A Comparison of Time-Space Schemes for Graphical Models

Robert Mateescu and **Rina Dechter**
Donald Bren School of Information and Computer Science
University of California, Irvine, CA 92697-3425
{*mateescu,dechter*}@*ics.uci.edu*

Abstract

We investigate three parameterized algorithmic schemes for graphical models that can accommodate trade-offs between time and space: 1) AND/OR Adaptive Caching (**AOC(i)**); 2) Variable Elimination and Conditioning (**VEC(i)**); and 3) Tree Decomposition with Conditioning (**TDC(i)**). We show that **AOC(i)** is better than the vanilla versions of both **VEC(i)** and **TDC(i)**, and use the guiding principles of **AOC(i)** to improve the other two schemes. Finally, we show that the improved versions of **VEC(i)** and **TDC(i)** can be simulated by **AOC(i)**, which emphasizes the unifying power of the AND/OR framework.

1 Introduction

This paper addresses some long-standing questions regarding the computational merits of several time-space sensitive algorithms for graphical models. All exact algorithms for graphical models, either search or inference based, are time and space exponentially bounded by the treewidth of the problem. For real life networks with large treewidth, the space limitation can be quite severe, therefore schemes that can trade space for time are of outmost importance.

In the past ten years, four types of algorithms have emerged, based on: (1) cycle-cutset and w-cutset [Pearl, 1988; Dechter, 1990]; (2) alternating conditioning and elimination controlled by induced-width w [Rish and Dechter, 2000; Larrosa and Dechter, 2002; Fishelson and Geiger, 2002]; (3) recursive conditioning [Darwiche, 2001], which was recently recast as context-based AND/OR search [Dechter and Mateescu, 2004]; (4) varied separator-sets for tree decompositions [Dechter and Fattah, 2001]. The question is how do all these methods compare and, in particular, is there one that is superior? A brute-force analysis of time and space complexities of the respective schemes does not settle the question. For example, if we restrict the available space to be linear, the cycle-cutset scheme is exponential in the cycle-cutset size while recursive conditioning is exponential in the depth of the pseudo tree (or d-tree) that drives the computation. However some graphs have small cycle-cutset and larger tree depth, while others have large cycle-cutsets and small tree depth (e.g., grid-like chains). The immediate conclusion seems to be that the methods are not comparable.

In this paper we show that by looking at all these schemes side by side, and analyzing them using the context minimal AND/OR graph data structure [Mateescu and Dechter, 2005b], each of these schemes can be improved via the AND/OR search principle and by careful caching, to the point that they all become identically good. Specifically, we show that the new algorithm *Adaptive Caching* (**AOC(i)**), inspired by the recently proposed AND/OR cutset conditioning [Mateescu and Dechter, 2005a] (improving cutset, and w-cutset schemes), can simulate any execution of alternating elimination and conditioning, if the latter is augmented with AND/OR search over the conditioning variables, and can also simulate any execution of separator controlled tree-clustering schemes [Dechter and Fattah, 2001], if the clusters are augmented with AND/OR cutset search, rather than regular search, as was initially proposed.

All the analysis is done assuming that the problem contains no determinism. When the problem has determinism all these schemes become incomparable, as was shown in [Mateescu and Dechter, 2005b], because they exploit the deterministic information in reversed ordering of variables.

2 Preliminaries

This section provides the basic definitions.

DEFINITION **1 (graphical model)** *A* graphical model *is a 3-tuple $\mathcal{M} = \langle \mathbf{X},\mathbf{D},\mathbf{F}\rangle$, where: $\mathbf{X} = \{X_1,\ldots,X_n\}$ is a set of variables; $\mathbf{D} = \{D_1,\ldots,D_n\}$ is the set of their finite domains of values; $\mathbf{F} = \{f_1,\ldots,f_r\}$ is a set of real-valued functions defined on variables from $\mathbf{X}$.*

DEFINITION **2 (primal graph)** *The* primal graph *of a graphical model is an undirected graph, $G = (\mathbf{X}, E)$, that has variables as its vertices and an edge connecting any two variables that appear in the scope (set of arguments) of the same function.*

DEFINITION **3 (pseudo tree)** *A* pseudo tree *of a graph $G = (\mathbf{X}, E)$ is a rooted tree $\mathcal{T}$ having the same set of nodes $\mathbf{X}$, such that every arc in E is a back-arc in $\mathcal{T}$ (i.e., it connects nodes on the same path from root).*

DEFINITION **4 (induced graph, induced width, treewidth)** *An* ordered graph *is a pair (G, d), where G is an undirected* *graph, and* $d = (X_1, ..., X_n)$ *is an ordering of the nodes. The* width of a node *in an ordered graph is the number of neighbors that precede it in the ordering. The* width of an ordering d*, denoted by* $w(d)$*, is the maximum width over all nodes. The* induced width of an ordered graph*,* $w^*(d)$*, is the width of the induced ordered graph obtained as follows: for each node, from last to first in d, its preceding neighbors are connected in a clique. The* induced width of a graph*,* w^**, is the minimal induced width over all orderings. The induced width is also equal to the* treewidth *of a graph.*

3 Description of Algorithms

In this section we describe the three algorithms that will be compared. They are all parameterized memory intensive algorithms that need to use space in order to achieve the worst case time complexity of $O(n\ k^{w^*})$, where k bounds domain size, and w^* is the treewidth of the primal graph. The task that we consider is one that is #P-hard (e.g., belief updating in Bayesian networks, counting solutions in SAT or constraint networks). We also assume that the model has no determinism (i.e., all tuples have a strictly positive probability).

