Chapter 7

Combining Search and Inference; Trading space for time

As we noted at the introduction, search and inference have complementary properties. Inference exploit the graph structure and therefore allows structure-based time guarantees but require substantial memory. Brute-force Search, does not posses good complexity time bounds but as AND/OR search does, as we will show in the forthcoming last two cahpters. The main virtue of search is that it can operate in linear space. Therefore, using a hybrid of search and inference allows a structure-driven tradeoff of space and time. Two such hybrids are presented next. We demonstrate the principles of the hybrids in the context of tree-search. However, all these ideas can be extended later when the cutset is traversed as an AND/OR search tree or a graph, as we will discuss.

7.1 The cycle-cutset and w-cutset schemes

The algorithms presented in this section exploit the fact that variable instantiation changes the effective connectivity of the primal graph. Consider a constraint problem whose graph is given in Figure 7.1a. For this problem, instantiating X_2 to some value, say a, renders the choices of values to X_1 and X_5 independent, as if the pathway $X_1 - X_2 - X_5$ were blocked at X_2. Similarly, this instantiation blocks dependency in the pathway $X_1 - X_2 - X_4$, leaving only one path between any two variables. In other words, given that X_2 was

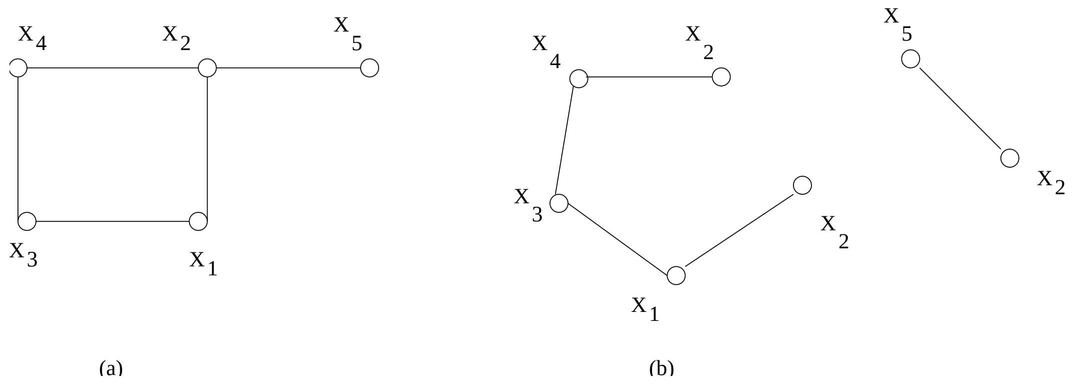


Figure 7.1: An instantiated variable cuts its own cycles.

assigned a specific value, the "effective" constraint graph for the rest of the variables is shown in Figure 7.1b. Here, the instantiated variable X_2 and its incident arcs are first deleted from the graph, and X_2 subsequently is duplicated for each of its neighbors. The constraint problem having the graph shown in Figure 7.1(a) when $X_2 = a$ is identical to the constraint problem having the graph in Figure 7.1(b) with the same assignment $X_2 = a$.

In general, when the group of instantiated variables constitutes a cycle-cutset; a set of nodes that, once removed, would render the constraint graph cycle-free. The remaining network is a tree (as shown in Figure 7.1b), and can be solved by *tree-solving* algorithm like belief propagation, or its constraint version or arc-consistency. In most practical cases it would take more than a single variable to cut all the cycles in the graph. Thus, a general way of solving a problem whose constraint graph contains cycles is to identify a subset of variables that cut all cycles in the graph, find a consistent instantiation of the variables in the cycle-cutset, and then solve the remaining problem by the *tree algorithm.* If a solution to this restricted problem (conditioned on the cycle-cutset values) is found, then a solution to the entire problem is at hand. If not, another instantiation of the cycle-cutset variables should be considered until a solution is found. If the task is to solve a constraint problem whose constraint graph is presented in Figure 7.1a, (assume X_2 has two values $\{a, b\}$ in its domain), first $X_2 = a$ must be assumed, and the remaining tree problem relative to this instantiation, is solved. If no solution is found, it is assumed that

$X_2 = b$ and another attempt is made.

The number of times the tree-solving algorithm needs to be invoked is bounded by the number of partial solutions to the cycle-cutset variables. A small cycle-cutset is therefore desirable. However, since finding a minimal-size cycle-cutset is computationally hard, it will be more practical to settle for heuristic compromises. One approach is to incorporate this scheme within depth-first search, which is called backtracking search in the context of constraint satisfaction problems. Because *backtracking* works by progressively instantiating sets of variables, we only need to keep track of the connectivity status of the constraint graph. As soon as the set of instantiated variables constitutes a cycle-cutset, the search algorithm is switched to the tree-solving algorithm on the restricted problem, i.e., either finding a consistent extension for the remaining variables (thus finding a solution to the entire problem) or concluding that no such extension exists (in which case backtracking takes place and another instantiation tried).

Example 7.1.1 Assume that backtracking instantiates the variables of the CSP represented in Figure 7.2a in the order C, B, A, E, D, F (Figure 7.2b). Backtracking will instantiate variables C, B and A, and then, realizing that these variables cut all cycles, will invoke a tree-solving routine on the rest of the problem: the tree-problem in Figure 7.2c with variables C, B and A assigned, should then be attempted. If no solution is found, control returns to backtracking which will go back to variable A. □

The idea of cutset-conditioning generalized to all graphical models. Indeed we already observed in Chapter 4 in Section **??** that when variable are assigned connectivity of the graph reduces thus yielding saving in computation. This yield algorithm that combines bucket-elimination algorithm with conditioning called VEC, trading space for time.

As noted in Chapter 4, the cycle-cutset scheme can be generalized. Rather than insisting on conditioning on a subset (cutset) that cuts all cycles and yields subproblems having induced-width 1, we can allow cutsets that create subproblems whose induced-width is higher than 1 but still bounded. This suggests a framework of hybrid algorithms parameterized by a bound w on the induced-width of subproblems solved by inference.

Definition 7.1.2 (w-cutset) *Given a graph G, a subset of nodes is called a w-cutset iff when the subset is removed the resulting graph has an induced-width less than or equal to*

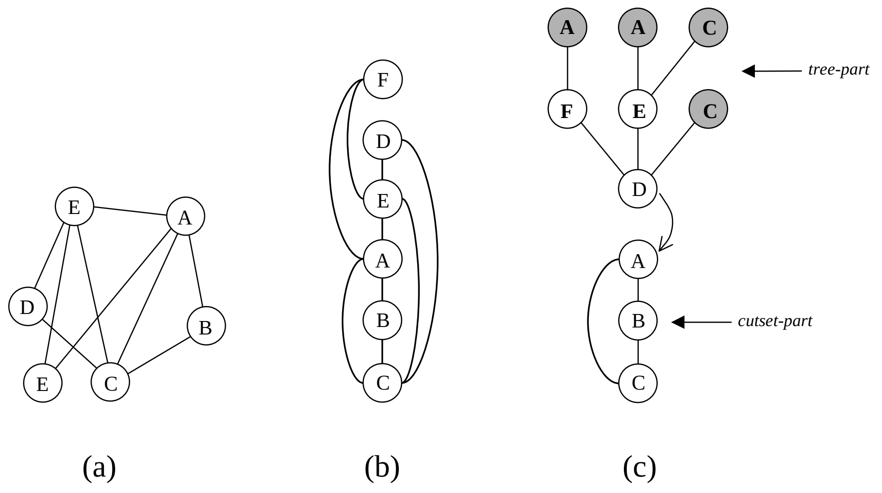


Figure 7.2: (a) a constraint graph (b) its ordered graph (c) The constraint graph of the cutset variable and the conditioned variable, where the assigned variables are darkened.

w. A minimal w-cutset of a graph has a smallest size among all w-cutsets of the graph. A cycle-cutset is a 1-cutset of a graph.

Finding a minimal w-cutset is a hard task. However, like in the special case of a cycle-cutset we can settle for a w-cutset relative to the given variable ordering. We can look for an initial set of the ordering that is a w-cutset. Then a DFS search algorithm can traverse the search space over the w-cutset and for each of its consistent assignment solve the rest of the problem by ADPATIVE-CONSISTENCY if it is a constraint problem or by BE, OR CTE, in the general case.

Algorithm cutset-decomposition(w) is described in Figure 7.3. The algorithm is presented in the context of constraint networks and for finding a single solution but it can be easily extended to any query over graphical models (exercise: Extend the algorithm to find the probability of evidence given a Bayesian network) It applies DFS (backtracking) search on the w-cutset and adaptive-consistency on the remaining variables. The constraint problem $\mathcal{R} = (X, D, C)$ conditioned on an assignment $Y = \bar{y}$ and denoted by $\mathcal{R}_{\bar{y}}$ is $\mathcal{R}$ augmented with the unary constraints dictated by the assignment $\bar{y}$. In the worst-case, all possible assignments to the w-cutset variables need to be enumerated. If c is the w-cutset size, k^c is the number of subproblems of induced-width bounded by w

```
Algorithm cutset-decomposition(w)
Input: A Graphical model such as a constraint network
$\mathcal{R} = (X, D, C)$, $Y \subseteq X$ which is a w-cutset. $d$ is an ordering that
starts with $Y$ such that the induced-width when $Y$ is removed,
along $d$, is bounded by $w$, $Z = X - Y$.
Output: A consistent assignment, if there is one.

1. while $\bar{y} \leftarrow$ next partial solution of $Y$ found by backtracking,
   do
     (a) $\bar{z} \leftarrow adaptive - consistency(\mathcal{R}_{Y=\bar{y}})$.
     (b) if $\bar{z}$ is not false, return solution $(\bar{y}, \bar{z})$.
2. endwhile.
3. return: the problem has no solutions.
```

Figure 7.3: Algorithm *cutset-decomposition(w)*

needed to be solved, each requiring $O(nk^{w+1})$ steps.

Theorem 7.1.3 *[27] Algorithm cutset-decomposition(w) has time complexity of $O(n \cdot k^{c+w+1})$ where n is the number of variables, c is the w-cutset size and k is the domain size. The space complexity of the algorithm is $O(k^w)$.* $\Box$

The special case of $w = 1$ yield the cycle-cutset decomposition algorithm whose time complexity is $O(nk^{c+2})$ and it operates in linear space. Thus, the constant w can control the balance between search and inference (e.g., variable-elimination), and can affect the tradeoff between time and space.

Another approach that uses the w-cutset principle is to alternate between conditioning-search and variable-elimination. Given a variable ordering we can apply BE as long as the induced-width of the variables does not exceed w. If a variable has induced-width higher than w, it will be conditioned upon. The algorithm alternates between conditioning and elimination. Clearly, a cutset uncovered via the *alternating algorithm* is also a w-cutset and therefore can be used within the cutset-decomposition scheme.

Both cutset-decomposition and the alternating cutset-elimination algorithm call for a new optimization task on graphs:

Definition 7.1.4 (finding a minimal w-cutset) *Given a graph $G = (V, E)$ and a constant w, find a smallest subset of nodes U, such that when removed the resulting graph has induced-width less than or equal w.*

Finding a minimal w-cutset is hard, but various greedy heuristic algorithms were investigated empirically. Several greedy and approximation algorithms for the special case of cycle-cutset can be found in the literature [2]. The general task of finding a minimal w-cutset was addressed in recent papers [44, 12] both for the cutset-decomposition version and for the alternating version. Note that verifying that a given subset of nodes is a w-cutset can be accomplished in polynomial time (linear in the number of nodes), by deleting the candidate cutset from the graph and verifying that the remaining graph has an induced width bounded by w [5].

In summary, the parameter w can be used within the cutset-decomposition scheme to control the trade-off between search and inference. If d is the ordering used by cutset-decomposition(w) and if $w \geq w^*(d)$, the algorithm coincides with a pure inference algorithm such as BUCKET-ELIMINATION . As w decreases, the algorithm requires less space and more time. It can be shown that the size of the smallest cycle-cutset (1-cutset), c_1^* and the smallest induced width, w^*, obey the inequality $c_1^* \geq w^* - 1$. Therefore, $1 + c_1^* \geq w^*$, where the left side of this inequality is the exponent that determines the time complexity of cutset-decomposition(w=1), while w^* governs the complexity of BUCKET-ELIMINATION. In general,

Theorem 7.1.5 *Given graph G, and denoting by c_w^* its minimal w-cutset then,*

$$1 + c_1^* \geq 2 + c_2^* \geq ...b + c_b^*, ... \geq w^* + c_{w^*}^* = w^*$$

(Prove as an exercise.)

We get a hybrid scheme controlled by w, whose time complexity decreases and its space increases as w changes from w^* to 1.

