Processing Probabilistic and Deterministic Graphical Models

Rina Dechter

DRAFT

May 1, 2013

Contents

Bibliography **155**

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
$\sigma_{x_1=d_1,...,x_k=d_k}(R)$	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

Over the last three decades, research in Artificial Intelligence has witnessed marked growth in the core disciplines of knowledge representation, learning and reasoning. This growth has been facilitated by a set of graph-based representations and reasoning algorithms known as *graphical models*.

The term graphical models describes a methodology for representing information, or knowledge, and for reasoning about that knowledge for the purpose of making decisions or for accomplishing other tasks by an intelligent agent. What makes these models *graphical* is that the structure used to represent the knowledge is often captured by a graph. The primary benefits of graph-based representation of knowledge are that it allows compact encoding of complex information and its efficient processing.

1.1 Probabilistic vs Deterministic Models

The concept of graphical models has mostly been associated exclusively with *probabilistic graphical models*. Such models are used in situations where there is uncertainty about the state of the world. The knowledge represented by these models concerns the joint probability distribution of a set of variables. An unstructured representation of such a distribution would be a list of all possible value combinations and their respective probabilities. This representation would require a huge amount of space even for a moderate

number of variables. Furthermore, reasoning about the information, for example, calculating the probability that a specific variable will have a particular value given some evidence would be very inefficient. A Bayesian network is a graph-based and a far more compact representation of a joint probability distribution (and, as such, a graphical model) where the information is encoded by relatively small number of conditional probability distributions as illustrated by the following example based on the early example by Spigelhalter and Lauritsen [41].

This simple medical diagnosis problem focuses on two diseases: Lung Cancer and Bronchitis. There is one sympton *dyspnoea* (shortness of breath), that may be associated with the presence of either disease (or both) and test results from X-ray that may be related to either cancer, or smoking or both. Whether or not the patient is a smoker also affects the likelihood of a patient having the diseases and symptoms. When a patient presents a particular combination of symptoms X-ray results it is usually impossible to say with certainty whether he suffers from either disease, from both, or from neither; at best, we would like to be able to calculate the probability of each of these possibilities. Calculating these probabilities (as well as many others) requires the knowledge of the joint probability distribution of the five variables (Lung Cancer (L) , Bronchitis (B), Dyspnea (D), Test of X-ray (T), and smoker (S)), that is, the probability of each of their 64 value combinations.

Alternatively, the joint probability distribution can be represented more compactly by factoring the distribution into a small number of conditional probabilities. One possible factorization, for example, is given by

$$P(S, L, B, D, T) = P(S)P(L|S)P(B|S)P(D|L, B)P(T|L)$$

This factorization corresponds to the directed graph in Figure 1.1 where each variable is represented by a node and there is an arrow connecting any two variables that have direct probabilistic (and may be causal) interactions between them, (that is, participate in one of the conditional probabilities).

In addition to allowing a more compact representation of the joint probability distribution, the graph also represents a set of independencies that are true for the distribution. For example, it shows that the variables Lung cancer and Bronchitis are conditionally independent on the variable smoking, that is, if smoking status is known then knowing that

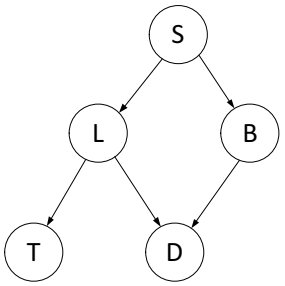


Figure 1.1: A simple medical diagnosis Bayesian network.

the patient has (or doesnt have) Lung cancer has no bearing on the probability that he has Bronchitis. However if it is also know that shortness of breath is present, Lung cancer and Bronchitis are no longer independent; knowing that the person has Lung cancer may explains away Bronchitis and reduces the likelihood of Dyspnea. Such independencies are very helpful for reasoning about the knowledge.

While the term graphical models has mostly been used for probabilistic graphical models, the idea of using a graph-based structure for representing knowledge has been used with the same amount of success in situations that seemingly have nothing to do with probability distributions or uncertainty. One example is that of constraint satisfaction problems. Rather than the probability of every possible combination of values assigned to a set of variables, the knowledge encoded in a constraint satisfaction problem concerns their feasibility, that is, whether these value combination satisfy a set of constraints that are often defined on relatively small subsets of variables. This structure is associated with a constraint graph where each variable is represented by a node and two nodes are connected by an edge if they are bound by at least one constraint. A constraint satisfaction problem along its constraint graph is often referred to as a constraint network and is illustrated by the following example.

Consider the map in Figure 1.2 showing eight neighboring countries and consider a set of three colors, red, blue, and yellow, for example. Each of the countries needs to be colored by one of the three colors so that no two countries that have a joint border have the same color. A basic question about this situation is to determine whether such a coloring scheme exists and, if so, to produce such a scheme. One way of answering these questions is to systematically generate all possible assignments of a color to a country and then

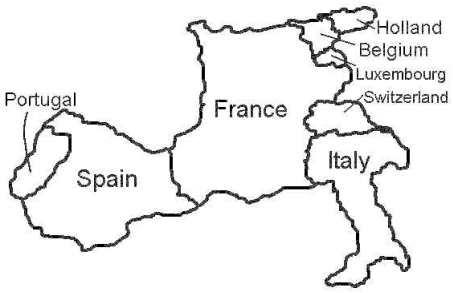


Figure 1.2: A map of eight neighboring countries

test each one to determine whether it satisfies the constraint. Such an approach would be very inefficient because the number of different assignments could be huge. The structure of the problem, represented by its constraint graph in Figure 1.3, could be helpful in accomplishing the task. In this graph each country is represented by a node and there is an edge connecting every pair of adjacent countries representing the constraint that prohibits that they be colored by the same color.

Just as in the Bayesian network graph, the constraint graph reveals the independencies in the map coloring problem. For example, it shows that if a color is selected for France the problem separates into three smaller problems (Portugal - Spain, Italy - Switzerland, and Belgium - Luxembourg - Holland) which could be solved independently of one another. This kind of information is extremely useful for expediting the solution of constraint satisfaction problems.

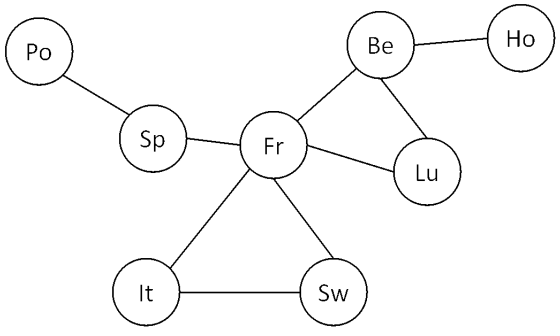


Figure 1.3: The map coloring constraint graph

Whereas a Bayesian network is an example of a probabilistic graphical model, a constraint network is an example of a deterministic graphical model. The graphs associated with the two problems are also different: Bayesian networks use directed graphs, indicating that the information regarding relationship between two variables is not symmetrical while constraint graphs are undirected graphs. Despite these differences, the significance of the graph-based structure and the way it is used to facilitate reasoning about the knowledge are sufficiently similar to place both problems in a general class of graphical models. Many other problem domains have similar graph based structures and are, in the view of this book, graphical models. Examples include propositional logic, integer linear programming, influence diagrams, and Markov networks.

1.2 Directed vs Undirected Models

The examples in previous section illustrate the two main classifications of graphical models. The first of these has to do with the kind information represented by the graph, primarily on whether the information is deterministic or probabilistic. Constraint networks are, for example, deterministic; an assignment of values to variables is either valid or it is not. Bayesian networks and Markov networks, on the other hand, represent probabilistic relationships; the nodes represent random variables and the graph as a whole encodes the joint probability distribution of those random variables. The distinction between these two categories of graphical models is not clear-cut, however. Cost networks, which represent preferences among assignments of values to variables are typically deterministic but they are similar to probabilistic networks as they are defined by real-valued functions just like probability functions.

The second classification of graphical models concerns how the information is encoded in the graph, primarily whether the edges in their graphical representation are directed or undirected. For example, Markov networks are probabilistic graphical models that have undirected edges while Bayesian networks are also probabilistic models but use a directed graph structure. Cost and constraint networks are primarily undirected yet some constraints are functional and can be associated with a directed model. For example, Boolean circuits encode functional constraints directed from inputs to outputs.

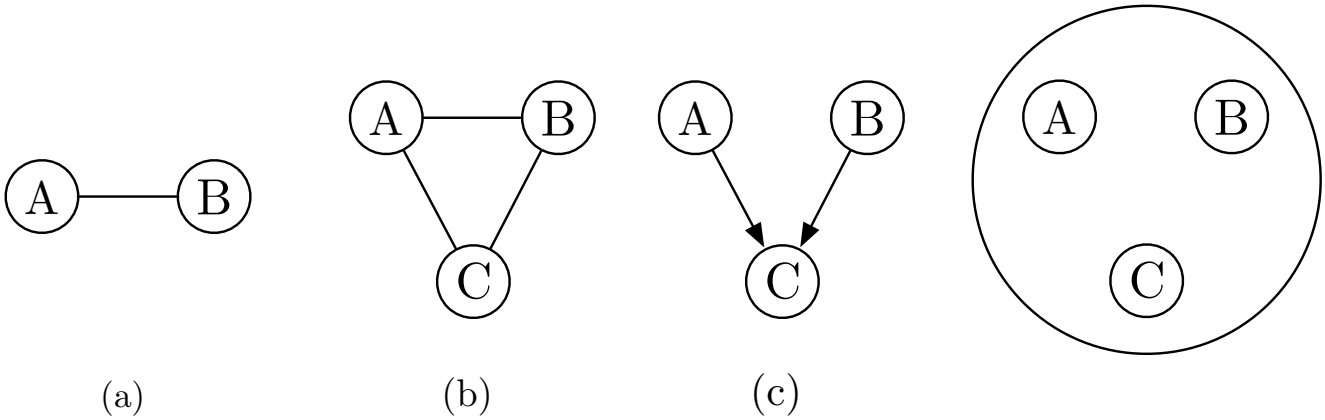


Figure 1.4: undirected and directed deterministic relationships

To make these classifications more concrete, consider a very simple example of a relationships between two variables. Suppose that we want to represent the logical relationship $A \vee B$ using a graphical model. We can do it by a constraint network of two variables and a single constraint (specifying that the relationship $A \vee B$ holds." The undirected graph representing this network is shown in (Figure 1.4(a)). We can add a third variable, C, that will be "true" if an only if the relation $A \vee B$ is "true," that is, $C = A \vee B$. This model may be expressed as a constraint on all three variables, resulting in the complete graph shown in Figure 1.4(b).