The algorithms we discuss work by processing variables either by *elimination* or by *conditioning*. These operations have an impact on the primal graph of the problem. When a variable is eliminated, it is removed from the graph along with its incident edges, and its neighbors are connected in a clique. When it is conditioned, it is simply removed from the graph along with its incident edges.

The algorithms we discuss typically depend on a variable ordering $d = (X_1, ..., X_n)$. Search proceeds by instantiating variables from X_1 to X_n, while Variable Elimination processes the variables backwards, from X_n to X_1. Given a graph G and an ordering d, an elimination tree, denoted by $\mathcal{T}(G, d)$, is uniquely defined by the Variable Elimination process. $\mathcal{T}(G, d)$ is also a valid pseudo tree to drive the AND/OR search. Note however that several orderings can give rise to the same elimination tree.

3.1 AND/OR Search Space

The AND/OR search space is a recently introduced [Dechter and Mateescu, 2004; Mateescu and Dechter, 2005b; 2005a] unifying framework for advanced algorithmic schemes for graphical models. Its main virtue consists in exploiting independencies between variables during search, which can provide exponential speedups over traditional search methods oblivious to problem structure.

Given a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, its primal graph G and a pseudo tree $\mathcal{T}$ of G (see Figure 6 for an example of a pseudo tree), the associated AND/OR search tree has alternating levels of OR and AND nodes (see Figure 7 for an example of OR and AND nodes; however, the figure shows a search graph, not a tree). 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 in the domains of the variables. The structure of the AND/OR search tree is based on the underlying pseudo tree $\mathcal{T}$. The root of the AND/OR search tree is an OR node labeled with the root of $\mathcal{T}$. The children of an OR node X_i are AND nodes labeled with assignments $\langle X_i, x_i \rangle$. The children of an AND node $\langle X_i, x_i \rangle$ are OR nodes labeled with the children of variable X_i in the pseudo tree $\mathcal{T}$.

The AND/OR search tree can be traversed by a depth first search algorithm, thus using linear space. It was already shown [Freuder and Quinn, 1985; Bayardo and Miranker, 1996; Darwiche, 2001; Mateescu and Dechter, 2005a] that:

THEOREM **1** *Given a graphical model* $\mathcal{M}$ *and a pseudo tree* $\mathcal{T}$ *of depth* m*, the size of the AND/OR search tree based on* $\mathcal{T}$ *is* $O(n\ k^m)$*, where* k *bounds the domains of variables. A graphical model of treewidth* w^* *has a pseudo tree of depth at most* $w^* \log n$*, therefore it has an AND/OR search tree of size* $O(n\ k^{w^* \log n})$*.*

The AND/OR search tree may contain nodes that root identical conditioned subproblems. These nodes are said to be *unifiable*. When unifiable nodes are merged, the search space becomes a graph. Its size becomes smaller at the expense of using additional memory by the search algorithm. The depth first search algorithm can therefore be modified to cache previously computed results, and retrieve them when the same nodes are encountered again. Some unifiable nodes can be identified based on their *contexts* [Darwiche, 2001]. We can define graph based contexts for both OR nodes and AND nodes, just by expressing the set of ancestor variables in $\mathcal{T}$ whose assignment would completely determine the conditioned subproblem. However, it can be shown that using caching based on OR contexts makes caching based on AND contexts redundant, so we will only use *OR caching*.

DEFINITION **5 (OR context)** *Given a pseudo tree* $\mathcal{T}$ *of an AND/OR search space, the context of an OR node* X*, denoted by* $context(X) = [X_1 \dots X_k]$*, is the set of ancestors of* X *in* $\mathcal{T}$ *ordered descendingly, that are connected in the primal graph to* X *or to descendants of* X*.*

It is easy to verify that the context of X d-separates [Pearl, 1988] the subproblem below X from the rest of the network. The *context minimal* AND/OR graph is obtained by merging all the context unifiable OR nodes. An example will appear later in Figure 7. It was already shown that [Bayardo and Miranker, 1996; Dechter and Mateescu, 2004]:

THEOREM **2** *Given a graphical model* $\mathcal{M}$*, its primal graph* G *and a pseudo tree* $\mathcal{T}$*, the size of the context minimal AND/OR search graph based on* $\mathcal{T}$ *is* $O(n\ k^{w^*_{\mathcal{T}}(G)})$*, where* $w^*_{\mathcal{T}}(G)$ *is the induced width of* G *over the depth first traversal of* $\mathcal{T}$*, and* k *bounds the domain size.*

3.2 AND/OR Cutset Conditioning - AOCutset(i)

AND/OR Cutset Conditioning (**AOCutset(i)**) [Mateescu and Dechter, 2005a] is a search algorithm that combines AND/OR search spaces with cutset conditioning. The conditioning (cutset) variables form a *start* pseudo tree. The remaining variables (not belonging to the cutset), have bounded conditioned context size that can fit in memory.

