REASONING WITH GRAPHICAL MODELS

Rina Dechter

October 30, 2007

Contents

Notation

$\mathcal{R}$	a constraint network
$x_1, \ldots, x_n$	variables
n	the number of variables in a constraint network
D_i	the domain of variable x_i
X, Y, Z	sets of variables
R, S, T	relations
r, s, t	tuples in a relation
$< x_1, a_1 >< x_2, a_2 >, ..., < x_n, a_n >$	an assignment tuple
$\sigma_{x_1=d_1,\ldots,x_k=d_k}(R)$	the selection operation on relations
$\Pi_Y(R)$	the projection operatoin on relations
$\lceil x \rceil$	the integer n such that $x \leq n \leq x+1$

Chapter 1

Introduction

Graphical models, including constraint networks, belief networks, Markov random fields and influence diagrams, are knowledge representation schemes that capture independencies in the knowledge base and support efficient, graph-based algorithms for a variety of reasoning tasks, including scheduling, planning, diagnosis and situation assessment, design, and hardware and software verification.

Algorithms for processing graphical models are of two primary types: inference-based and search-based. Inference-based algorithms (e.g., variable-elimination, join-tree clustering) are time and space exponentially bounded by the tree-width of the problem's graph. Search-based algorithms can be executed in linear space and often outperform their worst-case predictions. The thrust of advanced schemes is in combining inference and search yielding a spectrum of memory-sensitive algorithms universally applicable across graphical models.

Graphical models are a widely used knowledge representation framework that captures independencies in the data and allowing a concise representation and efficient query processing. Essential to a graphical model is the underlying graph that captures the problem structure. The vertices are the variables of interest, and the edges represent the interactions and dependencies between them (e.g., propositional clauses, constraints, probabilities, and utilities). Known examples include Bayesian (or belief) networks, constraint networks, Markov random fields and influence diagrams. There are numerous examples of problems defined as graphical models, including design, scheduling, planning, diagnosis, decision making or genetic linkage analysis. The class notes is focused on reasoning in graphical frameworks such as constraint and belief networks. Some reasoning tasks can be formulated as combinatorial optimization or constraint satisfaction problems, while others can be viewed as knowledge compilation, counting or likelihood computation. We approach those tasks using a general graph-based algorithmic framework that combines dynamic-programming techniques or *variable elimination*, often referred to as *inference* with *conditioning or search*, and investigate the effect of problem structure on the performance of such algorithms.

In this section we will define the frameworks of belief networks, constraint networks as well as cost networks and their associated tasks. we will subsequently generalize it all to the graphical model framework. We will use the following notations and definitions throughout the book.

Notations We denote variables or subsets of variables by uppercase letters (*e.g.*, $X, Y, \ldots$) and values of variables by lower case letters (*e.g.*, $x, y, \ldots$). Sets are usually denoted by bold letters, for example $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a set of variables. An assignment $(X_1 = x_1, \ldots, X_n = x_n)$ can be abbreviated as $x = (\langle X_1, x_1\rangle, \ldots, \langle X_n, x_n\rangle)$ or $x = (x_1, \ldots, x_n)$. For a subset of variables $\mathbf{Y}$, $D_{\mathbf{Y}}$ denotes the Cartesian product of the domains of variables in $\mathbf{Y}$. The projection of an assignment $x = (x_1, \ldots, x_n)$ over a subset $\mathbf{Y}$ is denoted by $x_{\mathbf{Y}}$ or $x[\mathbf{Y}]$. We will also denote by $Y = y$ (or y for short) the assignment of values to variables in $\mathbf{Y}$ from their respective domains. We denote functions by letters f, g, h etc., and the scope (set of arguments) of the function f by *scope*(f).

1.1 Example Graphical Models

Graphical models include constraint networks [14] defined by relations of allowed tuples, (directed or undirected) probabilistic networks [30], defined by conditional probability tables over subsets of variables, cost networks defined by costs functions and influence diagrams [23] which include both probabilistic functions and cost functions (*i.e.*, utilities) [13]. Each graphical model comes with its typical queries, such as finding a solution, or an optimal one (over constraint networks), finding the most probable assignment or updating the posterior probabilities given evidence, posed over probabilistic networks, or finding optimal solutions for cost networks. The task for influence diagrams is to choose a sequence of actions that maximizes the expected utility. Markov random fields are the undirected counterparts of probabilistic networks. They are defined by a collection of probabilistic functions called potentials, over arbitrary subsets of variables.

Throughout the book, we will use the two examples of graphical models: constraint networks and belief networks. In the case of constraint networks, the functions can be understood as relations. In other words, the functions (also called constraints) can take only two values, $\{0, 1\}$ (or $\{true, false\}$). A 0 value indicates that the corresponding assignment to the variables is inconsistent (not allowed), and a 1 value indicates consistency. Belief networks are an example of the more general case of graphical models (sometime called *weighted* graphical models). The functions are conditional probability tables, so the values of a function are any real number in the interval $[0, 1]$.

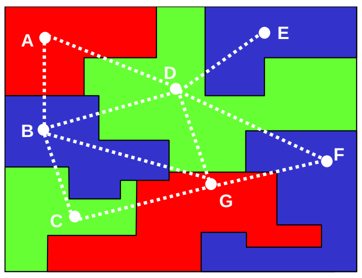
A *directed graph* is a pair $G = \{V, E\}$, where $V = \{X_1, \ldots, X_n\}$ is a set of vertices, and $E = \{(X_i, X_j) | X_i, X_j \in V\}$ is the set of edges (arcs). If $(X_i, X_j) \in E$, we say that X_i *points to* X_j. The degree of a variable is the number of arcs incident to it. For each variable X_i, $pa(X_i)$ or pa_i, is the set of variables pointing to X_i in G, while the set of child vertices of X_i, denoted $ch(X_i)$, comprises the variables that X_i points to. 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. An *undirected graph* is defined similarly to a directed graph, but there is no directionality associated with the edges.

Definition 1.1.1 (hypergraph) *A* hypergraph *is a pair $H = (X, S)$, where $S = \{S_1, \ldots, S_t\}$ is a set of subsets of V called* hyperedges.

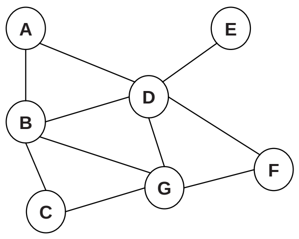
1.1.1 Constraint Networks

Constraint networks provide a framework for formulating real world problems, such as scheduling and design, planning and diagnosis, and many more as a set of constraints between variables. For example, one approach to formulating a scheduling problem as a constraint satisfaction problem (CSP) is to create a variable for each resource and time slice. Values of variables would be the tasks that need to be scheduled. Assigning a task to a particular variable (corresponding to a resource at some time slice) means that this resource starts executing the given task at the specified time. Various physical constraints (such as that a given job takes a certain amount of time to execute, or that a task can be executed at most once) can be modeled as constraints between variables. The *constraint satisfaction task* is to find an assignment of values to all the variables that does not violate any constraint, or else to conclude that the problem is inconsistent. Other tasks are finding all solutions and counting the solutions.

Definition 1.1.2 (constraint network, constraint satisfaction problem) *A* constraint network (CN) *is a 4-tuple, $\langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \bowtie \rangle$, where $\mathbf{X}$ is a set of variables $\mathbf{X} = \{X_1, \ldots, X_n\}$, associated with a set of discrete-valued domains, $\mathbf{D} = \{D_1, \ldots, D_n\}$, and a set of constraints $\mathbf{C} = \{C_1, \ldots, C_r\}$. Each constraint C_i is a pair $(\mathbf{S}_i, R_i)$, where R_i is a relation $R_i \subseteq D_{\mathbf{S}_i}$ defined on a subset of variables $\mathbf{S}_i \subseteq \mathbf{X}$. The relation denotes all compatible tuples of $D_{\mathbf{S}_i}$ allowed by the constraint. The $\bowtie$ simply note that the combination operator is join. T and will serve to unify constraint networks within graphical models. A solution is an assignment of a value to each variable that does not violate any constraint. A* solution *is an assignment of values to all the variables $x = (x_1, \ldots, x_n)$, $x_i \in D_i$, such that $\forall\ C_i \in \mathbf{C}$, $x_{\mathbf{S}_i} \in R_i$. The constraint network represents its set of solutions, $\bowtie_i C_i$.*



(a) Graph coloring problem



(b) Constraint graph

Figure 1.1: Constraint network

Definition 1.1.3 (constraint graph) *The* constraint graph *of a graphical model is an undirected graph that has variables as its vertices and an edge connects any two variables that appear in the scope of the same function.*

Example 1.1.4 Figure 1.1(a) shows a graph coloring problem that can be modeled by a constraint network. Given a map of regions, the problem is to color each region by one of the given colors {red, green, blue}, such that neighboring regions have different colors. The variables of the problems are the regions, and each one has the domain {red, green, blue}. The constraints are the relation *"different"* between neighboring regions. Figure 1.1(b) shows the constraint graph, and a solution (A=red, B=blue, C=green, D=green, E=blue, F=blue, G=red) is given in Figure 1.1(a). □

Cost Networks An immediate extension of constraint networks are *cost networks* where the constraints are replaces by cost components which are real-valued cost functions, and the primary task is optimization.

Definition 1.1.5 (cost network, combinatorial optimization) *A* cost network *is a 4-tuple, $\langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \sum \rangle$, where $\mathbf{X}$ is a set of variables $\mathbf{X} = \{X_1, \ldots, X_n\}$, associated with a set of discrete-valued domains, $\mathbf{D} = \{D_1, \ldots, D_n\}$, and a set of cost functions $\mathbf{C} = \{C_1, \ldots, C_r\}$. Each C_i is a real-valued function defined on a subset of variables $\mathbf{S}_i \subseteq \mathbf{X}$. The cost components are combined into a global cost function via the $\sum$ combination operator. The reasoning problem is to find a minimum cost solution for $F = \sum_i C_i$.*

The task of MAX-CSP, namely finding a solution that satisfies the maximum number of constraints (when the problem is inconsistent), can be formulated as a cost network task by treating each relation as a cost function that assigns "0" to consistent tuples and "1" otherwise.

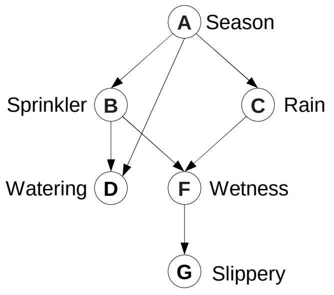
Propositional Satisfiability A special case of a CSP is *propositional satisfiability* (SAT). A formula φ in *conjunctive normal form* (CNF) is a conjunction of *clauses* $\alpha_1, \ldots, \alpha_t$, where a clause is a disjunction of *literals* (propositions or their negations). For example, $\alpha = (P \vee \neg Q \vee \neg R)$ is a clause, where P, Q and R are propositions, and P, $\neg Q$ and $\neg R$ are literals. The SAT problem is to decide whether a given CNF theory has a *model*, *i.e.*, a truth-assignment to its propositions that does not violate any clause. Propositional satisfiability (SAT) can be defined as a CSP, where propositions correspond to variables, domains are $\{0, 1\}$, and constraints are represented by clauses, for example the clause $(\neg A \vee B)$ is a relation over its propositional variables that allows all tuples over (A, B) except $(A = 1, B = 0)$.

1.1.2 Belief Networks

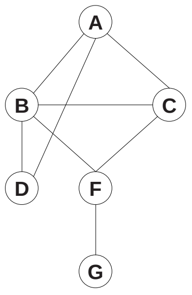
Belief networks [30], also known as Bayesian networks, provide a formalism for reasoning about partial beliefs under conditions of uncertainty. They are defined by a directed acyclic graph over vertices representing random variables of interest (*e.g.*, the temperature of a device, the gender of a patient, a feature of an object, the occurrence of an event). The arcs can signify the existence of direct causal influences between linked variables quantified by conditional probabilities that are attached to each cluster of parents-child vertices in the network. But these relationships need not necessarily be causal and we can still have a perfectly well defined belief network.

