Bucket Elimination: A Unifying Framework for Reasoning

Rina Dechter
Department of Computer and Information Science
University of California, Irvine
Irvine, California, USA 92697-3425
dechter@ics.uci.edu

Abstract

Bucket elimination is an algorithmic framework that generalizes dynamic programming to accommodate many problem-solving and reasoning tasks. Algorithms such as directional-resolution for propositional satisfiability, adaptive-consistency for constraint satisfaction, Fourier and Gaussian elimination for solving linear equalities and inequalities, and dynamic programming for combinatorial optimization, can all be accommodated within the bucket elimination framework. Many probabilistic inference tasks can likewise be expressed as bucket-elimination algorithms. These include: belief updating, finding the most probable explanation, and expected utility maximization. These algorithms share the same performance guarantees; all are time and space exponential in the induced-width of the problem's interaction graph.

While elimination strategies have extensive demands on memory, a contrasting class of algorithms called "conditioning search" require only linear space. Algorithms in this class split a problem into subproblems by instantiating a subset of variables, called a *conditioning set*, or a *cutset*. Typical examples of conditioning search algorithms are: backtracking (in constraint satisfaction), and branch and bound (for combinatorial optimization).

The paper presents the bucket-elimination framework as a unifying theme across probabilistic and deterministic reasoning tasks and show how conditioning search can be augmented to systematically trade space for time.

1 Introduction

Bucket elimination is a unifying algorithmic framework that generalizes dynamic programming to accommodate algorithms for many complex problem-solving and reasoning activities, including directional resolution for propositional satisfiability [11], adaptive consistency for constraint satisfaction [19], Fourier and

Gaussian elimination for linear equalities and inequalities, and dynamic programming for combinatorial optimization [5]. The bucket elimination framework will be demonstrated by presenting reasoning algorithms for processing both deterministic knowledge-bases such as constraint networks and cost networks as well as probabilistic databases such as belief networks and influence diagrams.

The main virtues of the bucket-elimination framework are *simplicity* and *generality*. By simplicity, we mean that a complete specification of bucket-elimination algorithms is possible without introducing extensive terminology, making the algorithms accessible to researchers in diverse areas. The primary importance of these algorithms is that their uniformity facilitates understanding which encourages cross-fertilization and technology transfer between disciplines. Indeed, all bucket-elimination algorithms are similar enough, allowing improvement to a single algorithm to be applicable to all others expressed in this framework. For example, expressing probabilistic inference algorithms as bucket-elimination methods clarifies the former's relationship to dynamic programming and to constraint satisfaction allowing the knowledge accumulated in those areas to be utilized in the probabilistic framework.

Normally, an input to a bucket elimination algorithm is a knowledge-base theory and a query specified by a collection of functions or relations over subsets of variables (e.g., clauses for propositional satisfiability, constraints, or conditional probability matrices for belief networks). The algorithm initially partitions these functions into buckets, and each is associated with a single variable. Given a variable ordering, the bucket of a particular variable contains the functions defined on that variable, provided the function is not defined on variables higher in the order. Subsequently, buckets are processed from last to first. When the bucket of variable X is processed, an "elimination procedure" is performed over the functions in its bucket yielding a new function that does not "mention" X. This function summarizes the "effect" of X on the remainder of the problem. The new function is placed in a lower bucket. Bucket-elimination algorithms are *knowledge-compilation* methods, since they generate not only an answer to a query, but also an equivalent representation of the input problem from which various queries are answerable in polynomial time.

An important property of variable elimination algorithms is that their performance can be predicted using a graph parameter called *induced width*, $w*$. In general the structure of a given theory will be associated with an *interaction graph* describing dependencies between variables. The induced-width describes the largest cluster in a tree-embedding of that graph (also known as tree-width). The complexity of bucket-elimination is *time and space* exponential in the induced width of the problem's interaction graph. The size of the induced width varies with various variable orderings, leading to different performance guarantees.

Since all variable elimination algorithms have *space* complexity exponential in the problem's induced width, bucket-elimination is unsuitable when a problem having a high induced-width is encountered. To alleviate space complexity, another universal method for problem solving, called *conditioning*, can be used.

Conditioning is a generic term for algorithms that search the space of partial value assignments or partial conditionings. Conditioning means splitting a problem into subproblems based on a certain condition. A subset of variables known as conditioning variables will generally be instantiated. This generates a subproblem that can be solved in different ways. If the resulting simplified subproblem has no solution or if more solutions are needed, the algorithm can try different assignments to the conditioning set. Algorithms such as *backtracking* search and *branch and bound* may be viewed as conditioning algorithms. *Cutset-conditioning* [12, 34] applies conditioning to a subset of variables that cut all cycles of the interaction graph and solve the resulting subproblem by bucket-elimination.

The complexity of conditioning algorithms is exponential in the conditioning set, however, their space complexity is only linear. Also, empirical studies show that their average performance is often far superior to their worst-case bound. This suggests that combining elimination with conditioning may be essential for improving reasoning processes. Tailoring the balance of elimination and conditioning to the problem instance may improve the benefits in each scheme on a case by case basis; we may have better performance guarantees, improved space complexity, and better overall average performance.

We begin (Section 2) with an overview of known algorithms for deterministic networks, rephrased as bucket elimination algorithms. These include *adaptive-consistency* for constraint satisfaction, *directional resolution* for propositional satisfiability and the Fourier elimination algorithm for solving a set of linear inequalities over real numbers. We summarize their performance as a function of the *induced-width*, and finally, contrast those algorithms with conditioning search methods.

Subsequent sections will provide a detailed derivation of bucket elimination algorithms for probabilistic tasks. Following additional preliminaries (Section 3), we develop the bucket-elimination algorithm for belief updating and analyze its performance in Section 4. The algorithm is extended to find the most probable explanation (Section 5), the maximum aposteriori hypothesis (Section 6) and the maximum expected utility (Section 7). Its relationship to dynamic programming is given in Section 8. Section 9 relates the algorithms to Pearl's poly-tree algorithm and to join-tree clustering. Schemes for combining the conditioning method with elimination are described in Section 10. We conclude with related work (Section 11) and concluding remarks (Section 12).

2 Bucket elimination for deterministic networks

This section provides an overview of known algorithms for reasoning with deterministic relationships, emphasizing their syntactic description as bucket elimi-

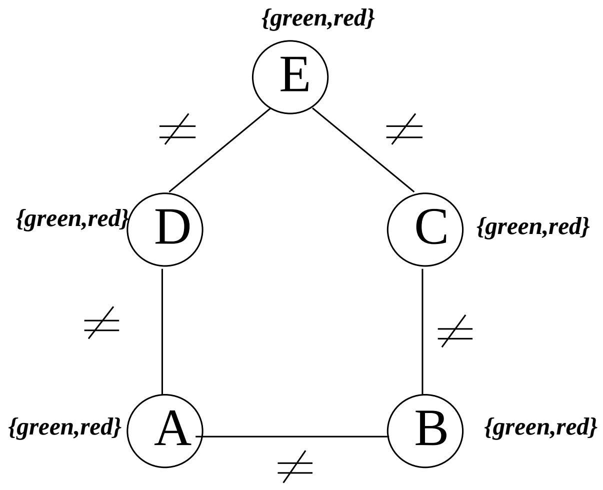

Figure 1: A graph coloring example

nation algorithms.

2.1 Bucket elimination for constraints

Constraint networks have been shown to be useful in formulating diverse problems such as scene labeling, scheduling, natural language parsing and temporal reasoning [13].

Consider the following graph coloring problem in Figure 1. The task is to assign a color to each node in the graph so that adjacent nodes will have different colors. Here is one way to solve this problem. Consider node E first. It can be colored either green or red. Since only two colors are available it follows that D and C must have identical colors, thus, $C = D$ can be added to the constraints of the problem. We focus on variable C next. ¿From the inferred $C = D$ and from the input constraint $C \neq B$ we can infer that $D \neq B$ and add this constraint to the problem, disregarding C and E from now on. Continuing in this fashion with node D, we will infer $A = B$. However, since there is an input constraint $A \neq B$ we can conclude that the original set of constraints is inconsistent.

The algorithm which we just executed is the well known algorithm for solving constraint satisfaction problems called *adaptive consistency* [19]. It works by eliminating variables one by one, while deducing the effect of the eliminated variable on the rest of the problem. Adaptive-consistency can be described using the bucket data-structure. Given a variable ordering such as $d = A, B, D, C, E$ in our example, we process the variables from last to first, namely, from E to A. Step one is to partition the constraints into *ordered buckets*. All the

$Bucket(E)$: $E \neq D$, $E \neq C$
$Bucket(C)$: $C \neq B$
$Bucket(D)$: $D \neq A$,
$Bucket(B)$: $B \neq A$,
$Bucket(A)$:

(a)

$Bucket(E)$: $E \neq D$, $E \neq C$
$Bucket(C)$: $C \neq B$ $||$ $D = C$
$Bucket(D)$: $D \neq A$, $||$, $D \neq B$
$Bucket(B)$: $B \neq A$, $||$ $B = A$
$Bucket(A)$: $||$

(b)

Figure 2: A schematic execution of adaptive-consistency

constraints mentioning the last variable E are put in a bucket designated as $bucket_E$. Subsequently, all the remaining constraints mentioning D are placed in D's bucket, and so on. The initial partitioning of the constraints is depicted in Figure 2a. In general, each constraint is placed in the bucket of its latest variable.

After this initialization step, the buckets are processed from last to first. Processing bucket E produces the constraint $D = C$, which is placed in bucket C. By processing bucket C, the constraint $D \neq B$ is generated and placed in bucket D. While processing bucket D, we generate the constraint $A = B$ and put it in bucket B. When processing bucket B inconsistency is discovered. The buckets' final contents are shown in Figure 2b. The new inferred constraints are displayed to the right of the bar in each bucket.

At each step the algorithm generates a reduced but equivalent problem with one less variable expressed by the union of unprocessed buckets. Once the reduced problem is solved its solution is guaranteed to be extendible to a full solution since it accounted for the deduced constraints generated by the rest of the problem. Therefore, once all the buckets are processed, and if there are no inconsistencies, a solution can be generated in a backtrack-free manner. Namely, a solution is assembled progressively assigning values to variables from the first variable to the last. A value of the first variable is selected satisfying all the current constraints in its bucket. A value for the second variable is then selected which satisfies all the constraints in the second bucket, and so on. Processing a bucket amounts to solving a subproblem defined by the constraints appearing in the bucket, and then restricting the solutions to all but the current bucket's variable. A more formal description requires additional definitions and notations.

A *Constraint Network* consists of a set of *Variables* $X = \{X_1, ..., X_n\}$, *Do-*

Algorithm Adaptive consistency
1. **Input:** A constraint problem $R_1, ...R_t$, ordering $d = X_1, ..., X_n$.
2. **Output:** An equivalent backtrack-free set of constraints and a solution.
3. **Initialize:** Partition constraints into $bucket_1, ...bucket_n$. $bucket_i$ contains all relations whose scope include X_i but no higher indexed variable.
4. **For** $p = n$ *downto* 1, process $bucket_p$ as follows

for all relations $R_1, ...R_m$ defined over $S_1, ...S_m \in bucket_p$ do
(Find solutions to $bucket_p$ and project out X_p:)
$A \leftarrow \bigcup_{j=1}^{m} S_j - \{X_i\}$

$R_A \leftarrow R_A \cap \Pi_A(\bowtie_{j=1}^{m} R_j)$

5. If R_A is not empty, add it to the bucket of its latest variable.
Else, the problem is inconsistent.

6. Return $\cup_j bucket_j$ and generate a solution: for $p = 1$ to n do assign a value to X_p that is consistent with previous assignments and satisfies all the constraints in $bucket_p$.

Figure 3: Algorithm Adaptive consistency

mains $D = \{D_1, ..., D_n\}$, $D_i = \{v_1, ..., v_k\}$ and *Constraints* $R_{S_1}, \ldots, R_{S_t}$, where $S_i \subseteq X$, $1 \leq i \leq n$. A *constraint* is a pair (R, S) where S is a subset of the variables $S = \{X_1, \ldots, X_r\}$, also called its *scope* and R is a relation defined over S, namely, $R \subseteq D_1 \times D_2, ..., \times D_r$, whose tuples denote the legal combination of values. The pair (R, S) is also denoted R_S. A constraint graph associates each variable with a node and connects any two nodes whose variables appear in the same scope. *A solution* is an assignment of a value to each variable that does not violate any constraint. Constraints can be expressed extensionally using relations or intentionally by a mathematical formula or a procedure.

For instance, in the graph-coloring example, the nodes are the variables, the colors are the domains of the variables and the constraints are the in-equation constraints for adjacent variables. The constraint graph is identical to the graph to be colored.

The computation in a bucket can be described in terms of the relational operators of *join* followed by *projection*. The join of two relations R_{AB} and R_{BC} denoted $R_{AB} \bowtie R_{BC}$ is the largest set of solutions over A, B, C satisfying the two constraints R_{AB} and R_{BC}. Projecting out a variable A from a relation R_{ABC}, written as $\Pi_{BC}(R_{ABC})$ removes the assignment to A from each tuple in R_{ABC} and eliminates duplicate rows from the resulting relation. Algorithm Adaptive-consistency is described in Figure 3. For instance, the computation in the bucket of E of our example of Figure 1 is $R_{ECD} \leftarrow R_{ED} \bowtie R_{EC}$ followed by $R_{CD} \leftarrow \Pi_{CD}(R_{ECD})$, where R_{ED} denotes the relation $E \neq D$, namely $R_{ED} = \{(green, red)(red, green)\}$ and R_{EC} stands for the relation $E \neq C$.

The complexity of adaptive-consistency is linear in the number of buckets and in the time to process each bucket. However, since processing a bucket amounts to solving a constraint-satisfaction subproblem its complexity is exponential in the number of variables mentioned in a bucket. If the constraint graph is ordered along the bucket processing, then the number of variables appearing in a bucket is bounded by the *induced-width* of the constraint graph along that ordering [19]. We will demonstrate and analyze this relationship more in Section 4, when discussing belief networks. In this section, we only provide a quick exposure to the concepts and refer the reader to the relevant literature.

Given an undirected graph G and an ordering $d = X_1, ..., X_n$ of its nodes, the *induced graph* of G relative to ordering d is obtained by processing the nodes in reverse order from last to first. For each node all its earlier neighbors are connected, while taking into account old and new edges created during processing. The *induced width of an ordered graph*, denoted $w^*(d)$, is the maximum number of earlier neighbors over all nodes, in the induced graph. The *induced width of a graph*, w^*, is the minimal induced width over all its ordered graphs. Another known related concept is *tree-width*. The tree-width of a graph is identical to its induced-width plus one.

Consider for example, a slightly different graph coloring problem as depicted in Figure 4. Generating the induced-graph along the ordering $d_1 = A, B, C, D, E$ or $d_2 = E, B, C, D, A$ leads to the two graphs in Figure 5. Note that in all drawings from now on, later nodes in the ordering appear on top of earlier ones. The broken arcs are the new added arcs. The induced-width along d_1 and d_2 are 2 and 3 respectively, suggesting different complexity bounds for adaptive-consistency. It was proven that [19],

Theorem 1 *Adaptive-consistency decides if a set of constraints are consistent, and if they are, generates an equivalent representation that is backtrack-free.* $\Box$

Theorem 2 *The time and space complexity of Adaptive-consistency along d is* $O(n \cdot exp(w^*(d)))$. $\Box$

As a result, problems having bounded induced-width ($w^* \leq b$) for some constant b, can be solved in polynomial time. In particular, Adaptive-consistency is linear for trees as demonstrated in Figure 6. The Figure depicts a constraint graph that has no cycles. When the graph is ordered along $d = A, B, C, D, E, F, G$ its width and induced width, equal 1. Indeed as is demonstrated by the schematic execution of adaptive-consistency along d, the algorithm generates only unary relationships and is therefore very efficient.