DEFINITION **6 (start pseudo tree)** *Given a primal graph* G *and a pseudo tree* $\mathcal{T}$ *of* G*, a* start pseudo tree $\mathcal{T}_{start}$ *is a connected subgraph of* $\mathcal{T}$ *that contains the root of* $\mathcal{T}$*.*

Algorithm **AOCutset(i)** depends on a parameter i that bounds the maximum size of a context that can fit in memory. Given a graphical model and a pseudo tree $\mathcal{T}$, we first find a start pseudo tree $\mathcal{T}_{start}$ such that the context of any node not in $\mathcal{T}_{start}$ contains at most i variables that are not in $\mathcal{T}_{start}$. This can be done by starting with the root of $\mathcal{T}$ and then including as many descendants as necessary in the start pseudo tree until the previous condition is met. $\mathcal{T}_{start}$ now forms the cutset, and when its variables are instantiated, the remaining conditioned subproblem has induced width bounded by i. The cutset variables can be explored by linear space (no caching) AND/OR search, and the remaining variables by using full caching, of size bounded by i. The cache tables need to be deleted and reallocated for each new conditioned subproblem (i.e., each new instantiation of the cutset variables).

3.3 Algorithm AOC(i) - Adaptive Caching

The cutset principle inspires a new algorithm, based on a more refined caching scheme for AND/OR search, which we call *Adaptive Caching* - **AOC(i)** (in the sense that it adapts to the available memory), that caches some values even at nodes with contexts greater than the bound i that defines the memory limit. Lets assume that $context(X) = [X_1 \dots X_k]$ and $k > i$. During search, when variables $X_1, \dots, X_{k-i}$ are instantiated, they can be regarded as part of a cutset. The problem rooted by X_{k-i+1} can be solved in isolation, like a subproblem in the cutset scheme, after variables $X_1, \dots, X_{k-i}$ are assigned their current values in all the functions. In this subproblem, $context(X) = [X_{k-i+1} \dots X_k]$, so it can be cached within space bounded by i. However, when the search retracts to X_{k-i} or above, the cache table for X needs to be deleted and will be reallocated when a new subproblem rooted at X_{k-i+1} is solved.

DEFINITION 7 **(i-context, flag)** *Given a graphical model, a pseudo tree $\mathcal{T}$, a variable X and $context(X) = [X_1 \dots X_k]$, the* i-context *of X is:*

$$i\text{-}context(X) = \begin{cases} [X_{k-i+1} \dots X_k], & if \quad i < k \\ context(X), & if \quad i \geq k \end{cases}$$

X_i is called the ***flag*** *of $i\text{-}context(X)$.*

```
Algorithm AOC(i)
  input  : $\mathcal{M}=\langle \mathbf{X},\mathbf{D},\mathbf{F}\rangle$; $G=(\mathbf{X},E)$; $d=(X_1,\dots,X_n)$; $i$
  output : Updated belief for $X_1$
1 Let $\mathcal{T} = \mathcal{T}(G,d)$ // create elimination tree
  for each $X \in \mathbf{X}$ do
2     allocate a table for $i\text{-}context(X)$
3 Initialize search with root of $\mathcal{T}$;
4 while search not finished do
5     Pick next successor not yet visited    // EXPAND;
6     Purge cache tables that are not valid;
7     if value in cache then
8         retrieve value; mark successors as visited;
9     while all successors visited do    // PROPAGATE
10        Save value in cache;
11        Propagate value to parent;
```

The high level pseudocode for **AOC(i)** is given here. The algorithm works similar to AND/OR search based on full context. The difference is in the management of cache tables. Whenever a variable X is instantiated (when an AND node is reached), the cache table is purged (reinitialized with a neutral value) for any variable Y such that X is the flag of $i\text{-}context(Y)$ (line 6). Otherwise, the search proceeds as usual, retrieving values from cache if possible (line 7) or else continuing to expand, and propagating the values up when the search is completed for subproblem below (line 11). We do not detail here the alternation of OR and AND type of nodes.

