigned by values along π_m. Formally, $w_{(n,m)}(X_i, x_i) = \bigotimes_{f \in B_{\mathcal{T}}(X_i)} f(asgn(\pi_m)[scope(f)])$.*

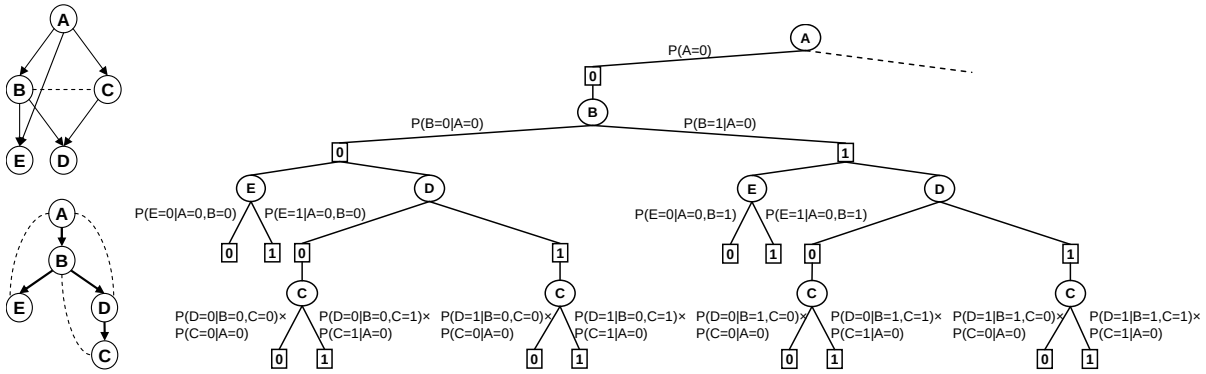


Figure 7.2: Arc weights for probabilistic networks

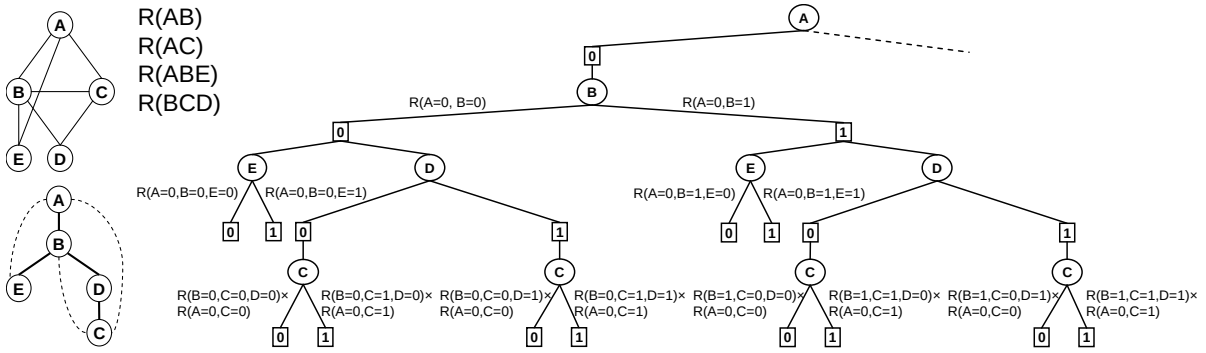
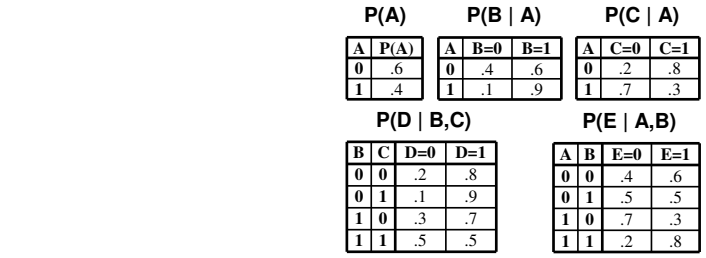


Figure 7.3: Arc weights for constraint networks

Definition 7.1.9 (weight of a solution subtree) *Given a weighted AND/OR tree $S_{\mathcal{T}}(\mathcal{R})$, of a graphical model $\mathcal{R}$, and given a solution subtree t having OR-to-AND set of arcs $arcs(t)$, the weight of t is defined by $w(t) = \bigotimes_{e \in arcs(t)} w(e)$.*

Example 7.1.10 Figure 7.2 shows a belief network, a DFS tree that drives its weighted AND/OR search tree, and a portion of the AND/OR search tree with the appropriate weights on the arcs expressed symbolically. In this case the bucket of E contains the function $P(E|A,B)$, and the bucket of C contains two functions, $P(C|A)$ and $P(D|B,C)$. Note that $P(D|B,C)$ belongs neither to the bucket of B nor to the bucket of D, but it is contained in the bucket of C, which is the last variable in its scope to be instantiated in a path from the root of the tree. We see indeed that the weights on the arcs from the OR node E and any of its AND value assignments include only the instantiated function $P(E|A,B)$, while the weights on the arcs connecting C to its AND child nodes are the products of the two functions in its bucket instantiated appropriately. Figure



P(A)

A	P(A)
0	.6
1	.4

P(B | A)

A	B=0	B=1
0	.4	.6
1	.1	.9

P(C | A)

A	C=0	C=1
0	.2	.8
1	.7	.3

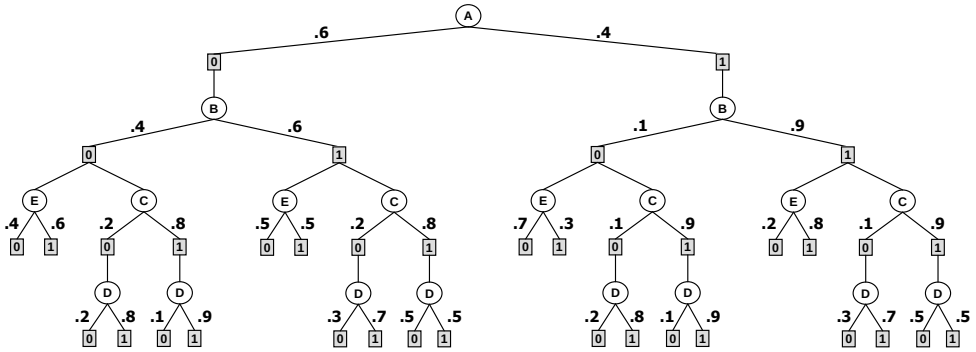
P(D | B,C)

B	C	D=0	D=1
0	0	.2	.8
0	1	.1	.9
1	0	.3	.7
1	1	.5	.5

P(E | A,B)

A	B	E=0	E=1
0	0	.4	.6
0	1	.5	.5
1	0	.7	.3
1	1	.2	.8

(a) CPTs



(b) Labeled AND/OR tree

Figure 7.4: Labeled AND/OR search tree for belief networks

7.3 shows a constraint network with four relations, a backbone DFS tree and a portion of the AND/OR search tree with weights on the arcs. Note that the complex weights would reduce to "0"s and "1"s in this case. However, since we use the convention that arcs appear in the search tree only if they represent a consistent extension of a partial solution, we will not see arcs having zero weights.

In Figure 7.4(b) we show the numerical values of the weights on the weighted AND/OR tree for the same belief network where 7.4(a) shows the conditional probability tables. □

7.1.2 Properties of AND/OR Search Tree

Any DFS tree $\mathcal{T}$ of a graph G has the property that the arcs of G which are not in $\mathcal{T}$ are backarcs. Namely, they connect a node and one of its ancestors in the backbone tree. This ensures that each scope of F will be fully assigned on some path in $\mathcal{T}$, a property that is essential for the validity of the AND/OR search tree. We can show that the AND/OR search tree is an alternative equivalent representation of the graphical model.

Theorem 7.1.11 (correctness) *Given a graphical model $\mathcal{R}$ having a primal graph G and a DFS spanning tree $\mathcal{T}$ of G, its weighted AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ then 1) there is a one-to-one correspondence between solution subtrees of $S_{\mathcal{T}}(\mathcal{R})$ and solutions of $\mathcal{R}$; 2) the weight of any solution tree equals the cost of the full solution it denotes; namely, if t is a solution tree of $S_{\mathcal{T}}(\mathcal{R})$ then $F(assn(t)) = w(t)$, where $assn(t)$ is the full solution defined by tree t.*

Proof: Prove as an exercise ∎

The virtue of an AND/OR search tree representation is that its size may be far smaller than the traditional OR search tree. The size of an AND/OR search tree depends on the depth of its backbone DFS tree $\mathcal{T}$. Therefore, DFS trees of smaller depth should be preferred to drive the AND/OR search *tree*. An AND/OR search tree becomes an OR search tree when its DFS tree is a chain.

Theorem 7.1.12 (size bounds of AND/OR search tree) *Given a graphical model $\mathcal{R}$, with domains size bounded by k, and a DFS spanning tree $\mathcal{T}$ having depth m and l leaves, the size of its AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ is $O(l \cdot k^m)$ (and therefore also $O(nk^m)$ and $O((bk)^m)$ when b bounds the branching degree of $\mathcal{T}$ and n bounds the number of nodes). In contrast the size of its OR search tree along any ordering is $O(k^n)$. The above bounds are tight and realizable for fully consistent graphical models. Namely, one whose all full assignments are consistent.*

Proof: Let p be an arbitrary directed path in the DFS 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 $O(k^m)$ when m bounds the path length. The DFS tree $\mathcal{T}$ is covered by l such directed paths, whose lengths are bounded by m. 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 $O(l \cdot k^m)$. Since $l \leq n$ and $l \leq b^m$, it concludes the proof. ∎

Table 7.1 demonstrates the size saving of AND/OR vs. OR search spaces for 5 random networks having 20 bivalued variables, 18 CPTs with 2 parents per child and 2 root nodes, when all the assignments are consistent (remember that this is the case when the

Table 7.1: OR vs. AND/OR search size, 20 nodes

		OR space		AND/OR space		
treewidth	height	time (sec.)	nodes	time (sec.)	AND nodes	OR nodes
5	10	3.154	2,097,151	0.03	10,494	5,247
4	9	3.135	2,097,151	0.01	5,102	2,551
5	10	3.124	2,097,151	0.03	8,926	4,463
5	10	3.125	2,097,151	0.02	7,806	3,903
6	9	3.124	2,097,151	0.02	6,318	3,159

probability distribution is strictly positive). The size of the OR space is the full binary tree of depth 20. The size of the full AND/OR space varies based on the backbone DFS tree.