Definition 1.1.6 (belief networks) *A* belief network (BN) *is a graphical model $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$, where $\mathbf{X} = \{X_1, \ldots, X_n\}$ is a set of variables over multi-valued domains $\mathbf{D} = \{D_1, \ldots, D_n\}$. Given a directed acyclic graph G over $\mathbf{X}$ as nodes, $P_G = \{P_1, \ldots, P_n\}$, where $P_i = \{P(X_i \,|\, pa\,(X_i)\,)\}$ are conditional probability tables (CPTs for short) associated with each X_i, where $pa(X_i)$ are the parents of X_i in a given acyclic graph G. A belief network represents a probability distribution over $\mathbf{X}$, $P(x_1, \ldots, x_n) = \prod_{i=1}^n P(x_i | x_{pa(X_i)})$. where x_S is the projection of $x = (x_1, \ldots, x_n)$ over a subset S. An evidence set e is an instantiated subset of variables. The argument set of a function h is called the scope of h and is denoted scope(h).*

Example 1.1.7 Figure 1.2(a) gives an example of a belief network over 6 variables, and Figure 1.2(b) shows its moral graph . The example expresses the causal relationship between variables "Season" (A), "The configuration of an automatic sprinkler system" (B), "The amount of rain expected" (C), "The amount of manual watering necessary" (D), "The wetness of the pavement" (F) and "Whether or not the pavement is slippery" (G). The belief network expresses the probability distribution $P(A, B, C, D, F, G) = P(A) \cdot P(B|A) \cdot P(C|A) \cdot P(D|B, A) \cdot P(F|C, B) \cdot P(G|F)$. □



(a) Directed acyclic graph



(b) Moral graph

Figure 1.2: Belief network

The following queries are defined over belief networks:

Definition 1.1.8 (queries) *Given a belief network over $X = \{X_1, ..., X_n\}$ and given evidence e (x is a tuple over all variables):*

1. **Belief assessment:** *The belief assessment task of $X_i = x_i$ is to find $bel(x_i) = P(x_i|e)$.*

2. **Most probable explanation** (*mpe*)*: The mpe task is to find an assignment $x^o = (x^o{}_1, ..., x^o{}_n)$ such that $P(x^o) = \max_x P(x|e)$.*

3. **Maximum a posteriori hypothesis** (*map*)**:** *Given a set of hypothesized variables $A = \{A_1, ..., A_k\}$, $A \subseteq X$, the map task is to find an assignment $a^o = (a^o{}_1, ..., a^o{}_k)$ such that $P(a^o) = \max_{\bar{a}_k} \sum_{x_{X-A}} P(x|e)$.*

4. **Maximum expected utility** (*meu*)**:** *Given a real-valued utility function $u(x)$ that is additively decomposable relative to $Q_1, ..., Q_j$, $Q_i \subseteq X$, $u(x) = \sum_{Q_j \in Q} f_j(x_{Q_j})$, and given a subset of decision variables $D = \{D_1, ..., D_k\}$ that are root variables in the belief network, the meu task is to find an assignment $d^o = (d^o{}_1, ..., d^o{}_k)$ such that $(d^o) = argmax_d \sum_{X-D} P(x|d)u(x)$.*

These queries are applicable to tasks such as situation assessment, diagnosis and probabilistic decoding, as well as planning and decision making.

Markov networks are graphical models very similar to belief networks. The only difference is that the set of compatibility functions P_i, called potentials, can be defined over any subset of variables. An important reasoning task for Markov networks is to find the partition function which is equivalent to finding the probability of evidence in a directed probabilistic network.

1.2 General Graphical models

Next we provide a general formulation of graphical models and of reasoning problems that unifies all the previous models and tasks.

A graphical model is defined by a collection of functions F, over a set of variables X, conveying probabilistic, deterministic or preferential information, whose structure is captured by a graph.

Definition 1.2.1 (graphical model) *A* graphical model $\mathcal{M}$ *is a 4-tuple,* $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes \rangle$*, where:*

1. $\mathbf{X} = \{X_1, \ldots, X_n\}$ *is a finite set of variables;*
2. $\mathbf{D} = \{D_1, \ldots, D_n\}$ *is the set of their respective finite domains of values;*
3. $\mathbf{F} = \{f_1, \ldots, f_r\}$ *is a set of positive real-valued discrete functions, each defined over a subset of variables* $\mathbf{S}_i \subseteq \mathbf{X}$*, called its scope, and denoted by scope*(f_i)*.*
4. $\otimes$ *is a combination operator*[1] *(e.g.,* $\otimes \in \{\prod, \sum, \bowtie\}$ *(product, sum, join)).*

The graphical model represents the combination of all its functions: $\otimes_{i=1}^{r} f_i$*.*

Definition 1.2.2 (primal graph) *The* primal graph *of a graphical model is an undirected graph that has variables as its vertices and an edge connects any two variables that appear in the scope of the same function.*

The primal graph captures the structure of the knowledge expressed by the graphical model. In particular, graph separation indicates independency of sets of variables given some assignments to other variables. As we will see all of the advanced algorithms for graphical models exploit the graphical structure. There are many additional graph representations that are used.

The hypergraph of a graphical models has the set of variables as its nodes and the set of scopes of functions as its edges.

Definition 1.2.3 (reasoning problem) *A* reasoning problem *over a graphical model* $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes \rangle$ *, is defined by a marginalization operator* $\Downarrow_{\mathbf{Y}}$*, and a set of subsets of* $\mathbf{X}$ *that are of interest. It is therefore a triplet,* $\mathcal{P} = \langle \mathcal{M}, \Downarrow_{\mathbf{Y}}, \{\mathbf{Z}_1, \ldots, \mathbf{Z}_t\} \rangle$*, where* $\mathbf{Z} = \{\mathbf{Z}_1, \ldots, \mathbf{Z}_t\}$ *is a set of subsets of variables of* $\mathbf{X}$*. If* $\mathbf{S}$ *is the scope of function* f *and* $\mathbf{Y} \subseteq \mathbf{X}$*, then* $\Downarrow_{\mathbf{Y}} f \in \{ \overset{max}{\mathbf{S}-\mathbf{Y}} f, \overset{min}{\mathbf{S}-\mathbf{Y}} f, \overset{\prod}{\mathbf{Y}} f, \overset{\sum}{\mathbf{S}-\mathbf{Y}} f \}$ *is a marginalization operator.* $\mathcal{P}$ *can be viewed as a vector function over the scopes* $\mathbf{Z}_1, \ldots, \mathbf{Z}_t$*. The reasoning problem is to compute* $\mathcal{P}_{\mathbf{Z}_1,\ldots,\mathbf{Z}_t}(\mathcal{M}) = (\Downarrow_{\mathbf{Z}_1} \otimes_{i=1}^{r} f_i, \ldots, \Downarrow_{\mathbf{Z}_t} \otimes_{i=1}^{r} f_i)$.

[1]The combination operator can also be defined axiomatically [41].

We will focus primarily on reasoning problems defined by $\mathbf{Z} = \emptyset$. The marginalization operator is sometimes called *elimination* operator because it removes some arguments from the scope of the input function. Specifically, $\Downarrow_{\mathbf{Y}} f$ is a function whose scope is $\mathbf{Y}$. It therefore removes variables $\mathbf{S} - \mathbf{Y}$ from $\mathbf{S} = \mathit{scope}(f)$. Note that here $\prod_{\mathbf{Y}} f$ is the relational projection operator and unlike the rest of the marginalization operators the convention is that is defined by the scope of variables that are *not* eliminated. We will now go back and show how each of the framework mentioned earlier fits the general graphical model definition.

Constraint satisfaction is a reasoning problem $\mathcal{P} = \langle \mathcal{R}, \Pi, \mathbf{Z} \rangle$, where $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \bowtie \rangle$ is a constraint network, and the marginalization operator is the projection operator Π. Namely, for constraint satisfaction $\mathbf{Z} = \{\emptyset\}$, and $\Downarrow_{\mathbf{Y}}$ is $\Pi_{\mathbf{Y}}$. So the task is to find $\Downarrow_{\emptyset} \otimes_i f_i = \Pi_{\emptyset}(\bowtie_i f_i)$ which corresponds to enumerating all solutions. When the combination operator is a product over the cost-based representation of the relations, and the marginalization operator is logical summation we get "1" if the constraint problem has a solution and "0" otherwise. For *counting*, the marginalization operator is summation and $\mathbf{Z} = \{\emptyset\}$ too.

The task of MAX-CSP, namely finding a solution that satisfies the maximum number of constraints (when the problem is inconsistent), can be defined by treating each relation as a cost function that assigns "0" to consistent tuples and "1" otherwise. The combination operator is summation and the marginalization operator is minimization. Namely, the task is to find $\Downarrow_{\emptyset} \otimes_i f_i = \min_{\mathbf{X}}(\sum_i f_i)$.

Max-CSP can be expressed as minimizing the number of constraints that are violated. Its set of functions F is the set of cost functions assigning 0 to all allowed tuples and 1 to all non-allowed tuples. It can be formalized as a reasoning task $P = \langle X, D, F, \sum, min, Z = \emptyset \rangle$, where (X, D, F) is a constraint network, the combination operator is summation and the marginalization operator is the minimization operator. Namely, the task is to find $\Downarrow_{\emptyset} \otimes_i f_i = \min_X \sum_i f_i$.

A *belief network* is a graphical model. When belief networks are formulated as a graphical model, functions in F denote conditional probability tables and the scopes of these functions are determined by the directed acyclic graph G: each function f_i ranges over variable X_i and its parents in G. The combination operator is product, $\otimes = \prod$. The primal graph of a belief network is called a moral graph. It connects any two variables appearing in the same CPT.

Given a belief network and evidence e, the *belief updating* task of computing the posterior marginal probability of variable X_i, conditioned on the evidence. can be formulated using the marginalization operator is $\Downarrow_{\mathbf{Y}} = \sum_{\mathbf{X}-\mathbf{Y}}$, and $\mathbf{Z}_i = \{X_i\}$. Namely, $\forall X_i, \Downarrow_{X_i} \otimes_k f_k = \sum_{\{X - X_i | E = e\}} \prod_k P_k$. The query of finding the probability of the evidence is defined by $Z = \emptyset$.

As a reasoning problem, an MPE task is to find $\Downarrow_{\emptyset} \otimes_i f_i = \max_X \prod_i P_i$. Namely, the marginalization operator is max and $\mathbf{Z} = \{\emptyset\}$. Other queries can be formulated as...

In the rest of the chapters in this book we will describe inference algorithms, search algorithms and their hybrids. We start with inference algorithms such as bucket elimination in Chapter 2.

Flat functions Each function in a graphical model having a "0" element expresses implicitly a constraint. The *flat* constraint of function f_i is a constraint R_i over its scope that includes all and only the consistent tuples. In the following chapters, when we talk about a constraint network, we refer also to the flat constraint network that can be extracted from the general graphical model. When all the full assignments are consistent we say that the graphical model is *strictly positive*.

Chapter 2

Bucket-elimination Algorithms

[1]

In this chapter we introduce the first Inference algorithm for graphical models focusing on probabilistic networks.

2.1 Introduction

Bucket elimination is a unifying algorithmic framework that generalizes dynamic programming and to accommodate algorithms for many complex problem-solving and reasoning activities, including directional resolution for propositional satisfiability [9], adaptive consistency for constraint satisfaction [15], Fourier and Gaussian elimination for linear equalities and inequalities, and dynamic programming for combinatorial optimization [4]. 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.

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

[1]Adapted from[12]

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.

We begin (Section 2) with briefly describing some algorithms for deterministic networks, phrased 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 show that their performance can be bounded exponentially by the *induced-width* of the graph.