It is known that finding w* (and the minimizing ordering) is NP-complete [2]. However greedy heuristic ordering algorithms [5, 25] and approximation

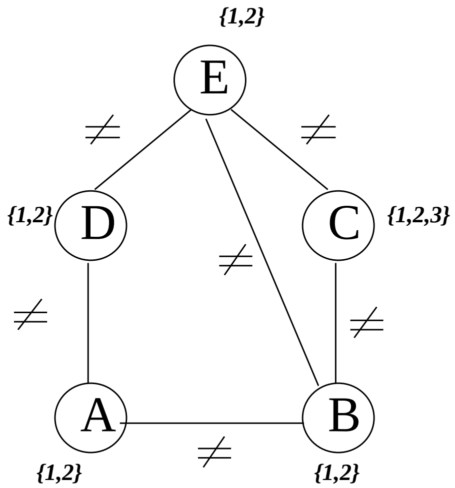

Figure 4: A modified graph coloring problem

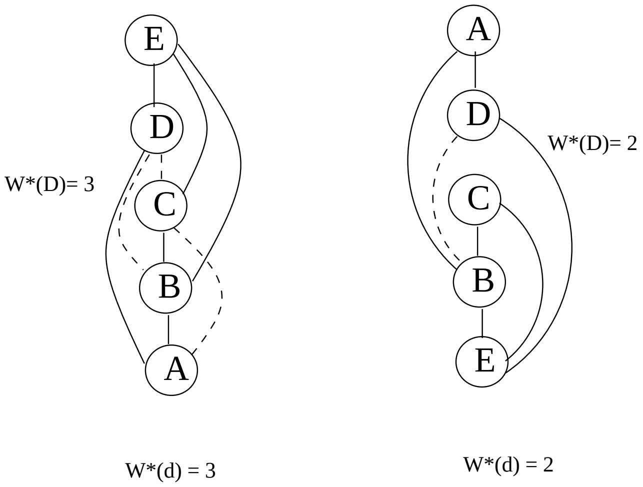

Figure 5: The induced-width along the orderings: $d_1 = A, B, C, D, E$ and $d_2 = E, B, C, D, A$

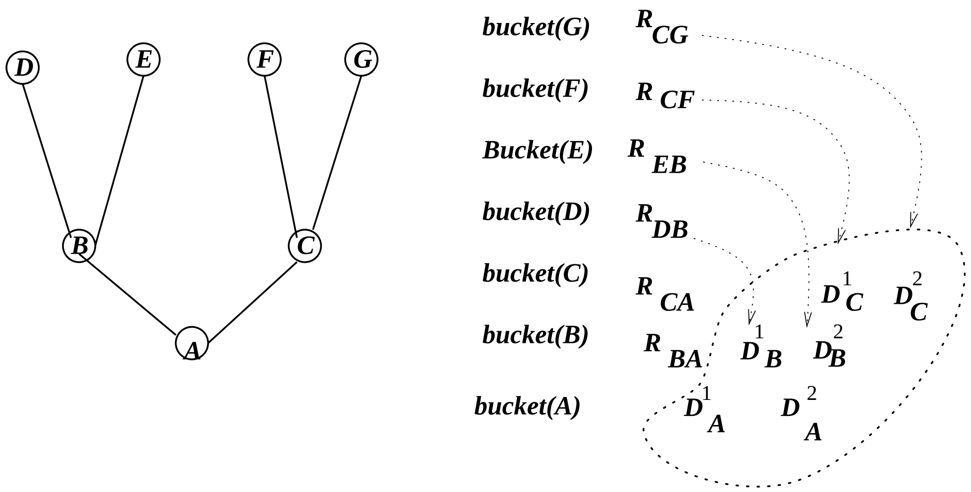

Figure 6: Schematic execution of adaptive-consistency on a tree network. D_X denotes unary constraints over X

.

orderings exist [4, 49]. Also, the induced width of a given ordering is easy to compute. Algorithm Adaptive-consistency and its properties are discussed at length in [19, 20].

2.2 Bucket elimination for Propositional CNFs

Bucket elimination generality can be further illustrated with an algorithm in deterministic reasoning for solving satisfiability [26].

Propositional variables take only two values $\{true, false\}$ or "1" and "0." We denote propositional *variables* by uppercase letters $P, Q, R, \ldots$, propositional literals (i.e., $P, \neg P$) stand for $P =$ "$true$" or $P =$ "$false$," and disjunctions of literals, or *clauses*, are denoted by $\alpha, \beta, \ldots$. A *unit clause* is a clause of size 1. The notation $(\alpha \vee T)$, when $\alpha = (P \vee Q \vee R)$ is shorthand for the disjunction $(P \vee Q \vee R \vee T)$. $\alpha \vee \beta$ denotes the clause whose literal appears in either α or β. The *resolution* operation over two clauses $(\alpha \vee Q)$ and $(\beta \vee \neg Q)$ results in a clause $(\alpha \vee \beta)$, thus eliminating Q. A formula φ in conjunctive normal form (*CNF*) is a set of clauses $\varphi = \{\alpha_1, \ldots, \alpha_t\}$ that denotes their conjunction. The set of *models* or *solutions* of a formula φ is the set of all truth assignments to all its symbols that do not violate any clause. Deciding if a theory is satisfiable is known to be NP-complete [26].

It can be shown that the join-project operation used to process and eliminate a variable by adaptive-consistency over relational constraints translates to pair-wise resolution when applied to clauses [23]. This yields a bucket-elimination

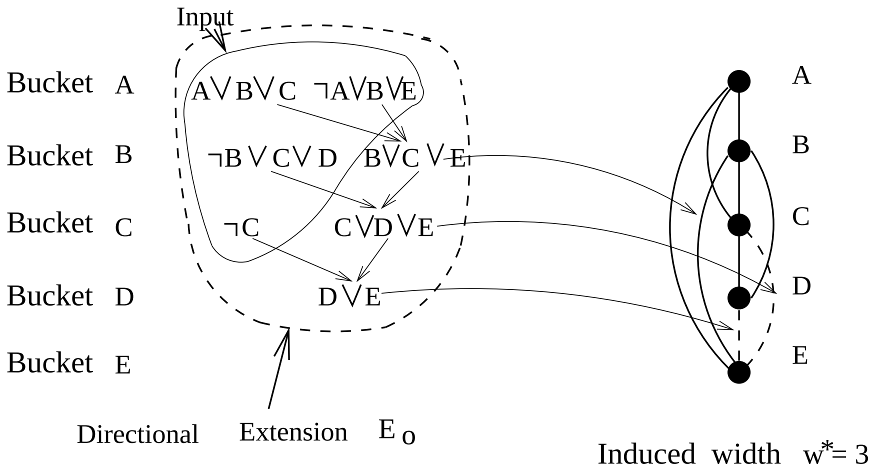

Figure 7: A schematic execution of directional resolution using ordering $d = E, D, C, B, A$

algorithm for propositional satisfiability which we call *directional resolution*. Algorithm *directional resolution*, (DR), is the core of the well-known Davis-Putnam algorithm for satisfiability [11, 21].

Algorithm DR (see Figure 8) is described using *buckets* partitioning the set of clauses in the theory φ. We call its output theory $E_d(\varphi)$, the *directional extension* of φ. Given an ordering $d = Q_1, ..., Q_n$, all the clauses containing Q_i that do not contain any symbol higher in the ordering are placed in the bucket of Q_i, denoted $bucket_i$. As previously noted, the algorithm processes the buckets in the reverse order of d. The processing $bucket_i$ resolves over Q_i all possible pairs of clauses in the bucket and inserts the resolvents into appropriate lower buckets.

Consider for example the following propositional theory:

$$\varphi = (A \vee B \vee C)(\neg A \vee B \vee E)(\neg B \vee C \vee D)$$

The initial partitioning into buckets along the ordering $d = E, D, C, B, A$ as well as the bucket's content generated by the algorithm following resolution over each bucket is depicted in Figure 7. As demonstrated [21], once all the buckets are processed, and if inconsistency was not encountered (namely the empty clause was not generated), a model can be assembled in a backtrack-free manner by consulting $E_d(\varphi)$ using the order d as follows: assign to Q_1 a truth value that is consistent with the clauses in $bucket_1$ (if the bucket is empty, assign Q_1 an arbitrary value); after assigning values to $Q_1, ..., Q_{i-1}$, assign a value to Q_i such that, together with the previous assignments, Q_i will satisfy all the clauses in $bucket_i$.

Algorithm directional resolution
Input: A *CNF* theory φ, an ordering $d = Q_1, ..., Q_n$.
Output: A decision of whether φ is satisfiable. If it is, a theory $E_d(\varphi)$, equivalent to φ; else, a statement "The problem is inconsistent".
1. **Initialize:** Generate an ordered partition of the clauses,
$bucket_1, ..., bucket_n$, where $bucket_i$ contains all the clauses whose highest literal is Q_i.
2. For $p = n$ to 1, do

- **if** $bucket_p$ contains a unit clause, perform only unit resolution. Put each resolvent in the appropriate bucket.
- **else,** resolve each pair $\{(\alpha \vee Q_p), (\beta \vee \neg Q_p)\} \subseteq bucket_p$. If $\gamma = \alpha \vee \beta$ is empty, return "the theory is not satisfiable"; else, determine the index of γ and add γ to the appropriate bucket.

3. **Return:** $E_d(\varphi) \Longleftarrow \bigcup_i bucket_i$ and generate a model in a backtrack-free manner.

Figure 8: Algorithm *directional resolution*

The complexity of DR is exponentially bounded (time and space) in the *induced width* of the theory's *interaction graph* in which a node is associated with a proposition and an arc connects any two nodes appearing in the same clause [21]. This is similar to adaptive-consistency. For example, the interaction graph of theory φ along the ordering d is depicted in Figure 7 by the solid arcs. The broken arcs reflect induced connection of the induced graph. Those are associated with the new clauses generated by resolution. The induced width of this ordering is 3 and, as shown, the maximum number of variables in a bucket, excluding the bucket's variable, is 3.

2.3 Bucket elimination for linear inequalities

A special type of constraint is one that can be expressed by linear inequalities. The domains may be the real numbers, the rationals or finite subsets. In general, a linear constraint between r variables or less is of the form $\sum_{i=1}^{r} a_i x_i \leq c$, where a_i and c are rational constants. For example, $(3x_i + 2x_j \leq 3) \wedge (-4x_i + 5x_j \leq 1)$ are allowed constraints between variables x_i and x_j. In this special case, variable elimination amounts to the standard Gaussian elimination. ¿From the inequalities $x - y \leq 5$ and $x > 3$ we can deduce by eliminating x that $y > 2$. The elimination operation is defined by:

Definition 1 *Let* $\alpha = \sum_{i=1}^{(r-1)} a_i x_i + a_r x_r \leq c$, *and* $\beta = \sum_{i=1}^{(r-1)} b_i x_i + b_r x_r \leq d$. *Then* $elim_r(\alpha, \beta)$ *is applicable only if* a_r *and* b_r *have opposite signs, in which*

Fourier algorithm
Input: A set of linear inequalities, an ordering o.
Output: An equivalent set of linear inequalities that is backtrack-free along o.
Initialize: Partition inequalities into $bucket_1, \ldots, bucket_n$, by the ordered partitioning rule.

For $p \leftarrow n$ **downto** 1
  for each pair $\{\alpha, \beta\} \subseteq bucket_p$, compute $\gamma = elim_p(\alpha, \beta)$.
    If γ has no solutions, return inconsistency.
    else add γ to the appropriate lower bucket.
return $E_o(\varphi) \leftarrow \bigcup_i bucket_i$.

Figure 9: Fourier elimination algorithm

$$
\begin{array}{l}
bucket_x : x - y \leq 5, \quad x > \ 3, \quad t - x \leq 10 \\
bucket_y : y \leq 10 \ \ || \ \ -y \leq 2, \ \ t - y \leq 15 \\
bucket_z : \\
bucket_t : \ || \ t \leq 25
\end{array}
$$

Figure 10: Bucket elimination for the set of linear inequalities: $x - y \leq 5, \quad x > \ 3, \quad t - x \leq 10$, $y \leq 10$ along the ordering $d = t, z, y, x$

case $elim_r(\alpha, \beta) = \sum_{i=1}^{r-1}(-a_i \frac{b_r}{a_r} + b_i)x_i \leq -\frac{b_r}{a_r}c + d$. *If* a_r *and* b_r *have the same sign the elimination implicitly generates the universal constraint.*

Applying adaptive-consistency to linear constraints and processing each pair of relevant inequalities in a bucket by linear elimination yields a bucket elimination algorithm which coincides with the well known Fourier elimination algorithm (see [30]). From the general principle of variable elimination, and as is already known, the algorithm decides the solvability of any set of linear inequalities over the rationals and generates a problem representation which is backtrack-free. The algorithm expressed as a bucket elimination algorithm is summarized in Figure 9. The complexity of Fourier elimination is not bounded exponentially by the induced-width, however. The reason is that the number of feasible linear inequalities that can be specified over a subset of i variables cannot be bounded exponentially by i. For a schematic execution of the algorithm see Figure 10, and for more details see [23].

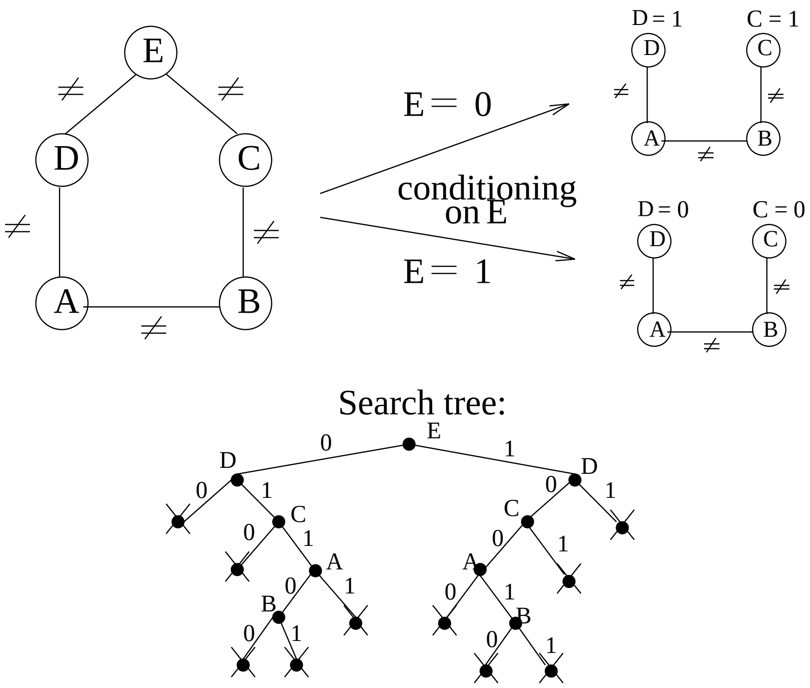

Figure 11: The search tree of the graph coloring problem

2.4 Conditioning

When a problem has a high induced-width the bucket-elimination algorithms such as adaptive-consistency and directional resolution are not applicable due to the algorithm's exponential space complexity. Instead, the respective problems can be solved by a simple conditioning *search*. The basic operation of the conditioning algorithm is assigning or guessing a value to a single variable, thus creating a smaller and simpler subproblem. If a solution to the subproblem is not found then a different value should be tried, leading to a branching search space of partial value assignments that can be traversed by a backtracking algorithm. Figure 11 shows the two subproblems generated by assigning $E = 1$ and $E = 0$ to the graph coloring of Figure 1, and the resulting search space. Although a backtracking algorithm is worst-case exponential in the number of variables, it has the important virtue of requiring only linear space. Only the currently pursued partial assignment needs to be maintained.

Intensive research in the last two decades has been done on improving the basic backtracking search for solving constraint satisfaction problems. For a recent survey see [18]. The most well known version of backtracking for propositional satisfiability is the Davis-Logemann-Loveland (DPLL) algorithm [10],

	Conditioning	Elimination
Worst-case time	O(exp(n))	O(n exp(w *)) w * $\leqslant$ n
Average time	better than worst-case	Same as worst-case
Space	O(n)	O(n exp(w *)) w * $\leqslant$ n
Output	one solution	knowledge compilation

Figure 12: Comparing elimination and conditioning

frequently called just (DP).