Now consider a probabilistic version of the above relationships, where, the case of $C = A \vee B$ we might employ a NOISY-OR relationship. A noisy-or function is the nondeterministic analog of the logical OR function and specifies that each input variable whose value is "1" produces an output of 1 with high probability $1 - \epsilon$ for some small ϵ. This can lead to the following encoding.

$$P(C = 1|A = 0, B = 0) = 0, \quad P(C = 1|A = 0, B = 1) = 1 - \epsilon_B,$$

$$P(C = 1|A = 1, B = 0) = 1 - \epsilon_A, \quad P(C = 1|A = 1, B = 1) = (1 - \epsilon_B)(1 - \epsilon_A)$$

This relationship is directional, representing the conditional probability of C for any given inputs to A and B and can parameterize the directed graph representation as in Figure 1.4(c). On the other hand, if we are interested to introduce some noise to an

A	B	$P(A \vee B)$
0	0	0
1	0	0.25
0	1	0.25
1	1	1/2

A	B	C	$P(A \vee B \vee C)$
0	0	0	0
1	0	0	1/15
0	1	0	1/15
0	0	1	1/15
1	1	0	2/15
1	0	1	2/15
0	1	1	2/15
1	1	1	6/15

Figure 1.5: parameterizing directed and undirected probabilistic relations

undirected relation $A \vee B$ we can do so by evaluating the strength of the OR relation in a way that fits our intuition or expertise, making sure that the resulting function is normalized. We could do the same for the ternary relation. These probabilistic functions are sometime called potentials or factors which frees them from the semantic coherency assumed when we talk about probabilities. Figure 1.5 shows a possible distribution of the noisy two and three-variable OR relation, which is symmetrical.

From an algorithmic perspective, the division between directed and undirected graphical models, is more salient and received considerable treatment in the literature [47]. Deterministic information seems to be merely a limiting case of nondeterministic information where probability values are limited to 0 and 1. Alternatively, it can be perceived as the limiting cost in preference description moving from 2-valued preference (consistent and inconsistent) to multi-valued preference, also called *soft constraints*. Yet, this book will be focused primarily on methods that are indifferent to the directionality aspect of the models, and be more aware of the deterministic vs non-deterministic distinction. The main examples used in this book will be constraint networks and Bayesian networks, since these are respective examples of both undirected and directed graphical models, and of Boolean vs numerical graphical models.

1.3 General Graphical models

Graphical models include constraint networks [23] defined by relations of allowed tuples, probabilistic networks [47], defined by conditional probability tables over subsets of variables or by a set of potentials, cost networks defined by costs functions and influence diagrams [35] which include both probabilistic functions and cost functions (*i.e.*, utilities) [22]. Mixed networks is a graphical model that distinguish between probabilistic information and deterministic constraints. Each graphical model comes with its typical queries, such as finding a solution (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.

The use of any model of knowledge (and graphical models are no exception) involves two, largely independent activities, the construction of the model and the extraction of useful information from the model. In the case of our medical diagnosis problem, for example, model construction involves the selection of the variables to be included, the structure of the Bayesian network, and the specification of the conditional probability distributions needed to specify the joint probability distribution. *Information extraction* involves answering queries about the effect of evidence on the probability of certain variables and about the best (most likely) explanation for such evidence. In the case of the map coloring problem the models structure is largely determined by the map to be colored. Information extraction involves answering queries like whether the map can be colored using a given set of colors, finding the minimum number of colors needed to color it and, if a map cannot be colored by a given number of colors, finding the minimum number of constraint violations that have to be incurred in order to color the map.

The construction of the graphical model, including learning its structure and parameters from data or from experts, depends very much on the specific type of problem. For example, constructing a Bayesian network would be a very different process from constructing an integer linear programming optimization problem. In contrast, the process of answering queries from graphical models, in particular when taking advantage of their graph-based structure, is more universal and common in many respects across many

types of problems. We call such activity as *reasoning* or, query processing, that is, deriving new conclusions from facts or data represented explicitly in the models. The focus of this book is on the common reasoning methods that are used to extract information from given graphical models. Reasoning over probabilistic models is often referred to as *inference*. We, however attribute a more narrow meaning to inference as discussed shortly.

Although the information extraction process for all the interesting questions posed over graphical models are computationally hard (i.e., NP-hard), and thus generally intractable, their structure invite effective algorithms for many graph structures as we show throughout the book. This includes answering optimization, constraint satisfaction, counting, and likelihood queries. And the breadth of these queries render these algorithms applicable to a variety of fields including scheduling, planning, diagnosis, design, hardware and software testing, bio-informatics and linkage analysis. Our goal is to present a unifying treatment in a way that goes beyond a commitment to the particular types of knowledge expressed in the model.

In chapter two, we will define the framework of graphical models and will review the various flavors of models. But, as already noted, the focus of this book is on query processing algorithms which exploit graph structures primarily and are thus applicable across all graphical models. These algorithms can be broadly classified as either inference-based or search-based, and each class will be discussed separately, for they share different characteristics. Inference-based algorithms perform a deductive step repeatedly while maintaining a single view of the model. Some example of inference-based algorithms are resolution, variable-elimination and join-tree clustering. These algorithms are exponentially bounded in both time and space by a graph parameter called *tree-width*. Search-based algorithms perform repeatedly a *conditioning step*, namely, fixing the value of a variable to a constant, and thus restrict the attention to a subproblem. This leads to a search over all subproblems that need to be solved eventually. Search algorithms can be executed in linear space, and this makes them attractive. These algorithms can be shown to be exponentially bounded by graph-cutset parameters that depend on the memory level the algorithm would use. When search and inference algorithms are combined they enable improved performance by flexibly trading off time and space.

Previous books on graphical models focused either on probabilistic networks, or on

constraint networks. The current book is therefore broader in its unifying perspective. Yet it has restricted boundaries along the following dimensions. We address only graphical models over discrete variables (no continuous variables), we cover only exact algorithms (a subsequent extension for approximation is forthcoming), we address only propositional graphical models (recent work on first-order graphical models is outside the scope of this book.) In addition, we will *not* focus on exploiting the local structure of the functions. what is knkoe the context-specific information. Such techniques are orthogonal to graph-based principles and can and should be combined with them.

Finally, and as already noted, the book will not cover issues of modeling (by knowledge acquisition or learning from data) which are the two primary approaches for generating probabilistic graphical models. For this, and for more we refer the readers to the books in the area. First and foremost is the classical book that introduced probabilistic graphical models [47] and a sequence of books that followed amongst which are [46, 36]. In particular note the comprehensive two recent textbooks [1, 39]. For deterministic graphical models of Constraint networks see [23].

1.4 Overview of the book, chapter by chapter

The focus in this book is on query processing algorithms which exploit the graph structure and are therefore applicable across all graphical models. It is useful to distinguish two types of algorithms: inference-based and search-based. Algorithms within each class share different characteristics. Inference-based algorithms (e.g., variable-elimination, join-tree clustering) are time and space exponentially bounded by a graph parameter called *tree-width*. Their complexity bounds are well studied and understood for more than 2 decades now.

Search-based algorithms on the other hand are attractive because they can be executed in linear space. Effective structure-based time bounds for search emerged only recently. By augmenting bounded ingerence ideas they can flexibly tradeoff time and space. Furthermore, search methods are more naturally poised to exploit the internal structure of the functions themselves, what is often called their *local structure*. The thrust of advanced reasoning schemes is in combining inference and search yielding a spectrum of

memory-sensitive algorithms universally applicable across many domains.

Chapter 2 presents the reader to the concepts of graphical models, provide definitions and the specific graphical models discussed throughout the book. Chapters 3-6 focus on inference algorithms, chapters 7-9 on search, while chapter 10 on hybrids of search and inference. Specifically, in the inference part, chapter 3 describes a variable-elimination scheme called *bucket-elimination* for constraint networks, chapter 4 use the motivation for graph parameters introduced in Chapter 3 to address and elaborate on graph properties that are relevant to the algorithms's design and analysis that will be exploited throughout the book. Then Chapter 5 focuses on bucket-elimination for probabilistic networks and chapter 6 shows how these variable elimination algorithms can be extended to tree-decompositions yielding the join-tree and junction-tree propagation schemes. Search is introduced through Chapter 7 and 8 through AND/OR decomposition. We conclude with chapter 10 giving hybrids of search an inference.

Chapter 2

What are Graphical Models

In this chapter, we will begin by introducing the general graphical model framework and continue with the most common types of graphical models, providing examples of each type: constraint networks [23], Bayesian networks, Markov networks [47] and cost networks. Another more involved example which we only briefly discuss is influence diagrams [35].