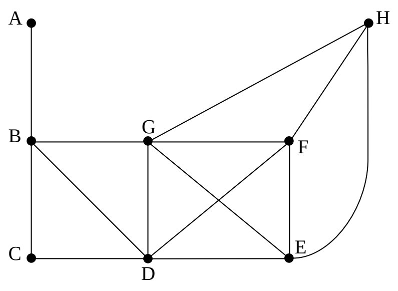


Figure 7.4: a primal constraint graph

7.2 The super-bucket and super-cluster schemes

We now present an orthogonal approach for combining search and inference. The inference algorithm CTE that process a tree-decomposition already contains a hidden combination of variable elimination and search. It computes functions on the separators using variable elimination and is space exponential in the separator's size. The clusters themselves can be processed by search in time exponential in the cluster size. Thus, one can trade even more space for time by allowing larger cliques but smaller separators.

Assume a problem whose tree-decomposition has tree-width r and maximum separator size s. Assume further that our space restrictions do not allow the necessary $O(k^s)$ memory required when applying CTE on such a tree. One way to overcome this problem is to combine the nodes in the tree that are connected by large separators into a single cluster. The resulting tree-decomposition has larger subproblems but smaller separators. This idea suggests a sequence of tree-decompositions parameterized by the sizes of their separators as follows.

Let T be a tree-decomposition of hypergraph $\mathcal{H}$. Let $s_0, s_1, ..., s_n$ be the sizes of the separators in T, listed in strictly descending order. With each separator size s_i we associate a secondary tree decomposition T_i, generated by combining adjacent nodes whose separator sizes are strictly greater than s_i. We denote by r_i the largest set of variables in any cluster of T_i, and by hw_i the largest number of constraints in T_i. Note that as s_i decreases, both r_i and hw_i increase. Clearly, from Theorem 6.3.3 it follows that,

Theorem 7.2.1 *Given a tree-decomposition T of graphical model having n variables and*

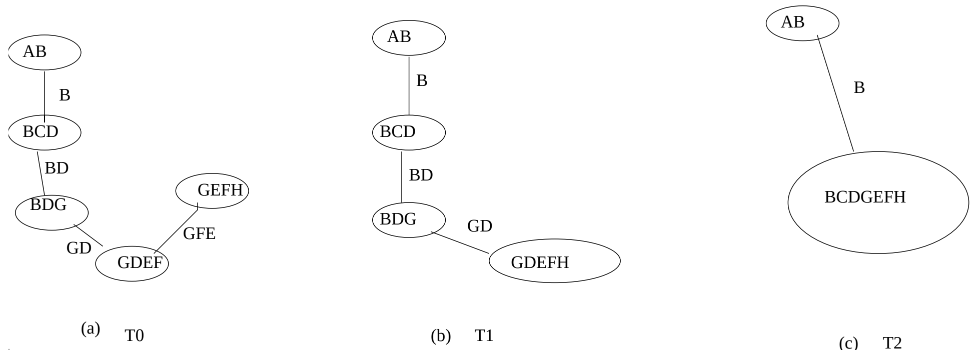


Figure 7.5: A tree-decomposition with separators equal to (a) 3, (b) 2, and (c) 1

m functions, separator sizes $s_0, s_1, ..., s_t$ and secondary tree-decompositions having a corresponding maximal number of nodes in any cluster, $r_0, r_1, ..., r_t$. The complexity of CTE when applied to each secondary tree-decompositions T_i is $O(m \cdot deg \cdot exp(r_i))$ time, and $O(n \cdot exp(s_i))$ space (i ranges over all the secondary tree-decomposition).

We will call the resulting algorithm SUPER-CLUSTER TREE ELIMINATION(s), or $SCTE(s)$. It takes a primary tree-decomposition and generates a tree-decomposition whose separator's size is bounded by s, which is subsequently processed by CTE. In the following example we assume that a naive depth-first search processes each cluster.

Example 7.2.2 Consider the constraint problem having the constraint graph in Figure 7.4. The graph can be decomposed into the join-tree in Figure 7.5(a). If we allow only separators of size 2, we get the join tree T_1 in Figure 7.5(b). This structure suggests that applying CTE takes time exponential in the largest cluster, 5, while requiring space exponential in 2. If space considerations allow only singleton separators, we can use the secondary tree T_2 in Figure 7.5(c). We conclude that the problem can be solved either in $O(k^4)$ time (k being the maximum domain size) and $O(k^3)$ space using T_0, or in $O(k^5)$ time and $O(k^2)$ space using T_1, or in $O(k^7)$ time and $O(k)$ space using T_2. □

7.2.1 Superbuckets

Since, as we saw in Chapter 4, bucket-elimination algorithms can be extended to bucket-trees and since a bucket-tree is a tree-decomposition, by merging adjacent buckets we generate a *super-bucket-tree* in a similar way to generating super clusters. This implies that in the top-down phase of bucket-elimination several variables are eliminated at once (see [29]). Algorithm SCTE suggests a new graph parameter.

Definition 7.2.3 *Given a graph G and a constant s find a tree-decomposition of G having the smallest induced-width, w_s^* when the separator size is bounded by s.*

A related problem of finding a tree-decomposition with a bounded tree-width w having the smallest separator, was shown to be polynomial [41]. Finding w_s^* is hard. However, it is easy for the special case of $s = 1$ as we show next.

7.2.2 Decomposition into non-separable Ccomponents

A special tree-decomposition occurs when all the separators are singleton variables. This type of tree-decomposition is attractive because it requires only linear space. While we generally cannot find the best tree-decompositions having a bounded separators' size in polynomial time, this is a feasible task when the separators are singletons. To this end, we use the graph notion of *non-separable components* [38].

Definition 7.2.4 (non-separable components) *A connected graph $G = (V, E)$ is said to have a separation node v if there exist nodes a and b such that all paths connecting a and b pass through v. A graph that has a separation node is called* separable*, and one that has none is called* non-separable*. A subgraph with no separation nodes is called a* non-separable component *or a* bi-connected component*.*

An $O(| E |)$ algorithm exists for finding all the non-separable components and the separation nodes. It is based on a depth-first search traversal of the graph. An important property of non-separable components is that that they are interconnected in a tree-structured manner [38]. Namely, for every graph G there is a *tree SG*, whose nodes are the non-separable components $C_1, C_2, \ldots, C_r$ of G. The separating nodes of these trees are $V_1, V_2, \ldots, V_t$ and any two component nodes are connected through a separating node vertex in SG.

Clearly the tree of non-separable components suggests a tree-decomposition where each node corresponds to a component, the variables of the nodes are those appearing in each component, and the constraints can be freely placed into a component that contains their scopes. Applying CTE to such a tree requires only linear space, but is time exponential in the components' sizes (see also [29]).

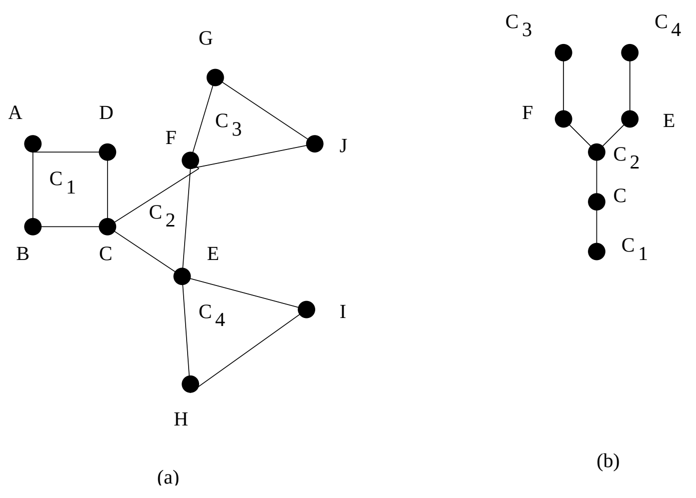
Example 7.2.5 Assume that the graph in Figure 7.6(a) represents a graphical model having unary, binary and ternary functions. Assume the functions are constraints as follows: $\mathcal{R} = \{R_{AD}, R_{AB}, R_{DC}, R_{BC}, R_{GF}, D_G, D_F, R_{EHI}, R_{CFE}\}$. The non-separable components and their tree-structure are given in Figure 7.6(b,c). The ordering of components $d = (C_1, C_2, C_3, C_4)$ dictates super-clusters associated with variables $\{G, J, F\}$, $\{E, H, I\}$, $\{C, F, E\}$ and $\{A, B, C, D\}$. The initial partition into super-clusters and a schematic execution of CTE are displayed in Figure 7.6d. □

Theorem 7.2.6 (non-separable components) *[47] If $\mathcal{R} = (X, D, C)$, $|X| = n$, is a constraint network whose constraint graph has non-separable components of at most size r, then the super-cluster-tree elimination algorithm, whose buckets are the non-separable components, is time exponential $O(n \cdot k^r)$ but requires only linear in space. k bounds the domain size.*

7.3 Bibliographical Notes

The loop-cutset conditioning for Bayesian networks was introduced by Pearl [74]. The cycle-cutset conditioning for constraint networks was introduced in [35]. Extensions to higher levels of w-cutsets appeared first in the context of satisfiability in [80] and were subsequently addressed for constraint processing by Javiere and Dechter [60]. The cutset-conditioning scheme was used both for solving SAT problems and for optimization tasks [79, 59] and is currently used for Bayesian networks applications [43, 42].

The problem of findingthe smallest cycle-cutset, or as is most commonly known, the feedback set problem was widely addressed[cite...]. Algorithms for finding a small cycle-cutsets also called the feedback set problems were proposed by [2]. An algorithm for finiding good w-cutset is given in [12]. [to complete...]



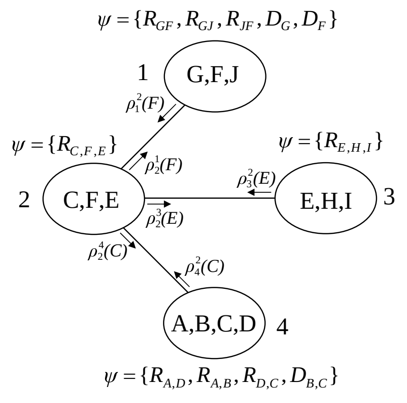


Figure 7.6: A graph and its decomposition into non-separable components.

Chapter 8

Search Algorithms over Graphical Models

In this chapter we start the discussion of the second type of reasoning algorithms, those that are based on the *conditioning* step. Namely, on assigning a single value to a variable. To recap, 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 displayed by the underlying graphical model and in avoiding redundant computation. They have worst case time guarantee which is exponential in the treewidth of the graph. Unfortunately, any method that is time-exponential in the treewidth is also space exponential in the treewidth or in the related separator-width parameter and, therefore, not feasible for models having 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 a full solution. Such search trees were discussed briefly towards the end of Chapter 4 when we discussed hybrids of conditioning and inference. Notice that the inherent linear structure of search spaces does not retain the independencies represented in the underlying graphical models and, therefore, algorithms exploring such traditional search spaces may be inferior.

The memory requirements of search algorithms, on the other hand, may be less severe than those of inference-based algorithms. In addition, search requires 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 algorithms are the only choice available for models with large treewidth and with implicit representation.

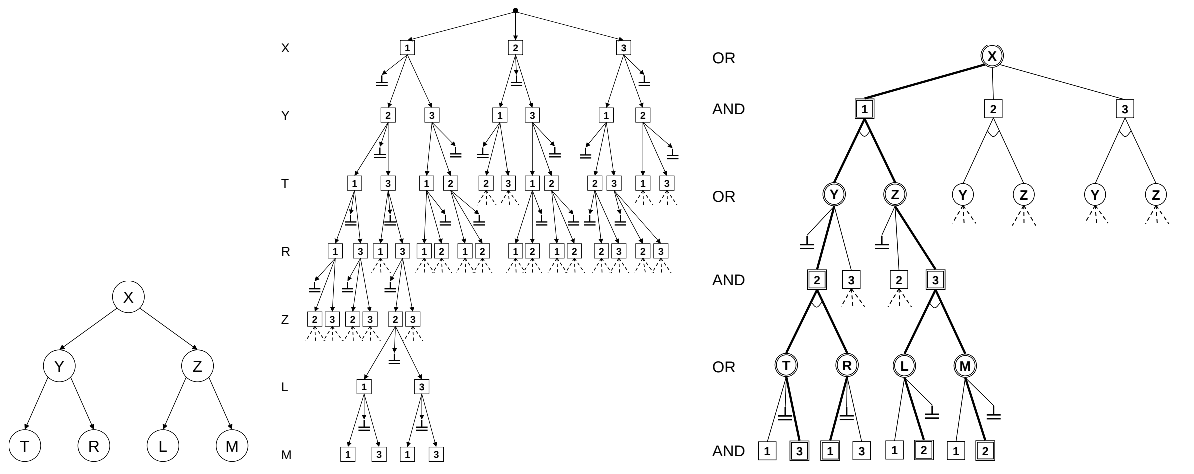
In this chapter we will show that AND/OR search spaces, originally introduced in the context of heuristic search [73], can be used to encode some of the structural information in the graphical models. In particular, they can capture the independencies in the graphical model to yield AND/OR search trees that are exponentially smaller than the standard OR tree. We will provide analysis of the size of the AND/OR search tree and show that it is bounded exponentially by the depth of a spanning pseudo-tree over the graphical model. Subsequently, we show that the search tree may contain significant redundancy that when identified, can be avoided by moving from AND/OR search tree to AND/OR search graph. This additional savings can reduce the size of the AND/OR search space which can then be bounded exponentially by the treewidth,

8.1 AND/OR Search Trees

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

Example 8.1.1 Consider the simple tree graphical model (*i.e.*, the primal graph is a tree) in Figure 8.1(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, $(X = 2)$ and $(X = 3)$. Applying the decomposition along the tree (in Figure 8.1(a) yields the AND/OR search tree in Figure 8.1(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 8.1(b). Clearly, the size of the AND/OR search tree is smaller than that of the regular OR



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

Figure 8.1: OR vs. AND/OR search trees; note the connector for AND arcs.

space. The OR search space has $3 \cdot 2^7$ nodes while the AND/OR one has $3 \cdot 2^5$ (compare 8.1(b) with 8.1(c)). If k is the domain size, a balanced binary tree graphical model (e.g., a map coloring problem) with n nodes has an OR search tree of size $O(k^n)$. The AND/OR search tree, whose underlying tree graphical model has depth $O(\log_2 n)$, has size $O((2k)^{\log_2 n}) = O(n \cdot k^{\log_2 n}) = O(n^{1+\log_2 k})$. When $k = 2$, this becomes $O(n^2)$. □

The AND/OR space is not restricted to tree graphical models. It only has to be guided by a 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 spanning tree of the primal graph that is generated by depth-first search (DFS tree) first, and will generalize to a broader class of spanning trees, called pseudo-trees, subsequently.