We can give a better analytic bound on the search space size by spelling out the depth m_i of each leaf node L_i in $\mathcal{T}$ as follows. Given a graphical model $\mathcal{R}$, with domains size bounded by k, and a backbone spanning tree $\mathcal{T}$ having $L = \{L_1, \ldots, L_l\}$ leaves, where depth of leaf L_i is m_i, then the size of its full AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ is $O(\sum_{k=1}^{l} k^{m_i})$. Alternatively, we can use the exact domain sizes for each variable yielding an even more accurate expression $O(\sum_{L_k \in L} \Pi_{\{X_j | X_j \in path_{\mathcal{T}}(L_k)\}} |D(X_j)|)$.

7.1.3 From DFS Trees to Pseudo Trees

You may wonder if DFS trees are the only type of trees that can serve as backbone trees to guide the AND?oR decomposition. You were right! Indeed, there is a larger class of trees that can be used as backbones for AND/OR search trees, called *pseudo trees* [27] and they just need to obey the above mentioned back-arc property.

Definition 7.1.13 (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 of G which is not included 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 $\mathcal{T}$ is defined as $G^{\mathcal{T}} = (V, E \cup E')$.*

Clearly, any DFS tree and any chain of a graph are pseudo trees.

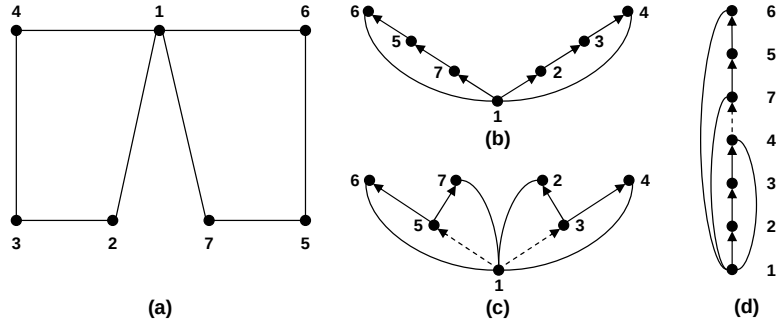


Figure 7.5: (a) A graph; (b) a DFS tree $\mathcal{T}_1$; (c) a pseudo tree $\mathcal{T}_2$; (d) a chain pseudo tree $\mathcal{T}_3$

Example 7.1.14 Consider the graph G displayed in Figure 7.5(a). Ordering $d_1 = (1, 2, 3, 4, 7, 5, 6)$ is a DFS ordering of a DFS tree $\mathcal{T}_1$ having the smallest DFS tree depth of 3 (Figure 7.5(b)). The tree $\mathcal{T}_2$ in Figure 7.5(c) 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. Tree $\mathcal{T}_3$ in Figure 7.5(d), is a chain. The extended graphs $G^{\mathcal{T}_1}$, $G^{\mathcal{T}_2}$ and $G^{\mathcal{T}_3}$ are presented in Figure 7.5(b),(c),(d) when we ignore directionality and include the dotted arcs. □

It is easy to see that the weighted AND/OR search tree is well defined when the backbone trees is a pseudo tree. Namely, the correctness properties (proposition 7.1.11) hold and the size bounds are extendible.

Theorem 7.1.15 (properties of AND/OR search trees) *Given a graphical model $\mathcal{R}$ and a backbone pseudo tree $\mathcal{T}$, its weighted AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ obey the correctness properties (1) and (2) of Proposition 7.1.11 and its size is $O(l \cdot k^m)$ where m is the depth of the pseudo tree, l bounds its number of leaves, and k bounds the domain size.*

Proof: All the arguments in the proof for Theorem 7.1.11 carry immediately to AND/OR search spaces that are defined relative to a pseudo tree. Likewise, the bound size argument in the proof of Theorem 7.1.12 holds relative to the depth of the more general pseudo tree. ■

Figure 7.6 shows the AND/OR search trees along the pseudo trees $\mathcal{T}_1$ and $\mathcal{T}_2$ from Figure 7.5. The domains of the variables are $\{a, b, c\}$ and the constraints are universal, namely this can represent a positive graphical model. We see that the AND/OR search

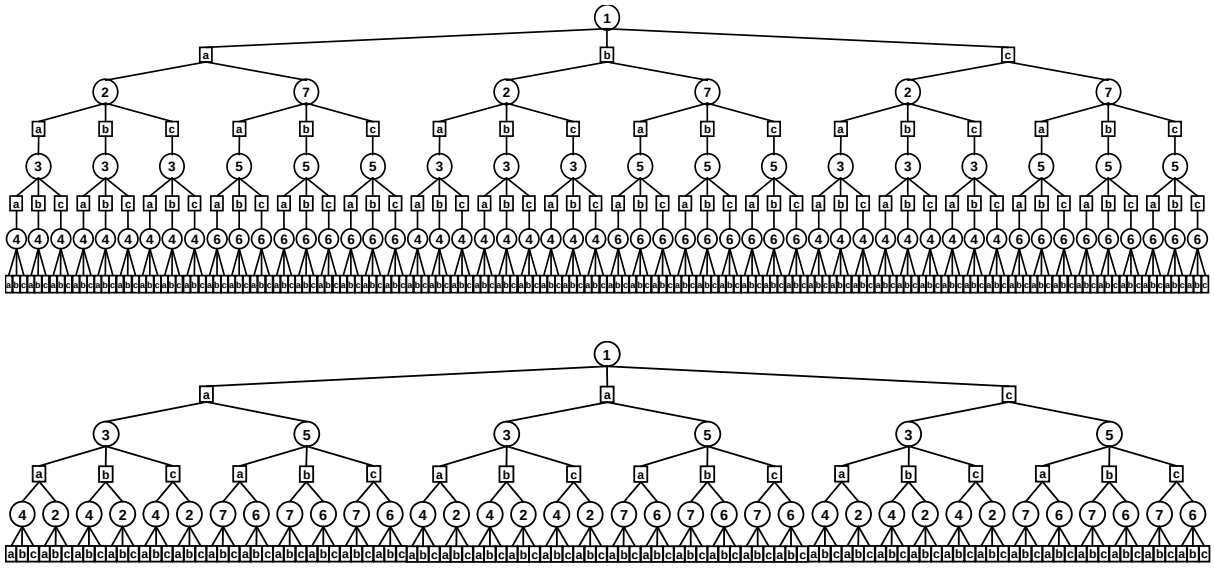


Figure 7.6: AND/OR search tree along pseudo trees $\mathcal{T}_1$ and $\mathcal{T}_2$

tree based on $\mathcal{T}_2$ is smaller, because $\mathcal{T}_2$ has a smaller depth than $\mathcal{T}_1$. The weights are not specified here.

Clearly, the question we should address is how can we find good pseudo-tree, namely, those having minimal depth. This is yet another graph problem (appended to finding good induced-width and cycle-cutset sizes) which was explored in the past 2 decades and I am sure the reader may not be surprised to learn that, finding a pseudo tree or a DFS tree of minimal depth is NP-complete, like other relevant graph problems which we encountered. Greedy algorithms and polynomial time strategies were developed in the graph literature and some will be discussed shortly (see section 7.1.4). A general approach to obtaining pseudo trees is by generating an induced graph along an ordering d and then traversing it in a depth-first manner, breaking ties in favor of earlier variables [5].

Interestingly, a pseudo-tree and the bucket-tree as defined in 6.1.1 are highly related. In fact. it is easy to show that a *bucket tree* yields a pseudo tree.

Proposition 7.1.16 *Given a graphical model* $\mathcal{R} =< X, D, F >$ *and a bucket-tree* $T_B = (\{B_{X_i} | X_i \in X\}, E_B)$*, the tree* $T = (X, E)$ *whose arcs are* $E = \{(X_i, X_j) | (B_{X_i}, B_{X_j}) \in E_B\}$ *is a pseudo tree of* $\mathcal{R}$*.*

Table 7.2: Average depth of pseudo trees vs. DFS trees; 100 instances of each random model. N is the number of variables, P is the number of variables in the scope of a function and C is the number of functions.

Model (DAG)	width	Pseudo tree depth	DFS tree depth
(N=50, P=2, C=48)	9.5	16.82	36.03
(N=50, P=3, C=47)	16.1	23.34	40.60
(N=50, P=4, C=46)	20.9	28.31	43.19
(N=100, P=2, C=98)	18.3	27.59	72.36
(N=100, P=3, C=97)	31.0	41.12	80.47
(N=100, P=4, C=96)	40.3	50.53	86.54

Proof: All one need to show is that all the arcs in R which are not in T are back-arcs, which is easy to verify based on the construction of a bucket-tree (prove formally as an exercise). ∎

Interestingly, there is a known relationship between the treewidth and the depth of pseudo trees. To gain intuition the reader should prove that a chain graph of size n has a pseudo-tree whose depth is $logn$.

Proposition 7.1.17 *[5, 30] The minimal depth m over all pseudo trees of a given graph G satisfies $m \leq w^* \cdot \log n$, where w^* is the treewidth of G, which is the primal graph of the given graphical model.*

This property suggets a bound on the size of the AND/OR search tree of a graphical model in terms of its treewidth.

Theorem 7.1.18 *A graphical model that has a treewidth w^* has an AND/OR search tree whose size is $O(n \cdot k^{(w^* \cdot \log n)})$, where k bounds the domain size and n is the number of variables.*

To illustrate, Table 7.2 shows the effect of DFS spanning trees against pseudo trees, both generated using brute-force heuristics over randomly generated graphs.

7.1.4 Finding Good Pseudo Trees

We describe next two heuristics for generating pseudo trees with relatively small depths/induced-widths which are common in practice.

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

Table 7.3: Bayesian Networks Repository (left); SPOT5 benchmarks (right).