2.1 General Graphical models

Graphical models include constraint networks [23] defined by relations of allowed tuples, probabilistic networks [47], defined by conditional probability tables over subsets of variables or by a set of potentials, cost networks defined by costs functions and influence diagrams [35] which include both probabilistic functions and cost functions (*i.e.*, utilities) [22]. Mixed networks is a graphical model that distinguish between probabilistic information and deterministic constraints. Each graphical model comes with its typical queries, such as finding a solution (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.

Simply put, a graphical model is a collection of *local* functions over subsets of variables that convey probabilistic, deterministic or preferential information and whose structure is described by a graph. The graph captures independency or irrelevance information

inherent in the model that can be useful for interpreting the data in the model and, most significantly, can be exploited by reasoning algorithms.

A graphical model is defined by a set of variables, their respective domains of values which we assume to be discrete and by a set of functions. Each function is defined on a subset of the variables called its *scope*, which maps any assignment over its scope, an instantiation of the scopes' variables, to a real value. The set of local functions can be *combined* in a variety of ways (e.g., by sum or product) to generate a *global function* whose scope is the set of all variables. Therefore, a *combination* operator is a defining element in a graphical model. As noted, common combination operators are summation and multiplication, but we also have *AND* operator, for Boolean functions, or the relational *join*, when the functions are relations.

We denote variables or sets of variables by uppercase letters (e.g., X, Y, Z, S) and values of variables by lower case letters (e.g., x, y, z, s). An assignment $(X_1 = x_1, ..., X_n = x_n)$ can be abbreviated as $x = (x_1, ..., x_n)$. For a set of variables S, D_S denotes the Cartesian product of the domains of variables in S. If $X = \{X_1, ..., X_n\}$ and $S \subseteq X$, x_S denotes the projection of $x = (x_1, ..., x_n)$ over S. We denote functions by letters f, g, h, etc., and the scope (set of arguments) of a function f by $scope(f)$. The projection of a tuple x on the scope of a function f, can also be denoted by $x_{scope(f)}$ or, for brevity x_f.

Definition 2.1.1 (elimination operators) *Given a function h defined over a scope S, the functions $(\min_X h)$, $(\max_X h)$, and $(\sum_X h)$ where $X \subseteq S$, are defined over $U = S - \{X\}$ as follows: For every $U = u$, and denoting by (u, x) the extension of tuple u by the tuple $X = x$, $(\min_X h)(u) = \min_x h(u, x)$, $(\max_X h)(u) = \max_x h(u, x)$, and $(\sum_X h)(u) = \sum_x h(u, x)$. Given a set of functions $h_1, ..., h_k$ defined over the scopes $S_1, ..., S_k$, the product function $\Pi_j h_j$ and the sum function $\sum_j h_j$ are defined over $U = \cup_j S_j$ such that for every $u \in D_U$, $(\Pi_j h_j)(u) = \Pi_j h_j(u_{S_j})$ and $(\sum_j h_j)(u) = \sum_j h_j(u_{S_j})$. Alternatively, $(\Pi_j h_j)(u) = \Pi_j h_j(u_{h_j})$ and $(\sum_j h_j)(u) = \sum_j h_j(u_{h_j})$.*

The formal definition of a graphical model is give next.

Definition 2.1.2 (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, defined over scopes of variables* $\mathbf{S}_i \subseteq \mathbf{X}$*. They are called* local *functions.*

4. $\otimes$ *is a combination operator. The combination operator can also be defined axiomatically as in [57]. (e.g.,* $\otimes \in \{\prod, \sum, \bowtie\}$ *(product, sum, join)). But for the sake of our discussion we can define it explicitly.*

The graphical model represents a global function *whose scope is* $\mathbf{X}$ *which is the combination of all its functions:* $\otimes_{i=1}^{r} f_i$*.*

Note that the local functions define the graphical model and are given as input. The global function provides the meaning of the graphical model but it cannot be computed explicitly (e.g., in a tabular form) due to its exponential size. Yet all the interesting reasoning tasks (called also 'problems' or 'queries') are defined relative to the global function. For instance, we may seek an assignment on all the variables (sometime called configuration, or a solution) having the *maximum* global value. Alternatively, we can ask for the number of solutions to a constraint problem, defined by a *summation.* We can therefore define a variety of *reasoning queries* using an additional operator called *marginalization.* For example, if we have a function defined on two variables, $F(X, Y)$, a maximization query can be specified by applying the *max* operator written as $\max_{x,y} F(x, y)$ which returns a function with no arguments, namely, a constant, or, we may seek the maximizing tuple $(x^*, y^*) = argmax_{x,y} F(x, y)$. Sometime we are interested to get $Y(x) = argmax_y F(x, y)$.

Since the marginalization operator, which is max in the above examples, operates on a function of several variables and returns a function on their subset, it can be viewed as *eliminating* some variables from the function's scope to which it is applied. Because of that it is also called an *elimination* operator. Consider another example when we have a joint probability distribution $P(X, Y)$ and we want to compute the marginal probability $P(X) = \sum_y P(X, y)$. In this case we use the *sum* marginalization operator to express our query. A formal definition of a reasoning task using the notion of a *marginalization* operator, is given next. We define *marginalization* by explicitly listing the specific operators

we consider, but those can also be characterized axiomatically ([37, 57, 12]).

Definition 2.1.3 (a reasoning problem) *A* reasoning problem *over a graphical model* $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes \rangle$ *and a subset of variable* $Y \subset \mathbf{X}$ *is defined by a marginalization operator* $\Downarrow_{\mathbf{Y}}$ *explicitly as follows. If* $\mathbf{S}$ *is the scope of function* f *then* $\Downarrow_{\mathbf{Y}} f \in \{ \max_{\mathbf{S}-\mathbf{Y}} f, \min_{\mathbf{S}-\mathbf{Y}} f, \pi_{\mathbf{Y}} f, \sum_{\mathbf{S}-\mathbf{Y}} f \}$ *is a marginalization operator. The reasoning problem* $\mathcal{P}\langle \mathcal{M}, \Downarrow_{\mathbf{Z}} \rangle$ *is the task of computing the function* $\mathcal{P}_{\mathcal{M}}(\mathbf{Z}) = \Downarrow_{\mathbf{Z}} \otimes_{i=1}^{r} f_i$*, where* r *is the number of functions in* F*.*

We will focus often on reasoning problems defined by $\mathbf{Z} = \{\emptyset\}$. Note that in our definition $\pi_{\mathbf{Y}} f$ is the relational projection operator and unlike the rest of the marginalization operators the convention is that it is defined by the scope of variables that are *not* eliminated.

Every graphical model can be associated with several graph representations. We next define the most common graph representation called the *primal graph*.

Definition 2.1.4 (primal graph) *The* primal graph *of a graphical model is an undirected graph in which each vertex corresponds to a variable in the model and in which an edge connects any two vertices if the corresponding variables appear in the scope of the same local function.*

We will now describe several specific graphical models and show how they fit the general definition.

2.2 Constraint Networks

Constraint networks provide a framework for formulating real world problems as satisfying a set of constraints among variables, and they are the simplest and most computationally tractable of the graphical models we will be considering. Problems in scheduling, design, planning and diagnosis are often encountered in real world scenarios and can be effectively rendered as constraint networks problems.

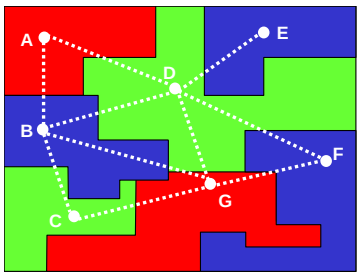
Let's take scheduling as an example. Consider the problem of scheduling several tasks, where each takes a certain time and each have different options for starting time. Tasks can be executed simultaneously, subject to some precedence restriction between them due

to certain resources that they need but cannot share. One approach to formulating such a scheduling problem is as a constraint satisfaction problem having a variable for each combination resource and time slice (e.g. the conference room at 3pm on Tuesday, for a class scheduling problem). The domain of each variable is the set of tasks that need to be scheduled, and assigning a task to a variable means that this task will begin at this resource at the specified time. In this model, various physical constraints can be described as constraints between variables (e.g. that a given task takes three hours to complete or that another task can be completed at most once).

The *constraint satisfaction task* is to find a solution to the constraint problem, that is, an assignment of a value to each variable such that no constraint is violated. If no such assignment can be found, we conclude that the problem is inconsistent. Other queries include finding all the solutions and counting them or, if the problem is inconsistent, finding a solution that satisfies the maximum number of constraints.

Definition 2.2.1 (constraint network, constraint satisfaction problem (CSP)) *A* constraint network (CN) *is a 4-tuple,* $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \bowtie \rangle$*, 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* join *operator* $\bowtie$ *is used to combine the constraints into a global relation. When it is clear that we discuss constraints we will refer to the problem as a triplet* $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$*. A* solution *is an assignment of values to all the variables, denoted* $x = (x_1, \ldots, x_n)$*,* $x_i \in D_i$*, such that* $\forall\ C_i \in \mathbf{C}$*,* $x_{\mathbf{S}_i} \in R_i$*. The constraint network represents its set of solutions,* $sol(\mathcal{R}) = \bowtie_i R_i$*. We see that a* constraint network *is a graphical model* $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \bowtie \rangle$ *whose functions are relations and the combination operator is the relational join. We define the minimal domain of a variable* X *to be all its values that participate in any solution. Namely,* $MinDom(X_i) = \pi_{X_i} \bowtie_j R_j$

The primary query over a constraint network is deciding if it has a solution. Other relevant queries are enumerating or counting the solutions. Namely, the primary reasoning tasks can be expressed as $\mathcal{P} = \langle \mathcal{R}, \pi, \mathbf{Z} \rangle$, when marginalization is the relational projection operator π. That is, $\Downarrow_{\mathbf{Y}}$ is $\pi_{\mathbf{Y}}$. Therefore the task of finding all solutions is expressed by $\Downarrow_{\emptyset} \otimes_i f_i = \pi_{\emptyset}(\bowtie_i f_i)$.