We will then provide a detailed derivation of bucket elimination algorithms for probabilistic tasks. Following additional preliminaries (Section 3), we will develop the bucket-elimination algorithm for belief updating and probability of evidence 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.

2.2 Adaptive-consistency; Bucket Elimination for Constraint Networks

This section describes algorithms for reasoning with deterministic relationships, emphasizing their syntactic description as bucket elimination algorithms.

2.2.1 Bucket elimination for constraints

Consider the following graph coloring problem in Figure 2.1. 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, called *Adaptive-consistency* [15] can solve any constraint satisfaction problem. 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 as follows. Given a variable ordering

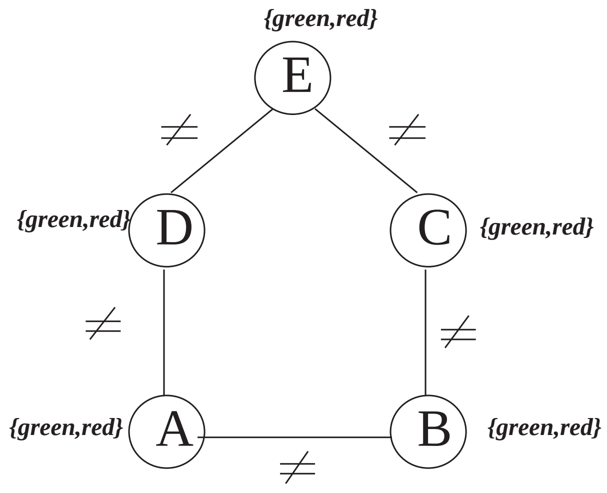


Figure 2.1: A graph coloring example

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 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 2.2a. 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 2.2b. 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 projecting the solutions to all but the current bucket's variable. A more

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

(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 2.2: A schematic execution of adaptive-consistency

formal description requires additional definitions and notations.

We assume that constraints are described by relations and the use of the following operations.

Definition 2.2.1 ((operations on constraints)) *Let R be a relation on a set S of variables, let $Y \subseteq S$ be a subset of the variables, and let Y_I be an instantiation of the variables in Y. We denote by $\sigma_{Y_I}(R)$ the selection of those tuples in R that agree with Y_I. We denote by $\Pi_Y(R)$ the projection of the relation R on the subset Y; that is, a tuple over Y appears in $\Pi_Y(R)$ if and only if it can be extended to a full tuple in R. Let R_{S_1} be a relation on a set S_1 of variables and let R_{S_2} be a relation on a set S_2 of variables. We denote by $R_{S_1} \bowtie R_{S_2}$ the natural join of the two relations. The join of R_{S_1} and R_{S_2} is a relation defined over $S_1 \cup S_2$ containing all the tuples t, satisfying $t[S_1] \in R_{S_1}$ and $t[S_2] \in R_{S_2}$.*

The computation in a bucket can be described in terms of the above 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. For instance, the computation in the bucket of E of our example of Figure 2.1 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 $E \neq C$. Algorithm Adaptive-consistency is described in Figure 2.3.

```
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 $X_p$:)
      $A \leftarrow \bigcup_{j=1}^{m} S_j - \{X_i\}$

      $R_A \leftarrow R_A \cap \Pi_A(\Join_{j=1}^{m} R_j)$
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 2.3: Algorithm Adaptive consistency

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 [15].

Definition 2.2.2 (induced-width,tree-width) *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.*

Consider for example, a slightly different graph coloring problem as depicted in Figure 2.4. 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 2.5. Note that in all drawings from now on, later nodes in the ordering appear on top of earlier ones. The broken arcs are the

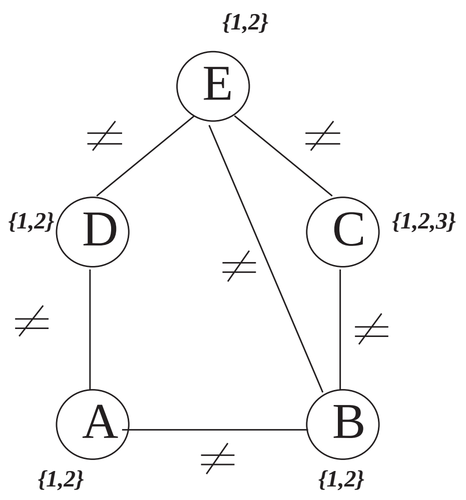


Figure 2.4: A modified graph coloring problem

new added arcs. The induced-width along d_1 and d_2 are 2 and 3 respectively, suggesting different complexity bounds for adaptive-consistency. We can show that,

Theorem 2.2.3 (correctness) *[15] Adaptive-consistency decides if a set of constraints are consistent, and if they are, generates an equivalent representation that is backtrack-free.* $\Box$

Theorem 2.2.4 (complexity) *The time and space complexity of Adaptive-consistency along d is $O(r \cdot exp(w^*(d)))$, when r is the number of input constraints.* $\Box$

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 because trees have induced-width of 1, as demonstrated in Figure 2.6. 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.

2.3 Bucket Elimination for Belief Assessment

Notation 2.3.1 (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 $U = u$, $(min_X h)(u) =$*

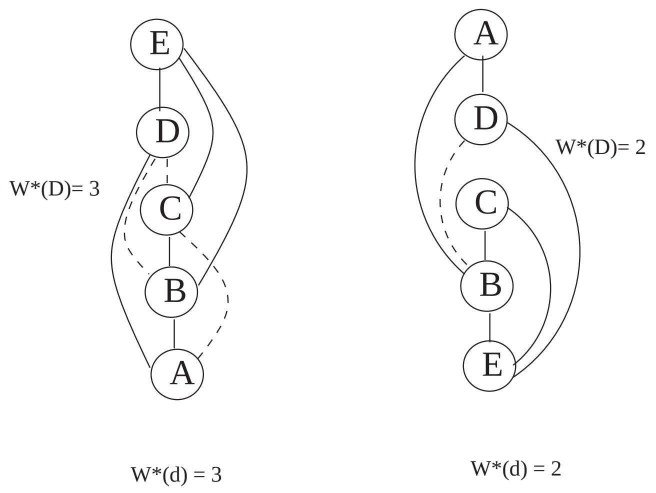


Figure 2.5: The induced-width along the orderings: $d_1 = A, B, C, D, E$ and $d_2 = E, B, C, D, A$

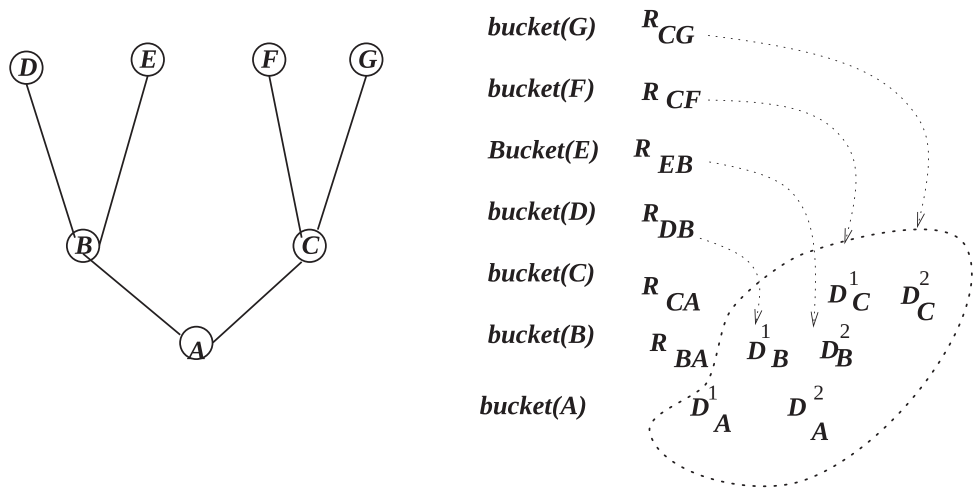


Figure 2.6: Schematic execution of adaptive-consistency on a tree network. D_X denotes unary constraints over X

.

*$\min_x h(u,x)$, $(max_X h)(u) = \max_x h(u,x)$,
$(\sum_X h)(u) = \sum_x h(u,x)$. 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 $(\sum_j h_j)(u) = \sum_j h_j(u_{S_j})$.*

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.

2.3.1 Deriving elim-bel

Belief updating was developed first for belief networks having no loops. The algorithm developed by Pearl is know as a belief Following propagation algorithm for singly-connected networks [30]. Subsequently researchers have investigated various general approaches to belief updating. The most common inference approaches are join-tree clustering [27] and variable-eliminations schemes [29, 10]. We next 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 and t o also compute $P(e)$. We wish to compute $P(X = x|e) = \alpha \cdot P(X = x, e)$, where α is a normalization constant $P(e)$. We will develop the algorithm using example 1.1.7 (Figure **??**). 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

$$= P(a)\sum_c P(c|a)\sum_b P(b|a)\sum_f P(f|b,c)\sum_d P(d|b,a)\sum_{g=1} P(g|f) \tag{2.1}$$

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

$$= P(a)\sum_c P(c|a)\sum_b P(b|a)\sum_f P(f|b,c)\lambda_G(f)\sum_d P(d|b,a) \tag{2.2}$$

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

$$= P(a)\sum_c P(c|a)\sum_b P(b|a)\lambda_D(a,b)\sum_f P(f|b,c)\lambda_G(f) \tag{2.3}$$

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

$$= P(a)\sum_c P(c|a)\sum_b P(b|a)\lambda_D(a,b)\lambda_F(b,c) \tag{2.4}$$

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

$$= P(a)\sum_c P(c|a)\lambda_B(a,c) \tag{2.5}$$

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

$$P(a)\lambda_C(a) \tag{2.6}$$

The answer to the query $P(a|g=1)$ can be computed by normalizing the last product. The probability of the evidence $P(g=1)$ is the normalizing constant itself.

The bucket-elimination algorithm mimics the above algebraic manipulation by the 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 2.18. Note that the observed variables are also placed in their corresponding bucket.

This initialization step corresponds to deriving the expression in Eq. (2.1). Now we process the buckets from last to first (or top to bottom in the figures), implementing the right to left computation of Eq. (2.1). 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. (2.2) from Eq. (2.1)). New functions are placed in lower buckets using the same placement rule.

$Bucket_D$ 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. (2.3) from Eq. (2.2)). The

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

Figure 2.7: Initial partitioning into buckets using $d_1 = A, C, B, F, D, G$

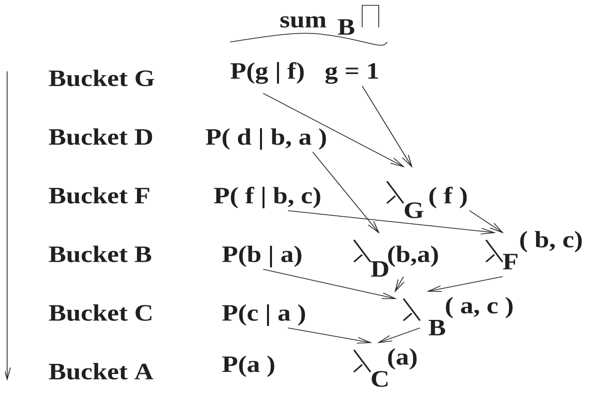


Figure 2.8: Bucket elimination along ordering $d_1 = A, C, B, F, D, G$.

next variable is F. $Bucket_F$ contains two functions $P(f|b,c)$ and $\lambda_G(f)$, and follows Eq. (2.4) 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. (2.4) from Eq. (2.3)). 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. (2.5) from Eq. (2.4)). 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. (2.6) from Eq. (2.5)). Finally, the belief in a can be computed in $bucket_A$, $P(a|g=1) = \alpha \cdot P(a) \cdot \lambda_C(a)$. Figure 2.8 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:

$P(a, g=1) = P(a) \sum_f \sum_d \sum_c P(c|a) \sum_b P(b|a)\ P(d|a,b) P(f|b,c) \sum_{g=1} P(g|f)=$
$P(a) \sum_f \lambda_G(f) \sum_d \sum_c P(c|a) \sum_b P(b|a)\ P(d|a,b) P(f|b,c)=$
$P(a) \sum_f \lambda_G(f) \sum_d \sum_c P(c|a) \lambda_B(a,d,c,f) =$

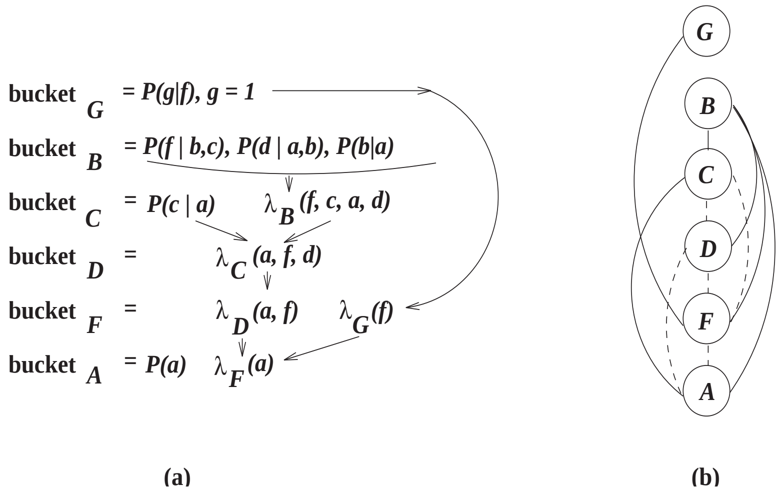

bucket G = P(g|f), g = 1
bucket B = P(f | b,c), P(d | a,b), P(b|a)
bucket C = P(c | a) λB(f, c, a, d)
bucket D = λC(a, f, d)
bucket F = λD(a, f) λG(f)
bucket A = P(a) λF(a)
G
B
C
D
F
A
(a)
(b)

Figure 2.9: The bucket's output when processing along $d_2 = A, F, D, C, B, G$

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

The bucket elimination process for ordering d_2 is summarized in Figure 2.9a. 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*, which yields as a byproduct also the probability of the evidence. 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

$$\lambda_n(x_{U_n}) = \sum_{x_n} P(x_n, e|x_{pa_n}) \Pi_{X_i \in ch_n} P(x_i, e|x_{pa_i}) \tag{2.7}$$