Min-Fill Heuristic

As we discussed in Chapter **??** *Min-Fill* is one of the best and most widely used heuristics for creating small induced width elimination orders. An ordering is generated by placing the variable with the smallest *fill set* (which is number of induced edges that need be added to fully connect the neighbors of a node) at the end of the ordering, connecting all of its neighbors and then removing the variable from the graph. The process continues until all variables have been eliminated.

Once an elimination order is given and once the induced-graph is generated, the pseudo tree can be extracted as a depth-first traversal of the induced graph, starting with the variable that initiated the ordering, always preferring as successor of a node the earliest adjacent node in the induced graph. This way a given ordering uniquely determines a pseudo tree. This approach was first used by [49].

Hypergraph Decomposition Heuristic

An alternative heuristic for generating a low height balanced pseudo tree is based on the recursive decomposition of the dual hypergraph associated with the graphical model.

Definition 7.1.19 (dual hypergraph) *The* dual hypergraph *of a graphical model* $\langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, *is a pair* $\mathcal{H} = (\mathbf{V}, \mathbf{E})$, *where each function in* $\mathbf{F}$ *is a vertex* $v_i \in \mathbf{V}$ *and Two functions are connected if they share variables.*

Definition 7.1.20 (hypergraph separators) *Given a dual hypergraph* $\mathcal{H} = (\mathbf{V}, \mathbf{E})$ *of a graphical model, a* hypergraph separator decomposition *of size* k *by a 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. [2].

Table7.3 illustrates the induced width and depth of the pseudo tree obtained with the hypergraph and min-fill heuristics for 10 Bayesian networks from the Bayesian Networks Repository[3] and 10 constraint networks derived from the SPOT5 benchmark [7] which are typical. In all the accumulated experience it is observed that the min-fill heuristic generates lower induced width pseudo trees, while the hypergraph heuristic produces much smaller depth pseudo trees.

7.2 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 reasoning subproblems, are identified. Any two such nodes can be *merged*, reducing the size of the search graph. For example, in Figure 7.1(c), the search trees below any appearance of $\langle Y, 2 \rangle$ are all identical, and therefore can be merged.

Sometimes, two nodes may not root identical subtrees, but they could still root search subspaces that correspond to equivalent subproblems. Such nodes that root equivalent subproblems can be *unified* even if their explicit weighted subtrees are not identical, yielding an even smaller search graph. We next formalize the notions of *merging* and *unifying* nodes and define the minimal AND/OR search graph.

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

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

In general two graphical models are equivalent if they have the same set of solutions, and if each is associated with the same *cost* when the cost defined by the combination (e.g., product) of the functions components in F. Formally,

Definition 7.2.1 (universal equivalent graphical model) *Given a graphical model* $\mathcal{R} = \langle X, D, F, \bigotimes \rangle$ *the universal equivalent model of* $\mathcal{R}$ *is* $u(\mathcal{R}) = \langle X, D, F = \{\bigotimes_{i=1}^{r} f_i\}, \bigotimes \rangle$.

we also need to define the cost of a partial solution and a graphical model conditioned on a partial assignment. Informally, a conditioned graphical model on a particular partial assignment is obtained by assigning all the variables and modifying cost in the obvious way.

Definition 7.2.2 (cost of an assignment, conditional model) *Given a graphical model* $\mathcal{R}$,

1. *the cost of a full assignment* $x = (x_1, ..., x_n)$ *is defined by* $c(x) = \bigotimes_{f \in F} f(x[scope(f)])$. *The* cost of a partial assignment y, *over* $Y \subseteq X$ *is the combination of all the functions whose scopes are included in* Y *(*F_Y*) evaluated at the assigned values. Namely,* $c(y) = \bigotimes_{f \in F_Y} f(y[scope(f)])$.

2. *The conditional graphical model give* $Y = y$ *is* $\mathcal{R}|_y = \langle X, D|_y, F|_y, \bigotimes \rangle$, *where* $D|_y = \{D_i \in D, X_i \notin Y\}$ *and* $F|_y = \{f|_{Y=y}, f \in F,$ *and* $scope(f) \not\subseteq Y\}$.

7.2.1 Minimal AND/OR Search Graphs

Next, we characterize the smallest search graph that may result from merging nodes.

By construction, a graphical model $\mathcal{R}$ is equivalent to its AND/OR search tree, $S_{\mathcal{T}}(\mathcal{R})$ if $u(\mathcal{R})$ coincides with the weighted solution substrees of $S_{\mathcal{T}}(\mathcal{R})$, (see Definition 7.1.4). We will next define merge and unify operations that transform AND/OR search trees into graphs that preserve equivalence.

Definition 7.2.3 (merge,unify) *Assume a given weighted AND/OR search graph* $S'_{\mathcal{T}}$ *of a graphical model* $(\mathcal{R})$. *and assume two paths* $\pi_1 = \pi_{n_1}(x_i)$ *and* $\pi_2 = \pi_{n_2}(x_i)$ *ending by AND nodes at level* i *having the same label* x_i.

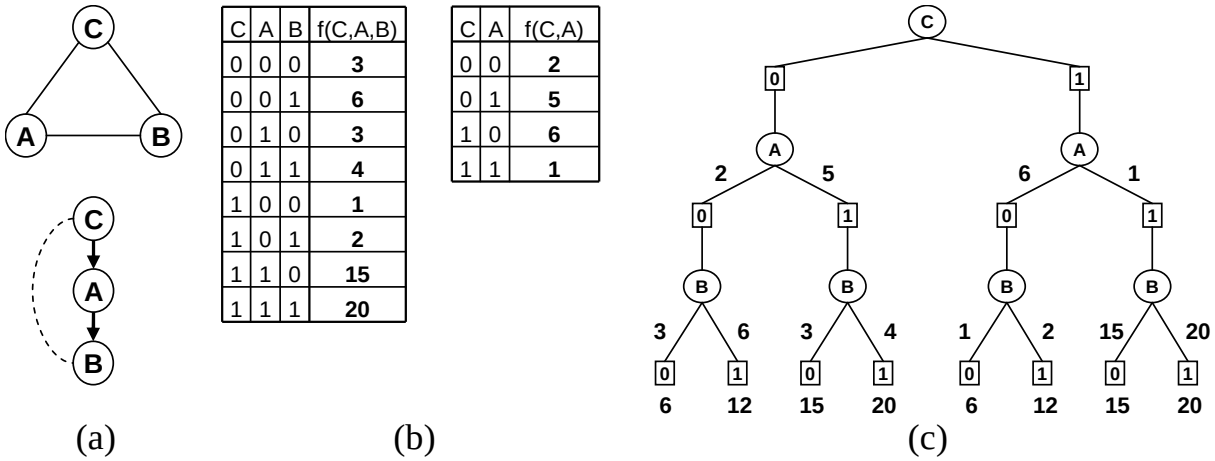


Figure 7.7: Merge vs. unify operators

1. *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. It thus transforms $S'_{\mathcal{T}}$ into a smaller graph. 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.*

2. *Nodes n_1 and n_2 are* unifiable, *iff they root equivalent conditioned subproblems (Definition 2). Namely, if $\mathcal{R}|_{asgn(\pi_1)} = \mathcal{R}|_{asgn(\pi_2)}$.*

Example 7.2.4 Let's follow the example in Figure 7.7 to clarify the difference between *merge* and *unify*. We have a graphical model defined by two functions (*e.g.* cost functions) over three variables. The search tree given in Figure 7.7(c) cannot be reduced to a graph by *merge*, because of the different arc weights. However, the two OR nodes labeled A root equivalent conditioned subproblems (the cost of each individual solution is given at the leaves). Therefore, the nodes labeled A can be *unified*, but they cannot be recognized as identical by the *merge* operator. □

Proposition 7.2.5 (minimal AND/OR graphs) *Given a weighted AND/OR search graph $\mathcal{G}$ based on pseudo tree $\mathcal{T}$:*

1. *The* merge *operator has a unique fix point, called the* ***merge-minimal*** *AND/OR search graph and denoted by $M_{\mathcal{T}}^{merge}(\mathcal{G})$.*

2. *The* unify *operator has a unique fix point, called the* ***unify-minimal*** *AND/OR search graph and denoted by* $M_{\mathcal{T}}^{unify}(\mathcal{G})$.

3. *Any two nodes* n_1 *and* n_2 *of* $\mathcal{G}$ *that can be merged can also be unified.*

Proof: All we need to show for (1) and (2) is that the *merge* and *unify* operators are not depandant on the order of their application (the details are left as an exercise). Claim (3) is clear because if two nodes can be merged they root identical subgraphs and therefore must correspond to a the same conditioned subproblems,thus unifiable. ∎

The unify-minimal AND/OR search graph of $\mathcal{R}$ relative to $\mathcal{T}$ is called the **minimal AND/OR search graph** and denoted by $M_{\mathcal{T}}(\mathcal{R})$. When $\mathcal{T}$ is a chain pseudo tree, the above definitions are applicable to the traditional OR search tree as well. However, note that 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 7.2.6 The smallest OR search graph of the graph-coloring problem in Figure 7.1(a) is given in Figure 7.9 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 7.11. We see that some variable-value pairs (AND nodes) must be repeated in Figure 7.9 while in the AND/OR case 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 7.8-7.11. □

7.2.2 Building AND/OR Search Graphs

The merging rule above seems 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, and is quite impractical.