(a) Graph coloring problem

Figure 2.1: A constraint network example of a map coloring

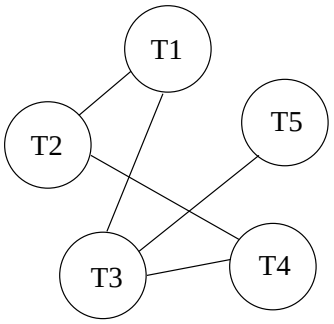
The primal graph of a constraint network is called a *constraint graph*. It is an undirected graph in which each vertex corresponds to a variable in the network and in which an edge connects any two vertices if the corresponding variables appear in the scope of the same constraint.

Example 2.2.2 The map coloring problem in Figure 2.1(a) can be modeled by a constraint network: given a map of regions and three colors {red, green, blue}, the problem is to color each region by one of the colors such that neighboring regions have different colors. Each region is a variable, and each has the domain {red, green, blue}. The set of constraints is the set of relations *"different"* between neighboring regions. Figure 2.1 overlays the corresponding constraint graph and one solution (A=red, B=blue, C=green, D=green, E=blue, F=blue, G=red) is given. The set of constraints are $A \neq B$, $A \neq D$, $B \neq D$, $B \neq C$, $B \neq G$,$D \neq G$, $D \neq F$,$G \neq F$, $D \neq E$.

□

Example 2.2.3 As noted earlier, constraint networks are particularly useful for expressing and solving scheduling problems. Consider the problem of scheduling five tasks (T1, T2, T3, T4, T5), each of which takes one hour to complete. The tasks may start at 1:00, 2:00 or 3:00. Tasks can be executed simultaneously subject to the restrictions that:

- T1 must start after T3,
- T3 must start before T4 and after T5,
- T2 cannot be executed at the same time as either T1 or T4, and



Unary constraint
$D_{T4} = \{1{:}00, 3{:}00\}$
Binary constraints
$R_{\{T1,T2\}}$: {(1:00,2:00), (1:00,3:00), (2:00,1:00), (2:00,3:00), (3:00,1:00), (3:00,2:00)}
$R_{\{T1,T3\}}$: {(2:00,1:00), (3:00,1:00), (3:00,2:00)}
$R_{\{T2,T4\}}$: {(1:00,2:00), (1:00,3:00), (2:00,1:00), (2:00,3:00), (3:00,1:00), (3:00,2:00)}
$R_{\{T3,T4\}}$: {(1:00,2:00), (1:00,3:00), (2:00,3:00)}
$R_{\{T3,T5\}}$: {(2:00,1:00), (3:00,1:00), (3:00,2:00)}

Figure 2.2: The constraint graph and constraint relations of the scheduling problem.

- T4 cannot start at 2:00.

We can model this scheduling problem by creating five variables, one for each task, where each variable has the domain {1:00, 2:00, 3:00}. The corresponding constraint graph is shown in Figure 2.2, and the relations expressed by the graph are shown beside the figure. □

Sometimes we express the relation R_i as a cost function $C_i(X_{i_1} = x_{i_1}, \ldots, X_{i_k} = x_{i_k}) = 1$ if $(x_{i_1}, \ldots, x_{i_k}) \in R_i$, and 0 otherwise. In this case the combination operator is a product. We will switch between these two views as needed. If we want to count the number of solutions we merely change the marginalization operator to be summation. If on the other hand we want merely to query whether the constraint network has a solution, we can let the marginalization operator be logical summation. We let $\mathbf{Z} = \{\emptyset\}$, so that the the summation occurs over all the variables. We will get "1" if the constraint problem has a solution and "0" otherwise.

Propositional Satisfiability One special case of the constraint satisfaction problem is what is called *propositional satisifiability* (usually referred to as SAT). Given a formula φ in *conjunctive normal form* (CNF), the SAT problem is to determine whether there is a truth-assignment of values to its variables such that the formula evaluates to true. A formula is in conjunctive normal form if it is a conjunction of *clauses* $\alpha_1, \ldots, \alpha_t$, where each clause is a disjunction of *literals* (propositions or their negations). For example, $\alpha = (P \vee \neg Q \vee \neg R)$ and $\beta = (R)$ are both clauses, where P, Q and R are propositions, and P, $\neg Q$ and $\neg R$ are literals. $\varphi = \alpha \wedge \beta = (P \vee \neg Q \vee \neg R) \wedge (R)$ is a formula in conjunctive normal form.

Propositional satisfiability can be defined as a constraint satisfaction problem in which each proposition is represented by a variable with domain $\{0, 1\}$, and a clause is represented by a constraint. For example, the clause $(\neg A \vee B)$ is a relation over its propositional variables that allows all tuple assignments over (A, B) except $(A = 1, B = 0)$.

2.3 Cost Networks

In constraint networks, the local functions are constraints, i.e., functions that assign a boolean value to a set of inputs. However, it is straightforward to extend constraint networks to accommodate real-valued relations using a graphical model called a *cost network*. In cost networks, the local functions represents cost-components, and the sum of these cost-components is the global cost function of the network. The primary task is to find an assignment of the variables such that the global cost function is optimized (minimized or maximized). Cost networks enable one to express preferences among local assignments and, through their global costs to express preferences among full solutions.

Often, problems are modeled using both constraints and cost functions. The constraints can be expressed explicitly as being functions of a different type than the cost functions, or they can be included as cost components themselves. It is straightforward to see that cost networks are graphical model where the combination operator is summation.

Definition 2.3.1 (cost network, combinatorial optimization) *A* cost network *is a 4-tuple graphical model,* $\mathcal{C} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$*, 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 local cost*

functions $\mathbf{F} = \{f_1, \ldots, f_r\}$. *Each* f_i *is a real-valued function (called also cost-component) defined on a subset of variables* $\mathbf{S}_i \subseteq \mathbf{X}$. *The local cost components are combined into a global cost function via the* $\sum$ *operator. Thus the cost network represents the function*

$$\mathcal{C}(x) = \sum_i f_i(x_{S_i})$$

which can also be written as

$$\mathcal{C}(x) = \sum_{f \in F} f(x_f)$$

where x_f *is the projection of* x *on the scope of* f.

The primary optimization task (which we will assume to be a minimization, w.l.o.g) is to find an optimal solution for the global cost function $\mathcal{F} = \sum_i f_i$. *Namely, finding a tuple* x *such that* $\mathcal{F}(x) = \min_x \sum_i f_i(x)$. *We can associate the cost model with its primal graph in the usual way. Like in the case of constraints, we will drop the* $\sum$ *notation whenever the nature of the functions and their combination into a global function is clear from the context.*

Weighted Constraint Satisfaction Problems A special class of cost networks that has gained considerable interest in recent years is a graphical model called the Weighted Constraint Satisfaction Problem (WCSP) [12]. These networks extends the classical constraint satisfaction problem formalism with *soft constraints*, that is, positive integer-valued local cost functions. Formally,

Definition 2.3.2 (WCSP) *A* Weighted Constraint Satisfaction Problem (WCSP) *is a graphical model* $\langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$ *where each of the functions* $f_i \in \mathbf{F}$ *assigns "0" (no penalty) to allowed tuples and a positive integer penalty cost to the forbidden tuples. Namely,* $f_i : D_{S_i} \rightarrow \mathbb{N}$, *where* S_i *is the scope of the function. More explicitly,* $f_i : D_{X_{i_1}} \times ... \times D_{X_{i_t}} \rightarrow \mathbb{N}$, *where* $S_i = \{X_{i_1}, ..., X_{i_t}\}$ *is the scope of the function.*

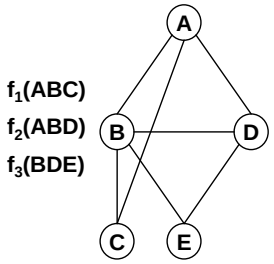
Many real-world problems can be formulated as cost networks and often fall into the weighted CSP class. This includes resource allocation problems, scheduling [8], bioinformatics [17, 62], combinatorial auctions [53, 23] and maximum satisfiability problems [16].

A	B	C	f_1(ABC)
0	0	0	∞
0	0	1	∞
0	1	0	∞
0	1	1	2
1	0	0	∞
1	0	1	2
1	1	0	∞
1	1	1	2

A	B	D	f_2(ABD)
0	0	0	1
0	0	1	∞
0	1	0	0
0	1	1	2
1	0	0	6
1	0	1	5
1	1	0	6
1	1	1	5

B	D	E	f_3(BDE)
0	0	0	∞
0	0	1	3
0	1	0	∞
0	1	1	4
1	0	0	∞
1	0	1	3
1	1	0	∞
1	1	1	4

(a) Cost functions



(b) Constraint graph

Figure 2.3: A cost network.