```
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 $\lambda_i$ 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 $\lambda_p$ to the largest-index variable in $U_p$.
3. Return: $Bel(x_1) = \alpha \Pi_i \lambda_i(x_1)$(where the $\lambda_i$ are in $bucket_1$), $P(e) = \alpha$ is the normalizing constant.
```

Figure 2.10: 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. (2.7). 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. If we also process the first bucket we get the probability of the evidence. Algorithm elim-bel is described in Figure 2.10. We conclude:

Theorem 2.3.2 *Algorithm elim-bel computes the posterior belief $P(x_1|e)$ for any given ordering of the variables which is initiated by X_1 and the probability of the evidence as the normalizing constant in the first bucket.* $\Box$

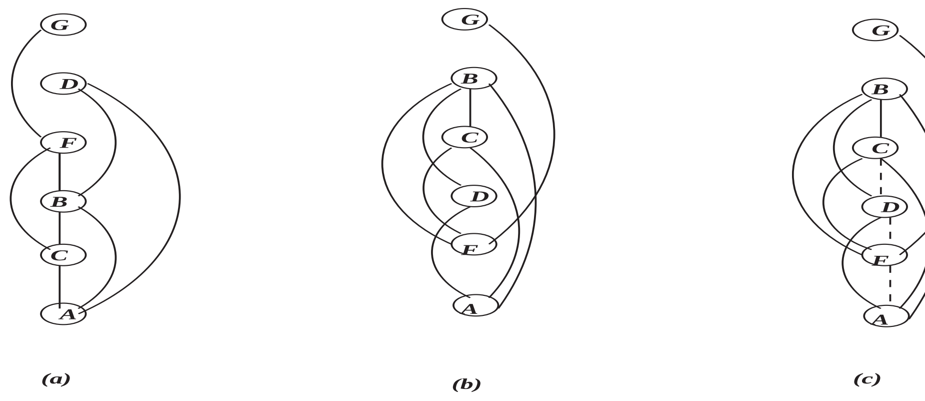


Figure 2.11: Two orderings of the moral graph of our example problem

The bucket's operation

[to be completed]

2.3.2 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 2.9a). 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, 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 1.2(a) is given in Figure 1.2(b). Let us take this moral graph and impose an ordering on its nodes. Figures 2.11a and 2.11b 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 2.9b).

Example 2.3.3 The induced moral graph of Figure **??**b, relative to ordering $d_1 = A, C, B, F, D, G$ is depicted in Figure 2.11a. 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 2.11c. The width of C is initially 2 (see Figure 2.11b) 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 2.3.4 (induced width) *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].*

Theorem 2.3.5 (Complexity of elim-bel) *Let w^* be the induced width of G along ordering d and k the maximum domain size of a variable. The time complexity of $elim-bel$ is $O(r \cdot k^{w^*+1})$ and its space complexity is $O(n \cdot k^{w^*})$.*

Proof. During elim-bel, each bucket sends a λ message to its parent and since it computes a function defined on all the variables in the bucket, the number of which is bounded by w^*, the size of the computed function is exponential in w^*. Since the number of functions that need to be consulted for each tuple in the generated function is bounded by the number of original functions in the bucket, r_{X_i} plus the messages received from its children, which is bounden by deg_i, the overall computation, summing over all buckets, is bounded by

$$\sum_{X_i} (r_{X_i} + deg_i - 1) \cdot k^{w^*+1}$$

The total complexity can be bound by $O((r+n) \cdot k^{w^*+1})$. Assuming $r > n$, this becomes $O(r \cdot k^{w^*+1})$. The size of each λ message is $O(k^{w^*})$. Since the total number of λ messages is $n-1$, the total space complexity is $O(n \cdot k^{w^*})$. $\Box$

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.

2.3.3 On finding small induced-width

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 as we briefly show in the next few paragraphs [14, 3, 43].

A rather important observation is that a graph is a tree (has no cycles) if and only if it has a width-1 ordering. The reason a width-1 graph cannot have a cycle is that for any ordering, at least one node on the cycle would have two parents, thus contradicting the width-1 assumption. And vice-versa: if a graph has no cycles, it can always be converted into a rooted directed tree by directing all edges away from a designated root node. In such a directed tree, every node has exactly one node pointing to it, – its parent. Therefore, any ordering in which every parent node precedes its child nodes in the rooted tree has a width of 1. Furthermore, given an ordering having width of 1, its induced-ordered graph has no additional arcs, yielding an induced width of 1, as well. In summary,

Proposition 2.3.6 *A graph is a tree iff it has both width and induced width of 1.* $\Box$

Finding a minimum-width ordering of a graph can be accomplished by the greedy algorithm *min-width* (see Figure 2.13). The algorithm orders variables from last to first as follows: in the first step, a variable with minimum number of neighbors is selected and put last in the ordering. The variable and all its adjacent edges are then eliminated from the original graph, and selection of the next variable continues recursively with the remaining graph. Ordering d_2 of G in Figure 2.12(c) could have been generated by a min-width ordering.

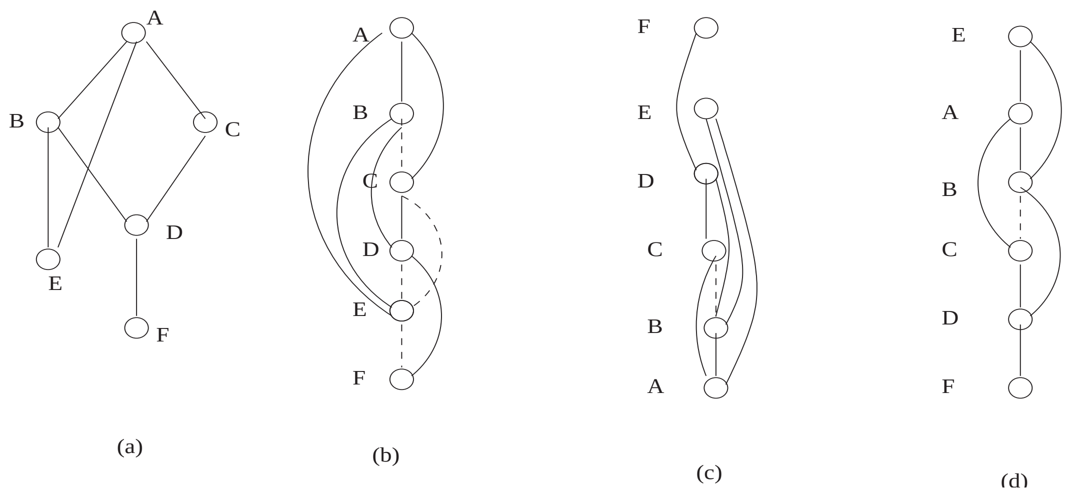


Figure 2.12: (a) Graph G, and three orderings of the graph; (b) $d_1 = (F, E, D, C, B, A)$, (c) $d_2 = (A, B, C, D, E, F)$, and (d) $d_3 = (F, D, C, B, A, E)$. Broken lines indicate edges added in the induced graph of each ordering.

Proposition 2.3.7 *[20] Algorithm min-width* (MW) *finds a minimum width ordering of a graph.*

Though finding the min-width ordering of a graph is easy, finding the minimum *induced width* of a graph is hard (NP-complete [2]). Nevertheless, deciding whether there exists an ordering whose induced width is less than a constant k, takes $O(n^k)$ time.

A decent greedy algorithm, obtained by a small modification to the min-width algorithm, is the *min-induced-width* (MIW) algorithm (Figure 2.14). It orders the variables from last to first according to the following procedure: the algorithm selects a variable with minimum degree and places it last in the ordering. The algorithm next connects the

```
MIN-WIDTH (MW)
input: a graph $G = (V, E)$, $V = \{v_1, ..., v_n\}$
output: A min-width ordering of the nodes $d = (v_1, ..., v_n)$.
1. for $j = n$ to 1 by -1 do
2.       $r \leftarrow$ a node in $G$ with smallest degree.
3.       put $r$ in position $j$ and $G \leftarrow G - r$.
         (Delete from $V$ node $r$ and from $E$ all its adjacent edges)
4. endfor
```

Figure 2.13: The min-width (MW) ordering procedure

node's neighbors in the graph to each other, and only then removes the selected node and its adjacent edges from the graph, continuing recursively with the resulting graph. The ordered graph in Figure 2.12(c) could have been generated by a min-induced-width ordering of G. In this case, it so happens that the algorithm achieves the minimum induced width of the graph, w^*.

Another variation yields a greedy algorithm known as *min-fill*. Rather than order the nodes in order of their min-degree, it uses the *min-fill set*, that is, the number of edges needed to be filled so that its parent set be fully connected, as an ordering criterion. This *min-fill* heuristic described in Figure 2.15, was demonstrated empirically to be somewhat superior to min-induced-width algorithm. The ordered graph in Figure 2.12(c) could have been generated by a min-fill ordering of G while the ordering d_1 or d_3 in parts (a) and (d) could not.