Some unifiable nodes can be identified based on their *contexts*, however. We can define graph-based contexts for both OR and AND nodes, identifying the set of ancestors variables in $\mathcal{T}$ that completely determine the conditioned subproblems. This leads to merging rules generating a small AND/OR graph called *the context minimal graph* without

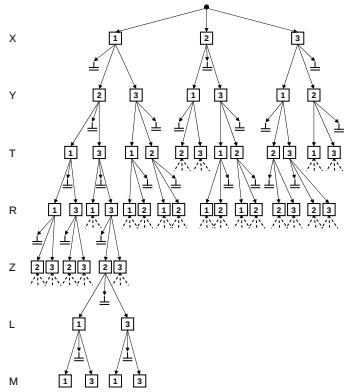


Figure 7.8: OR search tree for the tree problem in Figure 7.1(a)

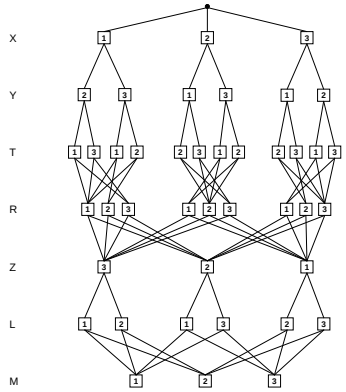


Figure 7.9: The minimal OR search graph of the tree graphical model in Figure 7.1(a)

creating the whole search tree $S_{\mathcal{T}}$ first. It is easiest to demonstrate the notion of context and the rule for unification when the graphical models are trees, and which are called *tree models*.

Example 7.2.7 Consider again the graph in Figure 7.1(a) and its AND/OR search tree in Figure 7.1(c) representing a constraint network. Observe that at level 3, node $\langle Y, 1\rangle$ appears twice, (and so are $\langle Y, 2\rangle$ and $\langle Y, 3\rangle$). Clearly however, the subtrees rooted at each of these two AND nodes are identical and we can reason that they can be merged because any specific assignment to Y uniquely determines its rooted subtree. Indeed, the AND/OR search graph in Figure 7.11 is equivalent to the AND/OR search tree in Figure 7.8 (same as Figure 7.1(c)). □

Indeed, in general, the AND/OR graph of a constraint tree model whose tree is $\mathcal{T}$ can be obtained by merging all AND nodes having the same label $\langle X, x\rangle$. This rule can be extended to any general *weighted* tree graphical models, and the AND/OR graph obtained by this rule is equivalent to $S_{\mathcal{T}}$ and that its size is $O(nk)$. We can now derive a merging rule for non-tree graphical models by generating a tree-decomposition of the graphical model first, and then applying the above merging rule to a tree over meta-variables. Instead, we propose a direct yet equivalent merging rule using the notion of context. The idea is to find a minimal set of variables from the current path to the node such that for

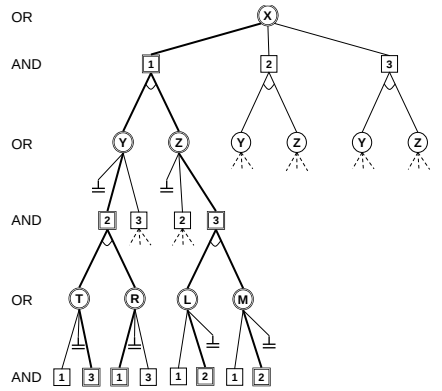


Figure 7.10: AND/OR search tree for the tree problem in Figure 7.1(a)

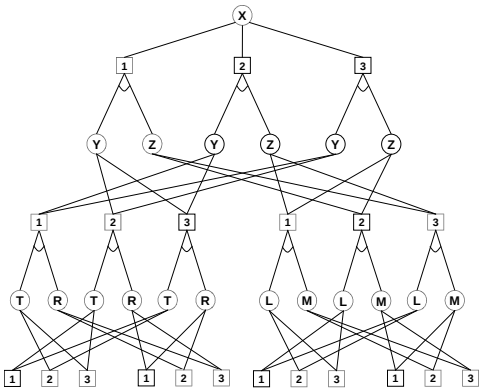


Figure 7.11: The minimal AND/OR search graph of the tree graphical model in Figure 7.1(a)

the same assignment will always generate the same conditioned subproblem, regardless of value assigned to other assignments.

Definition 7.2.8 (OR context) *Given a pseudo tree $\mathcal{T}$ of an AND/OR search space, $context(X) = [X_1 \dots X_p]$ is the set of ancestors of X in $\mathcal{T}$, ordered descendingly, that are connected in the primal graph to X or to descendants of X.*

Theorem 7.2.9 (context based merge) *Let π_{n_1} and π_{n_2} be any two partial paths in an AND/OR search graph guided by $\mathcal{T}$, ending with two nodes, n_1 and n_2.*

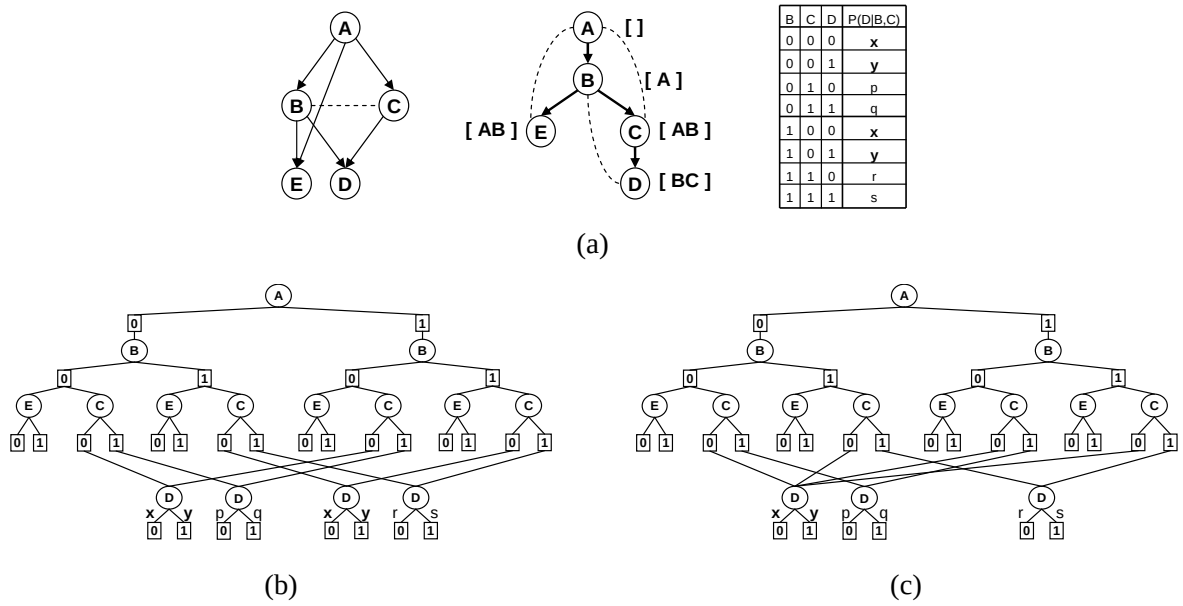
If n_1 and n_2 are OR nodes annotated by X_i and

$$asgn(\pi_{n_1})[context(X_i)] = asgn(\pi_{n_2})[context(X_i)] \tag{7.1}$$

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

Proof: The variables in $context(X_i)$ disconnect the subproblem below X_i from the ancestors of X_i along the pseudo-tree. Therefore the problem below X_i conditioned on $context(X_i)$ is independent of the rest of the ancestor variables, meaning that the context variables completely determines the subproblem below X_i. ∎

From Theorem 7.2.2 it is clear that in an AND/OR search graph, two OR nodes n_1 and n_2 are *context unifiable* if they have the same variable label X and the assignments



B	C	D	P(D|B,C)
0	0	0	x
0	0	1	y
0	1	0	p
0	1	1	q
1	0	0	x
1	0	1	y
1	1	0	r
1	1	1	s

Figure 7.12: Context minimal vs. minimal AND/OR graphs

of their contexts is identical. The *context minimal* AND/OR graph is obtained from the AND/OR search tree by merging all the context unifiable OR nodes.

Definition 7.2.10 (context minimal AND/OR search graph) *The AND/OR search graph of $\mathcal{R}$ based on $\mathcal{T}$ that is closed under context-based merge is called* context minimal *AND/OR search graph and is denoted $C_{\mathcal{T}}(\mathcal{R})$.*

As we have already seen (Example 7.2.4) that there exist nodes that can be *unified* but not *merged*. Here we give an example that shows that contexts can not identify all the nodes that can be *merged*. There could be paths whose contexts are not identical, yet they might root identical subgraphs.

Example 7.2.11 Let's return to the example of the Bayesian network given in Figure 7.12(a), where $P(D|B,C)$ is given in the table, and the OR-context of each node in the pseudo tree is given in square brackets. Figure 7.12(b) shows the context minimal graph. However, we can see that $P(D=0|B=0,C=0) = P(D=0|B=1,C=0) = x$ and $P(D=1|B=0,C=0) = P(D=1|B=1,C=0) = y$. This allows the *unification* of the corresponding OR nodes labeled with D, and Figure 7.12(c) shows the (unify) minimal graph. □

Since the number of nodes in the context minimal AND/OR search graph cannot exceed the number of different contexts, we can therefore bound the size of the context minimal graph.

Theorem 7.2.12 *Given a graphical model $\mathcal{R}$, its primal graph G, and a pseudo tree $\mathcal{T}$ such that the induced width of the graph along a DFS ordering $d_{\mathcal{T}}$ of $\mathcal{T}$ is w, the size of the context minimal AND/OR search graph based on $\mathcal{T}$, $C_{\mathcal{T}}(\mathcal{R})$, is $O(n \cdot k^w)$, when k bounds the domain size.*

Proof: For any variable, the number of contexts is bounded by the number of possible instantiations of the largest context in G along a dfs traversal of the pseudo-tree which is bounded by $O(k^w)$. For all the n variables, the bound $O(n \cdot k^w)$ follows. ∎

Therefore context based merge offers a powerful way of bounding the size of the minimal-context AND/OR search graph, and therefore also the minimal AND/OR search graph.