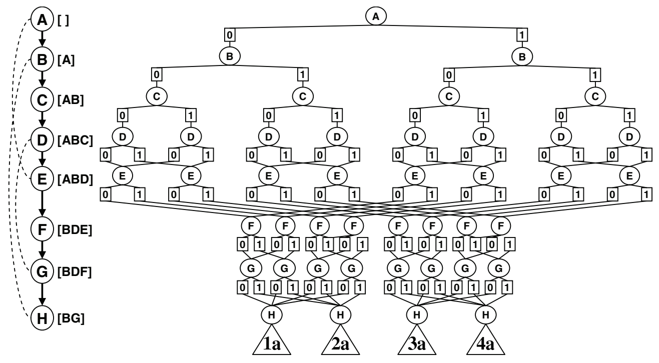

Figure 1: Context minimal graph (full caching)

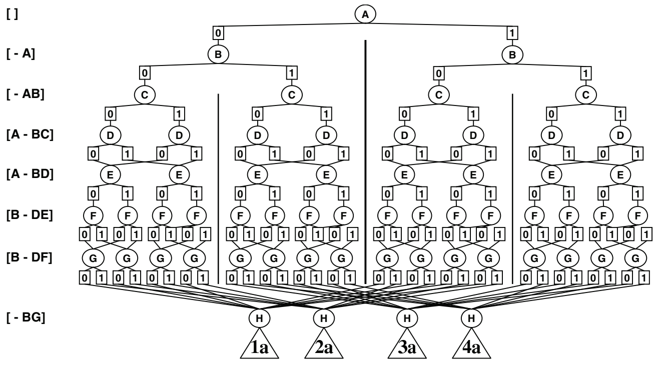

Figure 2: **AOC(2)** graph (Adaptive Caching)

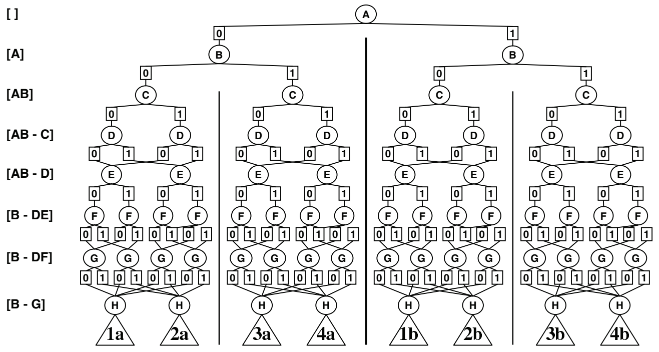

Figure 3: **AOCutset(2)** graph (AND/OR Cutset)

Example 1 *We will clarify here the distinction between AND/OR with full caching, AND/OR Cutset and AND/OR Adaptive Caching. We should note that the scope of a cache* *table is always a subset of the variables on the current path in the pseudo tree. Therefore, the caching method (e.g., full caching based on context, cutset conditioning cache, adaptive caching) is an orthogonal issue to that of the search space decomposition. We will show an example based on an OR search space (pseudo tree is a chain), and the results will carry over to the AND/OR search space.*

*Figure 1 shows a pseudo tree, with binary valued variables, the context for each variable, and the context minimal graph. If we assume the bound $i = 2$, some of the cache tables don't fit in memory. We could in this case use **AOCutset(2)**, shown in Figure 3, that takes more time, but can execute in the bounded memory. The cutset in this case is made of variables A and B, and we see four conditioned subproblems, the four columns, that are solved independently from one another (there is no sharing of subgraphs). Figure 2 shows **AOC(2)**, which falls between the previous two. It uses bounded memory, takes more time than full caching (as expected), but less time than **AOCutset(2)** (because the graph is smaller). This can be achieved because Adaptive Caching allows the sharing of subgraphs. Note that the cache table of H has the scope $[BG]$, which allows merging.*

3.4 Variable Elimination and Conditioning - VEC(i)

Variable Elimination and Conditioning (**VEC**) [Rish and Dechter, 2000; Larrosa and Dechter, 2002] is an algorithm that combines the virtues of both inference and search. One of its remarkably successful applications is the genetic linkage analysis software Superlink [Fishelson and Geiger, 2002]. **VEC** works by interleaving elimination and conditioning of variables. Typically, given an ordering, it prefers the elimination of a variable whenever possible, and switches to conditioning whenever space limitations require it, and continues in the same manner until all variables have been processed. We say that the conditioning variables form a *conditioning set*, or *cutset* (this can be regarded as a w-*cutset*, where the w defines the induced width of the problems that can be handled by elimination). The pseudocode for the vanilla version, called **VEC-OR(i)** because the cutset is explored by OR search rather than AND/OR, is shown below:

```
Algorithm VEC-OR(i)
  input  : $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$; $d = (X_1, \ldots, X_n)$
  output : Updated belief for $X_1$
1 if ($context(X_n) \leq i$) then
2     eliminate $X_i$;
3     call VEC-OR(i) on reduced problem
4 else for each $x_n \in D_n$ do
5     assign $X_n = x_n$;
6     call VEC-OR(i) on the conditioned subproblem
```

When there are no conditioning variables, **VEC** becomes the well known Variable Elimination (**VE**) algorithm. In this case **AOC** also becomes the usual AND/OR graph search (**AO**), and it was shown [Mateescu and Dechter, 2005b] that:

THEOREM **3 (VE and AO are identical)** *Given a graphical model with no determinism and a pseudo tree, **VE** traverses the full context minimal graph bottom-up by layers (breadth first), while **AO** is a top-down depth-first search that explores (and records) the full context minimal graph as well.*

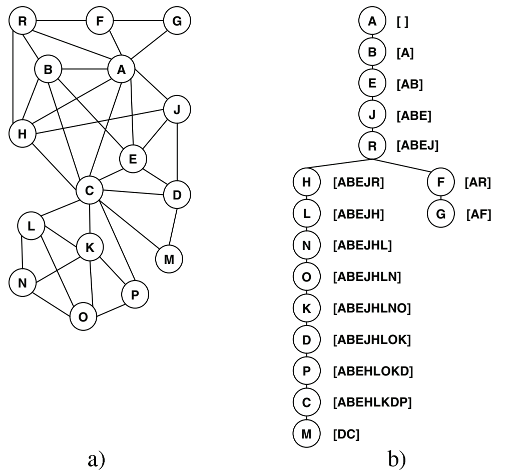

Figure 4: Primal graph and pseudo tree

3.5 Tree Decomposition with Conditioning - TDC

One of the most widely used methods of processing graphical models, especially belief networks, is tree clustering (also known as join tree or junction tree algorithm [Lauritzen and Spiegelhalter, 1988]). The work in [Dechter and Fattah, 2001] presents an algorithm called *directional join tree clustering*, that corresponds to an inward pass of messages towards a root in the regular tree clustering algorithm. If space is not sufficient for the separators in the tree decomposition, then [Dechter and Fattah, 2001] proposes the use of secondary join trees, which simply combine any two neighboring clusters whose separator is too big to be stored. The resulting algorithm that uses less memory at the expense of more time is called *space based join tree clustering*,

The computation in each cluster can be done by any suitable method. The obvious one would be to simply enumerate all the instantiations of the cluster, which corresponds to an OR search over the variables of the cluster. A more advanced method advocated by [Dechter and Fattah, 2001] is the use of cycle cutset inside each cluster. We can improve the cycle cutset scheme first by using an AND/OR search space, and second by using Adaptive Caching bounded by i, rather than simple AND/OR Cutset in each cluster. We call the resulting method *tree decomposition with conditioning* (**TDC(i)**).

4 AOC(i) Compared to VEC(i)

We will begin by following an example. Consider the graphical model given in Figure 4a having binary variables, the ordering $d_1 = (A, B, E, J, R, H, L, N, O, K, D, P, C, M$-$, F, G)$, and the space limitation $i = 2$. The pseudo tree corresponding to this ordering is given in Figure 4b. The context of each node is shown in square brackets.

If we apply **VEC** along d_1 (eliminating from last to first), variables G, F and M can be eliminated. However, C cannot be eliminated, because it would produce a function with scope equal to its context, $[ABEHLKDP]$, violating the

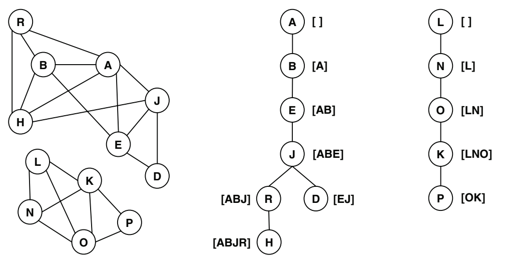

Figure 5: Components after conditioning on C

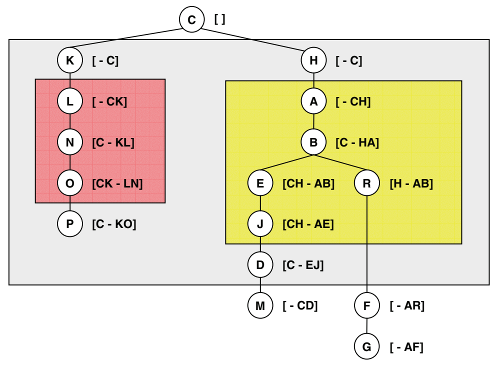

Figure 6: Pseudo tree for **AOC(2)**

bound $i = 2$. **VEC** switches to conditioning on C and all the functions that remain to be processed are modified accordingly, by instantiating C. The primal graph has two connected components now, shown in Figure 5. Notice that the pseudo trees are based on this new graph, and their shape changes from the original pseudo tree.

Continuing with the ordering, P and D can be eliminated (one variable from each component), but then K cannot be eliminated. After conditioning on K, variables O, N and L can be eliminated (all from the same component), then H is conditioned (from the other component) and the rest of the variables are eliminated. To highlight the conditioning set, we will box its variables when writing the ordering, $d_1 = (A, B, E, J, R, \boxed{H}, L, N, O, \boxed{K}, D, P, \boxed{C}, M, F, G)$.