```
MIN-INDUCED-WIDTH (MIW)
input: a graph $G = (V, E)$, $V = \{v_1, ..., v_n\}$
output: An ordering of the nodes $d = (v_1, ..., v_n)$.
1. for $j = n$ to 1 by -1 do
2.        $r \leftarrow$ a node in $V$ with smallest degree.
3.        put $r$ in position $j$.
4.        connect $r$'s neighbors: $E \leftarrow E \cup \{(v_i, v_j)|(v_i, r) \in E, (v_j, r) \in E\}$,
5.        remove $r$ from the resulting graph: $V \leftarrow V - \{r\}$.
```

Figure 2.14: The min-induced-width (MIW) procedure

```
MIN-FILL (MIN-FILL)
input: a graph $G = (V, E)$, $V = \{v_1, ..., v_n\}$
output: An ordering of the nodes $d = (v_1, ..., v_n)$.
1. for $j = n$ to 1 by -1 do
2.        $r \leftarrow$ a node in $V$ with smallest fill edges for his parents.
3.        put $r$ in position $j$.
4.        connect $r$'s neighbors: $E \leftarrow E \cup \{(v_i, v_j)|(v_i, r) \in E, (v_j, r) \in E\}$,
5.        remove $r$ from the resulting graph: $V \leftarrow V - \{r\}$.
```

Figure 2.15: The min-fill (MIN-FILL) procedure

The notions of width and induced width and their relationships with various graph parameters, have been studied extensively in the past two decades.

Chordal graphs

Computing the induced width for chordal graphs is easy. A graph is *chordal* if every cycle of length at least four has a chord, that is, an edge connecting two nonadjacent vertices. For example, G in Figure 2.12(a) is not chordal since the cycle (A, B, D, C, A) does not have a chord. The graph can be made chordal if we add the edge (B, C) or the edge (A, D).

Many difficult graph problems become easy on chordal graphs. For example, finding all the maximal (largest) *cliques* (completely connected subgraphs) in a graph – an NP-complete task on general graphs – is easy for chordal graphs. This task (finding maximal cliques in chordal graphs) is facilitated by using yet another ordering procedure called the *max-cardinality ordering* [44]. A *max-cardinality ordering* of a graph orders the vertices from *first to last* according to the following rule: the first node is chosen arbitrarily. From this point on, a node that is connected to a maximal number of already ordered vertices is selected, and so on. (See Figure 2.16.)

A max-cardinality ordering can be used to identify chordal graphs. Namely, a graph is chordal iff in a max-cardinality ordering each vertex and all its parents form a clique. One can thereby enumerate all maximal cliques associated with each vertex (by listing the sets of each vertex and its parents, and then identifying the maximal size of a clique). Notice that there are at most n cliques: each vertex and its parents is one such clique. In addition, when using a max-cardinality ordering of a chordal graph, the ordered graph is identical to its induced graph, and therefore its width is identical to its induced width. It is easy to see that,

Proposition 2.3.8 *If G^* is the induced graph of a graph G, along some ordering, then G^* is chordal.* □

Example 2.3.9 We see again that G in Figure 2.12(a) is not chordal since the parents of A are not connected in the max-cardinality ordering in Figure 2.12(d). If we connect B and C, the resulting induced graph is chordal. □

k-trees. A subclass of chordal graphs are *k-trees.* A *k-tree* is a chordal graph whose maximal cliques are of size $k + 1$, and it can be defined recursively as follows: (1) A complete graph with k vertices is a *k-tree.* (2) A *k-tree* with r vertices can be extended to $r + 1$ vertices by connecting the new vertex to all the vertices in any clique of size k.