Theorem 7.2.13 *The context minimal AND/OR search graph $C_{\mathcal{T}}$ of a graphical model having a pseudo tree with bounded treewidth w can be generated in time and space $O(nk^w)$.*

Proof: We can generate $C_{\mathcal{T}}$ using depth-first or breadth first search which caches all nodes via their contexts and avoids generating duplicate searches for the same contexts. Therefore, the generation of the search graph is linear in its size, which is exponential in w and linear in n. ∎

Since the unify minimal AND/OR graph $M_{\mathcal{T}}^{unify}$ and the merge minimal AND/OR graph $M_{\mathcal{T}}^{merge}$ are subsets of $C_{\mathcal{T}}$, both are bounded by $O(n \cdot k^w)$, where w is determined along ordering $d_{\mathcal{T}}$. It is easy to see that $min_{\mathcal{T}}\{w(d_{\mathcal{T}})\} = w*$, the treewidth of the graph, while $min_{\mathcal{T} \in \ chains}\{w(d_{\mathcal{T}}\}) = pw^*$, the pathwidth of G, we get:

Corollary 7.2.14 *Given a graphical model $\mathcal{R}$, there exists apseudo tree $\mathcal{T}$ such that the unify minimal, merge minimal and context minimal AND/OR search graphs of $\mathcal{R}$ are bounded exponentially by the treewidth of the primal graph. The unify, merge and context minimal OR search graphs can be bounded exponentially by the pathwidth only.*

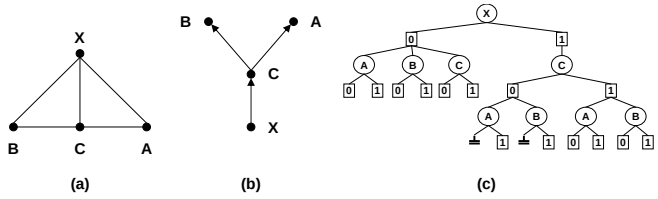


Figure 7.13: (a) A constraint graph; (b) a spanning tree; (c) a dynamic AND/OR tree

It is well known [30] that for any graph, $w^* \leq pw^* \leq w^* \cdot \log n$. It is easy to place m^* (the minimal depth over pseudo trees) in that relation yielding $w^* \leq pw^* \leq m^* \leq w^* \cdot \log n$. It is also possible to show that there exist primal graphs for which the upper bound on pathwidth is attained, that is $pw^* = O(w^* \cdot \log n)$ (show it as an exercise). Therefore, 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 $O(nk^{w \cdot \log n})$.

So, we have seen that AND/OR *trees* are characterized by the *depth* of the pseudo trees while minimal AND/OR *graphs* are characterized by their *induced width*. It turns out however that sometimes a pseudo tree that is optimal relative to w is far from optimal for m and vice versa. For example, a primal graph model that is a chain has a pseudo tree having $m_1 = n$ and $w_1 = 1$ on one hand, and another pseudo tree that is balanced having $m_2 = \log n$ and $w_2 = \log n$. There is no single pseudo tree having both $w = 1$ and $m = \log n$ for a chain. Thus, if we plan to have linear space search we should pick one kind of a pseudo tree, while if we plan to search a graph, and therefore cache some nodes, another pseudo tree should be used.

7.2.3 Using Dynamic 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 also be modified to allow dynamic variable ordering. We will touch very briefly on this issue here even though the ramification of dynamic variable ordering can be substantial (see for example [**?**, **?**]. A dynamic AND/OR tree that allows varied variable ordering has to satisfy that for every subtree rooted by the current path π, any arc of the primal graph that appears as a cross-arc (not a back-arc) in the subtree must be "inactive" conditioned on π.

Example 7.2.15 Consider the propositional formula $X \rightarrow A \vee C$ and $X \rightarrow B \vee C$. The constraint graph is given in Figure 7.13(a) and a DFS tree in 7.13(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 7.13(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 primal 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. In practice, after each new value assignment, the conditional constraint graph can be assessed as follows. For any constraint over the current variable X, if the current assignment $\langle X, x\rangle$ does not make the constraint *active* then the corresponding arcs can be removed from the graph. Then, a pseudo tree of the resulting graph is generated, its first variable is selected, and search continues. A full investigation of dynamic orderings is outside the scope of the current chapter.

7.3 Solving Reasoning Problems by AND/OR Search

7.3.1 Value Functions of Reasoning Problems

As we described earlier, there are a variety of reasoning problems over weighted graphical models. 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 there is a cost function defined we may also seek an optimal solution. The primary tasks over probabilistic networks are computing beliefs, finding the probability of the evidence and finding the most likely tuple given the evidence. Each of these reasoning problems can be expressed as finding the *value* of some nodes in the weighted AND/OR search space where different tasks call for different value definitions.

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 answers the consistency query. For solutions-counting the value function of each node is the number of solutions rooted at that node. It is easy to see that the value of nodes in the search graph

can then be computed recursively from leaves to root, to decide the relevant constraint query.

Proposition 7.3.1 (Recursive value computation for constraint queries) *1. For the consistency task the value of AND leaves is their labels 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 AND node has value "1" iff all its successor 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.

We now move to probabilistic queries. Remember that the label of an arc $(X_i, \langle X_i, x_i\rangle)$ along path $\pi(x_i)$, denoted as $l((X_i, \langle X_i, x_i\rangle)) = \prod_{f\in B(X_i)} f(\pi(x_i)_{scop(f)})$. Given a labeled AND/OR tree $S_{\mathcal{T}}(\mathcal{R})$ of a belief network, For the probability of evidence query, the value of a node (AND or OR) is the probability of evidence restricted to its subtree and to the assigned variables along the path to the node. For the MPE task, the value of each node is the probability of the most likely extension of its path given the evidence variables.

Proposition 7.3.2 (Recursive value computation for probabilistic queries) *1. For the probability of the evidence, the value of AND leaf nodes are "1" and the value of leaf OR nodes are "0". The value of an internal OR node is the weighted sum values of its child AND-nodes, each multiplied by the arc-label. The value of an internal AND node is the product of child nodes' values.*

2. For the mpe, the value of an internal OR node is the maximum among the values of its child nodes's each multiplied by the label of its OR-AND arc. The value of an AND node is the product of child values. G

Proof: Prove as an exercise ∎

We can now generalize to any graphical model and queries.

Definition 7.3.3 (Recursive value computation for general reasoning problems) *The value function of a reasoning problem $\mathcal{P} = \langle \mathcal{R}, \Downarrow_Y, Z\rangle$, where $\mathcal{R} = \langle X, D, F, \bigotimes\rangle$ and*

$Z = \emptyset$, 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 7.1.8) 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:*

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

Given a reasoning task, computing the value of the root node solves the given reasoning problem (we omit the formal proof which can be found in [25]). Search algorithms that traverse the AND/OR search space can compute the value of the root node yielding the answer to the problem. Algorithms that traverse the weighted AND/OR search tree in a depth-first manner or a breadth-first manner are guaranteed to have time bound exponential in the depth of the pseudo tree of the graphical model. Depth-first searches can be accomplished using either linear space only, or context based caching, bounded exponentially by the treewidth of the pseudo tree. Depth-first search is an anytime schemes and can, if terminated, provide an approximate solution for some tasks such as optimization.

The next subsection presents typical depth-first algorithms that search 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 reasoning query. For application of these ideas for combinatorial optimization tasks, such as MPE see [39].

7.3.2 Algorithm AND/OR Tree Search and Graph Search

Algorithm 1 presents the basic depth-first traversal of the AND/OR search tree (or graph, if caching is used) for counting the number of solutions of a constraint network, AO-COUNTING (or for probability of evidence for belief networks, AO-BELIEF-UPDATING). The context based caching is done based on 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 π_n. The children of the current node are denoted by *successors*(`n`).

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*(`n`) is set to the empty set, which will trigger the PROPAGATE step. If a node is not 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.

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, yielding the 0/1 labeling. 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. To find posterior beliefs of the root variable, we only need to keep the computation at the root of the search graph and normalize the results. If we want to find the belief for all variables we would need to make a more significant adaptation of the search scheme.

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.

From Theorems 7.1.15 and 7.1.18 we can conclude that:

Theorem 7.3.4 *For any reasoning problem,* AOT *runs in linear space and time* $O(nk^m)$*, when* m *is the depth of the pseudo tree of its graphical model and* k *is the maximum domain size. If the primal graph has a tree decomposition with treewidth* w^**, there exists a pseudo tree* $\mathcal{T}$ *for which AOT is* $O(nk^{w^* \cdot \log n})$*.*

Obviously, the algorithm for constraint satisfaction, that would terminate early with first solution, would potentially be much faster than the rest of the AOT algorithms, in practice. Based on Theorem 7.2.12 we get complexity bounds for graph searching algorithms.

Theorem 7.3.5 *For any reasoning problem, the complexity of algorithm* AOG *is time and space* $O(nk^w)$ *where* w *is the induced width of the pseudo tree and* k *is the maximum domain size.*

Thus the complexity of AOG can be time and space exponential in the treewidth, while the complexity of any algorithm searching the OR space can be time and space exponential in its pathwidth.

Space complexity and AO(i)

The space complexity of AOG can often be less than exponential in the treewidth. This is related to the space complexity of tree decomposition schemes which, as we know, can operate in space exponential only in the size of the cluster separators, rather than exponential in the cluster size.

We will use the term *dead caches* to address this issue presented in [12, 2]. Intuitively, a node that has only one incoming arc in the search tree will be traversed only once by search, and therefore its value does not need to be cached, because it will never be used again. For context based caching, such nodes can be recognized based only on their context.

Definition 7.3.6 (dead cache) *If X is the parent of Y in $\mathcal{T}$, and $context(X) \subset context(Y)$, then $context(Y)$ is a* dead cache.