2.5 Summary

We observe that elimination algorithms are efficient for problems having small induced width, otherwise their space requirements render them infeasible. Conditioning search algorithms, while they do not have nice worst-case guarantees, require only linear space. In addition, their average behavior is frequently much better than their worst-case bounds. Figure 12 summarizes the properties of elimination vs. conditioning search. This complementary behavior calls for algorithms that combine the two approaches. Indeed, such algorithms are being developed for constraint-satisfaction and propositional satisfiability [12, 40, 9, 15].

In the following sections we will focus in more detail on deriving bucket elimination algorithms for processing probabilistic networks. We are presenting a syntactic and uniform exposition emphasizing these algorithms' form as a straightforward elimination algorithm.

3 Preliminaries for probabilistic reasoning

The belief-network algorithms we present next have much in common with directional resolution and adaptive-consistency. They all possess the property of compiling a theory into a backtrack-free (i.e., greedy) theory, and their complexity is dependent on the induced width graph parameter. The algorithms are variations on known algorithms, and, for the most part, are not new in the sense that the basic ideas have existed for some time [8, 34, 31, 50, 28, 39, 32, 3, 45, 46, 48, 47].

Definition 2 (graph concepts) *A* directed graph *is a pair, $G = \{V, E\}$, where $V = \{X_1, ..., X_n\}$ is a set of elements and $E = \{(X_i, X_j) | X_i, X_j \in V, i \neq j\}$ is the set of edges. If $(X_i, X_j) \in E$, we say that X_i points to X_j. For each variable X_i, the set of parent nodes of X_i, denoted $pa(X_i)$, comprises the variables pointing to X_i in G, while the set of child nodes of X_i, denoted $ch(X_i)$, comprises the variables that X_i points to. We abbreviate $pa(X_i)$ by pa_i and $ch(X_i)$ by ch_i, when no possibility of confusion exists. The family of X_i, F_i, includes X_i and its parent variables. A directed graph is* acyclic *if it has no directed cycles. In an* undirected graph*, the directions of the arcs are ignored: (X_i, X_j) and (X_j, X_i) are identical.*

Definition 3 (belief network) *Let $X = \{X_1, ..., X_n\}$ be a set of random variables over multivalued domains, $D_1, ..., D_n$, respectively. A* belief network *is a pair (G, P) where $G = (X, E)$ is a directed acyclic graph over the variables, and $P = \{P_i\}$, where P_i denotes conditional probability matrices $P_i = \{P(X_i | pa_i)\}$. The belief network represents a probability distribution over X having the product form*

$$P(x_1, ..., x_n) = \Pi_{i=1}^n P(x_i | x_{pa_i})$$

where an assignment $(X_1 = x_1, ..., X_n = x_n)$ is abbreviated to $x = (x_1, ..., x_n)$ and where x_S denotes the restriction of a tuple x over a subset of variables S. An evidence set e is an instantiated subset of variables. $A = a$ denotes a partial assignment to a subset of variables A from their respective domains. We use upper case letters for variables and nodes in a graph and lower case letters for values in a variable's domain. We also call $X_i \cup pa_i$ the scope *of P_i.*

Belief networks provide a formalism for reasoning about partial beliefs under conditions of uncertainty. As we see, it is defined by a directed acyclic graph over nodes representing random variables of interest (e.g., the temperature of a device, the gender of a patient, a feature of an object, the occurrence of an event). The arcs signify the existence of direct causal influences between the linked variables. The strength of these influences are quantified by conditional probabilities that are attached to each cluster of parents-child nodes in the network.

Example 1 *The network in Figure 13a can express causal relationship between 'Season' (A), 'The configuration of an automatic sprinkler system,' (B), 'The amount of rain expected' (C), 'The wetness of the pavement' (F) whether or not the pavement is slippery (G) and 'the amount of manual watering necessary' (D). The belief-network is defined by*

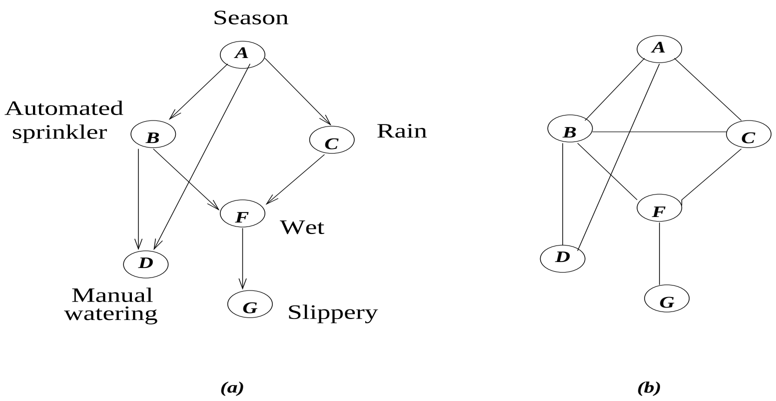

Figure 13: belief network $P(g, f, d, c, b, a) = P(g|f)P(f|c, b)P(d|b, a)P(b|a)P(c|a)P(a)$

$$\forall a, b, v, d, f, g, \quad P(g, f, d, c, b, a) = P(g|f)P(f|c, b)P(d|b, a)P(b|a)P(c|a)P(a).$$

In this case, $pa(F) = \{B, C\}$*.*

The following queries are defined over belief networks:

1. *belief updating*, given a set of observations, computing the posterior probability of each proposition,

2. *finding the most probable explanation (mpe)*, given some observed variables, finding a maximum probability assignment to all unobserved variables,

3. *finding the maximum aposteriori hypothesis (map)*, given some evidence, finding an assignment to a *subset* of the unobserved variables, called hypothesis variables, that maximizes their probability,

4. given also a utility function, finding an assignment to a subset of decision variables that *maximizes the expected utility (meu)* of the problem.

These queries are applicable to tasks such as situation assessment, diagnosis and probabilistic decoding, as well as planning and decision making. They are known to be NP-hard, nevertheless, they all permit a polynomial propagation algorithm for singly-connected networks [34]. The two main approaches for extending this propagation algorithm to multiply-connected networks are the *cycle-cutset* approach, (*cutset-conditioning*), and *tree-clustering* [34, 31, 45]. These methods work well for sparse networks with small cycle-cutsets or clusters. In subsequent sections bucket-elimination algorithms for each of these tasks will be presented and their relationship with existing methods will be discussed.

We conclude this section with some notational conventions. Let u be a partial tuple, S a subset of variables, and X_p a variable not in S. We use (u_S, x_p) to denote the tuple u_S appended by a value x_p of X_p.

Notation 1 (elimination functions) *Given a function h defined over a subset of variables S, called its scope and an $X \in S$, the functions $(min_X h)$, $(max_X h)$, $(mean_X h)$, and $(\sum_X h)$ are defined over $U = S - \{X\}$ as follows. For every $U = u$, $(min_X h)(u) = \min_x h(u,x)$, $(max_X h)(u) = \max_x h(u,x)$, $(\sum_X h)(u) = \sum_x h(u,x)$. Given a set of functions $h_1, ..., h_j$ defined over the subsets $S_1, ..., S_j$, the product function $(\Pi_j h_j)$ and $\sum_j h_j$ are defined over the scope $U = \cup_j S_j$ as follows. For every $U = u$, $(\Pi_j h_j)(u) = \Pi_j h_j(u_{S_j})$, and $(\sum_j h_j)(u) = \sum_j h_j(u_{S_j})$.*

4 Bucket Elimination for Belief Assessment

Belief updating is the primary inference task over belief networks. The task is to maintain the probability of singleton propositions once new evidence arrives. For instance, if we observe that the pavement is slippery, we want to assess the likelihood that the sprinkler was on in our example.

4.1 Deriving elim-bel

Following Pearl's propagation algorithm for singly-connected networks [34], researchers have investigated various approaches to belief updating. We will now present a step by step derivation of a general variable-elimination algorithm for belief updating. This process is typical for any derivation of elimination algorithms.

Let $X = x$ be an atomic proposition. The problem is to assess and update the belief in x_1 given evidence e. We wish to compute $P(X = x|e) = \alpha \cdot P(X = x, e)$, where α is a normalization constant. We will develop the algorithm using example 1 (Figure 13). Assume we have the evidence $g = 1$. Consider the variables in the order $d_1 = A, C, B, F, D, G$. By definition we need to compute

$$P(a, g = 1) = \sum_{c,b,f,d,g=1} P(g|f)P(f|b,c)P(d|a,b)P(c|a)P(b|a)P(a)$$

We can now apply some simple symbolic manipulation, migrating each conditional probability table to the left of the summation variables which it does not reference. We get

$$= P(a)\sum_{c} P(c|a)\sum_{b} P(b|a)\sum_{f} P(f|b,c)\sum_{d} P(d|b,a)\sum_{g=1} P(g|f) \qquad (1)$$

Carrying the computation from right to left (from G to A), we first compute the rightmost summation, which generates a function over f, $\lambda_G(f)$ defined by:

$\lambda_G(f) = \sum_{g=1} P(g|f)$ and place it as far to the left as possible, yielding

$$= P(a) \sum_c P(c|a) \sum_b P(b|a) \sum_f P(f|b,c)\lambda_G(f) \sum_d P(d|b,a) \tag{2}$$

Summing next over d (generating a function denoted $\lambda_D(a,b)$, defined by $\lambda_D(a,b) = \sum_d P(d|a,b)$), we get

$$= P(a) \sum_c P(c|a) \sum_b P(b|a)\lambda_D(a,b) \sum_f P(f|b,c)\lambda_G(f) \tag{3}$$

Next, summing over f (generating $\lambda_F(b,c) = \sum_f P(f|b,c)\lambda_G(f)$), we get,

$$= P(a) \sum_c P(c|a) \sum_b P(b|a)\lambda_D(a,b)\lambda_F(b,c) \tag{4}$$

Summing over b (generating $\lambda_B(a,c)$), we get

$$= P(a) \sum_c P(c|a)\lambda_B(a,c) \tag{5}$$

Finally, summing over c (generating $\lambda_C(a)$), we get

$$P(a)\lambda_C(a) \tag{6}$$

The answer to the query $P(a|g=1)$ can be computed by normalizing the last product.

The bucket-elimination algorithm mimics the above algebraic manipulation by the familiar organizational device of *buckets*, as follows. First, the conditional probability tables ($CPTs$, for short) are partitioned into buckets relative to the order used, $d_1 = A, C, B, F, D, G$. In bucket G we place all functions mentioning G. ¿From the remaining CPTs we place all those mentioning D in bucket D, and so on. The partitioning rule shown earlier for constraint processing and *cnf* theories can be alternatively stated as follows. In X_i's bucket we put all functions that mention X_i but do not mention any variable having a higher index. The resulting initial partitioning for our example is given in Figure 14. Note that the observed variables are also placed in their corresponding bucket.

This initialization step corresponds to deriving the expression in Eq. (1). Now we process the buckets from last to first (or top to bottom in the figures), implementing the right to left computation of Eq. (1). Processing a bucket amounts to eliminating the variable in the bucket from subsequent computation. $Bucket_G$ is processed first. To eliminate G we sum over all values of g. Since in this case we have an observed value $g = 1$, the summation is over a singleton value. The function $\lambda_G(f) = \sum_{g=1} P(g|f)$, is computed and placed in $bucket_F$ (this corresponds to deriving Eq. (2) from Eq. (1)). New functions are placed in lower buckets using the same placement rule.

$$
\begin{aligned}
bucket_G = &\quad P(g|f), g = 1 \\
bucket_D = &\quad P(d|b,a) \\
bucket_F = &\quad P(f|b,c) \\
bucket_B = &\quad P(b|a) \\
bucket_C = &\quad P(c|a) \\
bucket_A = &\quad P(a)
\end{aligned}
$$

Figure 14: Initial partitioning into buckets using $d_1 = A, C, B, F, D, G$

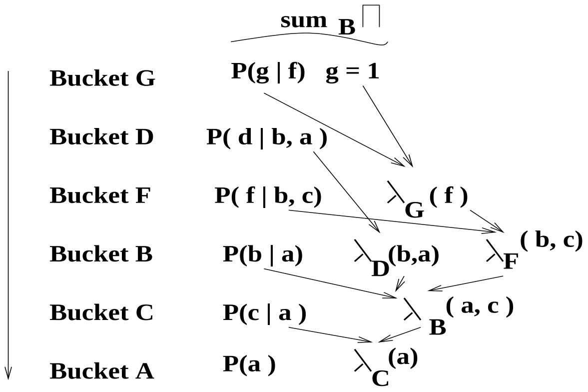

Figure 15: Bucket elimination along ordering $d_1 = A, C, B, F, D, G$.

$Bucket_D$ is processed next. We sum-out D getting $\lambda_D(b,a) = \sum_d P(d|b,a)$, which is placed in $bucket_B$, (which corresponds to deriving Eq. (3) from Eq. (2)). The next variable is F. $Bucket_F$ contains two functions $P(f|b,c)$ and $\lambda_G(f)$, and follows Eq. (4) we generate the function $\lambda_F(b,c) = \sum_f P(f|b,c) \cdot \lambda_G(f)$, which is placed in $bucket_B$ (this corresponds to deriving Eq. (4) from Eq. (3)). In processing the next $bucket_B$, the function $\lambda_B(a,c) = \sum_b P(b|a) \cdot \lambda_D(b,a) \cdot \lambda_F(b,c)$ is computed and placed in $bucket_C$ (deriving Eq. (5) from Eq. (4)). In processing the next $bucket_C$, $\lambda_C(a) = \sum_{c \in C} P(c|a) \cdot \lambda_B(a,c)$ is computed (which corresponds to deriving Eq. (6) from Eq. (5)). Finally, the belief in a can be computed in $bucket_A$, $P(a|g=1) = \alpha \cdot P(a) \cdot \lambda_C(a)$. Figure 15 summarizes the flow of computation. Throughout this process we recorded two-dimensional functions at the most; the complexity of the algorithm using ordering d_1 is (roughly) time and space quadratic in the domain sizes.

What will occur if we use a different variable ordering? For example, let's apply the algorithm using $d_2 = A, F, D, C, B, G$. Applying algebraic manipulation from right to left along d_2 yields the following sequence of derivations:
$P(a, g=1) = P(a) \sum_f \sum_d \sum_c P(c|a) \sum_b P(b|a)\ P(d|a,b) P(f|b,c) \sum_{g=1} P(g|f) =$
$P(a) \sum_f \lambda_G(f) \sum_d \sum_c P(c|a) \sum_b P(b|a)\ P(d|a,b) P(f|b,c) =$

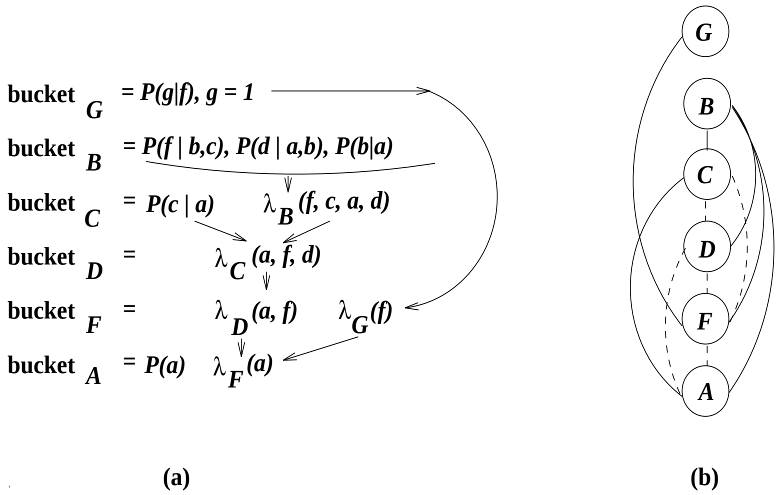

Figure 16: The bucket's output when processing along $d_2 = A, F, D, C, B, G$

$P(a)\sum_f \lambda_G(f)\sum_d \sum_c P(c|a)\lambda_B(a,d,c,f) =$
$P(a)\sum_f \lambda_g(f)\sum_d \lambda_C(a,d,f) =$
$P(a)\sum_f \lambda_G(f)\lambda_D(a,f) =$
$P(a)\lambda_F(a)$

The bucket elimination process for ordering d_2 is summarized in Figure 16a. Each bucket contains the initial $CPTs$ denoted by Ps, and the functions generated throughout the process, denoted by λs.