Example 2.3.3 Figure 2.3 shows an example of a WCSP instance with boolean variables. The cost functions are given in Figure 2.3(a), and the associated graph is shown in Figure 2.3(b). Note that a value of ∞ in the cost function denotes a hard constraint (i.e., high penalty). You should check that the minimal cost solution of the problem is 5, which corresponds to the assignment $(A = 0, B = 1, C = 1, D = 0, E = 1)$. □

The task of MAX-CSP, namely of finding a solution that satisfies the maximum number of constraints (when the problem is inconsistent), can be formulated as a cost network by treating each relation as a cost function that assigns "0" to consistent tuples and "1" otherwise. Since all violated constraints are penalized equally, the global cost function will simply count the number of violations. In this case the combination operator is summation and the marginalization operator is minimization. Namely, the task is to find $\Downarrow_\emptyset \otimes_i f_i = \min_{\mathbf{X}}(\sum_i f_i)$. Formally,

Definition 2.3.4 (MAX-CSP) *A* MAX-CSP *is a WCSP* $\langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$ *with all penalty costs equal to* 1. *Namely,* $\forall f_i \in \mathbf{F}$, $f_i : D_{S_i} \rightarrow \{0, 1\}$, *where scope*$(f_i) = S_i$

Maximum Satisfiability In the same way that propositional satisfiability (SAT) can be seen as a constraint satisfaction problem over logical formulas in conjunctive normal form, so can the problem of *maximum satisfiability* (MAX-SAT) be formulated as a MAX-CSP problem. In this case, given a set of boolean variables and a collection of clauses defined over subsets of those variables, the goal is to find a truth assignment that violates the least number of clauses. Naturally, if each clause is associated with a positive weight,

then the problem can be described as a WCSP. The goal of this problem, called *weighted maximum satisfiability* (Weighted MAX-SAT), is to find a truth assignment such that the sum weight of the violated clauses is minimized.

Integer Linear Programs. Another well known class of optimization task is integer linear programming. It is formulated over variables that can be assigned integer values (finite or infinite). The task is to find an optimal solution to a linear cost function $F(x) = \sum_i \alpha_i x_i$ that satisfies a set of linear constraints $C_1, ...C_l$ where each constraint can be specified by a linear function. Namely a constraint C_i over scopes S_i a constraint $\sum_{x \in S_i} \lambda_x \cdot x \leq 0$. Formally,

Definition 2.3.5 (Integer linear programming) *A* Integer Linear programming Problem (IP) *is a graphical model* $\langle \mathbf{X}, \mathbf{N}, \mathbf{F} = \{f_1, ...f_n, C, 1, .., C_l\}, \sum \rangle$ *having two types of functions. Linear cost components* $f(x_i) = \alpha_i x_i$ *for each variable* X_i*, where* α_i *is a real number. The constraints are of weighted csp type*

$$C_i(x_{S_i}) = \begin{cases} 0, & if \quad \sum_{x_j \in S_i} \lambda_{i_j} \cdot x \leq \lambda_i \\ \infty & otherwise \end{cases}$$

or infinity otherwise. The marginalization operator is minimization or maximization.

2.4 Probability Networks

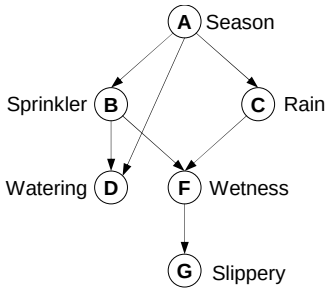
As mentioned previously, Bayesian networks and Markov networks are the two primary formalisms for expressing probabilistic information via graphical models. A *Bayesian network* [47] is defined by a directed acyclic graph over vertices that represent random variables of interest (e.g., the temperature of a device, the gender of a patient, a feature of an object, the occurrence of an event). The arc from one node to another is meant to signify a direct causal influence or correlation between the respective variables, and this influence is quantified by the conditional probability of the child given all of its parents. Therefore, to define a Bayesian network, one needs both a directed graph and the associated conditional probability functions. To be consistent with our graphical models description we define Bayesian network as follows.

Definition 2.4.1 (Bayesian networks) *A* Bayesian network (BN) *is a 4-tuple* $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$. $\mathbf{X} = \{X_1, \ldots, X_n\}$ *is a set of ordered variables defined over domains* $\mathbf{D} = \{D_1, \ldots, D_n\}$, *where* $o = (X_1, \ldots, X_n)$ *is an ordering of the variables. The set of functions* $P_G = \{P_1, \ldots, P_n\}$ *consist of conditional probability tables (CPTs for short)* $P_i = \{P(X_i \,|\mathbf{Y}_i)\}$ *where* $\mathbf{Y}_i \subseteq \{X_{i+1}, ..., X_n\}$. *These* P_i *functions can be associated with a directed acyclic graph* G *in which each node represents a variable* X_i *and* $\mathbf{Y}_i = pa\,(X_i)$ *are the parents of* X_i *in the graph. That is, there is a directed arc from each parent variable of* X_i *to* X_i. *The Bayesian network* $\mathcal{B}$ *represents the probability distribution over* $\mathbf{X}$, $P_{\mathcal{B}}(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|x_{pa(X_i)})$. *We define an evidence set* e *as an instantiated subset of the variables.*

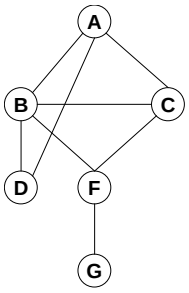
The parent/child relations of a Bayesian network, regardless of whether they actually represent causal relationships, always yields a valid joint probability distribution which is consistent with its input CPTs. Namely, for each X_i and its parent set Y_i, it can be shown that $P_{\mathcal{B}}(X_i|Y_i) = P(X_i|Y_i)$. Therefore a Bayesian network is a graphical model, where the functions in F denote conditional probability tables and the scope of each function f_i is X_i and its parents in the directed graph G, where the combination operator is product, $\otimes = \prod$. The primal graph of a Bayesian network is called a moral graph and it connects any two variables appearing in the same CPT. The moral graph can also be obtained from the directed graph G by connecting the parents of each child node and making all directed arcs undirected.

Example 2.4.2 [47] Figure 2.4(a) is a Bayesian network over six variables, and Figure 2.4(b) shows the corresponding moral graph. The example expresses the causal relationship between variables "season" (A), "the automatic sprinkler system is "on"' (B), "whether it rains or does not rain" (C), "manual watering is necessary" (D), "the wetness of the pavement" (F), and "the pavement is slippery" (G). The Bayesian network is defined by six conditional probability tables each associated with a node and its parents. For example, the CPT of F describes the probability that the pavement is wet ($F = 1$) for each status combination of the sprinkler and raining. Possible CPTs are given in Figure 2.4(c).

The conditional probability tables contain only half of the entries because the rest of the information can be derived based on the property that all the conditional probabilities



(a) Directed acyclic graph



(b) Moral graph

B	C	F	$P(F\|B,C)$
false	false	true	0.1
true	false	true	0.9
false	true	true	0.8
true	true	true	0.95

B	$A = winter$	D	$P(D\|A,B)$
false	false	true	0.3
true	false	true	0.9
false	true	true	0.1
true	true	true	1

A	C	$P(C\|A)$
Summer	true	0.1
Fall	true	0.4
Winter	true	0.9
Spring	true	0.3

A	B	$P(B\|A)$
Summer	true	0.8
Fall	true	0.4
Winter	true	0.1
Spring	true	0.6

F	G	$P(G\|F)$
false	true	0.1
true	true	1

(c) Possible CPTs that accompany our example

Figure 2.4: Belief network $P(G, F, C, B, A) = P(G|F)P(F|C, B)P(D|A, B)P(C|A)P(B|A)P(A)$

sum to 1. This Bayesian network expresses the probability distribution $P(A, B, C, D, F, G) = P(A) \cdot P(B|A) \cdot P(C|A) \cdot P(D|B, A) \cdot P(F|C, B) \cdot P(G|F)$. □

Next, we define the main queries over Bayesian networks:

Definition 2.4.3 (Queries over Bayesian networks) *Let $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$ be a Bayesian network. Given evidence $E = e$ the primary queries over Bayesian networks are to find the following quantities:*

1. **Posterior marginals, or belief.** *For every $X_i = x_i$ not in E the belief is defined by $bel(x_i) = P_{\mathcal{B}}(x_i|e)$.*

2. *The* **Probability of evidence** *is* $P_{\mathcal{B}}(e)$.

3. *The* **Most probable explanation** (*mpe*) *is an assignment* $x^o = (x^o{}_1, ..., x^o{}_n)$ *satisfying* $P_{\mathcal{B}}(x^o) = \max_x P_{\mathcal{B}}(x|e)$. *The mpe value is* $\max_x P_{\mathcal{B}}(x|e)$.

4. **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* $P(a^o) = \max_{\bar{a}_k} P(\bar{a}_k|e)$. *The mpe query is sometime also referred to as map query.*

These queries are applicable to a variety of applications such as situation assessment, diagnosis, probabilistic decoding and linkage analysis, to name a few. To answer the above queries over $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$ we use as marginalization operators either summation or maximization. In particular, the query of finding the probability of the evidence can be expressed as $\Downarrow_\emptyset \otimes_i f_i = \sum_X \prod_i P_i$. The *belief updating* task, when given evidence e, can be formulated using the summation as a marginalization operator, by $\Downarrow_{\mathbf{Y}} = \sum_{\mathbf{X}-\mathbf{Y}}$, where $\mathbf{Z}_i = \{X_i\}$. Namely, $\forall X_i, bel(X_i) = \Downarrow_{X_i} \otimes_k f_k = \sum_{\{X - X_i | E=e\}} \prod_k P_k$. An mpe task is defined by a maximization operator where $\mathbf{Z} = \{\emptyset\}$, yielding $mpe = \Downarrow_\emptyset \otimes_i f_i = \max_X \prod_i P_i$. If we want to get the actual mpe assignment we would need to use the *argmax* operator.