Definition 8.1.2 (DFS spanning tree of a graph) *Given a graph $G = (V, E)$ and given a node X_1, a* DFS *tree $\mathcal{T}$ of G is generated by applying a depth-first-search traversal over the graph, yielding an ordering $d = (X_1, \ldots, X_n)$. The* DFS spanning tree *$\mathcal{T}$ of G is defined as the tree rooted at the first node, X_1, which includes only the traversed (by DFS) arcs of G. Namely, $\mathcal{T} = (V, E')$, where $E' = \{(X_i, X_j) \mid X_j\ traversed\ from\ X_i\ by\ DFS\ traversal\}$.*

Definition 8.1.3 (AND/OR search tree) *Given a graphical model $\mathcal{M} = \langle X, D, F, \bigotimes \rangle$, its primal graph G and a spanning DFS tree $\mathcal{T}$ of G, the associated AND/OR search tree, denoted $S_{\mathcal{T}}(\mathcal{M})$, has alternating levels of AND and OR nodes. The OR nodes are labeled X_i and correspond to the variables. The AND nodes are labeled $\langle X_i, x_i \rangle$ (or simply x_i) and correspond to the value assignments of the variables. The structure of the AND/OR search tree is based on the underlying spanning tree $\mathcal{T}$. 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 $S_{\mathcal{T}}(\mathcal{M})$ 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 to the variables along the path. Namely to 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 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 constraints when we have a constraint problem, or relative to the flat constraints (see* **??***), otherwise.*

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 guiding spanning tree $\mathcal{T}$. Each OR arc, emanating from an OR to an AND node is associated with a weight to be defined shortly (see Definition 8.1.8).*

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 guiding spanning tree, denoted also by $path_{\mathcal{T}}(X_i)$.

A solution subtree is defined in the usual way:

Definition 8.1.4 (solution subtree) *A* solution subtree *of an AND/OR search tree contains the root node. For every OR node 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 8.1.5 In the example of Figure 8.1(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 8.1(c) is X. Its child nodes which are AND nodes, are labeled $\langle X, 1\rangle, \langle X, 2\rangle, \langle X, 3\rangle$ (only the values are noted in the Figure). From each of these AND nodes emanate two OR nodes, Y and Z, since these are the child nodes of X in the DFS tree of Figure 8.1(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 8.1(c) a solution tree is highlighted. $\square$

8.1.1 Weights of OR-AND Arcs

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

Definition 8.1.6 (arc weights for constraint networks) *Given an AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ of a constraint network $\mathcal{R}$, each terminal node is assumed to have a single, dummy, outgoing arc. The outgoing arc of a terminal AND node always has the weight "1" (namely it is consistent 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 general graphical model using the notion of buckets of functions. The concept is simple even if the formal definition may look involved. When considering an arc (n, m) having labels (X_i, x_i) (X_i labels n and x_i labels m), we identify all the functions over variable X_i that became fully instantiated now. We assign each such function a numerical weight based on the assignment along the path to n. The products of all these *function values* is the weight of the arc. The following definition identify those functions of X_i we want to consider in the product.

Definition 8.1.7 (buckets relative to a guiding tree) *Given a graphical model $\mathcal{M} = \langle X, D, F, \bigotimes\rangle$ and a guiding tree $\mathcal{T}$, the* bucket *of X_i relative to $\mathcal{T}$, denoted $B_{\mathcal{T}}(X_i)$, is*

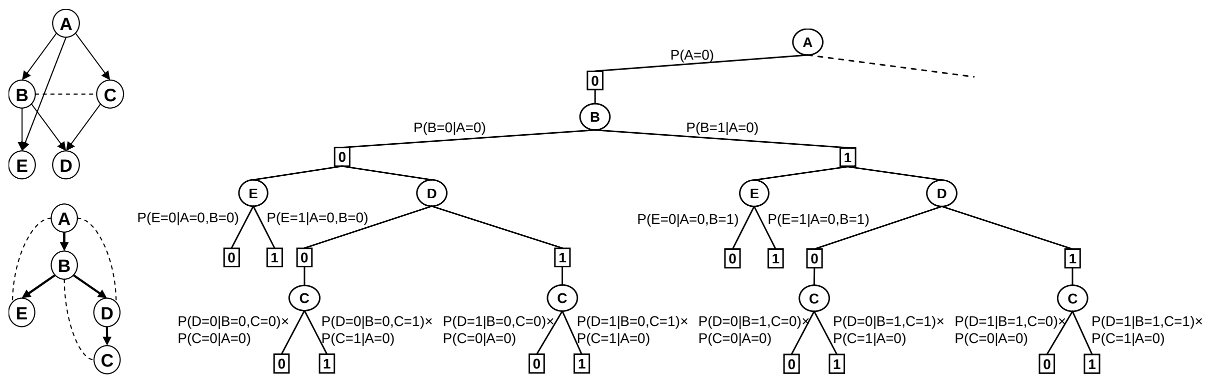


Figure 8.2: Arc weights for probabilistic networks

the set of functions whose scopes contain X_i and are included in $path_{\mathcal{T}}(X_i)$. Namely,

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

Weights are assigned only on arcs connecting an OR node to an AND node.

Definition 8.1.8 (OR-to-AND weights) *Given an AND/OR tree $S_{\mathcal{T}}(\mathcal{M})$, of a graphical model $\mathcal{M}$, 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 $B_{\mathcal{T}}(X_i)$ which are assigned by their values along π_m. Formally, $w_{(n,m)}(X_i, x_i) = \bigotimes_{f \in B_{\mathcal{T}}(X_i)} f(asgn(\pi_m))$. If the set of functions is empty the weight is the constant 1 (or the identity relative to the combination operator).*

The weight of a solution-tree is the product of weights on all its arcs.

Definition 8.1.9 (weight of a solution subtree) *Given a weighted AND/OR tree $S_{\mathcal{T}}(\mathcal{M})$, of a graphical model $\mathcal{M}$, and given a solution subtree t, the weight of t is $w(t) = \bigotimes_{e \in arcs(t)} w(e)$, where $arcs(t)$ is the set of arcs in subtree t.*

Example 8.1.10 Figure 8.2 shows a Bayesian network, a DFS tree that guides its 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 variable E contains the function $P(E|A, B)$, and the bucket of C contains two functions, $P(C|A)$ and $P(D|B, C)$.

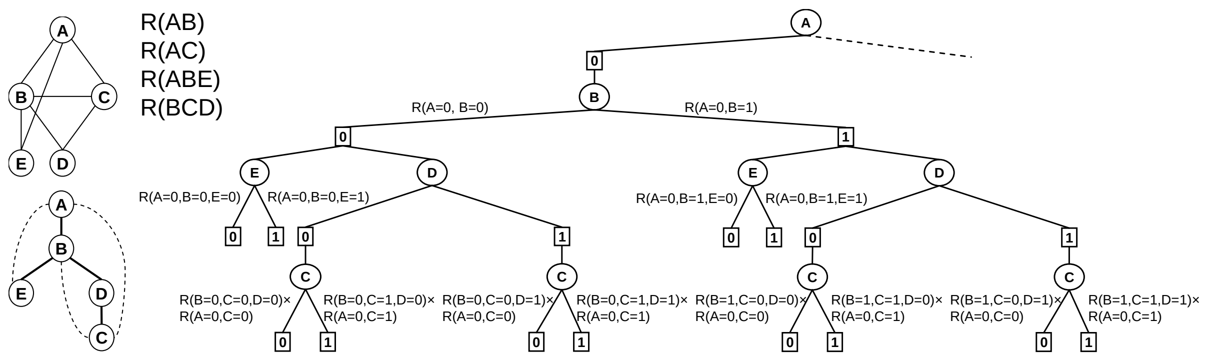

R(AB)
R(AC)
R(ABE)
R(BCD)
R(A=0, B=0)
R(A=0,B=1)
R(A=0,B=0,E=0)
R(A=0,B=0,E=1)
R(A=0,B=1,E=0)
R(A=0,B=1,E=1)
R(B=0,C=0,D=0)× R(A=0,C=0)
R(B=0,C=1,D=0)× R(A=0,C=1)
R(B=0,C=0,D=1)× R(A=0,C=0)
R(B=0,C=1,D=1)× R(A=0,C=1)
R(B=1,C=0,D=0)× R(A=0,C=0)
R(B=1,C=1,D=0)× R(A=0,C=1)
R(B=1,C=0,D=1)× R(A=0,C=0)
R(B=1,C=1,D=1)× R(A=0,C=1)

Figure 8.3: Arc weights for constraint networks

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 8.3 shows a constraint network with four relations, a guiding 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 8.4(b) we show the numerical values of the weights on the weighted AND/OR tree for the same belief network whose conditional probability tables are shown in 8.4(a). (Exercise: generate the AND/OR weighted tree for a DFS-tree ordering d= (C,B, D, A, E))

□

8.1.2 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 to one of its ancestors in the guiding tree. This ensures that each scope of a function in F will be fully assigned on some path in

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

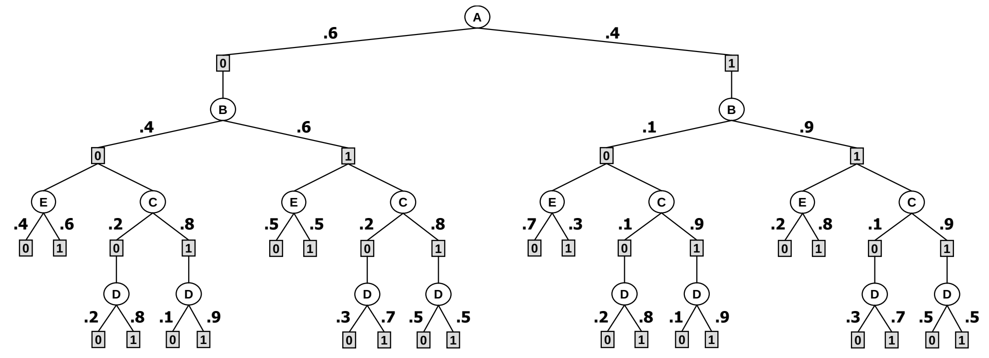
P(D | B,C)

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 8.4: Labeled AND/OR search tree for belief networks

$\mathcal{T}$, a property that is essential for the validity of the AND/OR search space. Indeed, the AND/OR search tree is an alternative equivalent representation of the graphical model.

Theorem 8.1.11 (correctness) *Given a graphical model* $\mathcal{M} = \langle X, D, F = \{f_1, ..., f_r\}, \bigotimes\rangle$ *having a primal graph G and a DFS spanning tree* $\mathcal{T}$ *of G and its associated weighted AND/OR search tree* $S_{\mathcal{T}}(\mathcal{M})$ *then 1) there is a one-to-one correspondence between solution subtrees of* $S_{\mathcal{T}}(\mathcal{M})$ *and solutions of* $\mathcal{M}$*; 2) the weight of any solution tree equals the cost of the full solution assignment it denotes; namely, if t is a solution tree of* $S_{\mathcal{T}}(\mathcal{M})$ *then* $F(assn(t)) = w(t)$*, where* $assn(t)$ *is the full solution defined by tree t. (Prove as an exercise.)*

The virtue of an AND/OR search tree representation is that its size can be far smaller than the traditional OR search tree. The size of an AND/OR search tree depends on the depth, also called height, of its DFS spanning tree $\mathcal{T}$. Therefore, DFS trees of smaller

Table 8.1: OR vs. AND/OR search size, 20 nodes

		OR space	AND/OR space	
treewidth	height	nodes	AND nodes	OR nodes
5	10	2,097,151	10,494	5,247
4	9	2,097,151	5,102	2,551
5	10	2,097,151	8,926	4,463
5	10	2,097,151	7,806	3,903
6	9	2,097,151	6,318	3,159

depth should be preferred. An AND/OR search tree becomes an OR search tree when its DFS tree is a chain.