We conclude with a general derivation of the bucket elimination algorithm, called *elim-bel*. Consider an ordering of the variables $X = (X_1, ..., X_n)$ and assume we seek $P(x_1|e)$. Using the notation $\bar{x}_i = (x_1, ..., x_i)$ and $\bar{x}_i^j = (x_i, x_{i+1}, ..., x_j)$, where F_i is the family of variable X_i, we want to compute:

$$P(x_1, e) = \sum_{x=\bar{x}_2^n} P(\bar{x}_n, e) = \sum_{\bar{x}_2^{(n-1)}} \sum_{x_n} \Pi_i P(x_i, e|x_{pa_i})$$

Separating X_n from the rest of the variables results in:

$$= \sum_{x=\bar{x}_2^{(n-1)}} \Pi_{X_i \in X-F_n} P(x_i, e|x_{pa_i}) \cdot \sum_{x_n} P(x_n, e|x_{pa_n}) \Pi_{X_i \in ch_n} P(x_i, e|x_{pa_i})$$

$$= \sum_{x=\bar{x}_2^{(n-1)}} \Pi_{X_i \in X-F_n} P(x_i, e|x_{pa_i}) \cdot \lambda_n(x_{U_n})$$

where

$$\lambda_n(x_{U_n}) = \sum_{x_n} P(x_n, e|x_{pa_n}) \Pi_{X_i \in ch_n} P(x_i, e|x_{pa_i}) \tag{7}$$

Algorithm elim-bel
Input: A belief network $BN = \{P_1, ..., P_n\}$; an ordering of the variables, $d = X_1, ..., X_n$; evidence e.
Output: The belief $P(x_1|e)$.
1. **Initialize:** Generate an ordered partition of the conditional probability matrices, $bucket_1, ..., bucket_n$, where $bucket_i$ contains all matrices whose highest variable is X_i. Put each observed variable in its bucket. Let $S_1, ..., S_j$ be the subset of variables in the processed bucket on which matrices (new or old) are defined.
2. **Backward:** For $p \leftarrow n$ downto 1, do
for all the matrices $\lambda_1, \lambda_2, ..., \lambda_j$ in $bucket_p$, do

- **If** (observed variable) $X_p = x_p$ appears in $bucket_p$, assign $X_p = x_p$ to each λ_i and then put each resulting function in appropriate bucket.
- **else**, $U_p \leftarrow \bigcup_{i=1}^{j} S_i - \{X_p\}$. Generate $\lambda_p = \sum_{X_p} \Pi_{i=1}^{j} \lambda_i$ and add λ_p to the largest-index variable in U_p.

3. **Return:** $Bel(x_1) = \alpha \Pi_i \lambda_i(x_1)$(where the λ_i are in $bucket_1$, α is a normalizing constant).

Figure 17: Algorithm *elim-bel*

and U_n denotes the variables appearing with X_n in a probability component, (excluding X_n). The process continues recursively with X_{n-1}.

Thus, the computation performed in bucket X_n is captured by Eq. (7). Given ordering $d = X_1, ..., X_n$, where the queried variable appears first, the CPTs are partitioned using the rule described earlier. Then buckets are processed from last to first. To process each bucket, all the bucket's functions, denoted $\lambda_1, ..., \lambda_j$ and defined over subsets $S_1, ..., S_j$ are multiplied. Then the bucket's variable is eliminated by summation. The computed function is $\lambda_p : U_p \rightarrow R$, $\lambda_p = \sum_{X_p} \Pi_{i=1}^{j} \lambda_i$, where $U_p = \cup_i S_i - X_p$. This function is placed in the bucket of its largest-index variable in U_p. Once all the buckets are processed, the answer is available in the first bucket. Algorithm elim-bel is described in Figure 17. We conclude:

Theorem 3 *Algorithm elim-bel computes the posterior belief $P(x_1|e)$ for any given ordering of the variables which is initiated by X_1.* $\Box$

The peeling algorithm for genetic trees [8], Zhang and Poole's algorithm [51], as well as the SPI algorithm by D'Ambrosio et. al., [39] are all variations of elim-bel. Decimation algorithms in statistical physics are also related and were applied to Boltzmann trees [43].

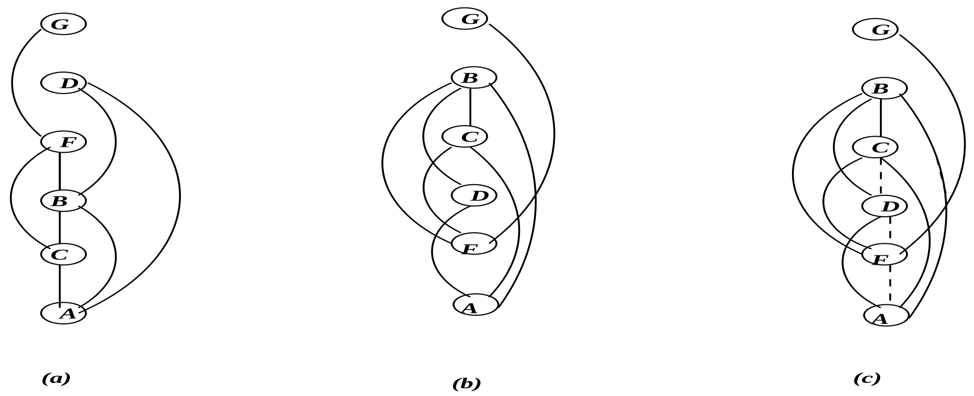

Figure 18: Two orderings of the moral graph of our example problem

4.2 Complexity

We see that although elim-bel can be applied using any ordering, its complexity varies considerably. Using ordering d_1 we recorded functions on pairs of variables only, while using d_2 we had to record functions on four variables (see $Bucket_C$ in Figure 16a). The arity of the function recorded in a bucket equals the number of variables appearing in that processed bucket, excluding the bucket's variable. Since computing and recording a function of arity r is time and space exponential in r we conclude that the complexity of the algorithm is exponential in the size (number of variables) of the largest bucket.

Fortunately, as was observed earlier for adaptive-consistency and directional-resolution, the bucket sizes can be easily predicted from an order associated with the elimination process. Consider the *moral graph* of a given belief network. This graph has a node for each variable and any two variables appearing in the same CPT are connected. The moral graph of the network in Figure 13a is given in Figure 13b. Let us take this moral graph and impose an ordering on its nodes. Figures 18a and 18b depict the ordered moral graph using the two orderings $d_1 = A, C, B, F, D, G$ and $d_2 = A, F, D, C, B, G$. The ordering is pictured with the first variable at the bottom and the last variable at the top.

The *width* of each variable in the ordered graph is the number of its *earlier* neighbors in the ordering. Thus, the width of G in the ordered graph along d_1 is 1 and the width of F is 2. Notice now that when using ordering d_1, the number of variables in the initial buckets of G and F, are 1 and 2, respectively. Indeed, the number of variables mentioned in a bucket in their initial partitioning (excluding the bucket's variable) is always identical to the width of that node in the ordered moral graph.

During processing we wish to maintain the correspondence that any two nodes in the graph are connected if there is a function (new or old) defined on both. Since, during processing, a function is recorded on all the variables appearing in a bucket of a variable (which is the set of earlier neighbors of the variable in the ordered graph) these nodes should be connected. If we perform this graph operation recursively from last node to first, (for each node connecting its earliest neighbors) we get the *the induced graph*. The width of each node in this induced graph is identical to the bucket's sizes generated during the elimination process (Figure 16b).

Example 2 *The induced moral graph of Figure 13b, relative to ordering* $d_1 = A, C, B, F, D, G$ *is depicted in Figure 18a. In this case, the ordered graph and its induced ordered graph are identical, since all the earlier neighbors of each node are already connected. The maximum induced width is 2. In this case, the maximum arity of functions recorded by the elimination algorithms is 2. For* $d_2 = A, F, D, C, B, G$ *the induced graph is depicted in Figure 18c. The width of* C *is initially 2 (see Figure 18b) while its induced width is* 3. *The maximum induced width over all variables for* d_2 *is 4, and so is the recorded function's dimensionality.*

A formal definition of all these graph concepts is given next, partially reiterating concepts defined in Section 2.

Definition 4 *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 the node's neighbors that precede it in the ordering. The* width of an ordering d, *denoted* $w(d)$, *is the maximum width over all nodes. The* induced width of an ordered graph, $w^*(d)$, *is the width of the ordered graph obtained by processing the nodes from last to first. When node* X *is processed, all its preceding neighbors are connected. The resulting graph is called Induced-graph or triangulated graph. The* induced width of a graph, $w*$, *is the minimal induced width over all its orderings. The induced graph suggests a hyper-tree embedding of the original graph whose tree-width equals the induced-width. Thus, the* tree-width *of a graph is the minimal induced width plus one [2].*

As noted before, the established connection between buckets' sizes and induced width motivates finding an ordering with a smallest induced width, a task known to be hard [2]. However, useful greedy heuristics as well as approximation algorithms are available [13, 4, 49].

In summary, the complexity of algorithm elim-bel is dominated by the time and space needed to process a bucket. Recording a function on all the bucket's variables is time and space exponential in the number of variables mentioned in the bucket. The induced width bounds the arity of the functions recorded: variables appearing in a bucket coincide with the earlier neighbors of the corresponding node in the ordered induced moral graph. In conclusion,

Theorem 4 *Given an ordering d the complexity of elim-bel is (time and space) exponential in the induced width $w^*(d)$ of the network's ordered moral graph.*

4.3 Handling observations

Evidence should be handled in a special way during the processing of buckets. Continuing with our example using elimination order d_1, suppose we wish to compute the belief in a, having observed $b = 1$. This observation is relevant only when processing $bucket_B$. When the algorithm arrives at that bucket, the bucket contains the three functions $P(b|a)$, $\lambda_D(b,a)$, and $\lambda_F(b,c)$, as well as the observation $b = 1$ (see Figure 15).

The processing rule dictates computing $\lambda_B(a,c) = P(b=1|a)\lambda_D(b=1,a)\lambda_F(b=1,c)$. Namely, generating and recording a two-dimensioned function. It would be more effective, however, to apply the assignment $b = 1$ to each function in the bucket separately then put the individual resulting functions into lower buckets. In other words, we can generate $P(b=1|a)$ and $\lambda_D(b=1,a)$, each of which will be placed in bucket A, and $\lambda_F(b=1,c)$, which will be placed in bucket C. By doing so, we avoid increasing the dimensionality of the recorded functions. Processing buckets containing observations in this manner automatically exploits the cutset conditioning effect [34]. Therefore, the algorithm has a special rule for processing buckets with observations. The observed value is assigned to each function in the bucket, and each resulting function is individually moved to a lower bucket.

Note that if bucket B had been last in ordering, as in d_2, the virtue of conditioning on B could have been exploited earlier. During its processing, $bucket_B$ contains $P(b|a), P(d|b,a), P(f|c,b)$, and $b = 1$ (see Figure 16a). The special rule for processing buckets holding observations will place $P(b=1|a)$ in $bucket_A$, $P(d|b=1,a)$ in $bucket_D$, and $P(f|c,b=1)$ in $bucket_F$. In subsequent processing only one-dimensional functions will be recorded. Thus, the presence of observations reduces complexity: Buckets of observed variables are processed in linear time, their recorded functions do not create functions on new subsets of variables, and therefore for observed variables no new arcs should be added when computing the induced graph.

To capture this refinement we use the notion of *adjusted induced graph* which is defined recursively. Given an ordering and given a set of observed nodes, the adjusted induced graph is generated (processing the ordered graph from last to first) by connecting only the earlier neighbors of unobserved nodes. The *adjusted induced width* is the width of the adjusted induced graph, disregarding observed nodes. For example, in Figure 19(a,b) we show the ordered moral graph and the induced ordered moral graph of Figure 13. In 19(c) the arcs connected to the observed nodes are marked by broken lines, resulting in the adjusted induced-graph given in (d). In summary,

Theorem 5 *Given a belief network having n variables, algorithm elim-bel when*

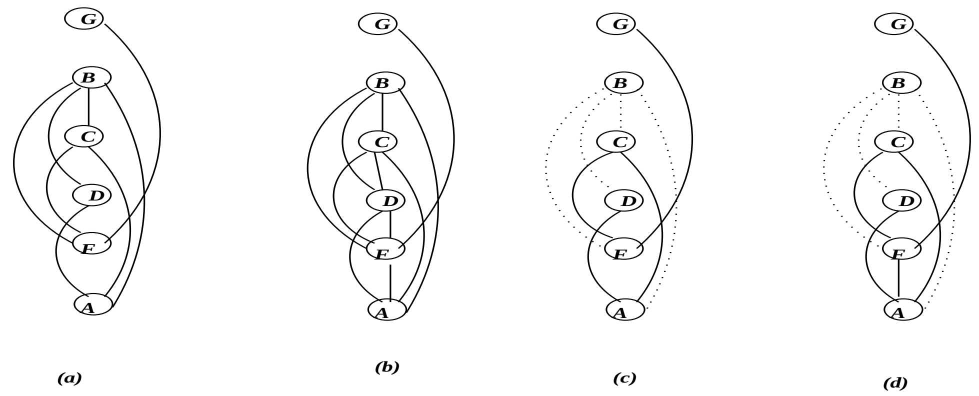

Figure 19: Adjusted induced graph relative to observing B

using ordering d and evidence e, is (time and space) exponential in the adjusted induced width $w^*(d, e)$ of the network's ordered moral graph. □

4.4 Relevant subnetworks

Here we will deviate from our emphasis on uniformity and focus instead on an improvement of the algorithm suitable for belief-updating only. The belief-updating task has special semantics which allows restricting the computation to relevant portions of the belief network. These restrictions are already available in the literature in the context of the existing algorithms [27, 44].

Since summation over all values of a probability function is 1, the recorded functions of some buckets will degenerate to the constant 1. If we can predict these cases in advance, we can avoid needless computation by skipping some buckets. If we use a *topological ordering* of the belief network's acyclic graph (where parents precede their child nodes), and assume that the queried variable initiates the ordering, we can identify skippable buckets dynamically during the elimination process.

Proposition 6 *Given a belief network and a topological ordering $X_1, ..., X_n$, that is initiated by a query variable X_1, algorithm elim-bel, computing $P(x_1|e)$, can skip a bucket if during processing the bucket contains no evidence variable and no newly computed function.*

Proof: If topological ordering is used, each bucket of a variable X contains initially at most one function, $P(X|pa(X))$. Clearly, if there is no evidence nor new functions in the bucket summation, $\sum_x P(x|pa(X))$ will yield the constant 1. □

Example 3 *Consider again the belief network whose acyclic graph is given in Figure 13a and the ordering $d_1 = A, C, B, F, D, G$. Assume we want to update* *the belief in variable A given evidence on F. Obviously the buckets of G and D can be skipped and processing should start with bucket$_F$. Once bucket$_F$ is processed, the remaining buckets are not skippable.*

Alternatively, the relevant portion of the network can be precomputed by using a recursive marking procedure applied to the ordered moral graph. (see also [51]). Since topological ordering initiated by the query variables are not always feasible (when query nodes are not root nodes) we will define a marking scheme applicable to an arbitrary ordering.