If we take the conditioning set $[HKC]$ in the order imposed on it by d_1, reverse it and put it at the beginning of the ordering d_1, then we obtain:

$$d_2 = \left(\boxed{C}, \underline{\left[\boxed{K}, \underline{\left[\boxed{H}, \underline{[A,B,E,J,R]}_H, L,N,O\right]}_K, D,P\right]}_C, M,F,G\right)$$

where the indexed squared brackets together with the underlines represent subproblems that need to be solved multiple times, for each instantiation of the index variable.

So we started with d_1 and bound $i = 2$, then we identified the corresponding conditioning set $[HKC]$ for **VEC**, and from this we arrived at d_2. We are now going to use d_2 to build the pseudo tree that guides **AOC(2)**, given in Figure 6. The outer box corresponds to the conditioning of C. The inner boxes correspond to conditioning on K and H, respectively. The context of each node is given in square brackets, and the *2-context* is on the right side of the dash. For example, $context(J) = [CH\text{-}AE]$, and $2\text{-}context(J) = [AE]$. The context minimal graph corresponding to the execution of **AOC(2)** is shown in Figure 7.

We can follow the execution of both **AOC** and **VEC** along this context minimal graph. After conditioning on C, **VEC** solves two subproblems (one for each value of C), which are the ones shown on the large rectangles. The vanilla version **VEC-OR** is less efficient than **AOC**, because it uses an OR search over the cutset variables, rather than AND/OR. In our example, the subproblem on A, B, E, J, R would be solved eight times by **VEC-OR**, once for each instantiation of C, K and H, rather than four times. It is now easy to make the first improvement to **VEC**, so that it uses an AND/OR search over the conditioning variables, an algorithm we call **VEC-AO(i)**, by changing line 6 of **VEC-OR** to:

Algorithm **VEC-AO(i)**

```
...
6 call VEC-AO(i) on each connected
  component of conditioned subproblem separately;
```

Let's look at one more condition that needs to be satisfied for the two algorithms to be identical. If we change the ordering to $d_3 = (A, B, E, J, R, \boxed{H}, L, N, O, \boxed{K}, D, P, F, G, \boxed{C}, M)$, ($F$ and G are eliminated after conditioning on C), then the pseudo tree is the same as before, and the context minimal graph for **AOC** is still the one shown in Figure 7. However, **VEC-AO** would require more effort, because the elimination of G and F is done twice now (once for each instantiation of C), rather than once as was for ordering d_1. This shortcoming can be eliminated by defining a pseudo tree based version for **VEC**, rather than one based on an ordering. The final algorithm, **VEC(i)** is given below (where $N_G(X_i)$ is the set of neighbors of X_i in the graph G). Note that the guiding pseudo tree is regenerated after each conditioning.

Algorithm **VEC(i)**

```
input  : $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$; $G = (\mathbf{X}, E)$; $d = (X_1, \ldots, X_n)$; $i$
output : Updated belief for $X_1$
Let $\mathcal{T} = \mathcal{T}(G, d)$  // create elimination tree;
while $\mathcal{T}$ not empty do
    if $((\exists X_i$ leaf in $\mathcal{T}) \wedge (|N_G(X_i)| \le i))$ then eliminate $X_i$
    else pick $X_i$ leaf from $\mathcal{T}$;
        for each $x_i \in D_i$ do
            assign $X_i = x_i$;
            call VEC(i) on each connected component of
            conditioned subproblem
        break;
```

Based on the previous example, we can prove:

THEOREM **4 (AOC(i) simulates VEC(i))** *Given a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$ with no determinism and an execution of* ***VEC(i)****, there exists a pseudo tree that guides an execution of* ***AOC(i)*** *that traverses the same context minimal graph.*

Proof: The pseudo tree of **AOC(i)** is obtained by reversing the conditioning set of **VEC(i)** and placing it at the beginning of the ordering. The proof is by induction on the number of conditioning variables, by comparing the corresponding contexts of each variable.