Theorem 8.1.12 (size bounds of AND/OR search tree) *Given a graphical model $\mathcal{M}$, with domains size bounded by k, having a DFS spanning tree $\mathcal{T}$ whose height is h and having l leaves, the size of its AND/OR search tree $S_{\mathcal{T}}(\mathcal{M})$ is $O(l \cdot k^h)$ (and therefore also $O(nk^h)$ and $O((bk)^h)$ when b bounds the branching degree of $\mathcal{T}$ and n bounds the number of nodes). The size of its OR search tree along any ordering is $O(k^n)$ and these bounds are tight.*

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 $O(k^h)$ when h bounds the path length. The DFS tree $\mathcal{T}$ is covered by l such directed paths, whose lengths are bounded by h. The union of their individual search trees covers the whole AND/OR search tree $S_{\mathcal{T}}$, where every distinct full path in the AND/OR tree appears exactly once, and therefore, the size of the AND/OR search tree is bounded by $O(l \cdot k^h)$. Since $l \leq n$ and $l \leq b^m$, it concludes the proof. The bounds are tight because they are realizable for graphical models whose all full assignments are consistent. ∎

Table 8.1 illustrates the size of AND/OR vs. OR search spaces for 5 random networks having 20 bi-valued 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 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 guiding DFS tree.

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

8.1.3 From DFS Trees to Pseudo Trees

You may have wondered whether DFS trees are the only type of trees that can guide the AND/OR decomposition. You were right! there is a larger class of spanning trees that can be appropriate to guide a decomposition for a graphical model called *pseudo trees* [47]. Such trees only need to obey the back-arc property mentioned earlier.

Definition 8.1.13 (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 $\mathcal{T}$ includes also the arcs in E' that are not in E. Namely the extended graph is defined as $G^{\mathcal{T}} = (V, E \cup E')$.*

Example 8.1.14 Consider the graph G displayed in Figure 8.5(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 8.5(b)). The tree $\mathcal{T}_2$ in Figure 8.5(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. The tree $\mathcal{T}_3$ in Figure 8.5(d), is a chain. The extended graphs $G^{\mathcal{T}_1}$, $G^{\mathcal{T}_2}$ and $G^{\mathcal{T}_3}$ are presented in Figure 8.5(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 guiding tree is a pseudo tree which is not necessarily a DFS-tree. Namely, the correctness properties (proposition 8.1.11) hold and the size complexity bounds are also extendible.

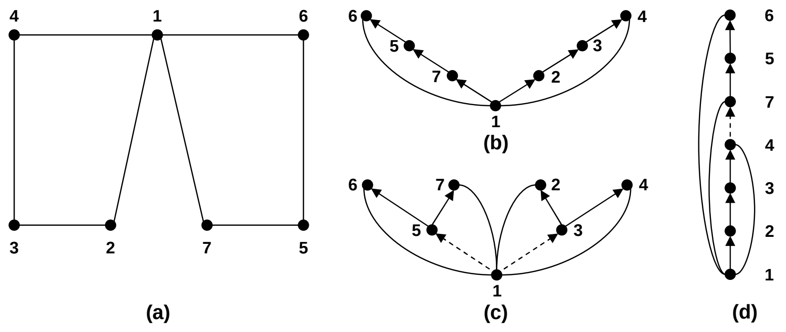


Figure 8.5: (a) A graph; (b) a DFS tree $\mathcal{T}_1$; (c) a pseudo tree $\mathcal{T}_2$; (d) a chain pseudo tree $\mathcal{T}_3$

Theorem 8.1.15 (properties of AND/OR search trees) *Given a graphical model $\mathcal{M}$ and a guiding pseudo tree $\mathcal{T}$, its weighted AND/OR search tree $S_{\mathcal{T}}(\mathcal{M})$ obeys the correctness properties (1) and (2) of Proposition 8.1.11 and its size is $O(l \cdot k^h)$ where h 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 8.1.11 carry over to AND/OR search spaces that are guided by a pseudo tree. Likewise, the bound size argument in the proof of Theorem 8.1.12 holds relative to the height of a pseudo tree. ∎

Figure 8.6 shows the AND/OR search trees along the pseudo trees $\mathcal{T}_1$ and $\mathcal{T}_2$ from Figure 8.5. The domains of the variables are $\{a, b, c\}$ and the constraints are universal, namely they can represent a positive probabilistic graphical model. We see that the AND/OR search tree based on $\mathcal{T}_2$ is smaller, because $\mathcal{T}_2$ has a smaller height than $\mathcal{T}_1$. The weights are not specified here.

To illustrate, Table 8.2 shows the effect of DFS spanning trees against pseudo trees, both generated using brute-force heuristics over randomly generated graphs.

8.2 AND/OR Search Graphs

It is often the case that a search space that is a tree can become a graph if nodes that root identical search subspaces, or correspond to identical subproblems, are identified. Any

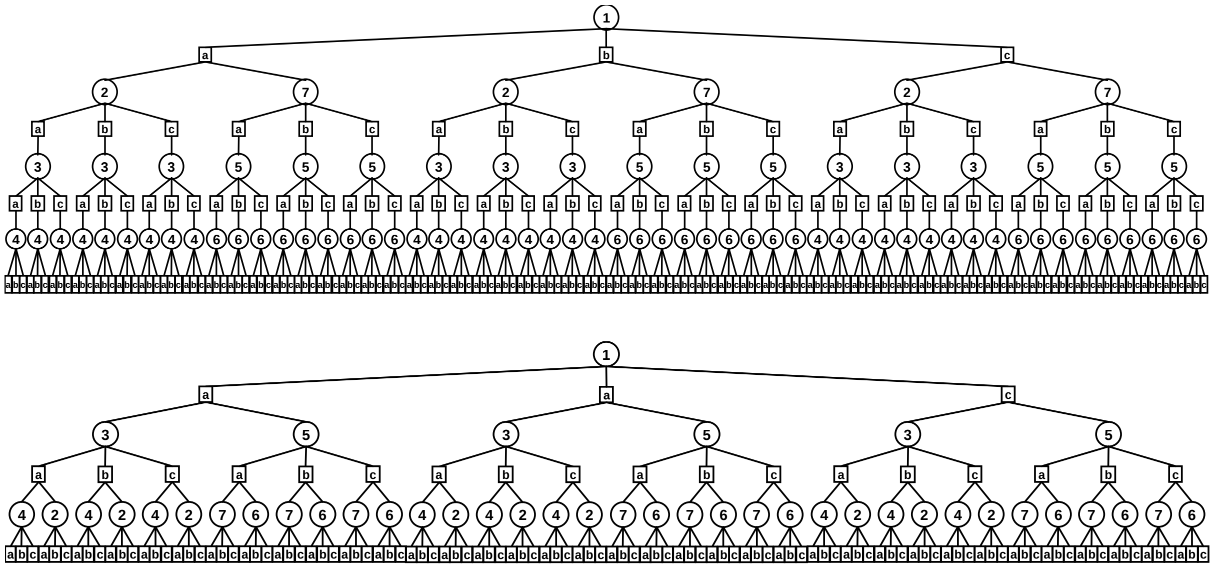


Figure 8.6: AND/OR search tree along pseudo trees $\mathcal{T}_1$ and $\mathcal{T}_2$

Table 8.2: 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

two such nodes can be *merged*, reducing the size of the search space (which will yields a graph).

Example 8.2.1 Consider again the graph in Figure 8.1(a) and its AND/OR search tree in Figure 8.1(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 $\langle Y, 3\rangle$) (not shown explicitly in the Figure). Clearly however, the subtrees rooted at each of these two AND nodes are identical and they can be merged because any specific assignment to Y uniquely determines its rooted subtree. Indeed, the resulting merged AND/OR search graph depicted in Figure 8.11 is equivalent to the AND/OR search tree in Figure 8.1(c). $\square$

It may also occur that two nodes that do not root identical subtrees still correspond to equivalent subproblems. Such nodes can also be *unified*, even if their *explicit* weighted subtrees do not look identical. In general two graphical models are equivalent if they have the same set of solutions, and if each is associated with the same *cost*. We will use the notion of *universal graphical model* to explain what we mean. A universal graphical model represents the *solutions* of a graphical model, through a single global function over all the variables.

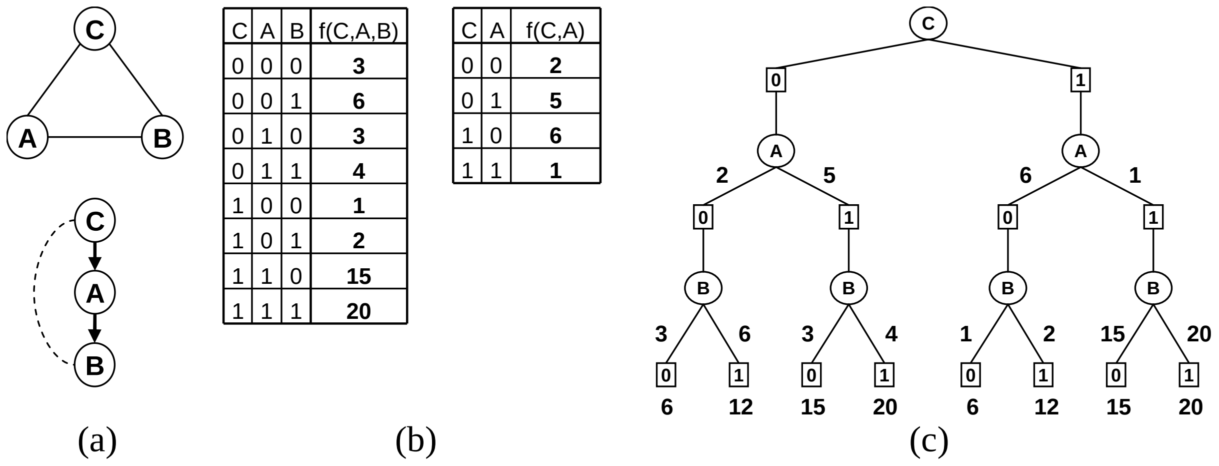
Definition 8.2.2 (universal equivalent graphical model) *Given a graphical model $\mathcal{M} = \langle X, D, F, \bigotimes \rangle$ the universal equivalent model of $\mathcal{M}$ is $u(\mathcal{M}) = \langle X, D, F = \{\bigotimes_{i=1}^{r} f_i\}, \bigotimes \rangle$.*

We also need to define the cost of a partial solution and the notion of a graphical model conditioned on a partial assignment. Informally, a conditioned graphical model on a particular partial assignment is obtained by assigning the appropriate values to all the relevant variables and modifying the costs appropriately.

Definition 8.2.3 (cost of an assignment, conditional model) *Given a graphical model $\mathcal{M}$,*

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 (denoted F_Y) evaluated at the assigned values. Namely, $c(y) = \bigotimes_{f \in F_Y} f(y_{scope(f)})$.*

2. *The conditional graphical model on $Y = y$ is $\mathcal{M}|_y = \langle X, D|_y, F|_y, \bigotimes \rangle$, where $D|_y = \{D_i \in D, X_i \notin Y\}$ and $F|_y = \{f|_{Y=y}, f \in F\}$.*



C	A	B	f(C,A,B)
0	0	0	**3**
0	0	1	**6**
0	1	0	**3**
0	1	1	**4**
1	0	0	**1**
1	0	1	**2**
1	1	0	**15**
1	1	1	**20**

C	A	f(C,A)
0	0	**2**
0	1	**5**
1	0	**6**
1	1	**1**

Figure 8.7: Merge vs. unify operators

8.2.1 Generating Compact AND/OR Search Spaces

We will next define merge and unify operations that transform AND/OR search trees into graphs. By construction, a graphical model $\mathcal{M}$ is equivalent to its AND/OR search tree, $S_{\mathcal{T}}(\mathcal{M})$ if $u(\mathcal{M})$ coincides with the weighted solution substrees of $S_{\mathcal{T}}(\mathcal{M})$, (see Definition 8.1.4).