Definition 5 *Given an acyclic graph and an ordering o that starts with the queried variable, and given evidence e, the marking process proceeds as follows.*

- *Initial marking: an evidence node is marked and any node having a child appearing earlier in o (namely violate the "parent preceding child rule"), is marked.*
- *Secondary marking: Processing the nodes from last to first in o, if a node X is marked, mark all its earlier neighbors.*

The marked belief subnetwork obtained by deleting all *unmarked* nodes can now be processed by elim-bel to answer the belief-updating query.

Theorem 7 *Let $R = (G, P)$ be a belief network, $o = X_1, ..., X_n$ and e set of evidence. Then $P(x_1|e)$ can be obtained by applying elim-bel over the marked network relative to evidence e and ordering o, denoted $M(R|e, o)$.*

Proof: We will show that if elim-bel was applied to the original network along ordering o, then any unmarked node is irrelevant, namely processing its bucket yields the constant 1. Let $R = (G, P)$ be a belief network processed along o by elim-bel, assuming evidence e. Assume the claim is incorrect and let X be the first unmarked node (going from last to first along o) such that when elim-bel process R the bucket of X does *not* yield the constant 1, and is therefore relevant. Since X is unmarked, it means that it is: 1) not an evidence, and 2) X does not have an earlier child relative to o, and 3) X does not have a later neighbor which is marked. Since X is not evidence, and since all its child nodes appear later in o, then, in the initial marking it cannot be marked and in the initial bucket partitioning its bucket includes its family $P(X|pa)$ only. Since the bucket is relevant, it must be the case that during the processing of prior buckets (of variables appearing later in o), a computed function is inserted to bucket X. Let Y be the variable during whose processing a function was placed in the bucket of X. This implies that X is connected to Y. Since Y is clearly relevant and is therefore marked (we assumed X was the first variable violating the claim, and Y appears later than X), X must also be marked, yielding a contradiction. $\Box$.

Corollary 1 *The complexity of algorithm elim-bel along ordering o given evidence e is exponential in the adjusted induced width of the marked ordered moral subgraph.* □

5 An Elimination Algorithm for mpe

In this section we focus on finding the most probable explanation. This task appears in applications such as diagnosis and design as well as in probabilistic decoding. For example, given data on clinical findings, it may suggest the most likely disease a patient is suffering from. In decoding, the task is to identify the most likely input message which was transmitted over a noisy channel, given the observed output. Although the relevant task here is finding the most likely assignment over a *subset* of hypothesis variables (known as map and analyzed in the next section), the mpe is close enough and is often used in applications. Researchers have investigated various approaches to finding the *mpe* in a belief network [34, ?, 35, 36]. Recent proposals include best first-search algorithms [48] and algorithms based on linear programming [41].

The problem is to find x^0 such that $P(x^0) = \max_x P(x,e) = \max_x \Pi_i P(x_i, e|x_{pa_i})$ where $x = (x_1, ..., x_n)$ and e is a set of observations, on subsets of the variables. Computing for a given ordering $X_1, ..., X_n$, can be accomplished as previously shown by performing the maximization operation along the ordering from right to left, while migrating to the left all components that do not mention the maximizing variable. We get,

$$M = \max_{\bar{x}_n} P(\bar{x}_n, e) = \max_{\bar{x}_{(n-1)}} \max_{x_n} \Pi_i P(x_i, e|x_{pa_i})$$

$$= \max_{\bar{x}_{n-1}} \Pi_{X_i \in X - F_n} P(x_i, e|x_{pa_i}) \cdot \max_{x_n} P(x_n, e|x_{pa_n}) \Pi_{X_i \in ch_n} P(x_i, e|x_{pa_i})$$

$$= \max_{x = \bar{x}_{n-1}} \Pi_{X_i \in X - F_n} P(x_i, e|x_{pa_i}) \cdot h_n(x_{U_n})$$

where

$$h_n(x_{U_n}) = \max_{x_n} P(x_n, e|x_{pa_n}) \Pi_{X_i \in ch_n} P(x_i, e|x_{pa_i})$$

and U_n are the variables appearing in components defined over X_n. Clearly, the algebraic manipulation of the above expressions is the same as the algebraic manipulation for belief assessment where summation is replaced by maximization. Consequently, the bucket-elimination procedure *elim-mpe* is identical to elim-bel except for this change. Given ordering $X_1, ..., X_n$, the conditional probability tables are partitioned as before. To process each bucket, we multiply all the bucket's matrices, which in this case are denoted $h_1, ..., h_j$ and defined over subsets $S_1, ..., S_j$, and then eliminate the bucket's variable by maximization as dictated by the algebraic derivation previously noted. The computed function in this case is $h_p : U_p \rightarrow R$, $h_p = \max_{X_p} \Pi_{i=1}^{j} h_i$, where $U_p = \cup_i S_i - X_p$. The function obtained by processing a bucket is placed in an earlier bucket of its largest-index variable in U_p. In addition, a function $x_p^o(u) = argmax_{X_p} h_p(u)$, which relates an optimizing value of X_p with each tuple of U_p, may be recorded and placed in the bucket of X_p.[1] Constant functions can be placed either in the preceding bucket or directly in the first bucket[2].

This procedure continues recursively, processing the bucket of the next variable, proceeding from the last to the first variable. Once all buckets are processed, the *mpe* value can be extracted as the maximizing product of functions in the first bucket. When this *backwards* phase terminates, the algorithm initiates a *forwards phase* to compute an *mpe* tuple by assigning values along the ordering from X_1 to X_n, consulting the information recorded in each bucket. Specifically, once the partial assignment $x = (x_1, ..., x_{i-1})$ is selected, the value of X_i appended to this tuple is $x_i^o(x)$, where x^o is the function recorded in the backward phase. Alternatively, if the functions x^o were not recorded in the backwards phase, the value x_i of X_i is selected to maximize the product in $bucket_i$ given the partial assignment x. This algorithm is presented in Figure 20. Observed variables are handled as in elim-bel. The notion of irrelevant buckets is not applicable here.

Example 4 *Consider again the belief network in Figure 13. Given the ordering $d = A, C, B, F, D, G$ and the evidence $g = 1$, process variables from last to first after partitioning the conditional probability matrices into buckets, such that $bucket_G = \{P(g|f), g = 1\}$, $bucket_D = \{P(d|b, a)\}$, $bucket_F = \{P(f|b, c)\}$, $bucket_B = \{P(b|a)\}$, $bucket_C = \{P(c|a)\}$, and $bucket_A = \{P(a)\}$. To process G, assign $g = 1$, get $h_G(f) = P(g = 1|f)$, and place the result in $bucket_F$. The function $G^o(f) = argmax h_G(f)$ may be computed and placed in $bucket_G$ as well. Process $bucket_D$ by computing $h_D(b, a) = \max_d P(d|b, a)$ and put the result in $bucket_B$. Bucket F, next to be processed, now contains two matrices: $P(f|b, c)$ and $h_G(f)$. Compute $h_F(b, c) = \max_f p(f|b, c) \cdot h_G(f)$, and place the resulting function in $bucket_B$. To eliminate B, we record the function $h_B(a, c) = \max_b P(b|a) \cdot h_D(b, a) \cdot h_F(b, c)$ and place it in $bucket_C$. To eliminate C, we compute $h_C(a) = \max_c P(c|a) \cdot h_B(a, c)$ and place it in $bucket_A$. Finally, the mpe value given in $bucket_A$, $M = \max_a P(a) \cdot h_C(a)$, is determined. Next the mpe tuple is generated by going forward through the buckets. First, the value a^0 satisfying $a^0 = argmax_a P(a) h_C(a)$ is selected. Subsequently the value of C, $c^0 = argmax_c P(c|a^0) h_B(a^0, c)$ is determined. Next $b^0 = argmax_b P(b|a^0) h_D(b, a^0) h_F(b, c^0)$ is selected, and so on. The schematics computation is summarized by Figure 15 where λ is replaced by h.*

The backward process can be viewed as a compilation phase in which we compile information regarding the most probable extension of partial tuples to variables higher in the ordering (see also section 7.2).

[1] This step is optional; the maximizing values can be recomputed from the information in each bucket.

[2] Those are necessary to determine the exact mpe value.

Algorithm elim-mpe
Input: A belief network $BN = \{P_1, ..., P_n\}$; an ordering of the variables, d; observations e.
Output: The most probable assignment.
1. **Initialize:** Generate an ordered partition of the conditional probability matrices, $bucket_1, \ldots, bucket_n$, where $bucket_i$ contains all matrices whose highest variable is X_i. Put each observed variable in its bucket. Let $S_1, ..., S_j$ be the subset of variables in the processed bucket on which matrices (new or old) are defined.
2. **Backward:** For $p \leftarrow n$ downto 1, do
for all the matrices $h_1, h_2, ..., h_j$ in $bucket_p$, do

- **If** (observed variable) $bucket_p$ contains $X_p = x_p$, assign $X_p = x_p$ to each h_i and put each in appropriate bucket.
- **else**, $U_p \leftarrow \bigcup_{i=1}^{j} S_i - \{X_p\}$. Generate functions $h_p = \max_{X_p} \Pi_{i=1}^{j} h_i$ and $x_p^o = argmax_{X_p} h_p$. Add h_p to bucket of largest-index variable in U_p.

3. **Forward:** The mpe value is obtained by the product in $bucket_1$.
An mpe tuple is obtained by assigning values in the ordering d
consulting recorded functions in each bucket as follows.
Given the assignment $x = (x_1, ..., x_{i-1})$ choose $x_i = x_i^o(x)$ (x_i^o is in $bucket_i$),
or Choose $x_i = argmax_{X_i} \Pi_{\{h_j \in \ bucket_i | \ x=(x_1,...,x_{i-1})\}} h_j$

Figure 20: Algorithm *elim-mpe*

As in the case of belief updating, the complexity of elim-mpe is bounded exponentially in the dimension of the recorded functions, and those functions are bounded by the induced width $w^*(d,e)$ of the ordered moral graph. In summary,

Theorem 8 *Algorithm elim-mpe is complete for the mpe task. Its complexity (time and space) is $O(n \cdot exp(w^*(d,e)))$, where n is the number of variables and $w^*(d,e)$ is the adjusted induced width of the ordered moral graph.*

6 An Elimination Algorithm for MAP

The map task is a generalization of both mpe and belief assessment. It asks for the maximal belief associated with a *subset of unobserved hypothesis variables* and is likewise widely applicable to diagnosis tasks. Since the map task by its definition is a mixture of the previous two tasks, in its corresponding algorithm some of the variables are eliminated by summation, others by maximization.

Given a belief network, a subset of hypothesized variables $A = \{A_1, ..., A_k\}$, and some evidence e, the problem is to find an assignment to the hypothesized variables that maximizes their probability given the evidence, namely to find $a^o = argmax_{a_1,...,a_k} P(a_1, ..., a_k, e)$. We wish to compute $\max_{\bar{a}_k} P(a_1, ..., a_k, e) = \max_{\bar{a}_k} \sum_{\bar{x}^n_{k+1}} \Pi_{i=1}^n P(x_i, e|x_{pa_i})$ where $x = (a_1, ..., a_k, x_{k+1}, ..., x_n)$. Algorithm *elim-map* in Figure 21 considers only orderings in which the hypothesized variables start the ordering. The algorithm has a backward phase and a forward phase, but the forward phase is relative to the hypothesized variables only. Maximization and summation may be somewhat interleaved to allow more effective orderings, however for simplicity of exposition we do not incorporate this option here. Note that the "relevant" graph for this task can be restricted by marking the summation variables as was done for belief updating.

Theorem 9 *Algorithm elim-map is complete for the map task. Its complexity is $O(n \cdot exp(w^*(d,e))$, where n is the number of variables in the relevant marked graph and $w^*(d,e)$ is the adjusted induced width of its marked moral graph.*

7 An Elimination Algorithm for MEU

The last and somewhat more complicated task is finding the maximum expected utility. Given a belief network, evidence e, a real-valued utility function $u(x)$ additively decomposable relative to functions $f_1, ..., f_j$ defined over $Q = \{Q_1, ..., Q_j\}$, $Q_i \subseteq X$, such that $u(x) = \sum_{Q_j \in Q} f_j(x_{Q_j})$, and given a subset of decision variables $D = \{D_1, ...D_k\}$ that are assumed to be root nodes,[3]

[3] We make this assumption for simplicity of presentation. The general case can be easily handled as is done for general influence diagrams.

Algorithm elim-map
Input: A belief network $BN = \{P_1, ..., P_n\}$; a subset of variables $A = \{A_1, ..., A_k\}$; an ordering of the variables, d, in which the A's are first in the ordering; observations e.
Output: A most probable assignment $A = a$.
1. **Initialize:** Generate an ordered partition of the conditional probability matrices, $bucket_1, \ldots, bucket_n$, where $bucket_i$ contains all matrices whose highest variable is X_i.
2. **Backwards** For $p \leftarrow n$ downto 1, do
for all the matrices $\beta_1, \beta_2, ..., \beta_j$ in $bucket_p$, do

- **If** (observed variable) $bucket_p$ contains the observation $X_p = x_p$, assign $X_p = x_p$ to each β_i and put each in appropriate bucket.
- **else**, $U_p \leftarrow \bigcup_{i=1}^{j} S_i - \{X_p\}$. If X_p is not in A, then $\beta_p = \sum_{X_p} \Pi_{i=1}^{j} \beta_i$; else, $X_p \in A$, and $\beta_p = \max_{X_p} \Pi_{i=1}^{j} \beta_i$ and $a^0 = argmax_{X_p} \beta_p$. Add β_p to the bucket of the largest-index variable in U_p.

3. **Forward:** Assign values, in the ordering $d = A_1, ..., A_k$, using the information recorded in each bucket.

Figure 21: Algorithm *elim-map*

the meu task is to find a set of decisions $d^o = (d^o{}_1, ..., d^o{}_k)$ $(d_i \in D_i)$, that maximizes the expected utility. We assume that variables *not* appearing in D are indexed $X_{k+1}, ..., X_n$. We want to compute

$$E = \max_{d_1,\ldots,d_k} \sum_{x_{k+1},\ldots x_n} \Pi_{i=1}^{n} P(x_i, e | x_{pa_i}, d_1, ..., d_k) u(x),$$

and

$$d^0 = argmax_D E$$

As in previous tasks, we will begin by identifying the computation associated with X_n from which we will extract the computation in each bucket. We denote an assignment to the decision variables by $d = (d_1, ..., d_k)$ and, as before, $\bar{x}_k^j = (x_k, ..., x_j)$. Algebraic manipulation yields

$$E = \max_{d} \sum_{\bar{x}_{k+1}^{n-1}} \sum_{x_n} \Pi_{i=1}^{n} P(x_i, e | x_{pa_i}, d) \sum_{Q_j \in Q} f_j(x_{Q_j})$$

We can now separate the components in the utility functions into those mentioning X_n, denoted by the index set t_n, and those not mentioning X_n, labeled with indexes $l_n = \{1,...,n\} - t_n$. Accordingly we produce

$$E = \max_d \sum_{\bar{x}_{k+1}^{(n-1)}} \sum_{x_n} \Pi_{i=1}^n P(x_i, e|x_{pa_i}, d) \cdot (\sum_{j \in l_n} f_j(x_{Q_j}) + \sum_{j \in t_n} f_j(x_{Q_j}))$$

$$E = \max_d [\sum_{\bar{x}_{k+1}^{(n-1)}} \sum_{x_n} \Pi_{i=1}^n P(x_i, e|x_{pa_i}, d) \sum_{j \in l_n} f_j(x_{Q_j})$$

$$+ \sum_{\bar{x}_{k+1}^{(n-1)}} \sum_{x_n} \Pi_{i=1}^n P(x_i, e|x_{pa_i}, d) \sum_{j \in t_n} f_j(x_{Q_j})]$$