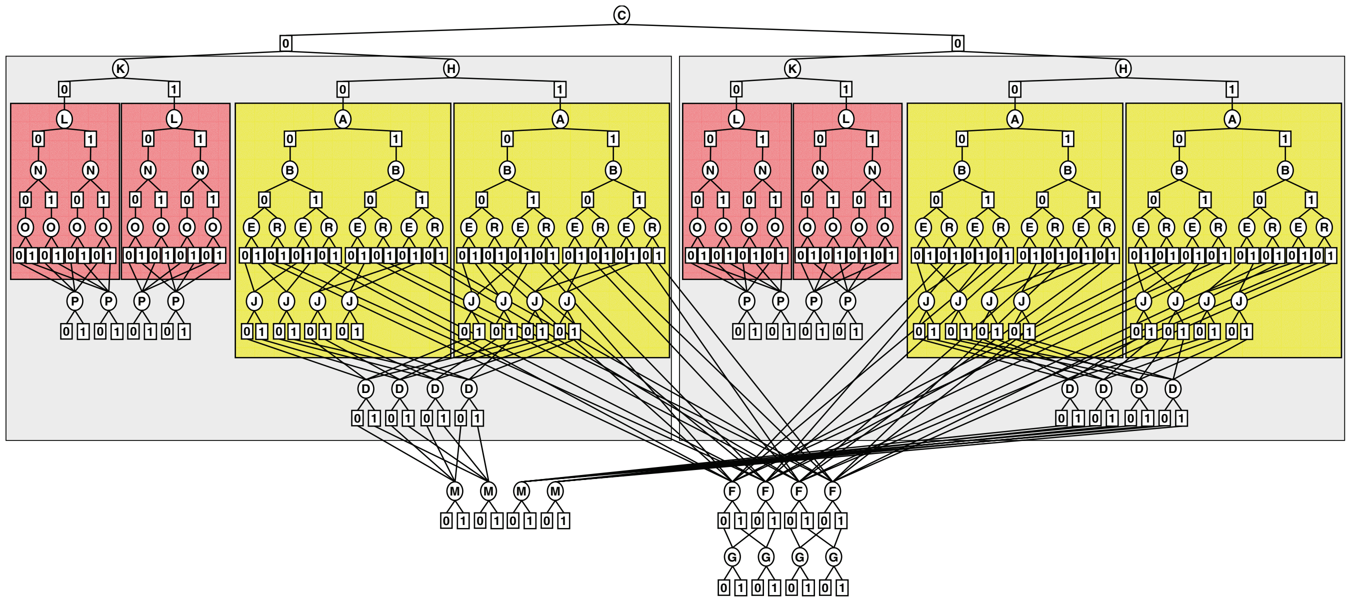

Figure 7: Context minimal graph

Basis step. If there is no conditioning variable, Theorem 3 applies. If there is only one conditioning variable. Given the ordering $d = (X_1, \ldots, X_j, \ldots, X_n)$, let's say X_j is the conditioning variable.

a) Consider $X \in \{X_{j+1}, \ldots, X_n\}$. The function recorded by **VEC(i)** when eliminating X has the scope equal to the context of X in **AOC(i)**.

b) For X_j, both **VEC(i)** and **AOC(i)** will enumerate its domain, thus making the same effort.

c) After X_j is instantiated by **VEC(i)**, the reduced subproblem (which may contain multiple connected components) can be solved by variable elimination alone. By Theorem 3, variable elimination on this portion is identical to AND/OR search with full caching, which is exactly **VEC(i)** on the reduced subproblem.

From a), b) and c) it follows that **VEC(i)** and **AOC(i)** are identical if there is only one conditioning variable.

Inductive step. We assume that **VEC(i)** and **AOC(i)** are identical for any graphical model if there are at most k conditioning variables, and have to prove that the same is true for $k + 1$ conditioning variables.

If the ordering is $d = (X_1, \ldots, X_j, \ldots, X_n)$ and X_j is the last conditioning variable in the ordering, it follows (similar to the basis step) that **VEC(i)** and **AOC(i)** traverse the same search space with respect to variables $\{X_{j+1}, \ldots, X_n\}$, and also for X_j. The remaining conditioned subproblem now falls under the inductive hypothesis, which concludes the proof. Note that it is essential that **VEC(i)** uses AND/OR over cutset, and is pseudo tree based, otherwise **AOC(i)** is better. □

5 AOC(i) Compared to TDC(i)

We will look again at the example from Figures 6 and 7, and the ordering d_2. It is well known that a tree decomposition

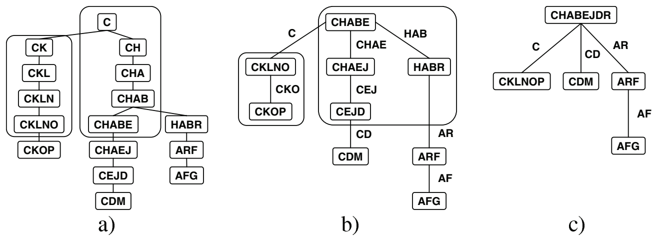

Figure 8: Tree decompositions: a) for d_2; b) maximal cliques only; c) secondary tree for $i = 2$

corresponding to d_2 can be obtained by inducing the graph along d_2 (from last to first), and then picking as clusters each node together with its parents in the induced graph, and connecting each cluster to that of its latest (in the ordering) induced parent. Because the induced parent set is equal to the context of a node, the method above is equivalent to creating a cluster for each node in the pseudo tree from Figure 6, and labeling it with the variable and its context. The result is shown in Figure 8a. A better way to build a tree decomposition is to pick only the maximal cliques in the induced graph, and this is equivalent to collapsing neighboring subsumed clusters from Figure 8a, resulting in the tree decomposition in Figure 8b. If we want to run **TDC** with bound $i = 2$, some of the separators are bigger than 2, so a secondary tree is obtained by merging clusters adjacent to large separators, obtaining the tree in Figure 8c. **TDC(2)** now runs by sending messages upwards, toward the root. Its execution, when augmented with AND/OR cutset in each cluster, can also be followed on the context minimal graph in Figure 7. The separators $[AF]$, $[AR]$ and $[CD]$ correspond to the contexts of G, F and M. The root cluster $[CHABEJDR]$ corresponds to the part of the context minimal graph that contains all these variables. If this cluster would be processed by enumeration (OR search), it would result in a tree with $2^8 = 256$ leaves. However, when explored by AND/OR search with adaptive caching the context minimal graph of the cluster is much smaller, as can be seen in Figure 7. By comparing the underlying context minimal graphs, it can be shown that:

THEOREM **5** *Given a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$ with no determinism, and an execution of* ***TDC(i)****, there exists a pseudo tree that guides an execution of* ***AOC(i)*** *that traverses the same context minimal graph.*

Proof: Algorithm **TDC(i)** is already designed to be an improvement over *space based join tree clustering* (Section 3.5), in that it uses AND/OR Adaptive Caching (rather than cutset conditioning) inside each cluster to compute the messages that are sent. **TDC(i)** is based on a rooted tree decomposition, which can serve as the skeleton for the underlying pseudo tree. Each cluster has its own pseudo tree to guide the AND/OR Adaptive Caching. Each message sent between neighboring clusters couples all the variables in its scope, therefore the scope variables have to appear on a chain at the top of the pseudo tree of the cluster where the message is generated, and also they have to appear on a chain in the pseudo tree of the cluster where the message is received. This implies that two neighboring clusters can agree on an ordering of the common variables (since we assume no determinism, the order of variables on a chain doesn't change the size of the context minimal graph). All this allows us to build a common pseudo tree for two neighboring clusters, by superimposing the common variables in their respective pseudo trees (which are the variables in the separator between the clusters). In this way we can build the pseudo tree for the entire problem. Now, for any variable X_i in the problem pseudo tree, $i\text{-}context(X_i)$ is the same as it was in the highest cluster (closest to the root of the tree decomposition) where X_i is mentioned. From this, it follows that **AOC(i)** based on the pseudo tree for the entire problem traverses the same context minimal graph as **TDC(i)**. □

6 Conclusion

We have compared three parameterized algorithmic schemes for graphical models that can accommodate time-space tradeoffs. They have all emerged from seemingly different principles: **AOC(i)** is search based, **TDC(i)** is inference based and **VEC(i)** combines search and inference.

We show that if the graphical models contain no determinism, **AOC(i)** can have a smaller time complexity than the vanilla versions of both **VEC(i)** and **TDC(i)**. This is due to a more efficient exploitation of the graphical structure of the problem through AND/OR search, and the adaptive caching scheme that benefits from the cutset principle. These ideas can be used to enhance **VEC(i)** and **TDC(i)**. We show that if **VEC(i)** uses AND/OR search over the conditioning set and is guided by the pseudo tree data structure, then there exists an execution of **AOC(i)** that is identical to it. We also show that if **TDC(i)** processes clusters by AND/OR search with adaptive caching, then there exists an execution of **AOC(i)** identical to it. AND/OR search with adaptive caching (**AOC(i)**) emerges therefore as a unifying scheme, never worse than the other two. All the analysis was done by using the context minimal data structure, which provides a powerful methodology for comparing the algorithms.

When the graphical model contains determinism, all the above schemes become incomparable. This is due to the fact that they process variables in reverse orderings, and will encounter and exploit deterministic information differently.

Acknowledgments

Research supported in part by the NSF grant IIS-0412854.

References

[Bayardo and Miranker, 1996] R. Bayardo and D. Miranker. A complexity analysis of space-bound learning algorithms for the constraint satisfaction problem. In *AAAI'96*, pages 298–304, 1996.

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

[Dechter and Fattah, 2001] R. Dechter and Y. El Fattah. Topological parameters for time-space tradeoff. *Artificial Intelligence*, 125:93–188, 2001.

[Dechter and Mateescu, 2004] Rina Dechter and Robert Mateescu. Mixtures of deterministic-probabilistic networks and their and/or search space. In *UAI'04*, pages 120–129, 2004.

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

[Fishelson and Geiger, 2002] M. Fishelson and D. Geiger. Exact genetic linkage computations for general pedigrees. *Bioinformatics*, 18(1):189–198, 2002.

[Freuder and Quinn, 1985] E. C. Freuder and M. J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *IJCAI'85*, pages 1076–1078, 1985.

[Larrosa and Dechter, 2002] J. Larrosa and R. Dechter. Boosting search with variable-elimination. *Constraints*, 7(3-4):407–419, 2002.

[Lauritzen and Spiegelhalter, 1988] 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.

[Mateescu and Dechter, 2005a] R. Mateescu and R. Dechter. And/or cutset conditioning. In *IJCAI'05*, pages 230–235, 2005.

[Mateescu and Dechter, 2005b] R. Mateescu and R. Dechter. The relationship between and/or search and variable elimination. In *UAI'05*, pages 380–387, 2005.

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

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