Given a pseudo tree $\mathcal{T}$, the induced graph along $\mathcal{T}$ can generate a tree decomposition based on the maximal cliques. The maximum separator size of the tree decomposition is the separator size of $\mathcal{T}$. We can conclude,

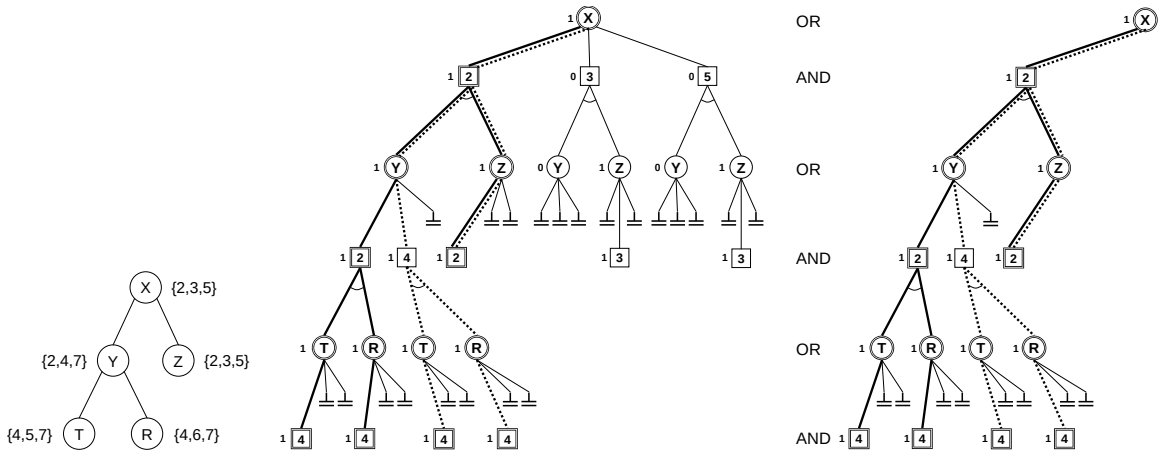
Proposition 7.3.7 *The space complexity of graph-caching algorithms can be reduced to being exponential in the separator's size only, while still being time exponential in the treewidth, if dead caches are not recorded.*

Proof: A bucket tree can be built by having a cluster for each variable X_i and its parents pa_i, and following the structure of the pseudo tree $\mathcal{T}$. Some of the clusters may not be maximal, and they have a one to one correspondence to the variables with dead caches. The parents pa_i that are not dead caches correspond to separators between maximal clusters in the bucket tree. ∎

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 backbone 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. The more precise evaluation of the time and space complexity of the algorithm can be derived. For more details see [24].

7.4 AND/OR Search Algorithms For Mixed Networks

All the advanced constraint processing algorithms, either incorporating no-good learning, and constraint propagation during search, or use variable elimination algorithms such as



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

Figure 7.14: AND/OR search tree and backtrack-free tree

adaptive-consistency and *directional resolution* generating all relevant no-goods prior to search can be incorporated over the AND/OR search space as well. We will address these issues in the rest of this section. We next define formally the *backtrack-free* AND/OR search tree.

Definition 7.4.1 (backtrack-free AND/OR search tree) *Given graphical model $\mathcal{R}$ and given an AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$, the* backtrack-free AND/OR search tree *of $\mathcal{R}$ based on $\mathcal{T}$, denoted $BF_{\mathcal{T}}(\mathcal{R})$, is obtained by pruning from $S_{\mathcal{T}}(\mathcal{R})$ all inconsistent subtrees, namely all nodes that root no consistent partial solution.*

Example 7.4.2 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 DFS tree rooted at X, are given in Figure 7.14(a). In 7.14(b) we present the $S_{\mathcal{T}}(\mathcal{R})$ search space whose nodes' consistency status (which will latter will be referred to as *values*) are already evaluated having value "1" is consistent and "0" otherwise. We also highlight two solutions subtrees; one depicted by solid lines and one by dotted lines. Part (c) presents $BF_{\mathcal{T}}(\mathcal{R})$, 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 space. Others can be made backtrack-free by massaging their representation using *constraint propagation* algorithms before or during search. In particular, variable-elimination algorithms described earlier in Chapter 3, such as *adaptive-consistency* and directional resolution, applied in a reversed order of d (where d is the DFS order of the pseudo tree), compile a constraint specification (resp., a Boolean CNF formula) that has a backtrack-free search space. We define now the directional extension formally (see also chapter 3).

Definition 7.4.3 (directional extension [20, 53]) *Let $\mathcal{R}$ be a constraint problem and let d be a DFS traversal ordering of a backbone pseudo tree of its primal graph, then we denote by $E_d(\mathcal{R})$ the constraint network (resp., the CNF formula) compiled by Adaptive-consistency (resp., directional resolution) in reversed order of d.*

We can summarize the above discussion by the following proposition.

Proposition 7.4.4 *Given a Constraint network $\mathcal{R}$, the AND/OR search tree of the directional extension $E_d(\mathcal{R})$ when d is a DFS ordering of $\mathcal{T}$, is identical to the backtrack-free AND/OR search tree of $\mathcal{R}$ based on $\mathcal{T}$. Namely $S_{\mathcal{T}}(E_d(\mathcal{R})) = BF_{\mathcal{T}}(\mathcal{R})$.*

Proof: 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 $\mathcal{R}$, however this 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 [20], a notion that is synonym with backtrack-freeness. ∎

Example 7.4.5 In Example 7.4.2, if we apply adaptive-consistency in reverse order of X, Y, T, R, Z, the algorithm will remove the values $3, 5$ from the domains of both X and Z yielding a tighter constraint network $\mathcal{R}'$. The AND/OR search tree in Figure 7.4.2(c) is both $S_{\mathcal{T}}(\mathcal{R}')$ and $BF_{\mathcal{T}}(\mathcal{R})$. □

Proposition 7.4.4 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, [32] and are currently used in most advanced solvers [40]. Namely, when we apply no-good learning we explore the search space whose many inconsistent subtrees are pruned.

For more involved queries like weighted-counting or sum-product or for optimization (e.g., max-product) , pruning inconsistent subtrees and searching the backtrack-free search tree yields a partial help only, as we see in Section 7.4.

Since search algorithms accommodate a host of constraint processing techniques we will now give more details on applying constraint techniques while searching and AND/OR search for processing queries over mixed networks. Remember that mixed networks are defined by a pair of a belief network and a constraint networks that together are equivalent to a Bayesian network that is conditioned on consistent solutions defined by the constraint network. The mixed network can be transformed into an equivalent representation by tightening the constraint network only. Therefore we can process the deterministic information separately (e.g., by enforcing some consistency level without losing any solution. Conditioning algorithms (search) offer a natural approach for exploiting the determinism through constraint propagation techniques. The intuitive idea is to search in the space of partial variable assignments, and use the wide range of readily available constraint processing techniques to limit the actual traversed space. We now describe these known basic principles in the context of AND/OR search spaces [19].

We will focus on the constraint probabilistic evaluation quer (CPE) described in Chapter **??** and describe the AND-OR-CPE Algorithm. Then, we will discuss how to incorporate in AND-OR-CPE the techniques that exploit the deterministic information: (1) constraint propagation (look-ahead), (2) backjumping and (3) good and nogood learning.

7.4.1 AND-OR-cpe Algorithm

Algorithm 2, AND-OR-CPE, presents the basic depth-first traversal of the AND/OR search tree (or graph, if caching is used) for solving the CPE task over a mixed network and it is indeed quite similar to Algorithm xxx. The input is a mixed network, a pseudo tree $\mathcal{T}$ of the moral mixed graph and the context of each variable. The output is the result of the CPE task, namely the probability that a random tuple generated from the belief network distribution satisfies the constraint query. As usual, AND-OR-CPE traverses

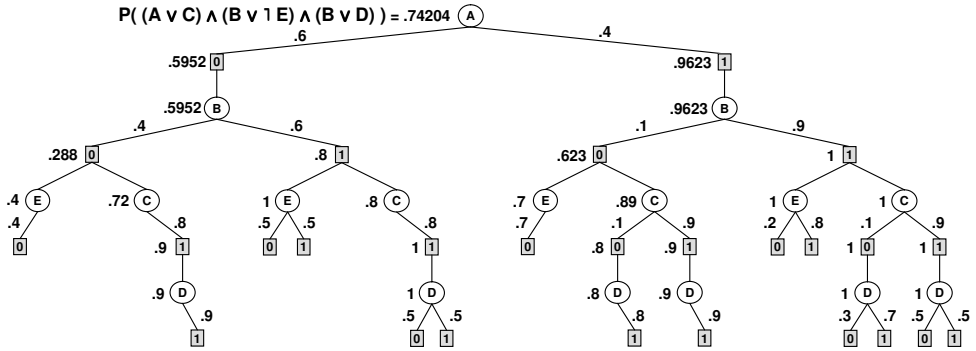


Figure 7.15: Mixed network defined by the query $\varphi = (A \vee C) \wedge (B \vee \neg E) \wedge (B \vee D)$

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 resulted from its subtree. 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 7.4.6 We refer back to the example in Figure 7.4. 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 7.15. 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. Figure 7.15 is similar to Figure 7.4. In Figure 7.4 the evidence can be modeled as the CNF formula with unit clauses $D \wedge \neg E$. □

It is clear that the algorithm inherits all the mentioned properties of AND/OR search.

7.4.2 Constraint Propagation in AND-OR-cpe

The virtue on 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 [17].

We next discuss here the use of constraint propagation during the look-ahead part of the search. This methods are used in any constraint or SAT solver. In general, constraint

propagation helps to discover (using limited computation) what variable and what value to instantiate next aiming to avoid deadends. The incorporation of these methods on top of AND/OR search for value computation is straightforward. For illustration, we will only consider static variable ordering, based on a pseudo tree, and therefore we will focus on the impact of constraint propagation on value selection.

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 the search. Using a polynomial time algorithm, `ConstraintPropagation` may discover some values that cannot be extended to a full solution. These values are marked as dead-ends and removed from the current domain of the variable. All the remaining values are returned by the procedure as good candidates to extend the search frontier. Of course, not all the values returned by `ConstraintPropagation` 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, and we denote this algorithm by AO-C.

For illustration consider two forms of constraint propagation on top of AO-C. The first, yielding algorithm AO-FC, is based on *forward checking*, which is one of the weakest forms of propagation. It propagates the effect of a value selection 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 current path. If the projection is empty an inconsistency is detected. AO-RFC is computationally more expensive than AO-FC, but its search space is smaller.

SAT solvers. One possibility that was explored with success (e.g., [2]) 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). The worst-case complexity of this level of constraint processing, at each node, is exponential. One very commonly used technique is *unit propagation*, or *unit resolution*, as a form of bounded resolution [53].

Such 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.