By migrating to the left of X_n all of the elements that are not a function of X_n, we get

$$\max_d [\sum_{\bar{x}_{k+1}^{n-1}} \Pi_{X_i \in X - F_n} P(x_i, e|x_{pa_i}, d) \cdot (\sum_{j \in l_n} f_j(x_{Q_j})) \sum_{x_n} \Pi_{X_i \in F_n} P(x_i, e|x_{pa_i}, d)$$
$$+ \sum_{\bar{x}_{k+1}^{n-1}} \Pi_{X_i \in X - F_n} P(x_i, e|x_{pa_i}, d) \cdot \sum_{x_n} \Pi_{X_i \in F_n} P(x_i, e|x_{pa_i}, d) \sum_{j \in t_n} f_j(x_{Q_j})] \quad (8)$$

We denote by U_n the subset of variables that appear with X_n in a probabilistic component, excluding X_n itself, and by W_n the union of variables that appear in probabilistic and utility components with X_n, excluding X_n itself. We define λ_n over U_n as (x is a tuple over $U_n \cup X_n$)

$$\lambda_n(x_{U_n}|d) = \sum_{x_n} \Pi_{X_i \in F_n} P(x_i, e|x_{pa_i}, d) \quad (9)$$

We define θ_n over W_n as

$$\theta_n(x_{W_n}|d) = \sum_{x_n} \Pi_{X_i \in F_n} P(x_i, e|x_{pa_i}, d) \sum_{j \in t_n} f_j(x_{Q_j})) \quad (10)$$

After substituting Eqs. (9) and (10) into Eq. (8), we get

$$E = \max_d \sum_{\bar{x}_{k+1}^{n-1}} \Pi_{X_i \in X - F_n} P(x_i, e|x_{pa_i}, d) \cdot \lambda_n(x_{U_n}|d) [\sum_{j \in l_n} f_j(x_{Q_j}) + \frac{\theta_n(x_{W_n}|d)}{\lambda_n(x_{U_n}|d)}] \quad (11)$$

The functions θ_n and λ_n compute the effect of eliminating X_n. The result (Eq. (11)) is an expression which does not include X_n, where the product has one more matrix λ_n and the utility components have one more element $\gamma_n = \frac{\theta_n}{\lambda_n}$. Applying such algebraic manipulation to the rest of the variables in order, yields the elimination algorithm *elim-meu* in Figure 22. Each bucket contains utility components, θ_i, and probability components, λ_i. Variables can be marked as

Algorithm elim-meu
Input: A belief network $BN = \{P_1, ..., P_n\}$; a subset of decision variables $D_1, ..., D_k$ that are root nodes; a utility function over X, $u(x) = \sum_j f_j(x_{Q_j})$; an ordering of the variables, o, in which the D's appear first; observations e.
Output: An assignment $d_1, ..., d_k$ that maximizes the expected utility.
1. **Initialize:** Partition components into buckets, where $bucket_i$ contains all matrices whose highest variable is X_i. Call probability matrices $\lambda_1, ..., \lambda_j$ and utility matrices $\theta_1, ..., \theta_l$. Let $S_1, ..., S_j$ be the scopes of probability functions and $Q_1, ..., Q_l$ be the scopes of the utility functions.
2. **Backward:** For $p \leftarrow n$ downto $k+1$, do
for all matrices $\lambda_1, ..., \lambda_j, \theta_1, ..., \theta_l$ in $bucket_p$, do

- **If** (observed variable) $bucket_p$ contains the observation $X_p = x_p$, then assign $X_p = x_p$ to each λ_i, θ_i and puts each resulting matrix in appropriate bucket.
- **else**, $U_p \leftarrow \bigcup_{i=1}^{j} S_i - \{X_p\}$ and $W_p \leftarrow U_p \cup (\bigcup_{i=1}^{l} Q_i - \{X_p\})\}$. If X_p is marked then $\lambda_p = \sum_{X_p} \Pi_i \lambda_i$ and $\theta_p = \frac{1}{\lambda_p} \sum_{X_p} \Pi_{i=1}^{j} \lambda_i \sum_{j=1}^{l} \theta_j$; else, $\theta_p = \sum_{X_p} \Pi_{i=1}^{j} \lambda_i \sum_{j=1}^{l} \theta_j$. Add θ_p and λ_p to the bucket of the largest-index variable in W_p and U_p, respectively.

3. **Forward:** Assign values in the ordering $o = D_1, ..., D_k$ using the information recorded in each bucket of the decision variables.

Figure 22: Algorithm *elim-meu*

relevant or irrelevant as in the elim-bel case. If a bucket is irrelevant λ_n is a constant. Otherwise, during processing, the algorithm generates the λ_i of a bucket by multiplying all its probability components and summing over X_i. The θ_i of bucket X_i is computed as the average utility of the bucket; if the bucket is marked, the average utility of the bucket is normalized by its λ. The resulting θ_i and λ_i are placed into the appropriate buckets.

Finally, the maximization over the decision variables can now be accomplished using maximization as the elimination operator. We do not include this step explicitly; given our simplifying assumption that all decisions are root nodes, this step is straightforward.

Example 5 *Consider the network of Figure 13 augmented by utility components and two decision variables D_1 and D_2. Assume that there are utility functions $u(f,g)$, $u(b,c)$, $u(d)$ such that the utility of a value assignment is the sum $u(f,g)+u(b,c)+u(d)$. The decision variables D_1 and D_2 have two options. Decision D_1 affects the outcome at G as specified by $P(g|f,D_1)$, while D_2 affects variable A as specified by $P(a|D_2)$. The modified belief network is shown in Figure 23. The bucket's partitioning and the schematic computation of this de-*

cision problem is given in Figure 24. Initially, bucket$_G$ contains $P(g|f, D_1)$, $u(f,g)$ and $g = 1$. Since the bucket contains an observation, we generate $\lambda_G(f, D_1) = P(g = 1|f, D_1)$ and $\theta_G(f) = u(f, g = 1)$ and put both in bucket F. Next, bucket D, which contains only $P(d|b,a)$ and $u(d)$, is processed. Since this bucket is not marked, *it will not create a probabilistic term. The utility term: $\theta_D(b,a) = \sum_d P(d|b,a)u(d)$ is created and placed in bucket B. Subsequently, when bucket F is processed, it generates the probabilistic component $\lambda_F(b, c, D_1) = \sum_f P(f|b,c)\lambda_G(f, D_1)$ and the utility component*

$$\theta_F(b, c, D_1) = \frac{1}{\lambda_F(b, c, D_1)} \sum_f P(f|b,c)\lambda_G(f, D_1)\theta_G(f)$$

Both new components are placed in bucket B. When bucket$_B$ is processed next, it creates the component $\lambda_B(a, c, D_1) = \sum_b P(b|a)\lambda_F(b, c, D_1)$ and

$$\theta_B(a, c, D_1) = \frac{1}{\lambda_B(a, c, D_1)} \sum_b P(b|a)\lambda_F(b, c, D_1)[u(b,c)+\theta_D(b,a)+\theta_G(b, c, D_1)].$$

Processing bucket$_C$ generates $\lambda_C(a, D_1) = \sum_c P(c|a)\lambda_B(a, c, D_1)$ and $\theta_C(a, D_1) = \frac{1}{\lambda_C(a,D_1)} \sum_c P(c|a)\lambda_B(a, c, D_1)\theta_B(a, c, D_1)$ while placing the two new components in bucket$_A$. Processing bucket$_A$ yields: $\lambda_A(D_1, D_2) = \sum_a P(a|D_2)\lambda_C(a, D_1)$ and $\theta_A(D_1, D_2) = \frac{1}{\lambda_A(D_1,D_2)} \sum_a P(a|D_2)\lambda_C(a, D_1)\theta_C(a, D_1)$, both placed in bucket$_{D_1}$. Bucket$_{D_1}$ is processed next by maximization generating $\theta_{D_1} D_2 = max_{D_1} \theta_A(D_1, D_2)$ which is placed in bucket$_{D_2}$. Now the decision of D_2 that maximizes $\theta_{D_1}(D_2)$, is selected. Subsequently, the decision that maximizes $\theta_A(D_1, D_2)$ tabulated in bucket$_{D_1}$, is selected.

As before, the algorithm's performance can be bounded as a function of the structure of its *augmented graph*. The augmented graph is the moral graph augmented with arcs connecting any two variables appearing in the same utility component f_i, for every i.

Theorem 10 *Algorithm elim-meu computes the meu of a belief network augmented with utility components (i.e., an influence diagram) in $O(n \cdot exp(w^*(d,e))$ time and space, where $w^*(d,e)$ is the adjusted induced width along d of the augmented moral graph.* □

Tatman and Schachter [50] have published an algorithm for the general influence diagram that is a variation of elim-meu. Kjaerulff's algorithm [29] can be viewed as a variation of elim-meu tailored to dynamic probabilistic networks.

8 Cost Networks and Dynamic Programming

As we have mentioned at the outset, bucket-elimination algorithms are variations of dynamic programming. Here we make the connection explicit, observing that elim-mpe is dynamic programming with some simple transformation.

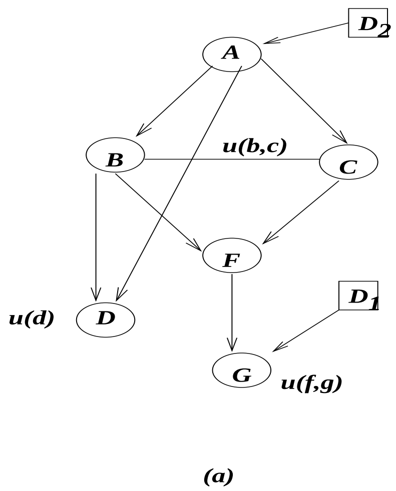

Figure 23: An influence diagram

$bucket_G$: $P(f|g, D_1)$, $g = 1$, $u(f, g)$
$bucket_D$: $P(d|b, a)$, $u(d)$
$bucket_F$: $P(f|b, c)$ $||$ $\lambda_G(f, D_1)$, $\theta_G(f)$
$bucket_B$: $P(b|a)$, $u(b, c)$ $||$ $\lambda_F(b, c, D_1)$, $\theta_D(b, a)$, $\theta_F(b, c, D_1)$
$bucket_C$: $P(c|a)$ $||$ $\lambda_B(a, c, D_1)$, $\theta_B(a, c, D_1)$
$bucket_A$: $P(a|D_2)$ $||$ $\lambda_C(a, D_1)$, $\theta_C(a, D_1)$
$bucket_{D_1}$: $||$ $\lambda_A(D_1, D_2)$, $\theta_A(D_1, D_2)$
$bucket_{D_2}$: $||$ $\theta_{D_1}(D_2)$

Figure 24: A schematic execution of elim-meu

Algorithm elim-opt
Input: A cost network $C = \{C_1, ..., C_l\}$; ordering o; assignment e.
Output: The minimal cost assignment.
1. **Initialize:** Partition the cost components into buckets.
2. **Process buckets** from $p \leftarrow n$ downto 1
For costs $h_1, h_2, ..., h_j$ in $bucket_p$, do:

- **If** (observed variable) $X_p = x_p$, assign $X_p = x_p$ to each h_i and put in buckets.
- **Else,** (sum and minimize)
$h^p = min_{X_p} \sum_{i=1}^{j} h_i$. Add h^p to its bucket.

3. **Forward:** Assign minimizing values in ordering o, consulting functions in each bucket.

Figure 25: Dynamic programming as elim-opt

That elim-mpe is dynamic programming becomes apparent once we transform the mpe's cost function, which has a product function, into the traditional additive function using the log function. For example, $P(a, b, c, d, f, g) = P(a)P(b|a)P(c|a)P(f|b, c)P(d|a, b)P(g|f)$ becomes $C(a, b, c, d, e) = -logP = C(a)+C(b, a)+C(c, a)+C(f, b, c)+C(d, a, b)+C(g, f)$ where each $C_i = -logP_i$. Indeed, the general dynamic programming algorithm is defined over *cost networks*. A *cost network* is a triplet (X, D, C), where $X = \{X_1, ..., X_n\}$ are variables over domains $D = \{D_1, ..., D_n\}$, C are real-valued cost functions $C_1, ..., C_l$. defined over subsets $S_i = \{X_{i_1}, ..., X_{i_r}\}$, $C_i : \bowtie_{j=1}^{r} D_{ij} \rightarrow R^+$. The *cost graph* of a *cost network* has a node for each variable and connects nodes denoting variables appearing in the same cost component. The task is to find an assignment to the variables that minimizes $\sum_i C_i$.

A straightforward elimination process similar to that of elim-mpe, (where the product is replaced by summation and maximization by minimization) yields the non-serial dynamic programming algorithm [5]. The algorithm, called *elim-opt*, is given in Figure 25.

A schematic execution of our example along ordering $d = G, A, F, D, C, B$ is depicted in Figure 26. Clearly,

Theorem 11 *Given a cost network, elim-opt generates a representation from which the optimal solution can be generated in linear time by a greedy procedure. The algorithm's complexity is time and space exponential in the cost-graph's adjusted induced-width.* $\square$

9 Relation with Other Methods

We show next that bucket-elimination is similar to a directional version of the poly-tree propagation for singly-connected networks and to a directional version

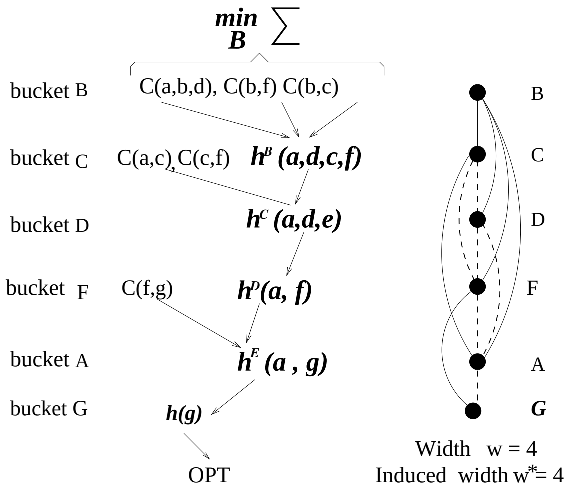

Figure 26: Schematic execution of elim-opt

of tree-clustering for general networks.

9.1 Poly-tree algorithm

When the belief network is a polytree, belief assessment, the mpe task and map task can be accomplished efficiently using Pearl's belief propagation algorithm [34]. As well, when the augmented graph is a tree, the *meu* can be computed efficiently. Bucket elimination is also guaranteed to be time and space linear on any polytree because the induced-width of its moral graph is bounded by its largest family and realizing an ordering having a minimal induced width is easy.

Theorem 12 *Given a polytree, a bucket-elimination's complexity is time and space exponential in the largest family size.* □

We next show that Pearl's *belief propagation* and bucket-elimination are very similar algorithms on polytrees. In fact, a directional version of belief propagation that computes the belief in a single proposition only, is identical to elim-bel that processes the buckets of each family as *super-buckets*.

A polytree is a directed acyclic graph whose underlying undirected graph has no cycles (see Figure 27(a)). Belief propagation (we assume familiarity with this algorithm) is a distributed, message-passing algorithm that computes the belief in each proposition by transmitting two messages, one in each direction, on each link. The messages are called λ's or π's, depending on whether they are transmitted upwards or downwards in the directed polytree.

If only a belief in a single proposition is desired, propagation can be restricted to one direction only. Messages will only propagate along the paths leading to

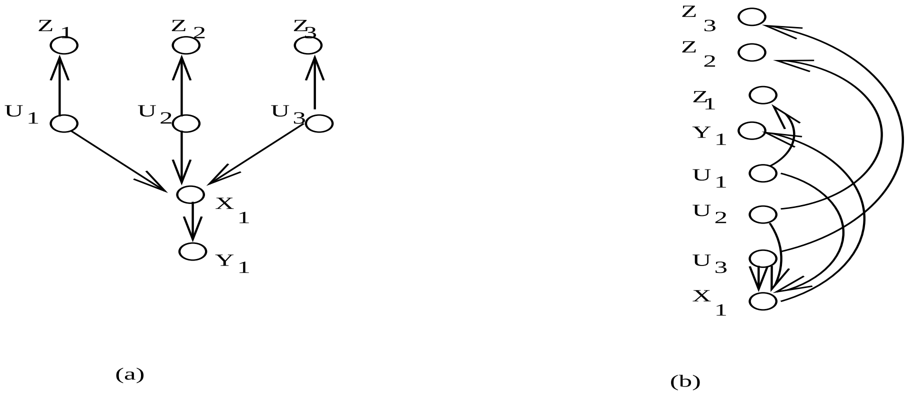

Figure 27: (a) A polytree and (b) a legal processing ordering

the queried variable. Many orderings can accommodate this message-passing and will be termed *legal orderings*. In particular, a reversed, breadth-first traversal ordering initiated at the queried proposition can be used.