Definition 8.2.4 (merge,unify) *Assume a given weighted AND/OR search graph $S'_{\mathcal{T}}$ of a graphical model $\mathcal{M}$ 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.*

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 $\mathcal{M}|_{asgn(\pi_1)} = \mathcal{M}|_{asgn(\pi_2)}$.*

Example 8.2.5 Let's follow the example in Figure 8.7 to clarify the difference between *merge* and *unify*. We have a graphical model defined by two functions (*e.g.* local cost functions) over three variables. The OR search tree given in Figure 8.7(c) cannot be reduced to a graph by *merge*, because the weights on the corresponding arcs are different. 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. In other words, the two nodes, one conditioned on $C = 0$ and one conditioned on $C = 1$ have exactly the same cost for each of the partial solutions. □

Proposition 8.2.6 (minimal AND/OR graphs) *Given a weighted AND/OR search graph $\mathcal{G}_T$ guided by a pseudo tree $\mathcal{T}$:*

1. *The* merge *operator has a unique fix point, called the* ***merge-minimal*** *AND/OR search graph.*

2. *The* unify *operator has a unique fix point, called the* ***unify-minimal*** *AND/OR search graph.*

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

The unify-minimal AND/OR search graph of $\mathcal{M}$ 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, 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 8.2.7 The smallest OR search graph of the graph-coloring problem in Figure 8.1(a) (same as Figure 8.8) is given in Figure 8.9 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 8.11. We see that some variable-value pairs (AND nodes) must be repeated in Figure 8.9 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 8.8-8.11 You should compare only by the AND nodes denoted by boxes in the figure. □

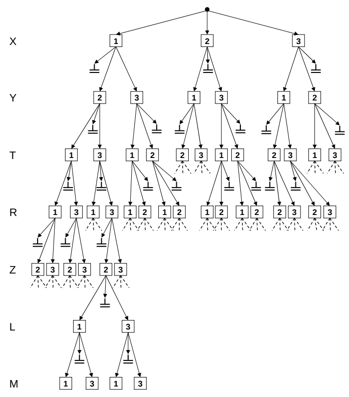


Figure 8.8: OR search tree for the tree problem in Figure 8.1(a)

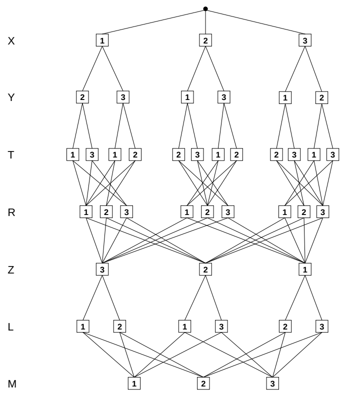


Figure 8.9: The minimal OR search graph of the tree graphical model in Figure 8.1(a)

8.2.2 Building Context-minimal AND/OR Search Graphs

The merging rule seems to be quite operational; we can generate the AND/OR search tree and then recursively merge identical subtrees going from leaves to root. This however, requires generating the whole search tree first, which is quite costly.

It turns out that for some nodes it is possible to recognize that they can be unified based on their *contexts*. The context is a set of ancestors variables pseudo tree $\mathcal{T}$ that completely determine the conditioned subproblems below a given variable. We will demonstrate the notion of context for graphical models that are trees, first.

We have already seen in Figure 8.1(a) that at level 3, node $\langle Y,1\rangle$ appears twice, (and so are $\langle Y,2\rangle$ and $\langle Y,3\rangle$). Clearly we see that Y uniquely determines its rooted subtree. We can say that Y is its own context. Indeed, as already observed, the AND/OR search graph in Figure 8.11 is equivalent to the AND/OR search tree in Figure 8.8 (same as Figure 8.1(c)).

In general, an AND/OR search graph of a constraint graph having no cycles along a pseudo-tree $\mathcal{T}$ 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* graphical model that is a tree, namely, that has no cycles. The resulting AND/OR graph is equivalent to the input $S_{\mathcal{T}}$ and its search space is $O(nk)$. The general idea of a context is to identify a minimal set of

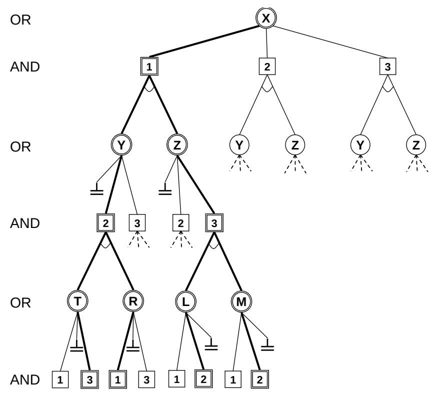


Figure 8.10: AND/OR search tree for the tree problem in Figure 8.1(a)

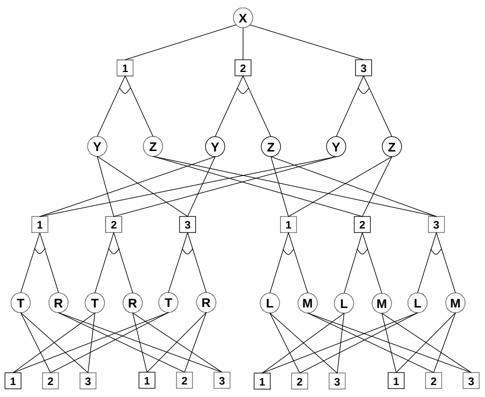


Figure 8.11: The minimal AND/OR search graph of the tree graphical model in Figure 8.1(a)

variables along the path from the root to the node such that when they are assigned the same assignments they yield the same conditioned subproblem, regardless of value assigned to the other variables. To derive a general and effective merging we would the following definition of induced-width of a pseudo-tree.

Definition 8.2.8 (induced width of a pseudo tree) *The induced width of G relative to a pseudo tree $\mathcal{T}$, is the maximum width of its* induced pseudo-tree *obtained by recursively connecting the parents of each node, going from leaves to root along each branch. In that process we consider both the extended arcs in the pseudo-tree and those in the graphical model.*

Definition 8.2.9 (parents) *Given a primal graph G and a pseudo tree $\mathcal{T}$ of a reasoning problem $\mathcal{P}$, the* parents *of an OR node X_i, denoted by pa_i or pa_{X_i}, are the ancestors of X_i that have connections in G to X_i or to descendants of X_i.*

Definition 8.2.10 (parent-separators) *Given a primal graph G and a pseudo tree $\mathcal{T}$ of a reasoning problem $\mathcal{P}$, the* parent-separators *of X_i (or of $\langle X_i, x_i\rangle$), denoted by pas_i or pas_{X_i}, are formed by X_i and its ancestors that have connections in G to descendants of X_i.*

It follows from these definitions that the parents of X_i, pa_i separate in the primal graph G the ancestors of X_i in $\mathcal{T}$, from X_i and its descendants. Similarly, the parents separators of X_i, pas_i, separate the ancestors of X_i from its descendants. It is also easy to see that each variable X_i and its parents pa_i form a clique in the induced pseudo-graph. The following proposition establishes the relationship between pa_i and pas_i. These two terms are clearly related. We use both in order to characterize two types of merging in the AND/OR search graph: AND merge and OR merge. The following claim follows directly from Definitions 8.2.9 and 8.2.10.

Proposition 8.2.11 *1. If Y is the single child of X in $\mathcal{T}$, then $pas_X = pa_Y$.*

2. If X has children $Y_1, \ldots, Y_k$ in $\mathcal{T}$, then $pas_X = \cup_{i=1}^{k} pa_{Y_i}$.

Theorem 8.2.12 (context based merge operators) *Let $G^{\mathcal{T}^*}$ be the induced pseudo-tree of $\mathcal{T}$ and let π_{n_1} and π_{n_2} be any two partial paths in an AND/OR search graph, ending with two nodes, n_1 and n_2.*

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

$$asgn(\pi_{n_1})[pas_{X_i}] = asgn(\pi_{n_2})[pas_{X_i}] \tag{8.1}$$

then the AND/OR search subtrees rooted by n_1 and n_2 are identical and n_1 and n_2 can be merged. The $asgn(\pi_{n_i})[pas_{X_i}]$ is called the ***AND context*** *of n_i.*

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

$$asgn(\pi_{n_1})[pa_{X_i}] = asgn(\pi_{n_2})[pa_{X_i}] \tag{8.2}$$

then the AND/OR search subtrees rooted by n_1 and n_2 are identical and n_1 and n_2 can be merged. The $asgn(\pi_{n_i})[pa_{X_i}]$ is called the ***OR context*** *of n_i.*

Definition 8.2.13 (context minimal AND/OR search graph) *The AND/OR search graph of $\mathcal{M}$ guided by a pseudo-tree $\mathcal{T}$ that is closed under context-based merge operator, is called the* context minimal *AND/OR search graph and is denoted by $C_{\mathcal{T}}(\mathcal{R})$.*

We should note that we can in general merge nodes based both on AND and OR contexts. However, Proposition 8.2.2 shows that doing just one of them renders the other unnecessary (up to some small constant factor). In practice, we would recommend just the OR context based merging, because it has a slight (albeit by a small constant factor) space advantage. In the examples that we give in this paper, $C_{\mathcal{T}}(\mathcal{R})$ refers to an AND/OR search graph for which either the AND context based or OR context based merging was performed exhaustively.

Example 8.2.14 For the balanced tree in Figure 8.1 consider the chain pseudo tree $d = (X, Y, T, R, Z, L, M)$. Namely the chain has arcs $\{(X, Y), (Y, T), (T, R), (R, Z), (Z, L), (L, M)\}$ and the extended graph includes also the arcs $(Z, X), (M, Z)$. The context of T along d is XYT (since the induced graph has the arc (T, X)), of R it is XR, for Z it is Z and for M it is M. Indeed in the first 3 levels of the OR search graph in Figure 8.9 there are no merged nodes. In contrast, if we consider the AND/OR ordering along the dfs tree, the context of every node is itself yielding a single appearance of each AND node having the same assignment annotation in the minimal AND/OR graph.

Since the number of nodes in the context minimal AND/OR search graph cannot exceed the number of different contexts, and since, as we will show, the context size is bounded by the induced-width of the pseudo-tree that guides it, the size of the context minimal graph can be bounded as exponentially by the induced-width along the pseudo-tree. So, we first extend the definition of induced-width to pseudo-trees as follows. We connect parents recursively going from leaves to root along each branch of the pseudo-tree. The induced-width is then the maximal width of a node in the resulting induced pseudo-tree.

Proposition 8.2.15 *Given a graphical model $\mathcal{M}$, and a pseudo tree $\mathcal{T}$ having induced width w, then the size of the context minimal AND/OR search graph based on $\mathcal{T}$, $CM_{\mathcal{T}}(\mathcal{R})$, is $O(n \cdot k^w)$, when k bounds the domain size.*

Proof:

For any variable, the number of its contexts is bounded by the number of possible instantiations to the variables in it context. Since, it can be shown that the context size of each variable is bounded by its induced-width along the pseudo-tree (see exercises), we get the bound of $O(k^w)$. Since we have n variables, the total bound is $O(n \cdot k^w)$. ∎

Therefore context-based merge (AND and/or OR) offers a powerful way of trimming the size of the AND/OR search space, and therefore bounding the truly minimal AND/OR search graph. We can generate $CM_{\mathcal{T}}$ using depth-first or breadth first traversals while figuring the converging arcs into nodes via their contexts. This way we avoid generating duplicate searches for the same contexts. All in all, the generation of the search graph is linear in its size, which is exponential in w and linear in n. Based on Proposition 8.2.15 we can conclude that

Theorem 8.2.16 *The context minimal AND/OR search graph $CM_{\mathcal{T}}$ of a graphical model having a pseudo tree with treewidth w can be generated in time and space $O(nk^w)$. The size of the generated context-minimal graph is bounded by $O(nk^w)$.*

8.3 Finding Good Pseudo Trees

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

Pseudo Trees created from induced-ordered graphs

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

A general scheme for generating pseudo trees is by considering induced graphs along some ordering d first. Subsequently, a pseudo-tree can be obtained via a depth-first traversal of the induced-ordered graph starting from the first node in d and breaking ties in favor of earlier variables in d. The algorithm is described in Figure 8.12. An alternative way for generating a pseudo-tree from an induced ordered graph is via its bucket-tree as presented in Definition 6.1.2. Namely, going from the the last variable to the first we connect each variable to its closest earliest neighbor in the induced ordered graph. Indeed a *bucket tree* is a pseudo tree. In summary:

```
GENERATE-PSEUDO-TREE
input: a graph $G = (V, E)$ an ordering $d$, $V = \{v_1, ..., v_n\}$.
output: A rooted tree $T = (V, E')$ which is a pseudo-tree of $G$.
1.  $G_d = (V, E")$ ← generate the induced graph of $G$ along $d$
2.  $E'$ ← generated as the forward arcs in a dfs search of $G_d$, starting at $v_1$.
```

Figure 8.12: The generate pseudo-tree procedure

Proposition 8.3.1 *Given a graphical model $\mathcal{M} =< X, D, F, \bigotimes >$ and an ordering d,*

1. *The bucket-tree derived from the induced ordered graph along d of $\mathcal{M}$, $T = (X, E)$ with $E = \{(X_i, X_j) | (B_{X_i}, B_{X_j}) \in bucket - tree\}$, is a pseudo tree of $\mathcal{M}$.*

2. *The DFS-tree generated by Algorithm Generate-Pseudo-tree in Figure 8.12 is a pseudo-tree.*

3. *Given an induced-graph of G, its bucket-tree is also a DFS-based spanning tree of the induced-graph along the given ordering. of G.*

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

It is interesting to note that a chain graphical model has a pseduo-tree of depth $\log n$, when n is the number of variables. On the other hand the induced-width of a chain is 1. Therefore, on the one hand a chain can be solved in constant space and in $O(k^{\log n})$ time along its height *logn* pseudo-tree, and on the other hand it can also be solved in $O(nk^2)$

time with $O(nk)$ memory using bucket elimination along its chain order having induced width of 1 and height of $n/2$. This special case generalizes into relationship between treewidth and pseudo-tree height:

Proposition 8.3.2 *[6, 51] The minimal height, h^*, over all pseudo trees of a given graph G satisfies $h^* \leq w^* \cdot \log n$, where w^* is the treewidth of G.*

Proof: If there is a tree-decomposition of G having a treewidth w, then it is possible to show that we can create a pseudo-tree whose height h satisfies $h \leq w \cdot \log n$ (do as exercise). From this it clearly follows that $h^* \leq w^* \cdot \log n$. ■

The above relationship suggests a bound on the size of the AND/OR search tree of a graphical model in terms of its treewidth.