```
MAX-CARDINALITY (MC)
input: a graph $G = (V, E)$, $V = \{v_1, ..., v_n\}$
output: An ordering of the nodes $d = (v_1, ..., v_n)$.
1. Place an arbitrary node in position 0.
2. for $j = 1$ to $n$ do
3.         $r \leftarrow$ a node in $G$ that is connected to a largest subset of nodes
           in positions 1 to $j-1$, breaking ties arbitrarily.
4. endfor
```

Figure 2.16: The max-cardinality (MC) ordering ordering procedure

2.3.4 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 2.8).

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 conditioning effect [30]. 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 2.9a). 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

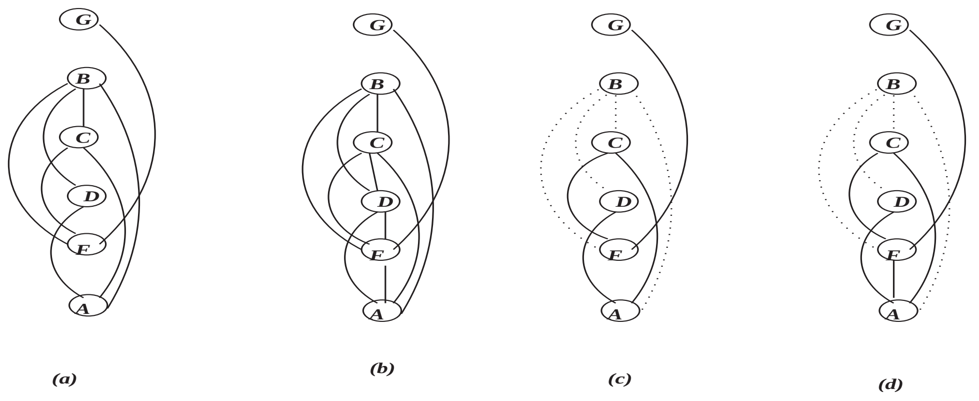


Figure 2.17: Adjusted induced graph relative to observing B

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, whose observed nodes. For example, in Figure 2.17(a,b) we show the ordered moral graph and the induced ordered moral graph of Figure 1.2. In 2.17(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 2.3.10 *Given a belief network having n variables, algorithm elim-bel when 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.* □

2.3.5 Relevant subnetworks

The belief-updating task has special semantics which allows restricting the computation to relevant portions of the belief network.

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 2.3.11 *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. $\square$

Example 2.3.12 Consider again the belief network whose acyclic graph is given in Figure 1.2(a) 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. $\square$

Alternatively, the relevant portion of the network can be precomputed by using a recursive marking procedure applied to the ordered moral graph. (see also [46]). 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 2.3.13 *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 2.3.14 *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 2.3.15 *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.* $\Box$

2.4 Bucket-Tree Elimination

The bucket-elimination algorithm, elim-bel, for belief updating can be viewed as one phase message propagation from leaves to root along a bucket-tree. The algorithm computes the belief of the first node in the ordering, given all the evidence (or, we can just compute the probablity of evidence). Often, it is desirable to get the belief of every variable in the network. A brute-force approach will require running elim-bel n times, each time with a different variable order. We will show next that this is unnecessary. By viewing bucket-elimination as message passing (of a function computed by variable elimination) from leaves to root along a rooted *bucket-tree*, we can augment it with a second message passing from root to leaves which is equivalent to running the algorithm for each variable separately. This yields a two-phase variable elimination algorithm up and down the bucket-tree, which can also be viewed as two-phase message passing (or propagation) along the tree.

Let G be a moral graph of a Bayesian network P, d an ordering of its variables $X_1, ..., X_n$. We denote by $B_1, ..., B_n$ the buckets of variables $X_1, ...X_n$ relative to ordering d. Each such bucket includes the functions in the initial partitioning, called the *bucket's functions* and the set of all variables that it mentions when its bucket is processed, called the *bucket's cluster*. A *bucket-tree* can then be formed by directing an edge from bucket B_i to B_j if the function computed at B_i is placed in B_j. It is easy to see that this definition is equivalent to the following graphical definition.

Definition 2.4.1 (buckets) *Let $P =< X, D, P, \Pi >$ be a Bayesian network and d an ordering of its variables $d = (X_1, ..., X_n)$. Let $B_{X_1}, ..., B_{X_n}$ be a set of buckets, one for each variable. The bucket B_{X_i} contains the CPTs whose latest variable with respect to d is X_i. The variable set (or cluster) of B_{X_i} are variable X and its induced-parents.*

Definition 2.4.2 (bucket tree) *Let G_d be the induced moral graph along d of a Bayesian network. Each bucket B_X points to B_Y (or, B_Y is the parent of B_X) if variable Y is the*

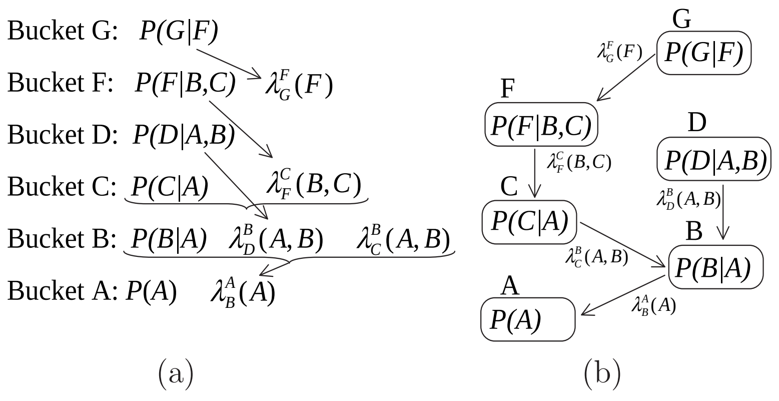


Figure 2.18: Execution of BE along the bucket-tree

latest earlier neighbor of X in G_d (namely it is the closest earlier neighbor to X in G_d). If B_Y is the parent of B_X in the bucket-tree, then the separator of B_X and B_Y, denoted $sepX, Y$ is $B_X \cap B_Y$. Note that by B_X we denote both the bucket name and its set of cluster variables.

It is easy to show that

Theorem 2.4.3 *The bucket-tree of a Bayesian network G is an i-map of G.*

Example 2.4.4 Consider the Bayesian network defined over the DAG in Figure 1.2(a). Figure 2.18a shows the initial buckets along the ordering $d = A, B, C, D, F, G$, and the messages (labeled λ in this case) that will be passed by $elim - bel$ from top to bottom. Figure 2.18b displays the same computation as a message-passing along its bucket-tree. □

Given a bucket tree, whose buckets are denoted by the integers $\{1, ...n\}$ and given a directed edge (i, j), $elim(i, j)$ is the set of variables in B_i and not in B_j, namely $elim(i, j) = B_i - sep(i, j)$.

Assume now that we computed the belief for variable A as in the previous example and that we now wants to compute the belief in D. Instead of doing all the computation from scratch using a different variable ordering we can take the bucket tree and reorient the edges towards D, making it the root. Then we can pass messages from the leaves to this new root when the computation in each bucket is basically the same variable elimination computation, when the eliminated variables are the eliminator sets.

The BTE algorithm *bucket-tree-elimination (BTE)* in figure 2.19 includes the two phases of message passing computations along the bucket-tree. The top down phase is identical to the elim-bel computation. The bottom-up message from the bucket of X to a child bucket Y takes the product of all the bucket functions, the π message from its parent and all the λ messages from it other child buckets and summing out over the eliminator from X to Y.

Theorem 2.4.5 *Algorithm BTE is sound. When terminates, each B_X has λ_j^X received from each child Z_j in the tree, its own original P functions and the π_Y^X sent from its parent Y. Then,*

$$P(B_X, e) = \alpha \prod_k P_k \cdot \prod_j \lambda_j \cdot \pi_Y^X$$

Proof: follows from the bucket-tree i-mapness. A full proof will be given later. □

Example 2.4.6 Figure 2.20 shows the complete execution of BTE along the linear order of buckets and along the bucket-tree. The π and λ messages are viewed as messages placed on the outgoing arcs.

The π functions computed in the up phase are:
$\pi_A^B(a) = P(a)$
$\pi_B^C(c,a) = P(b|a)\lambda_D(a,b)\pi_A^C(a)$
$\pi_B^D(a,b) = P(b|a)\lambda_C(a,b)\pi_C^B(a,b)$
$\pi_C^F(c,b) = \sum_a P(c|a)\pi_B^C(a,b)$
$\pi_F^G(f) = \sum_{b,c} P(f|b,c)\pi_C^F(c,b)$
□

Theorem 2.4.7 (Complexity of BTE) *Let w^* be the induced width of G along ordering d and k the maximum size of a domain of a variable. The time complexity of BTE is $O(r \cdot deg \cdot k^{w^*+1})$, where deg is the maximum degree in the bucket-tree. The space complexity of BTE is $O(n \cdot k^{w^*})$.*

Proof: complete

In theory the speedup expected from running BTE vs running n-BE (BE n times) is at most n. This may seem insignificant compared with the exponential complexity in w^*, however in practice it can be very significant. The actual speedup of BTE relative to n-BE may be smaller than n, however. We know that the complexity of n-BE is $O(n \cdot r \cdot k^{w^*+1})$, whereas the complexity of BTE is $O(deg \cdot r \cdot k^{w^*+1})$.

2.4.1 Bucket-tree propagation, an asynchronous version

The BTE algorithm can be described in an asynchronous manner when viewing the bucket-tree as an undirected tree and passing only one type of messages. Each bucket receives λ

Algorithm bucket-tree elimination (BTE)
Input: A problem $P =< X, D, F, \Pi >$, ordering d.
Output: Augmented buckets containing the original functions and all the π and λ functions received from neighbors in the bucket-tree.
0. Pre-processing:
Place each function in the latest bucket, along d, that mentions a variable in its scope. Connect two buckets B_x and B_y if variable Y is the latest earlier neighbor of X in the induced graph G_d.
1. Top-down phase: λ messages (BE)
For $i = n$ to 1, process bucket B_{X_i}:
Let $\lambda_1, ...\lambda_j$ be all the functions in B_{X_i} at the time B_{X_i} is processed, including the original functions of P. The message $\lambda_{X_i}^Y$ sent from X_i to its parent Y, is computed by

$$\lambda_{X_i}^Y(sep(X_i, Y)) = \sum_{elim(X_i,Y)} \Pi_{i=1}^j \lambda_i$$

where $sep(X_i, Y)$ is the separator of X_i and Y
2. bottom-up phase: π messages
For $i = 1$ to n, process bucket B_{X_i}:
Let $\lambda_1, ..., \lambda_j$ be all the functions in B_{X_i} at the time B_{X_i} is processed, including the original functions of P. B_{X_i} takes the π message received from its parent Y, $\pi_Y^{X_i}$, and computes a message $\pi_{X_i}^{Z_j}$ for each child bucket Z_j by

$$\pi_{X_i}^{Z_j}(sep(X_i, Z_j)) = \sum_{elim(X_i,Z_j)} \pi_Y^{X_i} \cdot (\Pi_{k \neq j} \lambda_{Z_k}^{X_i})$$

3. Deriving beliefs
The joint probabilities $P(B_X, E = e)$ in bucket B_X is computed by taking the product of all the functions in B_X (the original Ps, the λ functions and π function): Namely, given the functions $f_1, ..., f_t$ in B_X at termination,

$$P(B_X) = \alpha \prod_i f_i$$

and the belief of X is computed by

$$Bel(x) = \alpha \sum_{B_X - \{X\}} \prod_j f_j$$

Figure 2.19: Algorithm Bucket-Tree Elimination

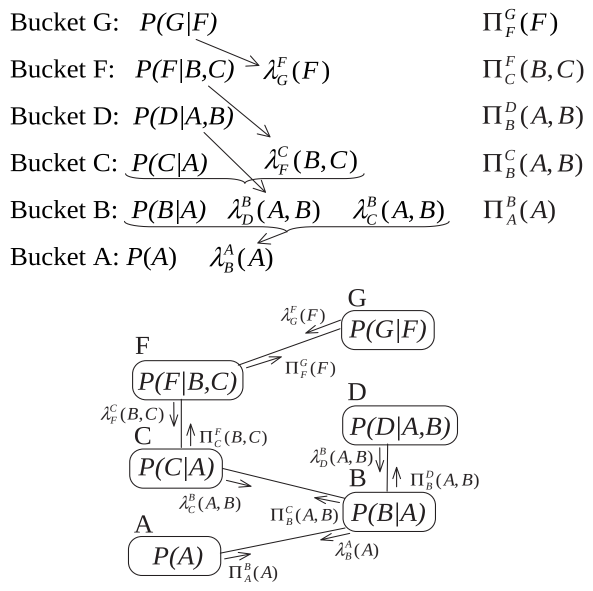

Bucket G: $P(G|F)$ $\Pi_F^G(F)$
Bucket F: $P(F|B,C)$ $\lambda_G^F(F)$ $\Pi_C^F(B,C)$
Bucket D: $P(D|A,B)$ $\Pi_B^D(A,B)$
Bucket C: $P(C|A)$ $\lambda_F^C(B,C)$ $\Pi_B^C(A,B)$
Bucket B: $P(B|A)$ $\lambda_D^B(A,B)$ $\lambda_C^B(A,B)$ $\Pi_A^B(A)$
Bucket A: $P(A)$ $\lambda_B^A(A)$
G
F
D
C
B
A
$P(G|F)$
$P(F|B,C)$
$P(D|A,B)$
$P(C|A)$
$P(B|A)$
$P(A)$
$\lambda_G^F(F)$
$\Pi_F^G(F)$
$\lambda_F^C(B,C)$
$\Pi_C^F(B,C)$
$\lambda_D^B(A,B)$
$\Pi_B^D(A,B)$
$\lambda_C^B(A,B)$
$\Pi_B^C(A,B)$
$\lambda_B^A(A)$
$\Pi_A^B(A)$

Figure 2.20: Propagation of π's and λ's along the bucket-tree

messages from each of its neighbors and each sends a λ message to every neighbors. The algorithm executed by node B_X is described next. We distinguish between the original P functions placed in bucket B_i and the messages that its received from its neighbors. The algorithm is described in Figure 2.21.

Let $\{P_i\}, i = 1, ...j$ be the original functions in B_X, let $Y_1, ...Y_k$. It is easy to see that algorithm BTP is guaranteed to converge, and when converged each B_X and its incoming messages will have the same content at the buckets B_X in BTE.

Bucket-Tree Propagation (BTP)
Input: For each node X, its bucket B_X and its neighboring buckets. Let $\lambda_{Y_j}^X$ be the message sent to X from its neighbor Y_i.
The message X sends to a neighbor Y_j is:

$$\lambda_X^{Y_j}(S_{XY_j}) = \sum_{B_X - S_{XY_j}} (\prod_i P_i) \cdot (\prod_{i \neq j} \lambda_i)$$

Figure 2.21: The Bucket-tree propagation (BTP) for X

2.4.2 From Buckets to Super-Buckets to Join-Tree Algorithm

The BTE and BTP algorithms are special cases of a wider class of algorithms all based on an underlying cluster-tree decomposition. We saw that in a chordal graph the maximal cliques form a tree which obeys the *running intersection property (r.i.p)*, also called *connectedness.*. This property guarantees that the clique-tree is an i-map of the original Bayes network.

In fact, given a chordal graph embedding of the original moralized Bayesian networks' dag, (which can be obtained by generating the induced graph along an ordering), there are many cluster-trees that have the desired running intersection property. The join-tree and the bucket-tree are just the two most popular candidates. another candidate. We next define *cluster-tree decompositions* as a generalization that captures all viable cluster-trees that support message propagation along a tree of clusters as described in BTE and BTP.

Definition 2.4.8 (tree-decomposition) *A* tree decomposition *of a Bayesian network is a set of subsets of variables, called clusters, $C_1, ..., C_l$ connected by a tree structure, T that satisfies the following two properties:*
1. Every function (CPT) has at least one cluster that contains its scope.
2. The tree-decomposition obeys the connectedness property, (running intersection property) namely for every variable Y, the set of clusters that contain y are connected in T.

A bucket-tree is a tree-decomposition because, by construction, each CPT fits into a bucket using the initial partitioning. Proving connectedness is a little more involved and we leave it as an exercise.

Theorem 2.4.9 *A bucket tree of a probabilistic network P is a tree-decomposition of P.*

Also,

Proposition 2.4.10 *If T is a tree-decomposition, then any tree obtained by merging adjacent clusters is also a tree-decomposition.*

Proof: Exercise.

So, to obtain a cluter-tree decomposition we can start from the bucket-tree and merge adjacent buckets, yielding *super-buckets.* The maximal cliques in the induced-order graph are a special kind of super-buckets which form a tree-decomposition which is called *join-tree* (see 2.22).

Both BTE and BTP can be immediately extended to any cluster-tree decomposition by applying the function computation defined originally for buckets to arbitrary clusters. When the tree-decomposition happens to be a join-tree, the BTE is often called join-tree

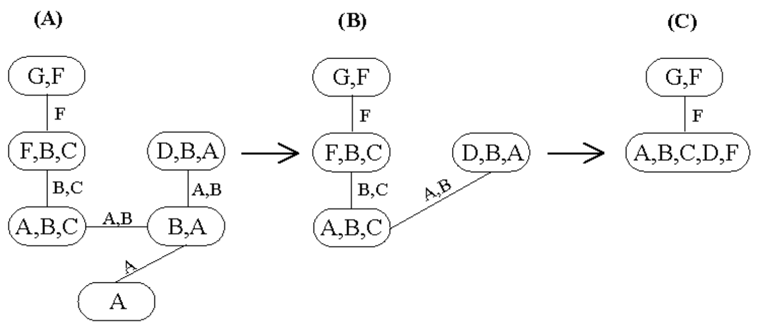


Figure 2.22: From a bucket-tree to join-tree to a super-bucket-tree

Algorithm Cluster-Tree Elimination (CTE)
Input: we assume a tree-decomposition T whose nodes are clusters of variables $C_1, ...C_t$ and each node has a collection of neighbors.
Output: Each cluster will have all input messages received from all its neighbors.
Initialize: put each original function in *any* cluster that contains its scope.
The algorithm for cluster C is: Let $\{P_i\}, i = 1, ...j$ be the original functions in C, let $C_1, ...C_k$ be its neighbors, let $\lambda_C^{C_i}$ be the message sent to C from its neighbor C_i. When C receives all messages from its neighbors except C_i, the message C sends to C_i is:

$$\lambda_C^{C_j}(Sep_{C,C_j}) = \sum_{C-Sep_{C,C_j}} (\prod_i P_i) \cdot (\prod_{i \neq j} \lambda_i) \tag{2.8}$$

Figure 2.23: Algorithm Cluster tree elimination (CTE)

clustering or junction-tree clustering (JTC) [15, 27]. The term we will use when moving from bucket-trees to arbitrary tree-decompositions is *cluster-tree elimination* or *CTE*. Algorithm cluster-tree elimination is presented in Figure 2.23.

We could clearly define the algorithm for synchronous execution from leaves to the root and back.

Theorem 2.4.11 *Algorithm cluster-tree clustering, CTE, is sound.*

Once the algorithm converged, each product of the cluster's functions provide the marginal probabilities over the cluster's scope, joint with the evidence.

Theorem 2.4.12 *The time complexity of CTE is exponential in the largest cluster size. The space complexity of CTE is exponential in the separator sizes*

proof: Clearly the time complexity is exponential in the cluster size since we consider all the assignments to all variables in each cluster, and there are $k^{|C|}$ tuples, when $|C|$ is the size of the cluster and k is the domain size. The space complexity however can be restricted to the output of the recorded function (the message). For each tuple of the recorded function we accumulate the running sum of probabilities over all the rest of the variables, and each such probability (of a tuple) can be computed in linear time. For example, this computation can be done by traversing the search space of all possible assignments in a depth-first manner. $\square$

A more refined analysis of the CTE algorithm is described in the following theorem which uses the notion of a tree-width of a tree-decomposition.

Definition 2.4.13 (tree-width,seperator) *The treewidth of a tree-decomposition T is the maximum number of variables in one of its clusters minus 1. The separator of a tree-decomposition is the maximum size of any intersection set over its tree edges.*

Theorem 2.4.14 *[24] [Complexity CTE] Let N be the number of vertices in the tree decomposition, w its tree-width, sep its maximum separator size, r be the number of input functions in F, deg be the maximum degree in T, and k be the maximum domain size of a variable. The time complexity of CTE is $O((r+N) \cdot deg \cdot k^w)$ and its space complexity is $O(N \cdot k^{sep})$.*

The separator sizes in the bucket-tree are equal to the cluster sizes (minus 1) and therefore the time complexity and space complexity for BTE/BTP are the same. In general however, for any cluster tree or, in particular, for the join-tree, the separators sizes may be far smaller than the maximal cliques sizes. Furthermore, it is sometimes worthwhile to combine two adjacent clusters having a *wide separator* in order to save space, as we will discuss in the sequel.

2.4.3 Pearl's Belief Propagation over Polytrees

It is clear that whenever we have a tree network its moral graph is also a tree having induced-width of 1. Consequently all the algorithms we discussed can be accomplished in linear time and space.

When the belief network is a *polytree* we have another interesting special case where belief assessment, and any other query, as we are about to see, can be accomplished efficiently as show by Pearl [30]. Indeed if we apply BTE to a polytree-based tree-decomposition we obtain Pearl's well known belief propagation algorithm [30].

Definition 2.4.15 (polytree) *A polytree is a directed acyclic graph whose underlying undirected graph has no cycles (see Figure 2.24(a)).*

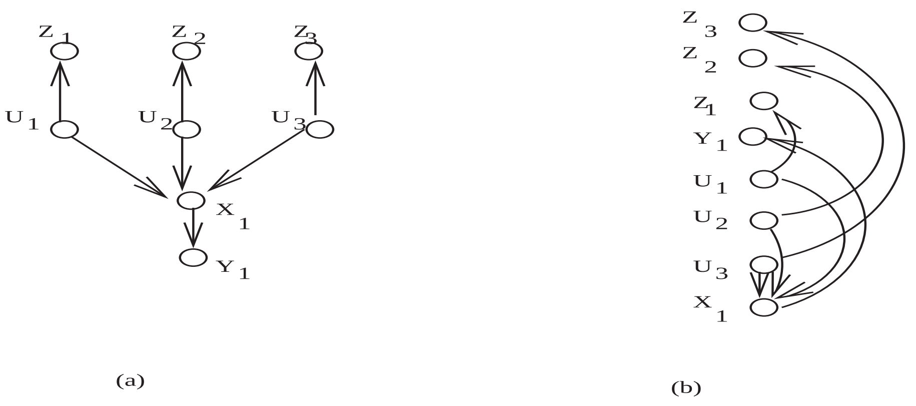


Figure 2.24: (a) A polytree and (b) a legal processing ordering

A polytree decomposition. Given a polytree Bayesian network, its set of families form a cluster tree-decomposition whose separators are singleton variables as follows. For each variable X and its parents $pa(X)$ we form a cluster C_X that includes the family of X and its CPT, $P(X|pa(X))$. Note, that the separators of this cluster tree are all singleton variables.

Proposition 2.4.16 *Given a polytree, a polytree-based tree-decomposition is a cluster tree-decomposition.*

It can be shown that if we direct the edges of the polytree decomposition from a cluster of a parent node X to the cluster of its child, the π and λ messages that propagate along the polytree tree decomposition using BTE (or CTE) are identical to Pearl's belief propagation messages applied along the original polytree. Indeed,

Theorem 2.4.17 *Given a polytree and its tree-decomposition, algorithms CTE or CTP (cluster tree propagation) applied along the polytree's tree-decomposition is time and space linear in the network's size.* $\square$

Note that although the above theorem is also exponential in the size of the largest family, it does not contradict linearity in the input since the CPTs are assumed to be exponential in their family size.

2.4.4 The semantic of the messages

Their meaning in Pearl's belief propagation, and their general meaning in BE and BTE. [to complete]

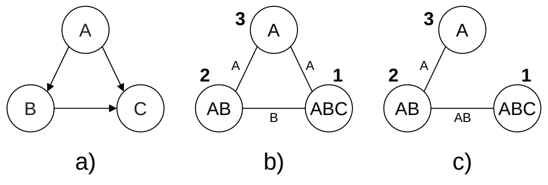


Figure 2.25: a) A belief network; b) A dual join-graph with singleton labels; c) A dual join-graph which is a join-tree

2.4.5 Iterative Belief Propagation over Dual Join-Graphs

Since Pearl's belief propagation algorithm over singly-connected networks is defined distributedly, it is still well defined if executed over a network with loops.

Iterative belief propagation (IBP) is an iterative application of Pearl's algorithm that was defined for poly-trees [30]. In this section we will present IBP as an instance of join-graph propagation over variants of the *dual graph*.

Consider a Bayesian network $\mathcal{B} =< X, D, G, P >$. The *dual graph* $\mathcal{D}_\mathcal{G}$ of the Belief network $\mathcal{B}$, is an arc-labeled graph defined over the CPTs as its functions. Namely, it has a node for each CPT and a labeled arc connecting any two nodes that share a variable in the CPT's scope. The nodes are labeled by the scopes of their CPTs. The arcs are labeled by the shared variables. A *dual join-graph* is a labeled arc subgraph of $\mathcal{D}_\mathcal{G}$ whose arc labels are subsets of the labels of $\mathcal{D}_\mathcal{G}$ such that the *running intersection property*, also called *connectedness property*, is satisfied. The running intersection property requires that any two nodes that share a variable in the dual join-graph be connected by a path of arcs whose labels contain the shared variable. Clearly the dual graph itself is a dual join-graph because any two nodes that share a variable are directly connected. An *arc-minimal* dual join-graph is a dual join-graph for which none of its labels can be further reduced while maintaining the connectedness property.

Interestingly, there are many dual join-graphs of the same dual graph and many of them are arc-minimal. We define Iterative Belief Propagation on a dual join-graph. Each node sends a message over an arc whose scope is identical to the label on that arc. Since Pearl's algorithm sends messages whose scopes are singleton variables only, we highlight arc-minimal singleton dual join-graph. One such graph can be constructed directly from the graph of the Bayesian network, labeling each arc with the parent variable. It can be shown that:

Proposition 2.4.18 *The dual graph of any Bayesian network has an arc-minimal dual join-graph where each arc is labeled by a single variable.*