We denote by DBP the *directional belief propagation* algorithm. Given a legal ordering, the algorithm processes the variables in a reverse ordering, where each node, in turn, computes and sends its message to its neighbor along the path to the root. We will show via an example that when using the same ordering, this directional version of belief propagation is identical to elim-bel.

Assume that we seek $P(x_1|z_1 = 0, z_2 = 0, z_3 = 0, y_1 = 0)$ for the network in Figure 27(a). A breadth-first ordering of the (underlying undirected) tree initiated at X_1 is $d = X_1, U_3, U_2, U_1, Y_1, Z_1, Z_2, Z_3$. (Clearly a breadth-first ordering is just one feasible ordering. In fact any ordering in which child nodes are eliminated before their parents is satisfactory.) DBP will send messages from the last variable to the first. The message sent by each Z_i towards U_i is $\lambda_{Z_i}(u_i) = P(z_i = 0|u_i) = P(z_i\prime|u_i)$ (we denote by primes instantiated variables and by lower case uninstantiated variables). The next message sent by Y_1 is $\lambda_{Y_1}(x_1) = P(y_1\prime|x_1)$. Subsequently, U_1, U_2, U_3, each multiply the messages they receive by $P(u_i)$, yielding $\pi_{U_i}(x_1) = P(u_i)\lambda_{Z_i}(u_i)$, which is sent to X_1. Variable X_1 now computes its π message as:

$$\pi(x_1) = P(x_1) \sum_{u_1,u_2,u_3} P(x_1|u_1, u_2, u_3)\pi_{U_1}(x_1)\pi_{U_2}(x_1)\pi_{U_3}(x_1). \qquad (12)$$

Finally, X_1 computes its own belief $Bel(x_1) = \pi(x_1) \cdot \lambda_{Y_1}(x_1)$.

Let us follow elim-bel's performance along the same ordering. The initial partitioning into buckets is:
$bucket(Z_3) = P(z_3|u_3),\ z_3 = 0.$
$bucket(Z_2) = P(z_2|u_2),\ z_2 = 0.$
$bucket(Z_1) = P(z_1|u_1),\ z_1 = 0.$

$bucket(Y_1) = P(y_1|x_1),\ y_1 = 0.$
$bucket(U_3) = P(u_3),\ P(x_1|u_1, u_2, u_3)$
$bucket(U_2) = P(u_2)$
$bucket(U_1) = P(u_1)$
$bucket(X_1) = P(x_1)$

Processing the initial buckets of Z_3, Z_2, and Z_1, generates $\lambda_{Z_i}(u_i) = P(z\prime_i|u_i)$. Each is placed in $bucket(U_i)$, $i \in \{1, 2, 3\}$. These functions are identical to the λ messages sent from the Z_is to the U_is by DBP. Likewise, bucket(Y_1) creates the function $\lambda_{Y_1}(x_1) = P(y\prime_1|x_1)$, which goes into bucket(X_1) and is identical to the message sent from Y_1 to X_1 by DBP. Processing buckets U_3, U_2 and U_1 produces $\lambda_{U_3}(u_1, u_2, x_1) = \sum_{u_3} P(x_1|u_1, u_2, u_3)\lambda_{z_1}(u_1)P(u_3)$, which is put in bucket(U_2). Processing bucket(U_2) generates $\lambda_{U_2}(u_1, x_1) = \sum_{u_2} \lambda_{U_2}(x_1, u_1, u_2)\lambda_{z_2}(u_2)P(u_2)$, which is positioned in bucket(U_1). Subsequently, bucket(U_1) yields $\lambda_{U_1}(x_1) = \sum_{u_1} \lambda_{U_2}(x_1, u_1)\lambda_{Z_1}(u_1)P(u_1)$ which is assigned to bucket(X_1). The combined computation in buckets U_1 U_2 and U_3 is equivalent to the message computed in Eq. (12). Notice that $\lambda_{U_1}(x_1)$ generated in bucket(U_1) is identical to the message $\pi(x_1)$ produced by BDP (Eq. (12)). Subsequently, in bucket(X_1) we take the product of all functions and normalize. The final resulting buckets are:
$bucket(Z_3) = P(z_3|u_3),\ z_3 = 0$.
$bucket(Z_2) = P(z_2|u_2),\ z_2 = 0.$
$bucket(Z_1) = P(z_1|u_1),\ z_1 = 0.$
$bucket(Y_1) = P(y_1|x_1),\ y_1 = 0.$
$bucket(U_3) = P(u_3), P(x_1|u_1, u_2, u_3),\ ||\ \lambda_{Z_3}(u_3)$
$bucket(U_2) = P(u_2),\ ||\ \lambda_{Z_2}(u_2),\ \lambda_{U_3}(x_1, u_2, u_1)$
$bucket(U_1) = P(u_1),\ ||\ \lambda_{Z_1}(u_1),\ \lambda_{U_2}(x_1, u_1)$
$bucket(X_1) = P(x_1)\ ||\ \lambda_{Y_1}(x_1),\ \lambda_{U_1}(x_1)$

We see that all the DBP messages map to functions recorded by elim-bel. However, in elim-bel we had two additional functions (generated in $bucket(U_3)$ and $bucket(U_2)$) that are avoided by DBP. A simple modification of elim-bel can avoid recording those functions: all the buckets that correspond to the same family can be processed simultaneously, as a single *super-bucket* where summation is applied over all the variables in the family. The two algorithms, directional belief propagation and elim-bel, become identical with these modifications.

The super-bucket idea can be implemented by using orderings that allow nodes of the same family to appear consecutively. These adjacent nodes are collected into super-buckets. Appropriate ordering is achieved by reversed breadth-first ordering. Processing a super-bucket amounts to eliminating all the super-bucket's variables without recording intermediate results.

Consider the polytree in Figure 27a and the ordering in 27b. Instead of processing each bucket(U_i) separately, we compute the function $\lambda_{U_1,U_2,U_3}(x_1)$ in the super-bucket bucket(U_1, U_2, U_3) and place the result in bucket(X_1), creating the unary function

$$\lambda_{U_1,U_2,U_3}(x_1) = \sum_{u_1,u_2,u_3} P(u_3)P(x_1|u_1,u_2,u_3)P(z'_3|u_3)P(u_2)P(z'_2|u_2)P(u_1)P(z'_1|u_1).$$

In summary, elim-bel with the super-bucket processing yields the following buckets:
$bucket(Z_3) = P(z_3|u_3),\ z_3 = 0$.
$bucket(Z_2) = P(z_2|u_2),\ z_2 = 0$.
$bucket(Z_1) = P(z_1|u_1),\ z_1 = 0$.
$bucket(Y_1) = P(y_1|x_1),\ y_1 = 0$.
$bucket(U_3, U_2, U_1) = P(u_3), P(u_2), P(u_1), P(x_1|u_1, u_2, u_3),\ ||\ \lambda_{Z_3}(u_3), \lambda_{Z_2}(u_2), \lambda_{Z_1}(u_1)$,
$bucket(X_1) = P(x_1)\ ||\ \lambda_{Y_1}(x_1),\ \lambda_{U_1,U_2,U_3}(x_1)$

We demonstrated that,

Proposition 13 *Given a polytree and a breadth-first ordering d, initiated at the queried variable, the set of functions generated by the modified elim-bel using d is identical to the messages generated by DBP when messages are generated in reverse order of d.* □.

9.2 Join-tree clustering

The polytree algorithm was extended to general networks by a method, similar to bucket elimination, known as *Join-tree clustering* [31]. The two algorithms (i.e., bucket-elimination and join-tree clustering) are closely related, and their worst-case complexity (time and space) is essentially the same (as already observed for constraint processing [20]).

Join-tree clustering is initiated by *triangulating* the moral graph along a given variable ordering. The maximal cliques (i.e., maximally fully connected subgraphs) of the triangulated graph are used to identify new subproblems that can be connected in a tree-like network called a join-tree. The complexity of tree-clustering is time and space exponential in the size of the largest clique because it is necessary to compile subproblems over the cliques (we assume familiarity with the join-tree algorithm). Since the triangulated graph is the same as the induced graph, the cliques' sizes in a join-tree clustering are identical to the induced-width (plus one). Therefore, the time and space complexity of join-tree clustering and bucket-elimination are the same.

This congruence in complexity is a result of the inherent similarity between the algorithms themselves. A directional version of join-tree clustering that updates a singleton belief is equivalent to elim-bel. In full tree-clustering, once the join-tree structure is identified, the cliques' potentials are compiled by taking the product of the conditional probabilities associated with each clique. Once the potentials are available, the problem resembles a tree. This allows message-passing between cliques in a manner similar to the message-passing in a polytree [34, 31].

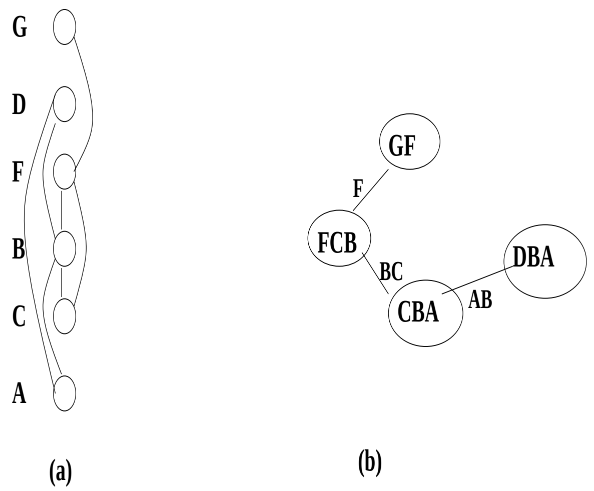

Figure 28: Clique-tree associated with the induced graph of Figure 7a

In a directional version of join-tree clustering, message-passing is restricted to one direction only: from leaf cliques towards a clique which contain the queried proposition, called a root. Moreover, instead of computing a clique's potential first, the computation of potentials and message-passing can be interleaved.

We will demonstrate next that a bucket-elimination trace corresponds to executing directional join-tree clustering. An ordering used by bucket-elimination generates a corresponding ordered induced moral graph (or triangulated moral graph). Each variable and its earlier neighbors in the induced moral graph form a clique, and each clique can be connected to an earlier clique with whom it shares a largest subset of variables [20]. For example, the induced graph in Figure 28(a) may generate the clique-tree in Figure 28(b).

The performance of directional join-tree parallels elim-bel in this example, assuming we seek the belief $P(a|g = 1)$. The ordering of cliques $d_1 = (CBA, FCB, DBA, GF)$ is used for join-tree clustering, while the corresponding ordering of $d_2 = (A, C, B, F, D, G)$ is used for elim-bel.

Join-tree processing GF**:** Its clique contains only $P(G|F)$, so no potential is updated. Message-passing assigns the value $g = 1$ resulting in the message $P(g = 1|f)$ that is propagated to clique FCB. The corresponding elim-bel step is to process bucket(G). $P(g = 1|F)$ is generated and put in Bucket(F).

Join-tree processing of DBA**:** The clique which contains $P(d|b, a)$, $P(b|a)$ and $P(a)$, results in the potential $h(d, b, a) = P(d|b, a)P(b|a)P(a)$. Subsequently, the message sent to clique CBA is the marginal $\sum_d h(a, b, d) = P(b|a)P(a)$. A corresponding elim-bel step processes Bucket(D), computing $\lambda_D(b, a) = \sum_D P(d|b, a)$, yields the constant 1. (Note that $P(b|a)$ already resides in bucket(B) and $P(a)$

is already in bucket(A)).

Join-tree process FCB**:** This clique contains $P(f|c,b)$ and the message $P(g=1|f)$. The potential remains $P(f|c,b)$. The message $\lambda_F(c,b) = \sum_F P(f|c,b)P(g=1|f)$ will then be computed and sent to clique CBA. Elim-bel processing $bucket(F)$ computes the same function $\lambda_F(b,c)$ and places it in $bucket(B)$.

Join-tree processing CBA: It computes the potential $h(a,b,c) = P(b|a)P(c|a)P(a)$, incorporates the message it received earlier, $\lambda_F(c,b)$, and computes the desired belief $P(a|g=1) = \alpha \sum_{b,c} h(a,b,c)\lambda_F(c,b)$. Elim-bel now processes buckets C, B, A in sequence (or alternatively the super-bucket $bucket(B,C,A)$) resulting in: $\lambda_C(a,b) = \sum_c P(c|a)\lambda_F(c,b)$ then $\lambda_B(a) = \sum_b P(b|a)\lambda_C((a,b)$ and finally, the desired belief: $P(a|g=1) = \alpha P(a)\lambda_B(a)$.

The relationship between directional join-tree clustering and bucket elimination provides semantics to the functions computed by bucket-elimination. Focusing first on the most probable explanation, mpe, the function $h_p(u)$ recorded in $bucket_p$ by elim-mpe and defined over $U_p = \cup_i S_i - \{X_p\}$, is the maximum probability extension of $U_p = u$, to variables appearing later in the ordering, and restricted to the clique-subtree rooted at the clique containing U_p. For instance, $h_F(b,c)$ recorded by elim-mpe equals $\max_{f,g} P(b,c,f,g)$, since F and G appear in the clique-tree rooted at clique FCB. For belief assessment the function $\lambda_p = \sum_{X_p} \Pi_{i=1}^{j} \lambda_i$, defined over $U_p = \cup_i S_i - X_p$, denotes the probability of all the evidence e^{+p} observed in the clique subtree rooted at a clique containing U_p, conjoined with u, specifically, $\lambda_p(u) = \alpha P(e^{+p}, u)$.

Algorithms for join-tree clustering in belief networks are sometimes ambiguous in that they seem to imply that only topological orderings are acceptable for triangulation. In fact, the tree-clustering algorithm is correct for any ordering. Its efficiency, however, (its clique size) indeed depends on the ordering selected. For tasks other than belief updating, the considerations for identifying good orderings are identical to those associated with constraint satisfaction. However, in belief updating, because of the ability to exploit relevant subgraphs, orderings that are consistent with the acyclic graph may be more suitable.

10 Combining Elimination and Conditioning

As noted earlier for deterministic reasoning, a serious drawback of elimination and clustering algorithms is that they require considerable memory for recording the intermediate functions. Conditioning search, on the other hand, requires only linear space. By combining conditioning and elimination, we may be able to reduce the amount of memory needed while still having performance guarantee.