Theorem 8.3.3 *A graphical model that has a treewidth w^* has an AND/OR search tree whose size is $O(k^{(w^* \cdot \log n)})$, where k bounds the domain size and n is the number of variables.*

Since, as noted, an induced ordered graph determine a pseudo-tree the question now is if the min-fill ordering heuristic which is so good for finding small induced-width is also good for finding pseudo-trees with small heights. Another question is what is the relative impact of the width and the height on the actual complexity. The AND/OR search graph is bounded exponentially by the induced-width while the AND/OR search tree is bounded exponentially by the height. We will have a glimpse into this question by comparing with an alternative scheme for generating pseudo-trees which is based on the hypergraph decompositions scheme.

Definition 8.3.4 (hypergraph separators) *Given a dual hypergraph $\mathcal{H} = (\mathbf{V}, \mathbf{E})$ of a graphical model, a* hypergraph separator decomposition *of size k by nodes S is obtained if removing S yields a hypergaph having k disconnected components. S is called a separator.*

It is well known that the problem of finding the minimal size hypergraph separator is hard. However heuristic approaches were developed over the years. [1]. Generating a pseudo

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

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

tree $\mathcal{T}$ for $\mathcal{M}$ using hypergraph decomposition is fairly straightforward. The vertices of the hypergraph are partitioned into two balanced (roughly equal-sized) parts, denoted by $\mathcal{H}_{left}$ and $\mathcal{H}_{right}$ respectively, while minimizing the number of hyperedges across. A small number of crossing edges translates into a small number of variables shared between the two sets of functions. $\mathcal{H}_{left}$ and $\mathcal{H}_{right}$ are then each recursively partitioned in the same fashion, until they contain a single vertex. The result of this process is a tree of hypergraph separators which can be shown to also be a pseudo tree of the original model where each separator corresponds to a subset of variables connected by a chain.

Table 8.3 illustrates the induced width and height of the pseudo tree obtained with the hypergraph and min-fill heuristics for 10 Bayesian networks from the Bayesian Networks Repository[2] and 10 constraint networks derived from the SPOT5 benchmarks [10]. It is generally observed that the min-fill heuristic generates lower induced width pseudo trees, while the hypergraph heuristic produces much smaller depth pseudo trees. Note that it is not possible to generate a pseudo-tree that is both optimal w.r.t. the treewidth and the height (remember our earlier example of a chain). Therefore, for graphical models

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

having a bounded treewidth w, the minimal AND/OR graph is bounded by $O(nk^w)$ while the minimal OR graph is bounded by $O(nk^{w \cdot \log n})$.

Example 8.3.5 Let's consider the graph of a graphical model given in Figure 8.13(a). We see the pseudo tree in part (b) having w=4 and h=8 and the corresponding context-minimal search graph in (c). The second pseudo-tred in part (d) has w=5, h=6 and the context-minimal graph appears in part (e). $\square$

8.3.1 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 only briefly on this issue here even though the ramification of dynamic variable ordering can be substantial (see for example [65, 40]. 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 8.3.6 Consider the propositional formula $X \rightarrow A \vee C$ and $X \rightarrow B \vee C$. The constraint graph is given in Figure 8.14(a) and a DFS tree in 8.14(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 8.14(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. $\square$

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.

8.4 Appendix: Proofs

Proof of Theorem 8.2.6

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

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

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

(2) Analogous to (1). (3) If the nodes can be merged, it follows that the subgraphs are identical, which implies that they define the same conditioned subproblems, and therefore the nodes can also be unified.

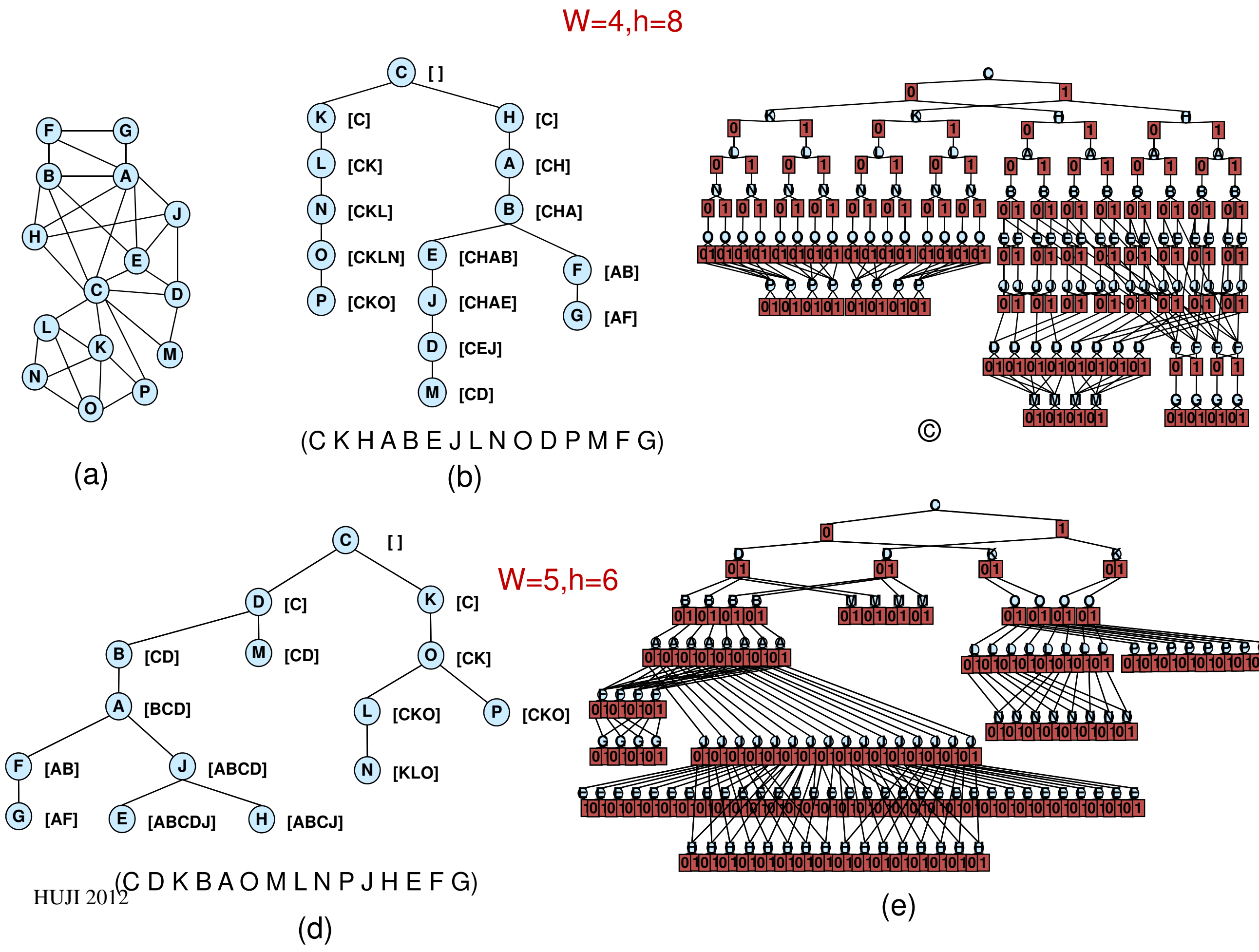

W=4,h=8
(a)
(C K H A B E J L N O D P M F G)
(b)
©
W=5,h=6
HUJI 2012
(C D K B A O M L N P J H E F G)
(d)
(e)

Figure 8.13: a graphical model; (a) one pseudo-tree; (b) Its context-minimal search graph ; (c) a second pseudo-tree; (d) Its corresponding context-minimal AND/OR search graph; (e)

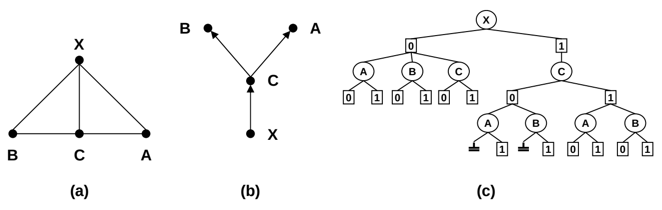


Figure 8.14: (a) A constraint graph; (b) a spanning tree; (c) a dynamic AND/OR tree

Chapter 9

Solving Reasoning problems by AND/OR Search

In this chapter we show how queries of likelihood and optimization can be answered by computing the value of the root node over the AND/OR search space discussed in the previous chapter. The value of a node in the search space is determined by the query we wish to answer. We will also show the impact of constraints and deterministic information on pruning the search space.

9.1 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 a cost function is defined by the graphical model we may also seek an optimal solution. The primary tasks over probabilistic networks are computing beliefs (i.e., the posterior marginals), 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 determines the consistency query. For solutions-counting the value function of each node is the number of solutions of the subproblem rooted at that node.

Definition 9.1.1 (value function for consistency and counting) *Given a weighted AND/OR tree $S_{\mathcal{T}}(\mathcal{R})$ of a constraint network. The value of a node (AND or OR) for* deciding consistency *is “1” if it roots a consistent subproblem and “0” otherwise. The value of a node (AND or OR) for* counting solutions *is the number of solutions in its subtree.*

We will next show that the value of nodes in the search graph can be expressed as a function of the values of their child nodes, thus allowing a recursive computation from leaves to root, to decide the relevant constraint query.

Proposition 9.1.2 (Recursive value computation for constraint queries)

1. *For the consistency task the value of AND leaves is ”1” and the value of OR leaves is “0” (they are inconsistent). An internal OR node is labeled “1” if one of its successor nodes is “1” and an internal AND node has value “1” iff all its child OR nodes have value “1”.*

2. *The counting values of leaf AND nodes are “1” and of leaf OR nodes are “0”. The counting value of an internal OR node is the sum of the counting-values of all its child nodes. The counting-value of an internal AND node is the product of the counting-values of all its child nodes. (prove as an exercise).*

We now move to probabilistic queries. Remember that the label of an arc $(X_i, \langle X_i, x_i \rangle)$ along path $\pi(x_i)$ is defined as $w((X_i, \langle X_i, x_i \rangle)) = \prod_{f \in B(X_i)} f(\pi(x_i)_{scop(f)})$, where $B(X_i)$ are the functions in its bucket.

Definition 9.1.3 (value for probabilistic queries) *Given a labeled AND/OR tree $S_{\mathcal{T}}(\mathcal{R})$ of a Bayesian network, for the probability of evidence query, the value of a node (AND or OR) is the probability of evidence restricted to its subtree variables and conditioned on 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 branch in the pseudo-tree conditioned on the evidence and on the assignment along the path to the node.*

Proposition 9.1.4 (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. (Prove as an exercise)

We can now generalize to any graphical model and queries. We define the recursive definition of values and then prove that it had the intended meaning of values.

Definition 9.1.5 (Recursive value computation for general reasoning problems) *The value function of a reasoning problem $\mathcal{P} = \langle \mathcal{M}, \Downarrow_Y, Z \rangle$, where $\mathcal{M} = \langle X, D, F, \bigotimes \rangle$ 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 8.1.8) 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*[1]*:*

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

Given a reasoning task, the value of the root node is the answer to the problem as stated next (the formal proof can be found in [37]).

Proposition 9.1.6 *Let $\mathcal{P} = \langle \mathcal{M}, \Downarrow_Y, Z \rangle$, where $\mathcal{M} = \langle X, D, F, \bigotimes \rangle$ and let X_1 be the root node in any AND/OR search graph $S'_{\mathcal{T}}(\mathcal{M})$. Then $v(X_1) = \Downarrow_X \bigotimes_{i=1}^{r} f_i$ when v is as defined in Definition 9.1.5.*

Search algorithms that traverse the AND/OR search space can compute the value of the root node yielding the answer to the problem. The next subsection presents typical

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

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 query. For application of these ideas for combinatorial optimization tasks, such as MPE see [64].