Markov networks also called *Markov Random Fields (MRF)* are undirected probabilistic graphical models very similar to Bayesian networks. However, unlike Bayesian networks they convey undirectional information, and are therefore defined over an undirected graph. Moreover, whereas the functions in Bayesian networks are conditional probability tables of children given their parents in the directed graph, in Markov networks the local functions, called potentials, can be defined over any subset of variables. These potential functions between random variables can be thought of as expressing some kind of a correlation information. When a configuration to a subset of variables is likely to occur together their potential value may be large. For instance in vision scenes, variables may represent the grey levels of pixels, and neighboring pixels are likely to have similar grey values. Therefore, they can be given a higher potential level. Other applications of Markov Random fields are in Physics (e.g., modeling magnetic behaviors of crystals).

Like a Bayesian network, a Markov network also represents a joint probability distribution, even though its defining local functions do not have a clear probabilistic semantics. In particular, they do not express local marginal probabilities (see [47] for a discussion).

Markov networks are useful when the notion of directionality in the information is unnatural. Example applications are image analysis and spatial statistics. They convey symmetrical information and can be viewed as the probabilistic counterpart of constraint or cost networks, whose functions are symmetrical as well.

Definition 2.4.4 (Markov Networks) *A Markov network is a graphical model* $\mathcal{M} = \langle \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{H}, \prod \rangle$ *where* $\boldsymbol{H} = \{\psi_1, \ldots, \psi_m\}$ *is a set of potential functions where each potential* ψ_i *is a non-negative real-valued function defined over a scope of variables* $\boldsymbol{S}_i$*. The Markov network represents a global joint distribution over the variables* $\boldsymbol{X}$ *given by:*

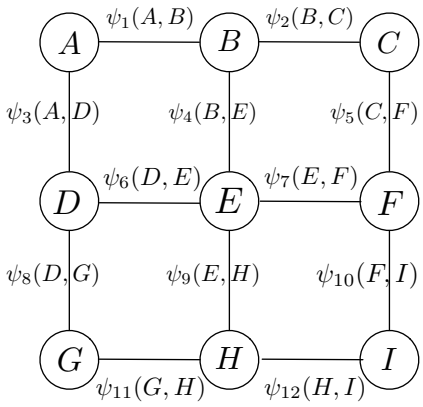
$$P(\boldsymbol{x}) = \frac{1}{Z} \prod_{i=1}^{m} \psi_i(\boldsymbol{x}) \quad , \quad Z = \sum_{\boldsymbol{x}} \prod_{i=1}^{m} \psi_i(\boldsymbol{x})$$

where the normalizing constant Z *is referred to as the partition function.*

The primary queries over Markov networks are the same as those of Bayesian network. That is, computing the posterior marginal distribution over all variables $X_i \in \mathbf{X}$, finding the *mpe* value and finding the partition function. It is not hard to see that this later query is mathematically identical to computing the probability of evidence. We see that as a graphical model, Markov networks are very similar to Bayesian networks, Markov networks are graphical models whose combination operator is the product operator, $\otimes = \prod$ and the marginalization operator can be summation, or maximization, depending on the query.

Example 2.4.5 Figure 2.5 shows a 3×3 square grid Markov network with 9 variables $\{A, B, C, D, E, F, G, H, I\}$. The twelve potentials are: $\psi_1(A, B)$, $\psi_2(B, C)$, $\psi_3(A, D)$, $\psi_4(B, E)$, $\psi_5(C, F)$, $\psi_6(C, D)$, $\psi_7(D, E)$, $\psi_8(D, G)$, $\psi_9(E, H)$, $\psi_{10}(F, I)$, $\psi_{11}(G, H)$ and $\psi_{12}(H, I)$. The Markov network represents the probability distribution formed by taking a product of these twelve functions and then normalizing. Namely, given that $x = (a, b, c, d, e, f, g, h, i)$

$$P(x) = \frac{1}{Z} \prod_{i=1}^{12} \psi_i(x_{\psi_i})$$



(a)

D	E	$\psi_6(D,E)$
0	0	20.2
0	1	12
1	0	23.4
1	1	11.7

(b)

Figure 2.5: (a) An example 3×3 square Grid Markov network (ising model) and (b) An example potential $H_6(D, E)$

where $Z = \sum_x \prod_i \psi_i(x_{\psi_i})$ is the partition function. □

Markov networks typicall are generated by starting with a graph model which describes the variables of interestand how they depend on each other, like in the case of image analysis whose graph is a grid. Then the user defines potential functions on the cliques of the graph. A well known example is the *ising model*. This model arize from statistical physics [**?**]. It was used to model the behavior of magnets. The structure is a grid, where the variables have values $\{-1, +1\}$. The potential express the desire to have neighboring variables have the same value. The resulting Markov network is called a Markov random field. Alternatively, like in the case of constraint networks, if the potential functions are specified with no explicit reference to a graph (perhaps representing some local probabilistic information or compatibility information) the graph emerges as the associated primal graph.

Markov networks provide some more freedom from the modeling perspective, allowing to express potential functions on any subset of variables. This however comes at the cost of loosing semantic clarity. The meaning of the input local functions relative to the emerging probability distribution is not coherent. In both Bayesian networks and Markov networks the modeling process starts from the graph. In the Bayesian network case the

graph restricts the CPTs to be defined for each node and its parents. In Markov networks, the potentials should be defined on the maximal cliques. For more see [47].

It is sometime convenient to represent potential as positive functions only (even when correlational information can be expressed also by negative numbers). In that case an exponential representation is common. This again, is motivated by work in statistical physics where using the following transformation into what is called energy function.

$$\psi_{S_i}(x_{S_i}) = e^{-E(x_{S_i})}$$

we get that

$$E(x_{S_i}) = -log\psi_i(x_{S_i})$$

and therefore

$$P(x) = \frac{1}{Z}\prod_{i=1}^{m}\psi_i(\mathbf{x}) = \frac{1}{Z}e^{-\sum_i E(x_{S_i})}$$

We see that high probability states correspond to low energy states. Such models. known as **energy-based models**, are common in Physics and Biochemistry and are particularly popular in machine learning.

2.5 Mixed networks

In this section, we introduce the mixed network, a graphical model which allows both probabilistic information and deterministic constraints and which provides a coherent meaning to the combination.

Definition 2.5.1 (mixed networks) *Given a belief network* $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$ *that expresses the joint probability* $P_{\mathcal{B}}$ *and given a constraint network* $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \bowtie \rangle$ *that expresses a set of solutions* ρ*, a mixed network based on* $\mathcal{B}$ *and* $\mathcal{R}$ *denoted* $\mathcal{M}_{(\mathcal{B},\mathcal{R})} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}, \mathbf{C} \rangle$ *is created from the respective components of the constraint network and a Bayesian network as follows: the variables* $\mathbf{X}$ *and their domains are shared, (we could allow non-common variables and take the union), and the functions include the CPTs in* $\mathbf{P}_G$ *and the constraints in* $\mathbf{C}$*. The mixed network expresses the conditional probability*

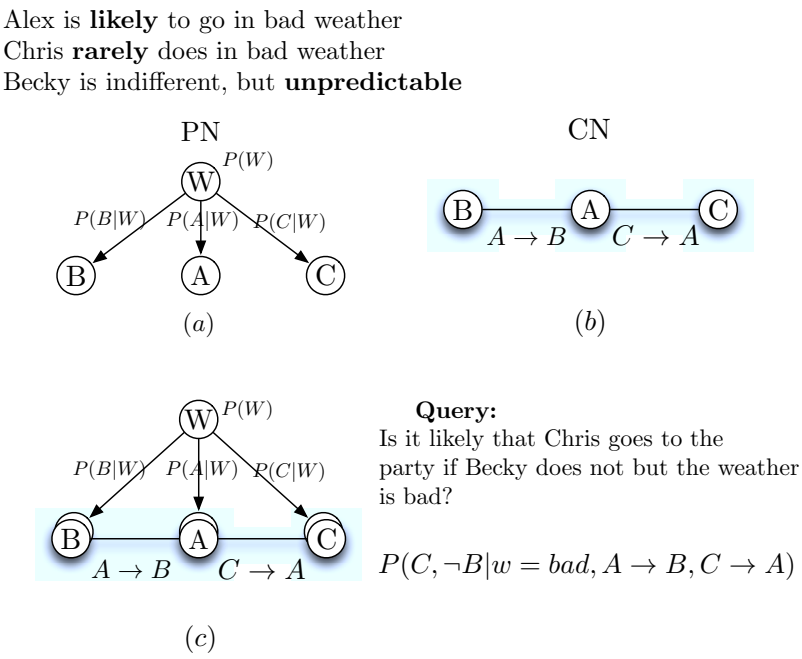


Figure 2.6: The part example, A Bayesian network (a), a constraint formula (b) and a mixed network (c)

$P_{\mathcal{M}}(\mathbf{X})$:

$$P_{\mathcal{M}}(\bar{x}) = \begin{cases} P_{\mathcal{B}}(\bar{x} \mid \bar{x} \in \rho), & if \quad \bar{x} \in \rho \\ 0, & otherwise. \end{cases}$$

Example 2.5.2 Consider a scenario involving social relationship between three individuals Alex (A), Becky (B) and Chris (C). We know that if Alex goes to a party Becky will go, and if Chris goes Alex goes. We also know the weather effects these three individuals differently and they will or will not go to a party with some differing likelihood. We can express the relationship between going to the party and the weather using a Bayesian network, while the social relationship using a propositional formula (see Figure 2.6). □

The mixed network have two types of functions, probabilistic local functions and constraints. This is a graphical model whose combination operator is product, when we assume that constraints have their cost-based representation.