Full conditioning for probabilistic networks is search, namely, traversing the tree of partial value assignments and accumulating the appropriate sums of probabilities. (It can be viewed as an algorithm for processing the algebraic expressions from left to right, rather than from right to left as was demonstrated

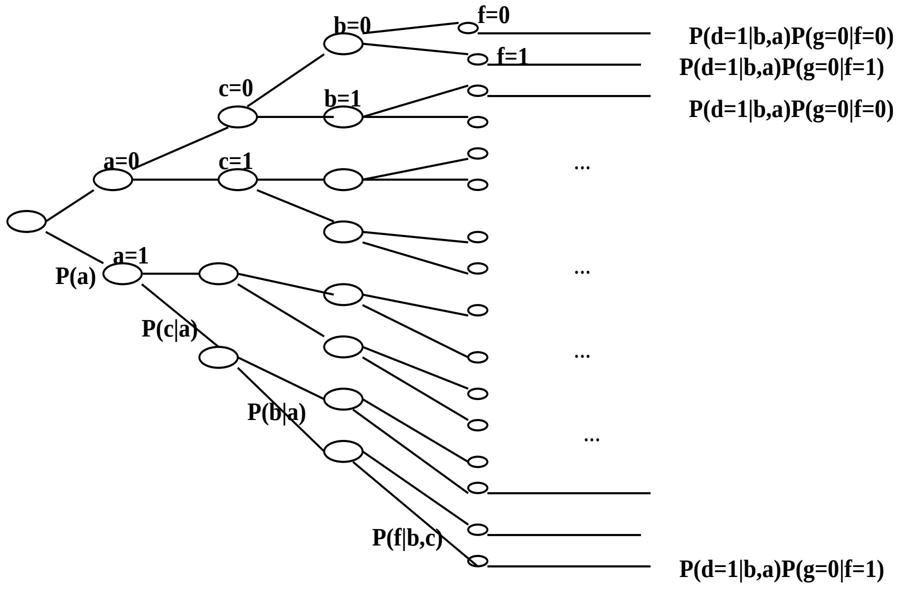

Figure 29: probability tree

for elimination). For example, we can compute the expression for mpe in the network of Figure 13:

$$M = \max_{a,c,b,f,d,g} P(g|f)P(f|b,c)P(d|a,b)P(c|a)P(b|a)P(a)$$

$$= \max_a P(a) \max_c P(c|a) \max_b P(b|a) \max_f P(f|b,c) \max_d P(d|b,a) \max_g P(g|f), \tag{13}$$

by traversing the tree in Figure 29, going along the ordering from first variable to last variable. The tree can be traversed either breadth-first or depth-first resulting in algorithms such as best-first search and branch and bound, respectively.

We will demonstrate one scheme of combining conditioning with elimination using the mpe task.

Notation: Let X be a subset of variables and $V = v$ be a value assignment to V. $f(X)|_v$ denotes the function f where the arguments in $X \cap V$ are assigned the corresponding values in v.

Let C be a subset of conditioned variables, $C \subseteq X$, and $V = X - C$. We denote by v an assignment to V and by c an assignment to C. Obviously,

$$\max_x P(x,e) = \max_c \max_v P(c,v,e) = max_{c,v} \Pi_i P(x_i|x_{pa_i})|_{(c,v,e)}$$

Therefore, for every partial tuple c, we can compute $\max_v P(v,c,e)$ and a cor-

Algorithm elim-cond-mpe
Input: A belief network $BN = \{P_1, ..., P_n\}$; an ordering of the variables, d; a subset C of conditioned variables; observations e.
Output: The most probable assignment.
Initialize: $p = 0$.

1. For every assignment $C = c$, do
   - $p_1 \leftarrow$ The output of elim-mpe with $c \cup e$ as observations.
   - $p \leftarrow \max\{p, p_1\}$ (update the maximum probability).
2. **Return** p and a maximizing tuple.

Figure 30: Algorithm *elim-cond-mpe*

responding maximizing tuple

$$(x_V^o)(c) = argmax_V \Pi_{i=1}^n P(x_i|x_{pa_i})|_{(c,e)}$$

using variable elimination, while treating the conditioned variables as observed variables. This basic computation will be enumerated for all value combinations of the conditioned variables, and the tuple retaining the maximum probability will be kept. This straightforward algorithm is presented in Figure 30.

Given a particular value assignment c, the time and space complexity of computing the maximum probability over the rest of the variables is bounded exponentially by the induced width $w^*(d, e \cup c)$ of the ordered moral graph along d adjusted for both observed and conditioned nodes. Therefore, the induced graph is generated without connecting earlier neighbors of both evidence and conditioned variables.

Theorem 14 *Given a set of conditioning variables, C, the space complexity of algorithm elim-cond-mpe is $O(n \cdot exp(w^*(d, c \cup e))$, while its time complexity is $O(n \cdot exp(w^*(d, e \cup c) + |C|))$, where the induced width $w^*(d, c \cup e)$, is computed on the ordered moral graph that was adjusted relative to e and c.* □

When the variables in $e \cup c$ constitute a cycle-cutset of the graph, the graph can be ordered so that its adjusted induced width equals 1 and elim-cond-mpe reduces to the known loop-cutset algorithms [34, 12].

In general Theorem 14 calls for a secondary optimization task on graphs:

Definition 6 (secondary-optimization task) *Given a graph $G = (V, E)$ and a constant r, find a smallest subset of nodes C_r, such that $G\prime = (V - C_r, E\prime)$, where $E\prime$ includes all the edgs in E that are not incident to nodes in C_r, has induced-width less or equal r.*

Clearly, the minimal cycle-cutset corresponds to the case where the induced-width is $r = 1$. The general task is clearly NP-complete.

Clearly, algorithm elim-cond-mpe can be implemented more effectively if we take advantage of shared partial assignments to the conditioned variables. There are a variety of possible hybrids between conditioning and elimination that can refine this basic procedure. One method imposes an upper bound on the arity of functions recorded and decides dynamically, during processing, whether to process a bucket by elimination or by conditioning (see [40]). Another method which uses the super-bucket approach collects a set of consecutive buckets into one super-bucket that it processes by conditioning, thus avoiding recording some intermediate results [15, 24]. See also [9].

11 Additional related work

We have mentioned throughout this paper algorithms in probabilistic and deterministic reasoning that can be viewed as bucket-elimination algorithms. Among those are the peeling algorithm for genetic trees [8], Zhang and Poole's VE1 algorithm [51] which is identical to elim-bel, SPI algorithm by D'Ambrosio et.al., [39] which preceded both elim-bel and VE1 and provided the principle ideas in the context of belief updating. Decimation algorithms in statistical physics are also related and were applied to Boltzmann trees [43]. We also made explicit the observation that bucket elimination algorithms resemble tree-clustering methods, an observation that was made earlier in the context of constraint satisfaction tasks [20].

The observation that a variety of tasks allow efficient algorithms of hyper-trees and therefore can benefit from a tree-clustering approach was recognized by several works in the last decade. In [38] the connection between optimization and constraint satisfaction and its relationship to dynamic programming is explicated. In the work of [33, 47] and later in [6] an axiomatic framework that characterize tasks that can be solved polynomially over hyper-trees, is introduced. Such tasks can be described using *combination* and *projection* operators over real-valued functions, and satisfy a certain set of axioms. The axiomatic framework [47] was shown to capture optimization tasks, inference problems in probabilistic reasoning, as well as constraint satisfaction. Indeed, the tasks considered in this paper can be expressed using operators obeying those axioms and therefore their solution by tree-clustering methods follows. Since, as shown in [20] and here, tree-clustering and bucket elimination schemes are closely related, tasks that fall within the axiomatic framework [47] can be accomplished by bucket elimination algorithms as well. In [6] a different axiomatic scheme is presented using semi-ring structures showing that impotent semi-rings characterize the applicability of constraint propagation algorithms. Most of the tasks considered here do not belong to this class.

In contrast, the contribution of this paper is in making the derivation process of variable elimination algorithms from the algebraic expression of the tasks, explicit. This makes the algorithms more accessible and their properties better understood. The associated complexity analysis and the connection to graph parameters are also made explicit. Task specific properties are also studied (e.g, irrelevant buckets in belief updating).

The work we show here also fits into the framework developed by Arnborg and Proskourowski [2, 1]. They present table-based reductions for various NP-hard graph problems such as the independent-set problem, network reliability, vertex cover, graph k-colorability, and Hamilton circuits. Here and elsewhere [22, 16] we extend the approach to a different set of problems.

The following paragraphs summarize and generalizes the bucket elimination algorithm using two operators of *combination* and *marginalization*. The task at hand can be defined in terms of a triplet (X, D, F) where $X = \{X_1, ..., X_n\}$ is a set of variables having domain of values $\{D_1, ..., D_n\}$. and $F = \{f_1, ..., f_k\}$ is a set of functions, where each f_i is defined over a scope $S_i \subseteq X$. Given a function h defined over scope $S \subseteq X$, and given $Y \subseteq S$, the (generalized) projection operator $\Downarrow_Y f$ is defined by enumeration as $\Downarrow_Y h \in \{max_{S-Y} h, min_{S-Y} h, \Pi_{S-Y} h, \sum_{S-Y} h\}$ and the (generalized) combination operator $\otimes_j f_j$ is defined over $U = \cup_j S_j$. $\otimes_{j=1}^k f_j \in \{\Pi_{j=1}^k f_j, \sum_{j=1}^k f_j, \Join_j f_j\}$.

The problem is to compute

$$\Downarrow_Y \otimes_{i=1}^n f_i$$

(In [47] the f_i are called valuations.) We showed that such problems can be solved by the bucket-elimination algorithm, stated using this general form in Figure 31. For example, elim-bel is obtained when $\Downarrow_Y = \sum_{S-Y}$ and $\otimes_j = \Pi_j$, elim-mpe is obtained when $\Downarrow_Y = \max_{S-Y}$ and $\otimes_j = \Pi_j$, and adaptive consistency corresponds to $\Downarrow_Y = \Pi_{S-Y}$ and $\otimes_j = \Join_j$. Similarly, Fourier elimination, directional resolution as well as elim-meu can be shown to be expressible in terms of such operators.

12 Conclusion

The paper describes the bucket-elimination framework which unifies variable elimination algorithms appearing for deterministic and probabilistic reasoning as well as for optimization tasks. In this framework, the algorithms exploit the structure of the relevant network without conscious effort on the part of the designer. Most bucket-elimination algorithms[4] are time and space exponential in the induced-width of the underlying dependency graph of the problem.

The simplicity of the proposed framework highlights the features common to bucket-elimination and join-tree clustering, and allows focusing belief-assessment procedures on the relevant portions of the network. Such enhancements were

[4] all, except Fourier algorithm.

Algorithm bucket-elimination
Input: A set of functions $F = \{f_1, ..., f_n\}$ over scopes $S_1, ..., S_n$; an ordering of the variables, $d = X_1, ..., X_n$; A subset Y.
Output: A new compiled set of functions
from which $\Downarrow_Y \otimes_{i=1}^{n} f_i$ can be derived in linear time.
1. **Initialize:** Generate an ordered partition of the functions into $bucket_1, ..., bucket_n$, where $bucket_i$ contains all the functions whose highest variable in their scope is X_i. Let $S_1, ..., S_j$ be the subset of variables in the processed bucket on which functions (new or old) are defined.
2. **Backward:** For $p \leftarrow n$ downto 1, do
for all the functions $\lambda_1, \lambda_2, ..., \lambda_j$ in $bucket_p$, do

- **If** (observed variable) $X_p = x_p$ appears in $bucket_p$, assign $X_p = x_p$ to each λ_i and then put each resulting function in appropriate bucket.
- **else**, $U_p \leftarrow \bigcup_{i=1}^{j} S_i - \{X_p\}$. Generate $\lambda_p = \Downarrow_{U_p} \otimes_{i=1}^{j} \lambda_i$ and add λ_p to the largest-index variable in U_p.

3. **Return:** all the functions in each bucket.

Figure 31: Algorithm *bucket-elimination*

accompanied by graph-based complexity bounds which are even more refined than the standard induced-width bound.

The performance of bucket-elimination and tree-clustering algorithms suffers from the usual difficulty associated with dynamic programming: exponential space and exponential time in the worst case. Such performance deficiencies also plague resolution and constraint-satisfaction algorithms [21, 16]. Space complexity can be reduced using conditioning search. We have presented one generic scheme showing how conditioning can be combined on top of elimination, reducing the space requirement while still exploiting topological features.

In summary, we provided a uniform exposition across several tasks, applicable to both probabilistic and deterministic reasoning, which facilitates the transfer of ideas between several areas of research. More importantly, the organizational benefit associated with the use of buckets should allow all the bucket-elimination algorithms to be improved uniformly. This can be done either by combining conditioning with elimination as we have shown, or via approximation algorithms as is shown in [16].

Finally, no attempt was made in this paper to optimize the algorithms for distributed computation, nor to exploit compilation vs. run-time resources. These issues should be addressed. In particular, improvements exploiting the structure of the conditional probability matrices as presented recently in [42, 7, 37] can be incorporated on top of bucket-elimination.

13 Acknowledgment

A preliminary version of this paper appeared in [14]. An extension restricted to probabilistic reasoning only appears in [17]. I would like to thank Irina Rish and Nir Freidman for their useful comments on different versions of this paper. This work was partially supported by NSF grant IIS-9610015 , and Rockwell MICRO grants ACM-20775 and 95-043.

References

[1] S. Arnborg and A. Proskourowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete and Applied Mathematics.*

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

[3] F. Bacchus and P. van Run. Dynamic variable ordering in csps. In *Principles and Practice of Constraints Programming (CP'95)*, Cassis, France, 1995. Available as Lecture Notes on CS, vol 976, pp 258 – 277, 1995.

[4] 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.

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

[6] 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.

[7] C. Boutilier. Context-specific independence in bayesian networks. In *Uncertainty in Artificial Intelligence (UAI'96)*, pages 115–123, 1996.

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

[9] A. Darwiche. Conditioning methods for exact and approximate inference in ca usal networks. In *Proceedings of the 11th Conference on Uncertainty in Artifi cial Intelligence (UAI95)*, pages 99–107, 1995.

[10] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

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

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

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

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

[15] R. Dechter. Topological parameters for time-space tradeoffs. In *Uncertainty in Artificial Intelligence (UAI'96)*, pages 220–227, 1996.

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

[17] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference algorithms. In *M. I. Jordan (Ed.), Learning in Graphical Models, Kluwer Academic Press*, 1998.

[18] R. Dechter and D. Frost. Backtracking algorithms for constraint satisfaction problems – a tutorial survey. In *UCI technical report. Also on web page www.ics.uci.edu/~dechter*, 1998.

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

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

[21] 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.

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

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

[24] Y. El-Fattah and R. Dechter. An evaluation of structural parameters for probabilistic reasoning: results on benchmark circuits. In *Uncertainty in Artificial Intelligence (UAI'96)*, pages 244–251, 1996.

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

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

[27] D. Geiger, T. Verma, and J. Pearl. Identifying independence in bayesian networks. *Networks*, 20:507–534, 1990.

[28] F.V. Jensen, S.L Lauritzen, and K.G. Olesen. Bayesian updating in causal probabilistic networks by local computation. *Computational Statistics Quarterly*, 4:269–282, 1990.

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

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

[31] 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.

[32] Z Li and B. D'Ambrosio. Efficient inference in bayes networks as a combinatorial optimization problem. *International Journal of Approximate Reasoning*, 11:1–58, 1994.

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

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

[35] Y. Peng and J.A. Reggia. Plausability of diagnostic hypothesis. In *National Conference on Artificial Intelligence (AAAI'86)*, pages 140–145, 1986.

[36] Y. Peng and J.A. Reggia. A connectionist model for diagnostic problem solving. *IEEE Transactions on Systems, Man and Cybernetics*, 1989.

[37] D. Poole. Probabilistic partial evaluation: Exploiting structure in probabilistic inference. In *IJCAI-97: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1997.

[38] 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.

[39] 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.

[40] I. Rish and R. Dechter. To guess or to think? hybrid algorithms for sat. In *Principles of Constraint Programming (CP-96)*, pages 555–556, 1996.

[41] E. Santos. On the generation of alternative explanations with implications for belief revision. In *Uncertainty in Artificial Intelligence (UAI'91)*, pages 339–347, 1991.

[42] E. Santos, S.E. Shimony, and E. Williams. Hybrid algorithms for approximate belief updating in bayes nets. *International Journal of Approximate Reasoning*, in press.

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

[44] R. D. Shachter. An ordered examination of influence diagrams. *Networks*, 20:535–563, 1990.

[45] R.D. Shachter. Evaluating influence diagrams. *Operations Research*, 34, 1986.

[46] R.D. Shachter. Probabilistic inference and influence diagrams. *Operations Research*, 36, 1988.

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

[48] S.E. Shimony and E. Charniak. A new algorithm for finding map assignments to belief networks. In *P. Bonissone, M. Henrion, L. Kanal, and J. Lemmer (Eds.), Uncertainty in Artificial Intelligence*, volume 6, pages 185–193, 1991.

[49] 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.

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

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