9.1.1 Algorithm AND/OR Tree Search and Graph Search

Algorithm 2 presents the basic depth-first traversal of the AND/OR search tree or search 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 using tables. For each variable X_i, a table is reserved in memory for each possible assignment to its parent set pa_i which is its context. Initially each entry has a predefined value, in our case "-1". The fringe of the search is maintained on a stack called `OPEN`. The current node is denoted by `n`, its parent by `p`, and the current path by π_n. The children of the current node are denoted by *successors*(`n`).If caching is set to "false" the algorithm searches the AND/OR tree and we will refer to it as *AOT*.

The algorithm is based on two mutually recursive steps: EXPAND and PROPAGATE, which call each other (or themselves) until the search terminates. Before expanding an OR node, its cache table is checked (line 5). If the same context was encountered before, it is retrieved from cache, and *successors*(`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. 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 marginal query of the root variable, we only need to keep the computation at the root of the search graph and normalize the results. However, 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 spaces are identical to the above algorithms when product is replaced by the appropriate combination operator and summation is replaced by the appropriate marginalization operator.

Complexity

From Theorems 8.1.15 and 8.3.3 we can clearly conclude that:

Theorem 9.1.7 *For any reasoning problem, algorithm* AOT *runs in linear space and time $O(nk^m)$, when m is the depth of the guiding 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, for constraint satisfaction that would terminate early with first solution, the algorithm would potentially be much faster than the rest of the AOT algorithms.

Based on Theorem 8.2.15 we can also derive a complexity bounds when caching, namely when searching the AND/OR context-minimal search graph.

Theorem 9.1.8 *For any reasoning problem, the complexity of algorithm* AOG *(i.e., the algorithm when caching is true) is time and space* $O(nk^w)$ *where* w *is the induced width of the guiding pseudo tree and* k *is the maximum domain size.*

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 (prove as an exercise).

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 in the size of the cluster separators only, rather than exponential in the cluster size.

We will use the term *dead caches* to address this issue. Intuitively, a node that has only one incoming arc in the search tree will be traversed only once by 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 based on their context.

Definition 9.1.9 (dead cache) *If* X *is the parent of* Y *in pseudo-tree* $\mathcal{T}$*, and* $context(X) \subset context(Y)$*, then* $context(Y)$ *represents a* dead cache.

We know that a pseudo-tree is also a bucket-tree. So given a pseudo-tree we can generate a bucket tree by having a cluster for each variable X_i and its parents pa_i in the induced-graph. Following the structure of the pseudo tree $\mathcal{T}$, some of the clusters may not be maximal. These are the ones that correspond to dead caches. The parents pa_i that are not dead caches are actually the separators between maximal clusters in the bucket tree.

Example 9.1.10 Consider the graphical models and the pseudo-tree in Figure 8.13. The context in the left branch (C, CK, CKL, $CKLN$) are all dead-caches. The only one which is not is CKO of P. As you can see, there are converging arcs into P only along this branch. Indeed if we describe the clusters of the corresponding bucket-tree. we would have just two maximal clusters: $CKLNO$ and $PCKO$ whose separator is CKO, the context of P. □

We can conclude,

Proposition 9.1.11 *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 recorded.*

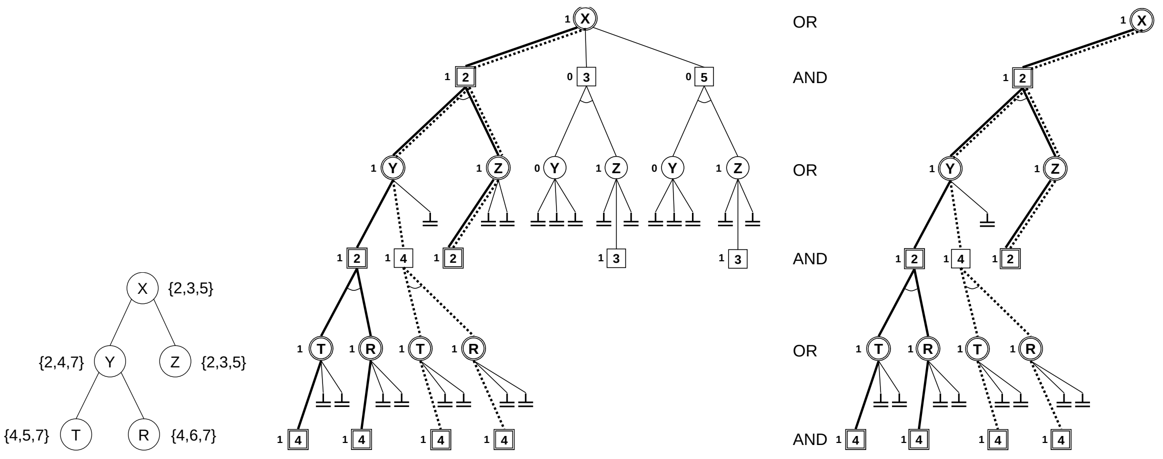
■

We can view the AND/OR tree algorithm (which we will denote AOT) and the AND/OR graph algorithm (denoted AOG) as two extreme cases in a parameterized collection of algorithms that trade space for time via a controlling parameter i. We denote this class of algorithms as $AO(i)$ where i determines the size of contexts that the algorithm caches. Algorithm $AO(i)$ records nodes whose context size is i or smaller (the test in line 21 needs to be a bit more elaborate and check if the context size is smaller than i). Thus AO(0) is identical to AOT, while $AO(w)$ is identical to AOG, where w is the induced width of the used pseudo tree. For any intermediate i we get an intermediate level of caching, which is space exponential in i and whose execution time will increase as i decreases. A more refined characterization of the time and space complexity of the algorithm can be obtained (see [36]).

9.2 AND/OR Search Algorithms For Mixed Networks

All the advanced constraint processing algorithms, either incorporating no-good learning and constraint propagation during search, or using variable elimination algorithms such as *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 next define formally the *backtrack-free* AND/OR search tree.

Definition 9.2.1 (backtrack-free AND/OR search tree) *Given graphical model $\mathcal{M}$ and given an AND/OR search tree $S_{\mathcal{T}}(\mathcal{M})$, the* backtrack-free AND/OR search tree *of*



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

Figure 9.1: AND/OR search tree and backtrack-free tree

$\mathcal{M}$ based on $\mathcal{T}$, denoted $BF_{\mathcal{T}}(\mathcal{M})$, is obtained by pruning from $S_{\mathcal{T}}(\mathcal{M})$ all inconsistent subtrees, namely all nodes that root no consistent partial solution.

Example 9.2.2 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 guiding DFS tree rooted at X, are given in Figure 9.1(a,b). In 9.1(b) we present the $S_{\mathcal{T}}(\mathcal{R})$ search space whose nodes' consistency status are already evaluated having value "1" if consistent and "0" otherwise. We also highlight two solutions subtrees; one depicted by solid lines and one by dotted lines. Part (c) presents $BF_{\mathcal{T}}(\mathcal{R})$, where all nodes that do not root a consistent solution are pruned. $\square$

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. In particular, the variable-elimination algorithms described in Chapter 3, such as *adaptive-consistency* and directional resolution, compile a constraint specification (resp., a Boolean CNF formula) that has a backtrack-free search space, if applied from leaves to root of the pseudo-tree. We remind now the edfinition of directional extension (see also chapter 3).

Definition 9.2.3 (directional extension [31, 79]) *Let $\mathcal{R}$ be a constraint problem and let d be a DFS ordering of a guiding pseudo tree, then $E_d(\mathcal{R})$ denotes the constraint network (resp., the CNF formula) compiled by Adaptive-consistency (resp., directional resolution) in reversed order of d.*

It is possible to show that the backtrack-free portion of an AND/OR tree can be obtained by generating a tighter representation along d using a variable elimination algorithm, first and then creating the AND/OR tree. In summary,

Proposition 9.2.4 *Given a Constraint network $\mathcal{R}$, and a pseudo-tree $\mathcal{T}$, the AND/OR search tree of the directional extension $E_d(\mathcal{R})$ when d is a DFS ordering of $\mathcal{T}$, coincides with the backtrack-free AND/OR search tree of $\mathcal{R}$ based on $\mathcal{T}$. 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 $\mathcal{R}$, This however is obvious, because Adaptive consistency makes $E_d(\mathcal{R})$ strongly directional $w^*(d)$ consistent, where $w^*(d)$ is the induced width of R along ordering d [31], a notion that is synonym with backtrack-freeness. ∎

Example 9.2.5 In Example 9.2.2, 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 9.1(c) is both $S_{\mathcal{T}}(\mathcal{R}')$ and $BF_{\mathcal{T}}(\mathcal{R})$. □

Proposition 9.2.4 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 [55] and are currently used in most advanced solvers [66]. Namely, when we apply no-good learning we explore a pruned search space whose many inconsistent

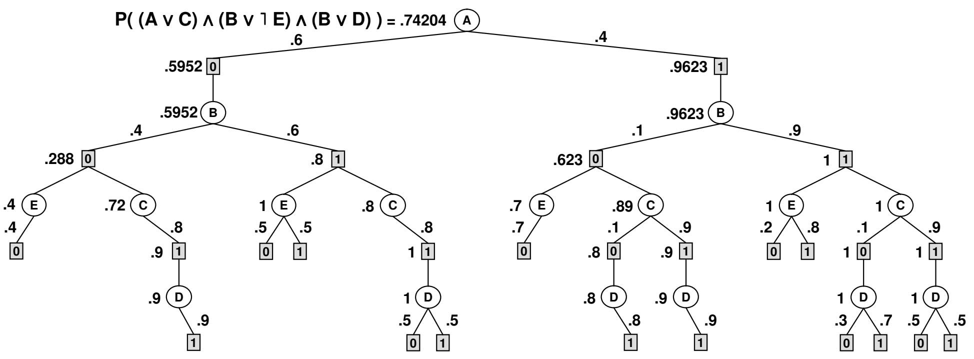


Figure 9.2: Mixed network defined by the query $\varphi = (A \vee C) \wedge (B \vee \neg E) \wedge (B \vee D)$

subtrees are removed. 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 space for processing queries over mixed networks.

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). We now describe these basic principles of constraint processing in the context of AND/OR search spaces [30].

9.2.1 AND-OR-cpe Algorithm

Algorithm AND-OR-CPE for the constraint probabilistic evaluation query (CPE) is given in Algorithm 3. It 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 AOT. 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 probability that a random tuple generated from the belief network distribution is consistent (satisfies the constraint portion). As usual, AND-OR-CPE traverses the AND/OR search tree or graph corresponding to $\mathcal{T}$ in a DFS manner and each node maintains a value v which accumulates the computation 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 9.2.6 We refer back to the example in Figure 8.4. 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 9.2. 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 9.2 is similar to Figure 8.4. In Figure 8.4 the evidence can be modeled as the CNF formula with unit clauses $D \wedge \neg E$. $\square$

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

9.2.2 Constraint Propagation in AND-OR-cpe

The virtue of having the mixed network view is that the constraint portion can be processed by a wide range of constraint processing techniques, both statically before search or dynamically during search [29].

We next discuss the use of constraint propagation during the look-ahead part of the search. This methods are used in any constraint or SAT/CSP (see chapters 5 and 6 in [29]). In general, constraint propagation helps to discover (using limited computation) what variable and what value to instantiate in order to avoid dead-ends as much as popssible. 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 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 search. `ConstraintPropagation` which requires polynomial time, may discover that some values 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 possible candidates to extend the search frontier. Clearly, 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 use and only the initial constraints of $\mathcal{R}$ are checked for consistency. We denote this algorithm by AO-C.

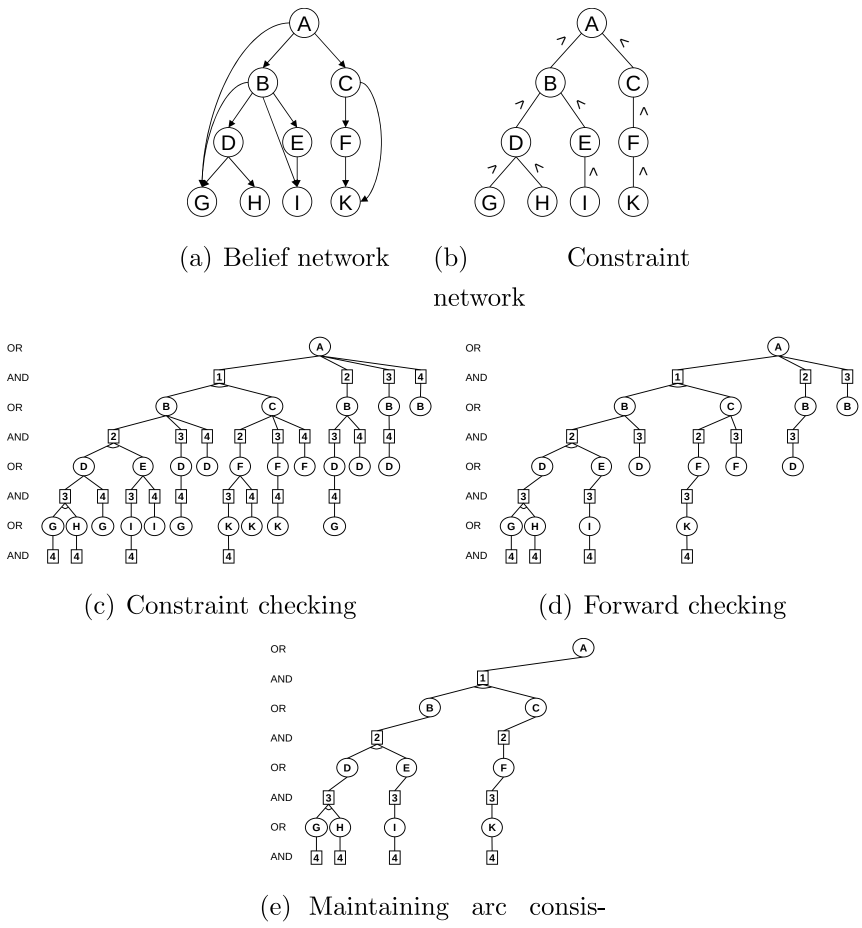
For illustration consider two forms of constraint propagation on top of AO-C. The first algorithm AO-FC, is based on *forward checking*, which is one of the weakest forms of propagation. It propagates the effect of a value selection to each future uninstantiated variable separately, and checks consistency 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 yields a more pruned search space.