Belief updating, MPE and MAP queries over probabilistic networks can be extended to mixed networks straight-forwardly. They are well defined relative to the mixed probability distribution $P_{\mathcal{M}}$. Since $P_{\mathcal{M}}$ is not well defined for *inconsistent* constraint networks we always assume that the constraint network portion is consistent.

Mixed networks give rise to a new query, which is to find the probability of a consistent tuple; namely, we want to determine $P_{\mathcal{B}}(\bar{x} \in sol(\mathcal{R}))$. We will call this *a Constraint Probability Evaluation (CPE)*. Note that evidence is a special type of constraint. We will elaborate on this next.

The problem of evaluating the probability of CNF queries over Bayesian networks has various applications. One example is network reliability: Given a communication graph with a source and a destination, one seeks to diagnose the failure of communication. Since several paths may be available between source and destination, the failure condition can be described by a CNF formula as follows. Failure means that for all paths (conjunctions) there is a link on that path (disjunction) that fails. Given a probabilistic fault model of the network, the task is to assess the probability of a failure [48]. There are many examples in modeling travel patterns of human and in natural language processing. [**?**, **?**, **?**]. [**comment: either have citations or remove this last sentence**]

Definition 2.5.3 (queries on mixed networks) *We consider the following 2 new queries:*

- *Given a mixed network $\mathcal{M}_{(\mathcal{B},\mathcal{R})}$, where $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{G}, \mathbf{P} \rangle$ and $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ the* constraint*, Probability Evaluation (CPE) task is to find the probability $P_{\mathcal{B}}(\bar{x} \in sol(\mathcal{R}))$. If R is a CNF expression φ, the cnf probability evaluation seeks $P_{\mathcal{B}}(\bar{x} \in m(\varphi))$, where $m(\varphi)$ are the models (solutions of φ).*

- Belief assessment of a constraint or on a CNF expression *is the task of assessing $P_{\mathcal{B}}(X|\varphi)$ for every variable X. Since $P(X_i|\varphi) = \alpha \cdot P(X_i \wedge \varphi)$ where α is a normalizing constant relative to X, computing $P_{\mathcal{B}}(X|\varphi)$ reduces to a CPE task over $\mathcal{B}$ for the query $((X_i = x_i) \wedge \varphi)$. In other words we want to compute $P_B(\bar{x}|X_i = x_i, \bar{x} \in m(\varphi))$. More generally, $P_{\mathcal{B}}(\varphi|\psi) = \alpha_{\varphi} \cdot P_{\mathcal{B}}(\varphi \wedge \psi)$ where α_{φ} is a normalization constant relative to all the models of φ.*

We conclude with some common alternative formulations of queries over mixed networks which contain both constraints and probabilistic functions.

Definition 2.5.4 (The Weighted Counting Task) *Given a mixed network $\mathcal{M} = \langle X, D, P_G, C \rangle$, where $P_G = \{P_1, ..., P_m\}$ the weighted counting task is to compute the normalization constant given by:*

$$Z = \sum_{\mathbf{x} \in Sol(\mathbf{C})} \prod_{i=1}^{m} P_i(\mathbf{x}_{\mathbf{S_i}}) \tag{2.1}$$

where $sol(\boldsymbol{C})$ is the set of solutions of the constraint portion $\boldsymbol{C}$. Equivalently, if we have a cost-based representation of the constraints in C as $0/1$ functions, we can rewrite Z as:

$$Z = \sum_{\mathbf{x} \in \mathbf{X}} \prod_{i=1}^{m} P_i(\mathbf{x}_{\mathbf{S_i}}) \prod_{j=1}^{p} C_j(\mathbf{x}) \tag{2.2}$$

We will refer to Z as **weighted counts** and we can see that mathematically, it is identical to the partition function.

Definition 2.5.5 (Marginal task) *Given a mixed network $\mathcal{M} = \langle X, D, F, C \rangle$, the marginal task is to compute the marginal distribution at each variable. Namely, for each variable X_i and $x_i \in D_i$, compute:*

$$P(x_i) = \sum_{\mathbf{x} \in \mathbf{X}} \delta_{x_i}(\mathbf{x}) P_{\mathcal{M}}(\mathbf{x}), \text{ where } \delta_{x_i}(\mathbf{x}) = \begin{cases} 1 & \text{if } X_i \text{ is assigned the value } x_i \text{ in } \boldsymbol{x} \\ 0 & \text{otherwise} \end{cases}$$

When we are given a probabilistic network that has zeros, we can extract a constraint portion from it, generating an explicit mixed network as we show below.

Definition 2.5.6 (Modified Mixed network) *Given a mixed network $\mathcal{M} = \langle \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{F}, \boldsymbol{C} \rangle$, a modified mixed network is a four-tuple $\mathcal{M}' = \langle \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{F}, \boldsymbol{C}' \rangle$ where $\boldsymbol{C}' = \boldsymbol{C} \cup \{FC_i\}_{i=1}^{m}$ where*

$$FC_i(\mathbf{S}_i = \mathbf{s}_i) = \begin{cases} 0 & \text{if } F_i(\boldsymbol{s}_i) = 0 \\ 1 & \text{Otherwise} \end{cases} \tag{2.3}$$

FC_i can be expressed as a relation. It is sometimes called the flat constraints of the probability function.

Clearly, the modified mixed network M' and the original mixed network M are equivalent in that $P_{\mathcal{M}'} = P_{\mathcal{M}}$.

It is easy to see that the weighted counts over a mixed network specialize to (a) the probability of evidence in a Bayesian network, (b) the partition function in a Markov network and (c) the number of solutions of a constraint network. The marginal problem can express the posterior marginals in a Bayesian or Markov network.

2.5.1 Mixed Markov and Bayesian networks

2.5.2 Mixed cost networks

We often have combinatorial optimization tasks that distinguish between local cost functions and local constraints. While constraints can be expressed as cost functions, maintaining them seperate can invite specific constraint processing algorithms that would improve performance.

Definition 2.5.7 (mixed cost and constraint networks) *Given...*

A classical example is integer linear programs.

2.6 Summary and bibliographical comments

The work on graphical models can be seen as originating from two communities. The one that centers on statistics, probabilities and aim at capturing probability distributions vs the one that centers on deterministic relationships, such as constraint networks and logic systems. Each represent an extreme point in a spectrum of models. Each went through the process of generalization and extensions towards the other; probablistic models were augmented with constraint processing and utility information (e.g., leading to influence diagrams), and constraint networks were extended to soft constraints and into fuzzy type information.

The seminal work by Bistareli, Rossi and Montanari [12] titled semiring-based constraint satisfaction and optimization and the whole line of work that followed provides a foundational unifying treatment of graphical models, using the mathematical framework of semirings. Varoius semirings yield different graphical models, using the umbrealla name Soft Constraints. The work emerged from and generalizes the area of constraint

networks. Constraint networks were distinguished as semirings that are *idempotent*. For a complete treatment see [11]. Another line of work rooted at probabilistic networks was introduced by Shenoy and Shafer providing an axiomatic treatment for probability and belief-function propagation [56, 57]. Their framework is focued on an axiomatic formulation of the two operators of combination and marginalization in graphical models. The work by Dechter [19, 21] focusing on unifying variable elimination algorithms across constraint networks, cost networks and probabilistic networks demonstrate that common algorithms can be applied across all these graphical models such as constraints networks, cost-networks, propositional cnfs, influence diagrams and probabilistic networks. can be expressed using also the two operation of combination and marginalization [37]. This work is the basis of the exposition in this book. Other work that observe the applicability of common message-passing algorithms over certain restricted graphs beyond probabilistic networks only, or constraint networks only is the work by Srinivas and McEliece [3] titled the generalized distributive law.

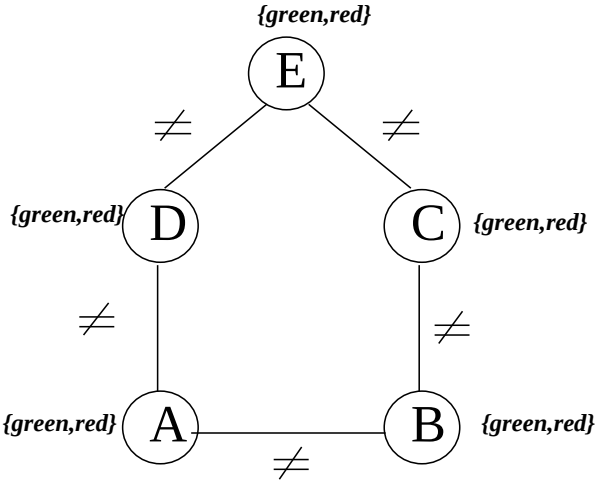
Chapter 3

Bucket-Elimination for Deterministic Networks

This chapter is the first of three chapters in which we introduce the *bucket-elimination* inference scheme. This scheme characterizes all inference algorithms over graphical models, where by *inference* we mean algorithms that solve queries by inducing equivalent model representations according to some set of inference rules. These are sometimes called *reparameterization schemes* because they generate an equivalent specification of the problem from which answers can be produced easily. We will see that the bucket elimination scheme is applicable to most, if not all, of the types of queries and graphical models we discussed in Chapter 2, but its general structure and properties are most readily understood in the context of constraint networks. Therefore, this chapter introduces bucket elimination in its application to constraint networks. In the following chapter, we will apply this scheme to probabilistic reasoning and combinatorial optimization.

Bucket-elimination algorithms are *knowledge-compilation* methods: they generate an equivalent representation of the input problem from which various queries are answerable in polynomial time. In this chapter, the target query is whether or not an input constraint network is consistent.