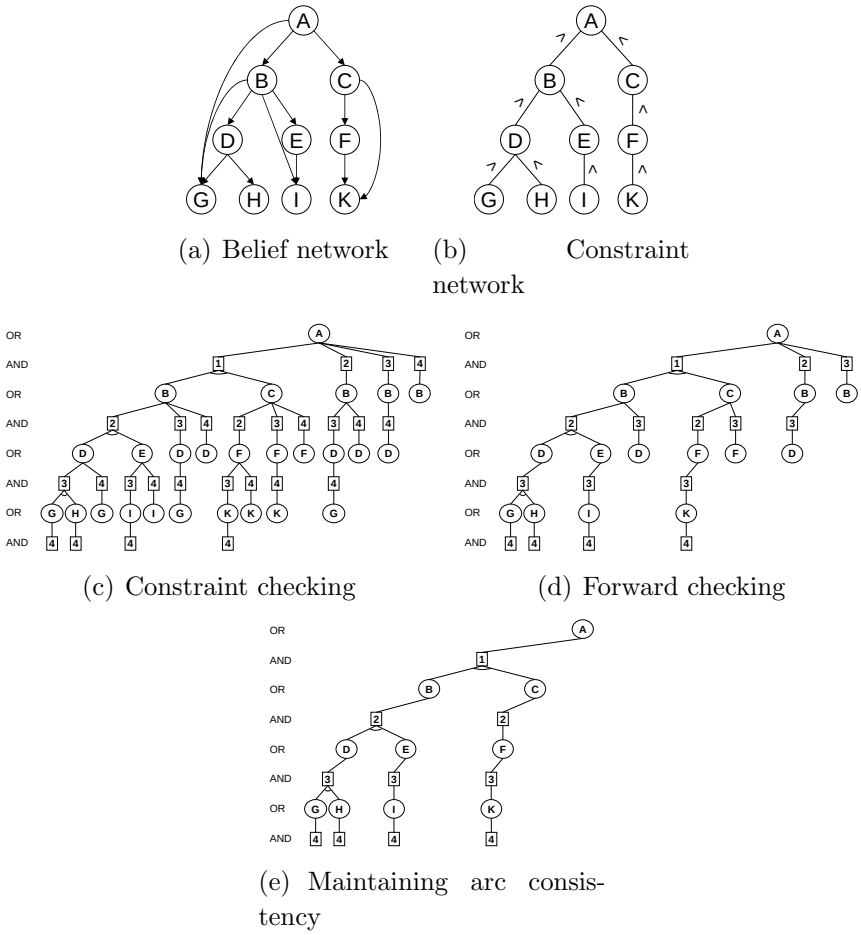
Example 7.4.7 Figure 7.16(a) shows the belief part of a mixed network, and Figure 7.16(b) the constraint part. All variables have the same domain, {1,2,3,4}, and the constraints express "less than" relations. Figure 7.16(c) shows the search space of AO-C. Figure 7.16(d) shows the space traversed by AO-FC. Figure 7.16(e) shows the space when consistency is enforced with Maintaining Arc Consistency (which enforces full arc-consistency after each new instantiation of a variable). □

7.4.3 Backjumping

Backjumping algorithms [29, 48, 5, 17] are backtracking search algorithms applied to the OR space, which uses the problem structure to jump back from a dead-end as far back as possible. They have been known for a long time in the constraint processing community. For probabilistic models, backjumping is very useful in the context of determinism.

In *graph-based backjumping* (GBJ) each variable maintains a graph-based induced ancestor set which ensures that no solutions are missed by jumping back to its deepest variable.

If the ordering of the OR space is a DFS ordering of the primal graph, it is known [17] that all the backjumps are from a variable to its DFS parent. In [41] it was shown that this means that a simple AND/OR search automatically incorporates graph-based backjumping, when the pseudo tree is a DFS tree of the primal graph.



(a) Belief network (b) Constraint network

(c) Constraint checking (d) Forward checking

(e) Maintaining arc consistency

Figure 7.16: Traces of AND-OR-CPE with various levels of constraint propagation

When the pseudo tree is not a DFS tree of the primal graph, it may happen that the parent of a node in the pseudo tree is not the node where graph-based backjumping would retreat in the case of OR search. An example is provided in Figure 7.17. Figure 7.17a shows a graphical model, 7.17b a pseudo tree and 7.17c a chain driving the OR search (top down). If a deadend is encountered at variable 3, graph-based backjumping retreats to 8 (see 7.17c), while simple AND/OR would retreat to 1, the pseudo tree parent. When the deadend is encountered at 2, both algorithms backtrack to 3 and then to 1. Therefore, in such cases, augmenting AND/OR with a graph-based backjumping mechanism can provide some improvement.

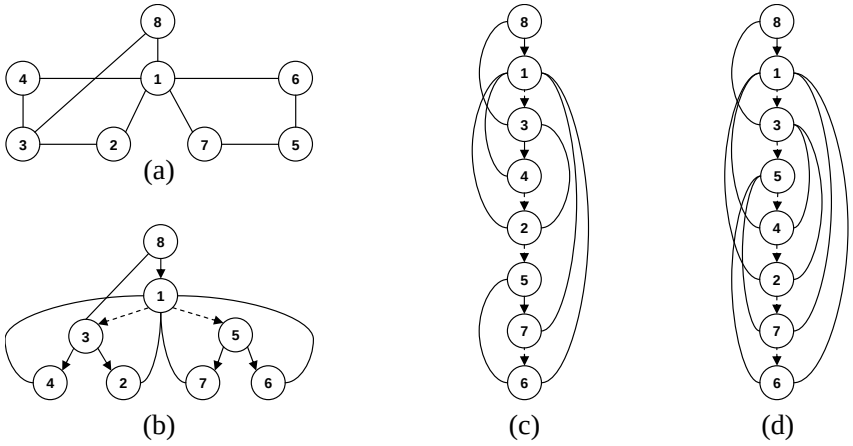


Figure 7.17: Graph-based backjumping and AND/OR search

We want to emphasize that the graph-based backjumping behavior is in most cases intrinsic to AND/OR search. The more advanced and computationally intensive forms of conflict directed backjumping [17] are not captured by the AND/OR graph, and can be implemented on top of it as in the case of OR search by focusing on and analyzing the constraint portion only.

7.4.4 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, 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 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 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 the same technique as graph-based backjumping to identify the ancestor variables that generate the nogood. The information on conflicts is generated from the primal graph information alone. Similar to the case of backjumping, it is easy to see that AND/OR search already provides this information in the context of the nodes. Therefore, if caching is used, just saving the

information about the nogoods encountered amounts to graph-based nogood learning in the case of OR search.

If deeper types of nogood learning are desirable, they need to be implemented 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, deep learning is computationally more expensive and it can be facilitated via a focus on the constraint portion of the mixed network.

In recent years [12, 63, 19], several schemes propose not only the learning of nogoods, but also that of their logical counterparts, the *goods*. This is in fact the well known technique of caching, or memoization, and in recent years it became appealing due to the availability of computer memory and when the task to be solved requires the enumeration of many solutions. The idea is to store the value of a solved conditioned subproblem, associating it with a minimal set of ancestor assignments that are guaranteed to root the same conditioned subproblem, and retrieve that value whenever the same set of ancestor assignments is encountered again during search. This is exactly what AND/OR context-based caching does.

Overall traversing the context minimal AND/OR graph provides both good and nogood graph-based learning.

7.5 Chapter Notes

7.6 Conclusion to Chapter 7

The primary contribution of this chapter is in viewing search for graphical models in the context of AND/OR search spaces rather than OR spaces. We 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 savings for any linear space algorithms traversing the AND/OR search tree. Specifically, if the graphical model has treewidth w^*, the depth of the pseudo tree is $O(w^* \cdot \log n)$.

The AND/OR search tree was extended 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 (*e.g.*, balanced pseudo trees) 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 goods 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. For finding a single solution, pruning the search space is the most significant action. For counting and probabilistic inference, using AND/OR graphs can be of much help even on top of no-good recording.

Algorithm 1: AO-COUNTING / AO-BELIEF-UPDATING

A constraint network $\mathcal{R} = \langle X, D, C \rangle$, or a belief network $\mathcal{P} = \langle X, D, P \rangle$; a pseudo tree $\mathcal{T}$ rooted at X_1; parents pa_i (OR-context) for every variable X_i; `caching` set to *true* or $false$. 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` $\neq \varphi$ **do**
4. `n` $\leftarrow$ *top*(`OPEN`); remove `n` from `OPEN`
5. **if** `caching` $==$ *true* **and** `n` *is OR, labeled* X_i **and** $Cache(asgn(\pi_n)[pa_i]) \neq -1$ **then** `// In cache`
6. $v($`n`$) \leftarrow Cache(asgn(\pi_n)[pa_i])$ `// Retrieve value`
7. $successors($`n`$) \leftarrow \varphi$ `// No need to expand below`
8. **else** `// EXPAND`
9. **if** `n` *is an OR node labeled* X_i **then** `// OR-expand`
10. $successors($`n`$) \leftarrow \{\langle X_i, x_i \rangle \mid \langle X_i, x_i \rangle$ is consistent with $\pi_n \}$
11. $v(\langle X_i, x_i \rangle) \leftarrow 1$, for all $\langle X_i, x_i \rangle \in successors($`n`$)$
12. $v(\langle X_i, x_i \rangle) \leftarrow \prod_{f \in B_{\mathcal{T}}(X_i)} f(asgn(\pi_n)[pa_i])$, for all $\langle X_i, x_i \rangle \in successors($`n`$)$ `// AO-BU`
13. **if** `n` *is an AND node labeled* $\langle X_i, x_i \rangle$ **then** `// AND-expand`
14. $successors($`n`$) \leftarrow children_{\mathcal{T}}(X_i)$
15. $v(X_i) \leftarrow 0$ for all $X_i \in successors($`n`$)$
16. Add $successors($`n`$)$ to top of `OPEN`
17. **while** $successors($`n`$) == \varphi$ **do** `// PROPAGATE`
18. **if** `n` *is an OR node labeled* X_i **then**
19. **if** $X_i == X_1$ **then** `// Search is complete`
20. **return** $v($`n`$)$
21. **if** `caching` $==$ *true* **then**
22. $Cache(asgn(\pi_n)[pa_i]) \leftarrow v($`n`$)$ `// Save in cache`
23. $v($`p`$) \leftarrow v($`p`$) * v($`c`$)$
24. **if** $v($`p`$) == 0$ **then** `// Check if p is dead-end`
25. remove $successors($`p`$)$ from `OPEN`
26. $successors($`p`$) \leftarrow \varphi$
27. **if** `n` *is an AND node labeled* $\langle X_i, x_i \rangle$ **then**
28. let `p` be the parent of `n`
29. $v($`p`$) \leftarrow v($`p`$) + v($`n`$)$;
30. remove `n` from $successors($`p`$)$
31. `n` $\leftarrow$ `p`