```
Algorithm **IBP**
**Input:** An arc-labeled dual join-graph $DJ = (V, E, L)$ for a Bayesian network $BN =< X, D, G, P >$. Evidence $e$.
**Output:** An augmented graph whose nodes include the original CPTs and the messages received from neighbors. Approximations of $P(X_i|e)$, $\forall X_i \in X$. Approximations of $P(F_i|e)$, $\forall F_i \in \mathcal{B}$.
Denote by: $h_u^v$ the message from $u$ to $v$; $ne(u)$ the neighbors of $u$ in $V$; $ne_v(u) = ne(u) - \{v\}$; $l_{uv}$ the label of $(u, v) \in E$; $elim(u, v) = scope(u) - scope(v)$.
• **One iteration of IBP**
   **For** every node $u$ in $DJ$ in a topological order and back, do:
      1. **Process observed variables**
         Assign evidence variables to the each $p_i$ and remove them from the labeled arcs.
      2. **Compute and send to $v$ the function:**
         $h_u^v = \sum_{elim(u,v)} (p_u \cdot \prod_{\{h_i^u, i \in ne_v(u)\}} h_i^u)$
    **Endfor**
• **Compute approximations of $P(F_i|e)$, $P(X_i|e)$:**
   For every $X_i \in X$ let $u$ be the vertex of family $F_i$ in $DJ$,
   $P(F_i|e) = \alpha(\prod_{h_i^u, u \in ne(i)} h_i^u) \cdot p_u$;
   $P(X_i|e) = \alpha \sum_{scope(u)-\{X_i\}} P(F_i|e)$.
```

Figure 2.26: Algorithm Iterative Belief Propagation

Example 2.4.19 Consider the belief network on 3 variables A, B, C with CPTs 1.$P(C|A, B)$, 2.$P(B|A)$ and 3.$P(A)$, given in Figure 2.25a. Figure 2.25b shows a dual graph with singleton labels on the arcs. Figure 2.25c shows a dual graph which is a join tree, on which belief propagation can solve the problem exactly in one iteration (two passes up and down the tree). □

We will next present IBP algorithm that is applicable to any dual join-graph (Figure 2.26). The algorithm is a special case of IJGP introduced in [35]. It is easy to see that one iteration of IBP is time and space linear in the size of the belief network, and when IBP is applied to the singleton labeled dual graph it coincides with Pearl's belief propagation applied directly to the acyclic graph representation. Also, when the dual join-graph is a tree IBP converges after one iteration (two passes, up and down the tree) to the exact representation, from which beliefs can be computed.

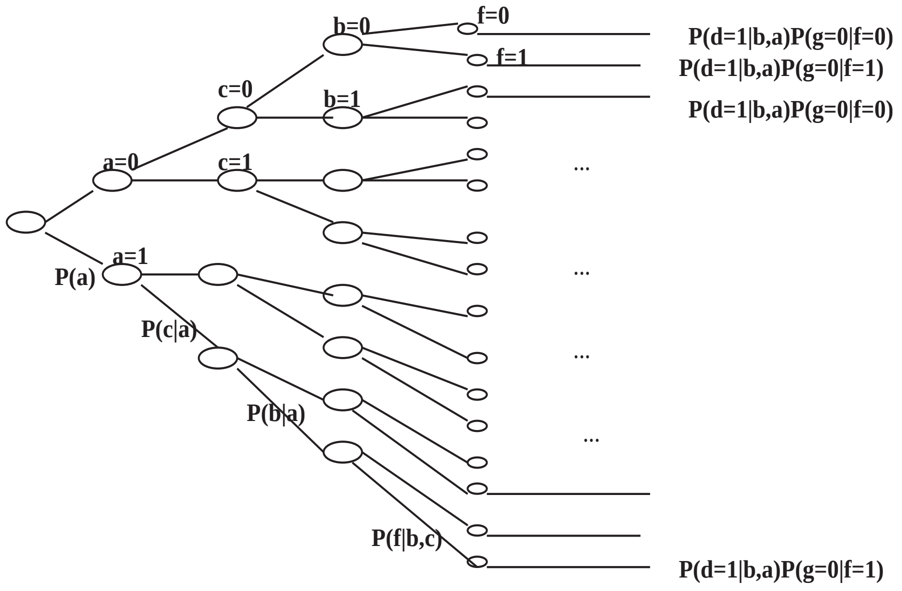


Figure 2.27: probability tree

2.5 Combining Elimination and Conditioning

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 for elimination). For example, we can compute the expression for belief updating or the probability of evidence in the network of Figure **??**:

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

$$= P(a)\sum_c P(c|a)\sum_b P(b|a)\sum_f P(f|b,c)\sum_d P(d|b,a)\sum_g P(g|f), \tag{2.9}$$

by traversing the tree in Figure 2.27, 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

```
Algorithm elim-cond-bel
Input: A belief network $BN = \{P_1, ..., P_n\}$; an ordering of the variables, $d$; a subset $C$ of conditioned variables; observations $e$.
Output: $Bel(A)$.
Initialize: $\lambda = 0$.

  1. For every assignment $C = c$, do
       • $\lambda_1 \leftarrow$ The output of elim-bel with $c \cup e$ as observations.
       • $\lambda \leftarrow \lambda + \lambda_1$. (update the sum).