SAT solvers. One possibility that was explored with success (e.g., [4]) 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 (see chapter 3 and [79]).

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 9.2.7 Figure 9.3(a) shows the belief part of a mixed network, and Figure 9.3(b) the constraint part. All variables have the same domain, {1,2,3,4}, and the constraints express "less than" relations. Figure 9.3(c) shows the search space of AO-C. Figure 9.3(d)



(a) Belief network (b) Constraint network

(c) Constraint checking (d) Forward checking

(e) Maintaining arc consistency

Figure 9.3: Traces of AND-OR-CPE with various levels of constraint propagation

shows the space traversed by AO-FC. Figure 9.3(e) shows the space when consistency is enforced with Maintaining Arc Consistency (which enforces full arc-consistency after each new instantiation of a variable). □

9.2.3 Backjumping

Backjumping algorithms [50, 76, 6, 29] are depth-first search, known as backtracking search algorithms, applied to the OR space, which use the problem structure to jump back from a dead-end as far back as possible. Backjumping can be very useful in the

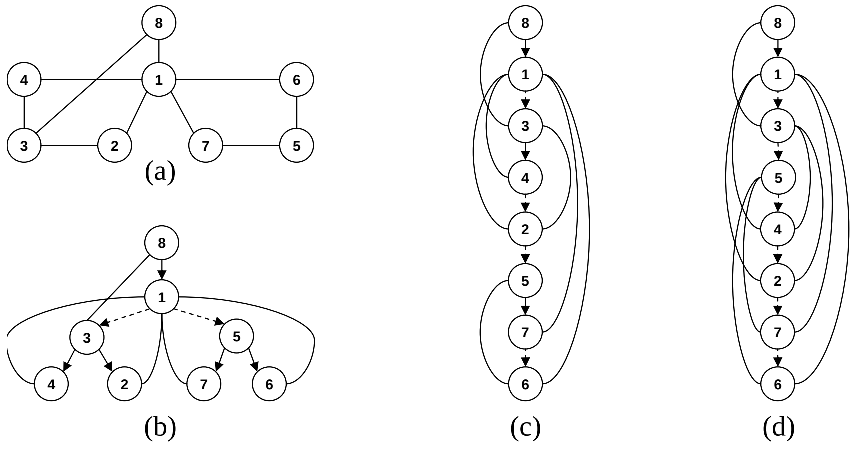


Figure 9.4: Graph-based backjumping and AND/OR search

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 the deepest variable in this induced-ancestor set.

If the ordering of the OR space is a DFS ordering of the primal graph, it is known [29] that all the backjumps are from a variable to its DFS parent. It was shown ([67]) 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. Indeed, when we search an AND/OR graph, a certain level of backjumping occurs automatically (for more details see [29]).

9.2.4 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 when focusing on the constraint portion of the mixed network.

In recent years [19, 94, 30], 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. Overall traversing the context minimal AND/OR graph and caching appropriately implements both good and nogood graph-based learning.

9.3 Summary and Bibliographical Notes for Chapters 8 and 9

Chapters 8 and 9 present 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.

It is possible to show [67] that Variable Elimination can be understood as bottom up layer by layer traversal of the context minimal AND/OR search graph. If the graphical model is strictly positive (has no determinism), then context based AND/OR search and Variable Elimination are essentially identical. When determinism is present, they may differ, because they traverse the AND/OR graph in different directions and encounter determinism (and can take advantage of it) differently. Therefore, for graphical models with no determinism, there is no principled difference between memory-intensive AND/OR search with fixed variable ordering and inference beyond: (1) different direction of exploring a common search space (top down for search vs. bottom up for inference); (2) different assumption of control strategy (depth-first for search and breadth-first for inference).

The AND/OR search space is inspired by search advances introduced sporadically in the past three decades for constraint satisfaction and more recently for probabilistic inference and for optimization tasks. Specifically, it resembles pseudo tree rearrangement [47, 48], briefly introduced two decades ago, which was adapted subsequently for distributed constraint satisfaction [17, 18] and more recently in [70], and was also shown to be related to graph-based backjumping [24]. This work was extended in [6] and more recently applied to optimization tasks [53]. Another version that can be viewed as exploring the AND/OR graphs was presented recently for constraint satisfaction [92] and for optimization [91]. Similar principles were introduced recently for probabilistic inference (in algorithm Recursive Conditioning [19] as well as in Value Elimination [40, 39]) and currently provide the backbones of the most advanced SAT solvers [94].

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

```
A constraint network $\mathcal{M} = \langle X, D, C\rangle$, 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 3: 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              $\boxed{successors(\mathtt{n}) \leftarrow \mathtt{ConstraintPropagation}(\langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle, asgn(\pi_n))}$
                // 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] Bar-Yehuda R A. Becker and D. Geiger. Random algorithms for the loop-cutset problem. In *Uncertainty in AI (UAI'99)*, pages 81–89, 1999.

[3] Srinivas M. Aji and Robert J. McEliece. The generalized distributive law. *IEEE Transactions on Information Theory*, 46(2):325–343, 2000.

[4] 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.

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

[6] 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.

[7] 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.

[8] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database ochemes. *Journal of the ACM*, 30(3):479–513, 1983.

[9] R.E. Bellman. *Dynamic Programming.* Princeton UNiversity Press, 1957.

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

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

[12] B. Bidyuk and R. Dechter. On finding w-cutset in bayesian networks. In *Uncertainty in AI (UAI04)*, 2004.

[13] S. Bistarelli. *Semirings for Soft Constraint Solving and Programming (Lecture Notes in Computer Science.* Springer-Verlag, 2004.

[14] 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.

[15] H.L. Bodlaender. Treewidth: Algorithmic techniques and results. In *MFCS-97*, pages 19–36, 1997.

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

[17] Z. Collin, R. Dechter, and S. Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of the twelfth International Conference of Artificial Intelligence (IJCAI-91)*, pages 318–324, Sidney, Australia, 1991.

[18] Z. Collin, R. Dechter, and S. Katz. Self-stabilizing distributed constraint satisfaction. *The Chicago Journal of Theoretical Computer Science*, 3(4), special issue on self-stabilization, 1999.

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

[20] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association of Computing Machinery*, 7(3), 1960.

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

[22] 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.

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

[24] R. Dechter. Constraint networks. *Encyclopedia of Artificial Intelligence*, pages 276–285, 1992.

[25] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Proc. Twelfth Conf. on Uncertainty in Artificial Intelligence*, pages 211–219, 1996.

[26] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference algorithms. In *Uncertainty in Artificial Intelligence (UAI'96)*, pages 211–219, 1996.

[27] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.

[28] R. Dechter. A new perspective on algorithms for optimizing policies under uncertainty. In *International Conference on Artificial Intelligence Planning Systems (AIPS-2000)*, pages 72–81, 2000.

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

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

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

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

[33] R. Dechter and I. Rish. Directional resolution: The davis-putnam procedure, revisited. In *Principles of Knowledge Representation and Reasoning (KR-94)*, pages 134–145, 1994.

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

[35] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

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

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

[38] S. Even. Graph algorithms. In *Computer Science Press*, 1979.

[39] S. Dalmo F. Bacchus and T. Piassi. Algorithms and complexity results for #sat and bayesian inference. In *FOCS 2003*, 2003.

[40] S. Dalmo F. Bacchus and T. Piassi. Value elimination: Bayesian inference via backtracking search. In *Uncertainty in AI (UAI03)*, 2003.

[41] G. Greco F. Scarcello and N. Leone. Weighted hypertree decompositions and optimal query plans. *PODS'04*, pages 210–221, 2004.

[42] M. Fishelson, N. Dovgolevsky, and D. Geiger. Maximum likelihood haplotyping for general pedigrees. *Human Heredity*, 2005.

[43] M. Fishelson and D. Geiger. Exact genetic linkage computations for general pedigrees. *Bioinformatics*, 2002.

[44] Myan Fishelson and Dan Geiger. Optimizing exact genetic linkage computations. *RECOMB*, pages 114–121, 2003.

[45] Freuder. Partial constraint satisfaction. *Artificial Intelligence*, 50:510–530, 1992.

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

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

[48] E. C. Freuder and M. J. Quinn. The use of lineal spanning trees to represent constraint satisfaction problems. Technical Report 87-41, University of New Hampshire, Durham, 1987.

[49] M. R Garey and D. S. Johnson. Computers and intractability: A guide to the theory of NP-completeness. In *W. H. Freeman and Company, New York*, 1979.

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

[51] 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.

[52] R. A. Howard and J. E. Matheson. *Influence diagrams.* 1984.

[53] P. Meseguer J. Larrosa and M Sanchez. Pseudo-tree search with soft constraints. In *European conference on Artificial Intelligence (ECAI02)*, 2002.

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

[55] 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.

[56] J. Larrosa K. Kask, R. Dechter and A. Dechter. Unifying tree-decompositions for reasoning in graphical models. *Artificial Intelligence*, 166(1-2):165–193, 2005.

[57] 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.

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

[59] J Larrosa and R. Dechter. Dynamic combination of search and variable-elimination in csp and max-csp. *Submitted*, 2001.

[60] Javier Larrosa and Rina Dechter. Boosting search with variable elimination in constraint optimization and constraint satisfaction problems. *Constraints*, 8(3):303–326, 2003.

[61] J.-L. Lassez and M. Mahler. On fourier's algorithm for linear constraints. *Journal of Automated Reasoning*, 9, 1992.

[62] 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.

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

[64] 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.

[65] Radu Marinescu and Rina Dechter. And/or branch-and-bound search for combinatorial optimization in graphical models. *Artif. Intell.*, 173(16-17):1457–1491, 2009.

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

[67] 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.

[68] Robert Mateescu, Kalev Kask, Vibhav Gogate, and Rina Dechter. Join-graph propagation algorithms. *J. Artif. Intell. Res. (JAIR)*, 37:279–328, 2010.

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

[70] P. J. Modi, W. Shena, M. Tambea, and M. Yokoo. Adopt: asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161:149–180, 2005.

[71] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(66):95–132, 1974.

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

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

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

[75] L. Portinale and A. Bobbio. Bayesian networks for dependency analysis: an application to digital control. In *Proceedings of the 15th Conference on Uncertainty in Artifi cial Intelligence (UAI99)*, pages 551–558, 1999.

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

[77] 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.

[78] 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.

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

[80] Irina Rish and Rina Dechter. Resolution versus search: Two strategies for sat. *J. Autom. Reasoning*, 24(1/2):225–275, 2000.

[81] D. G. Corneil S. A. Arnborg and A. Proskourowski. Complexity of finding embeddings in a k-tree. *SIAM Journal of Discrete Mathematics.*, 8:277–284, 1987.

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

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

[84] R. Seidel. A new method for solving constraint satisfaction problems. In *International Joint Conference on Artificial Intelligece (Ijcai-81)*, pages 338–342, 1981.

[85] G. R. Shafer and P.P. Shenoy. Axioms for probability and belief-function propagation. volume 4, 1990.

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

[87] P.P. Shenoy. Binary join trees for computing marginals in the shenoy-shafer architecture. *International Journal of approximate reasoning*, pages 239–263, 1997.

[88] 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.

[89] Kirill Shoikhet and Dan Geiger. A practical algorithm for finding optimal triangulations. In *AAAI/IAAI*, pages 185–190, 1997.

[90] 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.

[91] C. Terrioux and P. Jegou. Bounded backtracking for the valued constraint satsfaction problems. In *Constraint Progamming (CP2003)*, pages 709–723, 2003.

[92] C. Terrioux and P. Jegou. Hybrid backtracking bounded by tree-decomposition of constraint networks. In *Artificial Intelligence*, 2003.

[93] 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.

[94] 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.

[95] Yair Weiss and Judea Pearl. Belief propagation: technical perspective. *Commun. ACM*, 53(10):94, 2010.

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