Algorithm 2: AND-OR-CPE

A mixed network $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{G}, \mathbf{P}, \mathbf{C} \rangle$; 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 $false$. 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` $\neq \varphi$ **do**

4 `n` $\leftarrow$ *top*(`OPEN`); remove `n` from `OPEN`

5 **if** `caching` == *true* **and** `n` *is OR, labeled* X_i **and** $Cache(asgn(\pi_n)[pa_i]) \neq -1$ **then** // If in cache

6 $v(\mathtt{n}) \leftarrow Cache(asgn(\pi_n)[pa_i])$ // Retrieve value

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

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

9 **if** `n` *is an OR node labeled* X_i **then** // OR-expand

10 $successors(\mathtt{n}) \leftarrow$ `ConstraintPropagation`$(\langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle, asgn(\pi_n))$

// CONSTRAINT PROPAGATION

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

12 **if** `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** $successors(\mathtt{n}) == \varphi$ **do** // **Update values (backtrack)**

17 **if** `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 $Cache(asgn(\pi_n)[pa_i]) \leftarrow v(\mathtt{n})$ // Save in cache

22 let `p` be the parent of `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 $successors(\mathtt{p}) \leftarrow \varphi$

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

28 let `p` be the parent of `n`

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

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

31 `n` $\leftarrow$ `p`

Procedure `ConstraintPropagation`($\mathcal{R}$, $\bar{x}_i$)

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

return *reduced domain of* X_i

Bibliography

[1] Darwiche A. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[2] D. Allen and A. Darwiche. New advances in inference by recursive conditioning. In *Proceedings of the 19th Conference on uncertainty in Artificial Intelligence (UAI03)*, pages 2–10, 2003.

[3] S. Arnborg and A. Proskourowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete and Applied Mathematics*, 23:11–24, 1989.

[4] S. A. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. *BIT*, 25:2–23, 1985.

[5] R. Bayardo and D. Miranker. 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*, pages 298–304, 1996.

[6] A. Becker and D. Geiger. A sufficiently fast algorithm for finding close to optimal junction trees. In *Uncertainty in AI (UAI'96)*, pages 81–89, 1996.

[7] E. Bensana, M. Lemaitre, and G. Verfaillie. Earth observation satellite management. *Constraints*, 4(3):293–299, 1999.

[8] U. Bertele and F. Brioschi. *Nonserial Dynamic Programming*. Academic Press, 1972.

[9] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the Association of Computing Machinery*, 44, No. 2:165–201, 1997.

[10] C. Cannings, E.A. Thompson, and H.H. Skolnick. Probability functions on complex pedigrees. *Advances in Applied Probability*, 10:26–61, 1978.

[11] R. McEliece D. C. MacKay and J. Cheng. Turbo decoding as an instance of pearl's "belief propagation" algorithm. 1996.

[12] A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 125(1-2):5–41, 2001.

[13] S. de Givry, J. Larrosa, and T. Schiex. Solving max-sat as weighted csp. *In Principles and Practice of Constraint Programming (CP-2003)*, 2003.

[14] S. de Givry, I. Palhiere, Z. Vitezica, and T. Schiex. Mendelian error detection in complex pedigree using weighted constraint satisfaction techniques. In *ICLP Workshop on Constraint Based Methods for Bioinformatics*, 2005.

[15] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[16] R. Dechter. Mini-buckets: A general scheme of generating approximations in automated reasoning. In *IJCAI-97: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1297–1302, 1997.

[17] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[18] R. Dechter and D. Larkin. Hybrid processing of belief and constraints. *Proceeding of Uncertainty in Artificial Intelligence (UAI01)*, pages 112–119, 2001.

[19] R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171(2-3):73–106, 2007.

[20] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.

[21] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, pages 353–366, 1989.

[22] R. Dechter and P. van Beek. Local and global relational consistency. In *Principles and Practice of Constraint programming (CP-95)*, pages 240–257, 1995.

[23] R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, pages 283–308, 1997.

[24] Rina Dechter. Tractable structures for constraint satisfaction problems. In *Handbook of Constraint Programming, part I, chapter 7*, pages 209–244. Elsevier, 2006.

[25] Rina Dechter and Robert Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171(2-3):73–106, 2007.

[26] E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[27] E. C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(1):755–761, 1985.

[28] M. R Garey and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness. In *W. H. Freeman and Company, San Francisco*, 1979.

[29] J. Gaschnig. Performance measurement and analysis of search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University, 1979.

[30] H. Hasfsteinsson H.L. Bodlaender, J. R. Gilbert and T. Kloks. Approximating treewidth, pathwidth and minimum elimination tree-height. In *Technical report RUU-CS-91-1, Utrecht University*, 1991.

[31] F.V. Jensen. *Bayesian networks and decision graphs*. Springer-Verlag, New-York, 2001.

[32] R. J. Bayardo Jr. and R. C. Schrag. Using csp look-back techniques to solve real world sat instances. In *14th National Conf. on Artificial Intelligence (AAAI97)*, pages 203–208, 1997.

[33] U. Kjæaerulff. Triangulation of graph-based algorithms giving small total state space. In *Technical Report 90-09, Department of Mathematics and computer Science, University of Aalborg, Denmark*, 1990.

[34] U. Kjæaerulff. A computational scheme for reasoning in dynamic probabilistic networks. In *Uncertainty in Artificial Intelligence (UAI'93)*, pages 121–149, 1993.

[35] D. Koller and N. Friedman. *Probabilistic Graphical Models*. MIT Press, 2009.

[36] F. R. Kschischang and B.H. Frey. Iterative decoding of compound codes by probability propagation in graphical models. *submitted*, 1996.

[37] S.L. Lauritzen and D.J. Spiegelhalter. Local computation with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society, Series B*, 50(2):157–224, 1988.

[38] D. Maier. The theory of relational databases. In *Computer Science Press, Rockville, MD*, 1983.

[39] R. Marinescu and R. Dechter. AND/OR branch-and-bound for graphical models. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 224–229, 2005.

[40] J. P. Marques-Silva and K. A. Sakalla. Grasp-a search algorithm for propositional satisfiability. *IEEE Transaction on Computers*, pages 506–521, 1999.

[41] R. Mateescu and R. Dechter. The relationship between AND/OR search and variable elimination. In *Proceedings of the Twenty First Conference on Uncertainty in Artificial Intelligence (UAI'05)*, pages 380–387, 2005.

[42] R.J. McEliece, D.J.C. MacKay, and J.-F.Cheng. Turbo decoding as an instance of Pearl's belief propagation algorithm. *To appear in IEEE J. Selected Areas in Communication*, 1997.

[43] L. G. Mitten. Composition principles for the synthesis of optimal multistage processes. *Operations Research*, 12:610–619, 1964.

[44] R.E. Neapolitan. *Learning Bayesian Networks*. Prentice hall series in Artificial Intelligence, 2000.

[45] N. J. Nillson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, CA, 1980.

[46] Jurg Ott. *Analysis of Human Genetics*. Cambridge University Press, 1999.

[47] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

[48] P. Prosser. Hybrid algorithms for constraint satisfaction problems. *Computational Intelligence*, 9(3):268–299, 1993.

[49] Jr. R. Bayardo and D. P. Miranker. On the space-time trade-off in solving constraint satisfaction problems. In *Fourteenth International Joint Conference on Artificial Intelligence(I95)*, pages 558–562, 1995.

[50] A. Dechter R. Dechter and J. Pearl. Optimization in constraint networks. In *Influence Diagrams, Belief Nets and Decision Analysis*, pages 411–425. John Wiley & Sons, 1990.

[51] B. D'Ambrosio R.D. Shachter and B.A. Del Favero. Symbolic probabilistic inference in belief networks. In *National Conference on Artificial Intelligence (AAAI'90)*, pages 126–131, 1990.

[52] R.G.Gallager. A simple derivation of the coding theorem and some applications. *IEEE Trans. Information Theory*, IT-11:3–18, 1965.

[53] I. Rish and R. Dechter. Resolution vs. search; two strategies for sat. *Journal of Automated Reasoning*, 24(1/2):225–275, 2000.

[54] T. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. *Proc. IJCAI-99*, pages 542–547, 1999.

[55] L. K. Saul and M. I. Jordan. Learning in boltzmann trees. *Neural Computation*, 6:1173–1183, 1994.

[56] C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423,623–656, 1948.

[57] P.P. Shenoy. Valuation-based systems for bayesian decision analysis. *Operations Research*, 40:463–484, 1992.

[58] K. Shoiket and D. Geiger. A proctical algorithm for finding optimal triangulations. In *Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 185–190, 1997.

[59] C.E. Leiserson T. H. Cormen and R.L. Rivest. In *Introduction to algorithms*. The MIT Press, 1990.

[60] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs. *SIAM Journal of Computation.*, 13(3):566–579, 1984.

[61] J.A. Tatman and R.D. Shachter. Dynamic programming and influence diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, pages 365–379, 1990.

[62] P. Thbault, S. de Givry, T. Schiex, and C. Gaspin. Combining constraint processing and pattern matching to describe and locate structured motifs in genomic sequences. In *Fifth IJCAI-05 Workshop on Modelling and Solving Problems with Constraints*, 2005.

[63] P. Beam H. Kautz Tian Sang, F. Bacchus and T. Piassi. Cobining component caching and clause learning for effective model counting. In *SAT 2004*, 2004.

[64] N.L. Zhang and D. Poole. Exploiting causal independence in bayesian network inference. *Journal of Artificial Intelligence Research (JAIR)*, 1996.