  2. Return $\lambda$.
```

Figure 2.28: Algorithm *elim-cond-bel*

as best-first search and branch and bound, respectively. The sum can be accumulated for each value of variable A.

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,

$$\sum_x P(x,e) = \sum_c \sum_v P(c,v,e) = \sum_{c,v} \Pi_i P(x_i|x_{pa_i})|_{(c,v,e)}$$

Therefore, for every partial tuple c, we can compute $\sum_v P(v,c,e)$ 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 sum will be accumulated. This straightforward algorithm is presented in Figure 2.28.

Given a particular value assignment c, the time and space complexity of computing the 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 2.5.1 *Given a set of conditioning variables, C, the space complexity of algorithm elim-cond-bel 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.* $\square$

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-bel reduces to the known loop-cutset algorithm (see Pearl, chapter 4).

Definition 2.5.2 *Given an undirected graph, G a cycle-cutset is a subset of the nodes that breaks all its cycles. Namely, when removed, the graph has no cycles.*

In general Theorem 2.5.1 calls for a secondary optimization task on graphs:

Definition 2.5.3 (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 loop-cutset corresponds to the case when conditioning creates a poly-tree. The general task is clearly NP-complete.

Clearly, algorithm elim-cond-bel 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 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

2.6 Bucket elimination for optimization tasks

2.6.1 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 [30, 8, 31, 32]. Recent proposals include best first-search algorithms [42] and algorithms based on linear programming [37].

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.[2] Constant functions can be placed either in the preceding bucket or directly in the first bucket[3].

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

[2]This step is optional; the maximizing values can be recomputed from the information in each bucket.
[3]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, $bucket_1$, ..., $bucket_n$, 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 2.29: Algorithm *elim-mpe*

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 2.29. Observed variables are handled as in elim-bel. The notion of irrelevant buckets is not applicable here.

Example 2.6.1 Consider again the belief network in Figure **??**. 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 2.8 where λ is replaced by h. $\square$

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).

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 2.6.2 *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.*

2.6.2 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 2.30 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.

```
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, $bucket_1$, ..., $bucket_n$, 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 $\beta_i$ 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 $\beta_p$ 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 2.30: Algorithm *elim-map*

Theorem 2.6.3 *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.*

2.6.3 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,[4] 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

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

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

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

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

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

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

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

$$\max_d[\sum_{\bar{x}_{k+1}^{n-1}}\Pi_{X_i\in X-F_n}P(x_i,e|x_{pa_i},d)\cdot(\sum_{j\in l_n}f_j(x_{Q_j}))\sum_{x_n}\Pi_{X_i\in F_n}P(x_i,e|x_{pa_i},d) \tag{2.10}$$

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

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 $U_n \cup X_n$)

$$\lambda_n(x_{U_n}|d)=\sum_{x_n}\Pi_{X_i\in F_n}P(x_i,e|x_{pa_i},d) \tag{2.11}$$

We define θ_n over W_n as

$$\theta_n(x_{W_n}|d)=\sum_{x_n}\Pi_{X_i\in F_n}P(x_i,e|x_{pa_i},d)\sum_{j\in t_n}f_j(x_{Q_j})) \tag{2.12}$$

After substituting Eqs. (2.11) and (2.12) into Eq. (2.10), we get

$$E=\max_d\sum_{\bar{x}_{k+1}^{n-1}}\Pi_{X_i\in X-F_n}P(x_i,e|x_{pa_i},d)\cdot\lambda_n(x_{U_n}|d)[\sum_{j\in l_n}f_j(x_{Q_j})+\frac{\theta_n(x_{W_n}|d)}{\lambda_n(x_{U_n}|d)}] \tag{2.13}$$

The functions θ_n and λ_n compute the effect of eliminating X_n. The result (Eq. (2.13)) 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 2.31. Each bucket contains utility components, θ_i, and probability components, λ_i. Variables can be marked as 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.

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 2.31: Algorithm *elim-meu*

Example 2.6.4 Consider the network of Figure **??** 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 2.32. The bucket's partitioning and the schematic computation of this decision problem is given in Figure 2.33. 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 $\theta_C(a,D_1) = \frac{1}{\lambda_C(a,D_1)} \sum_c P(c|a)\lambda_B($
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 2.6.5 *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 [45] have published an algorithm for the general influence diagram that is a variation of elim-meu. Kjaerulff's algorithm [25] can be viewed as a variation of elim-meu tailored to dynamic probabilistic networks.

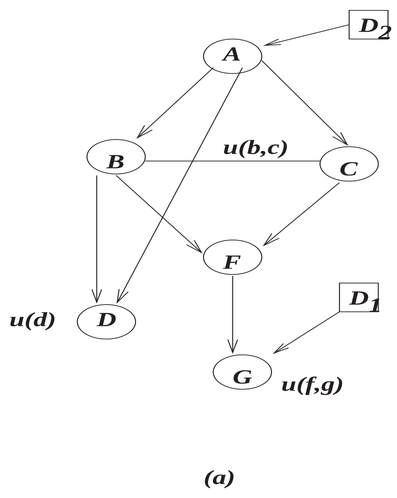


Figure 2.32: An influence diagram

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

Figure 2.33: 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 func-
tions in each bucket.
```

Figure 2.34: Dynamic programming as elim-opt

2.6.4 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.

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 [4]. The algorithm, called *elim-opt*, is given in Figure 2.34.

A schematic execution of our example along ordering $d = G, A, F, D, C, B$ is depicted in Figure 2.35. Clearly,

Theorem 2.6.6 *Given a cost network, elim-opt generates a representation from which*

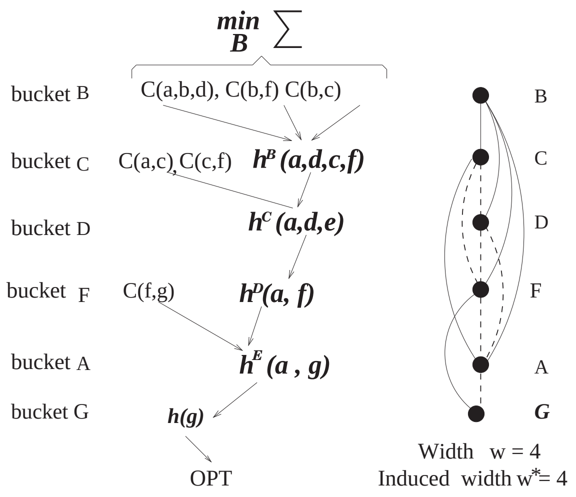


Figure 2.35: Schematic execution of elim-opt

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. □

2.7 General bucket elimination and 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 [7], Zhang and Poole's VE1 algorithm [46] which is identical to elim-bel, SPI algorithm by D'Ambrosio et.al., [36] 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 [39]. 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 [16].

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 [34] the connection between optimization and constraint satisfaction and its relationship to dynamic programming is explicated. In the work of [28, 41] and later in [5] 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 [41] 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 [16] and here, tree-clustering and bucket elimination schemes are closely related, tasks that fall within the axiomatic framework [41] can be accomplished by bucket elimination algorithms as well. In [5] 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 [18, 11] 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 $U = \cup_j S_j$. $\otimes_{j=1}^k f_j \in \{\Pi_{j=1}^k f_j,\ \sum_{j=1}^k f_j,\ \bowtie_j f_j\}$.

The problem is to compute

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

(In [41] 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 2.36. 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 $\otimes_j = \bowtie_j$. Similarly, Fourier elimination, directional resolution as well as elim-meu can be shown to be expressible in terms of such operators.

```
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 $\lambda_i$ 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 $\lambda_p$ to the largest-index variable in $U_p$.
3. Return: all the functions in each bucket.
```

Figure 2.36: Algorithm *bucket-elimination*

2.8 Back to deterministic graphical models

2.8.1 Bucket elimination for Propositional CNFs

We will next describe the bucket elimination for solving satisfiability of a propositional formula expressed in CNF [21] .

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 [21].

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 [19]. This yields a bucket-elimination algorithm for propositional satisfiability which we call *directional resolution.* Consider the following algorithm for deciding the satisfiability of a propositional theory in conjunctive normal form (CNF). Given a set of clauses and an ordering of the propositional variables, assign to each clause the index of the highest ordered literal in the clause. Then resolve in order, from last to first, only clauses having the same index, and only on their highest literal. This restriction results in a systematic elimination of literals from the set of clauses that are candidates for future resolution. Algorithm *directional resolution*, (DR), is the core of the well-known Davis-Putnam algorithm for satisfiability [9, 17].

Algorithm DR (see Figure 2.38) 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

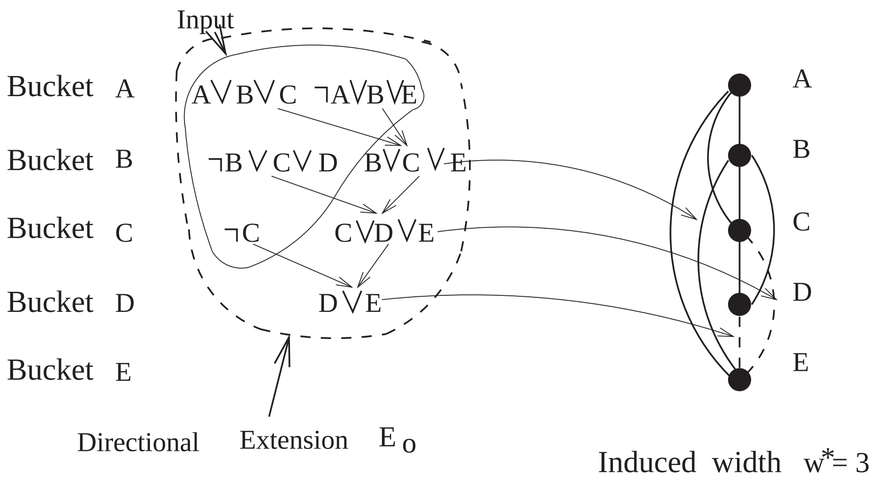


Figure 2.37: A schematic execution of directional resolution using ordering $d = E, D, C, B, A$

depicted in Figure 2.37. As demonstrated [17], 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 $bucket_i$.

We can easily show that the complexity of DR is exponentially bounded (time and space) in the *induced width* of the theory's *primal graph* in which a node is associated with a proposition and an arc connects any two nodes appearing in the same clause [14]. For example, the primal graph of theory φ along the ordering d is depicted in Figure 2.37 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.

2.8.2 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

```
Algorithm directional resolution
Input: A CNF theory $\varphi$, an ordering $d = Q_1, ..., Q_n$.
Output: A decision of whether $\varphi$ is satisfiable. If it is, a theory
$E_d(\varphi)$, equivalent to $\varphi$; 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 $\gamma$ and add $\gamma$ to the appropriate
      bucket.
3. Return: $E_d(\varphi) \Longleftarrow \bigcup_i bucket_i$ and generate a model in a
backtrack-free manner.
```

Figure 2.38: Algorithm *directional resolution*

```
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 $\gamma$ has no solutions, return inconsistency.
        else add $\gamma$ to the appropriate lower bucket.
return $E_o(\varphi) \leftarrow \bigcup_i bucket_i$.
```

Figure 2.39: Fourier elimination algorithm

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 2.8.1 *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 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.*

It is possible to show that the pair-wise join-project operation applied in a bucket can be accomplished by *linear elimination* as defined above. 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 [26]). 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 2.39. 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 set of i variables cannot be bounded exponentially by i. For a schematic execution of the algorithm see Figure 2.40, and for more details see [19].

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

Figure 2.40: 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 $d = t, z, y, x$

2.9 Summary

The chapter 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[5] 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 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 [17, 11]. 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 [11].

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

[5]all, except Fourier algorithm.

be addressed. In particular, improvements exploiting the structure of the conditional probability matrices as presented recently in [38, 6, 33] can be incorporated on top of bucket-elimination.

These restrictions are already available in the literature in the context of the existing algorithms [22, 40]

Bibliography

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

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

[3] 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.

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

[5] 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.

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

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

[8] G.F. Cooper. Nestor: A computer-based medical diagnosis aid that integrates causal and probabilistic knowledge. Technical report, Computer Science department, Stanford University, Palo-Alto, California, 1984.

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

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

[11] 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.

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

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

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

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

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

[17] 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.

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

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

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

[21] 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.

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

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

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

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

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

[27] 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.

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

[29] R. Qi N. L. Zhang and D. Poole. A computational theory of decision networks. *International Journal of Approximate Reasoning*, pages 83–158, 1994.

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

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

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

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

[34] 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.

[35] R. Mateescu R. Dechter and K. Kask. Iterative join-graph propagation. In *Uncertainty in Artificial Intelligence (UAI02)*, pages 128–138, 2002.

[36] 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.

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

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

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

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

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

[42] 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.

[43] 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.

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

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

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