To illustrate the basic idea behind bucket elimination, let's walk through a simple constraints problem. Consider the graph coloring problem in Figure **??**. The task is to assign one of two colors (green or red) to each node in the graph so that adjacent nodes



(a)

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

(b)

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

(c)

Figure 3.1: A graph coloring example (a) and a schematic execution of adaptive-consistency (b,c)

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 inferred, and we can add this as a new constraint in our network without changing its solutions set. We can ignore variable E from now on since we already summarized his impact on the rest of the problem when we added $C = D$. We focus on variable C next. Together, the inferred constraint $C = D$ and the input constraint $C \neq B$ imply that $D \neq B$, and we add this constraint to the problem. Having taken into account the effect of C on the other variables in the network, we can ignore C also from now on. Continuing in this fashion with node D, we infer $A = B$. However, since there is an input constraint $A \neq B$ we have reached a contradiction and can conclude that the original set of constraints is inconsistent.

The algorithm which we just executed, is known as *adaptive-consistency* in the constraint literature [24] and it can solve any constraint satisfaction problem. The algorithm works by processing and *eliminating* variables one by one, while deducing the effect of the

eliminated variable on the rest of the problem. The elimination operation first *joins* all the relations that are defined on the current variable and then projects out the variable. Adaptive-consistency can be described using a data structure called *buckets* as follows: given an ordering of the variables, we process the variables from last to first. In the previous example, the ordering was $d = A, B, D, C, E$, and we processed the variables from E to A. Note that we will use this convention throughout: we assume that the inference algorithm process the variables from last to first w.r.t to a given ordering. The reason for that will be clear later. The first step is to partition the constraints into *ordered buckets*, so that the bucket for the current variable contains all constraints that mention the current variable and that have not been placed in a previous bucket. In our example, 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 $bucket_D$, and so on. The initial partitioning of the constraints is depicted in Figure 3.1a. The general partition rule given an ordering is that each constraint identifies the variable L in its scope that appears latest in the ordering, and then places the constraint in $bucket_L$.

After this initialization step, the buckets are processed from last to first. Processing a bucket means solving the subproblem defined by the constraints in the bucket and then inferring the constraint that is imposed by that subproblem on the rest of the variables excluding the bucket's variable. In other words, we compute the constraint that the bucket-variable induces on the variables that precede it in the ordering. As we saw, 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 between the inferred $A \neq B$ and the input constraint $A = B$. The buckets' final contents are shown in Figure 3.1b. The new inferred constraints are displayed to the right of the bar in each bucket.

Observe that at each step, one variable and all its related constraints are, in fact, solved, and a new constraint is inferred on all of the rest of the participating variables. Observe also that because the new added constraints are inferred, the problem itself does not change and with or without the added constraints, it has the same set of solutions.

However, what is significant is that once all the buckets are processed, and if no inconsistencies were discovered, a solution can be generated in a so called *backtrack-free* manner. This means that a solution can be assembled by assigning values to the variables progressively, starting with the first variable in ordering d and this process is guaranteed to continue until all the variables are assigned a value from their respective domains, thus yielding a *solution* to the problem. The notion of *backtrack-free* constraint network relative to an ordering is central to the theory of constraint processing and will be defined shortly.

The bucket-elimination algorithm illustrated above for constraints is applicable to general graphical models as we will show. The algorithm is applied to the given model (e.g., a Bayesian network or a cost network) for the given particular query.

3.1 The case of Constraint Networks

We have presented an informal definition of the bucket elimination algorithm on constraint networks called adaptive-consistency. Here we will provide a formal definition of the algorithm, using the formalism of constraint networks introduced in the previous chapter and utilizing the the following operations:

Definition 3.1.1 (operations on constraints: select, project, join) *Let R be a relation on a set S of variables, let $Y \subseteq S$ be a subset of the variables, and let $Y = y$ (or y) be an instantiation of the variables in Y. We denote by $\sigma_y(R)$ the selection of those tuples in R that agree with $Y = y$. We denote by $\Pi_Y(R)$ the projection of the relation R on the subset Y, that is, a tuple $Y = y$ 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 $t_{S_2} \in R_{S_2}$.*

Using the above operations, adaptive-consistency can be specified as in Figure 3.2. In step 1 the algorithm partitions the constraints into buckets whose structure depends on the variable ordering used. The main bucket operation is given in steps 4 and 5.

ADAPTIVE-CONSISTENCY (AC)

Input: a constraint network $\mathcal{R} = (\mathcal{X}, \mathcal{D}, \mathcal{R})$, an ordering $d = (x_1, \ldots, x_n)$

output: A backtrack-free network, denoted $E_d(\mathcal{R})$, along d, if the empty constraint was not generated. Else, the problem is inconsistent

1. Partition constraints into $bucket_1, \ldots, bucket_n$ as follows:
   for $i \leftarrow n$ **downto** 1, put in $bucket_i$ all unplaced constraints mentioning x_i.
2. **for** $p \leftarrow n$ **downto** 1 **do**
3. **for** all the constraints $R_{S_1}, \ldots, R_{S_j}$ in $bucket_p$ **do**
4. $A \leftarrow \bigcup_{i=1}^{j} S_i - \{x_p\}$
5. $R_A \leftarrow \Pi_A(\bowtie_{i=1}^{j} R_{S_i})$
6. **if** R_A is not the empty relation **then** add R_A to the bucket of the latest variable in scope A,
7. **else** exit and return the empty network
8. **return** $E_d(\mathcal{R}) = (X, D, bucket_1 \cup bucket_2 \cup \cdots \cup bucket_n)$

Figure 3.2: Adaptive-Consistency as a bucket-elimination algorithm

Algorithm adaptive-consistency specifies that it returns a "backtrack-free" network along the ordering d. This concept is related to the search approach that is common for solving constraint satisfaction, and in particular, to backtracking search (for more see Chapter **??**). Backtracking search assign values to the variables in a certain order in a depth-first manner, checking the relevant constraints, until an assignment is made to all the variables or a *dead-end* is reached where no consistent values exist. If a dead-end is reached, search will *backtrack* to a previous variable, change its value, and proceed again along the ordering. We say that a constraint network is *backtrack-free* along an ordering d of its variables if it is guaranteed that a dead-end will never be encountered by backtrack search.

We next define the notion of backtrack-free network. It is based on the notion of a *partial solution.*

Definition 3.1.2 (partial solution) *Given a constraint network $\mathcal{R}$, we say that an assignment of values to a subset of the variables $S = \{X_1, ..., X_j\}$ given by $\bar{a} = (< X_1, a_1 > , < X_2, a_2 >, ..., < X_j, a_j >)$ is consistent relative to $\mathcal{R}$ iff it satisfies every constraint whose scope is subsumed in S. The assignment $\bar{a}$ is also called a partial solution of $\mathcal{R}$.*

Definition 3.1.3 (backtrack-free search) *A constraint network is backtrack-free relative to a given ordering $d = (X_1, ..., X_n)$ if for every $i \leq n$, every partial solution over $(X_1, ..., X_i)$ can be consistently extended to include X_{i+1}.*

We are now ready to state the main property of adaptive-conistency.

Theorem 3.1.4 (Correctness and Completeness of Adapative-consistency) *[24] Given a set of constraints and an ordering of variables, adaptive-consistency decides if a set of constraints is consistent and, if it is, the algorithm always generates an equivalent representation that is backtrack-free along the input variable ordering.* □

Proof: See exercises ∎

Example 3.1.5 Consider the graph coloring problem depicted in Figure **??** (modified from example **??** with colors represented by numbers). The figure shows a schematic

execution of adaptive-consistency using the bucket data structure for the two orderings $d_1 = (E, B, C, D, A)$ and $d_2 = (A, B, D, C, E)$. The initial constraints, partitioned into buckets for both orderings, are displayed in the figure to the left of the double bars, while the constraints generated by the algorithm are displayed to the right of the double bar, in their respective buckets. Let's focus first on ordering d_2: as shown in 3.4, adaptive-consistency proceeds from E to A and imposes constraints on the parents of each processed variable, which are those variables appearing in its bucket. To process $bucket_E$ all three constraints in the buckets are solved and the set of solutions is projected over D, C, and B, recording the ternary constraint R_{DCB} which is placed in $bucket_C$ (see Figure 3.4 for details). Next, the algorithm processes $bucket_C$ which contains $C \neq A$ and the new constraint R_{DCB}. Joining these two constraints and projecting *out* C yields a constraint R_{DB} that is placed in the bucket of D, and so on. In our case $R_{DB} \equiv D = B$ and its generation is depicted in two steps of Figure 3.4.

Notice that adaptive-consistency along ordering d_1 generates a different set of constraints, and in particular it generates only binary constraints, while along ordering d_2 the algorithm generates a ternary constraint. Notice also that for the ordering d_1, the constraint $B \neq E$ generated in the bucket of D is displayed for illustration only in the bucket of B (in parentheses), since there is already an identical original constraint. Indeed the constraint is redundant. □

Example 3.1.6 An alternative graphical illustration of the algorithm's performance using d_2 is given in Figure 3.4. The figure shows, through the changing graph, how constraints are generated in the reverse order of $d_2 = A, B, D, C, E$, and how a solution is created in the forward order of d_2. The first step is processing the constraints that mention variables E. These are all joined to create a relation over $EDBC$ and then E is projected out, yielding a constraint on DBC whose relation is explicitly given in the figure. The relation is added to the set of constraints and is depicted as an added clique over the 3 nodes. Then E is removed, yielding the 3rd graph that includes only nodes A, D, B, C. The next variable to be processed is C. the constraints that include C are the original constraint $B \neq C$ and the new constraint over DBC. Joining both yields the new constraint on DBC as depicted, and projecting out C from this relation yields a constraint on DB whose meaning is the equality constraint. variable D is eliminated

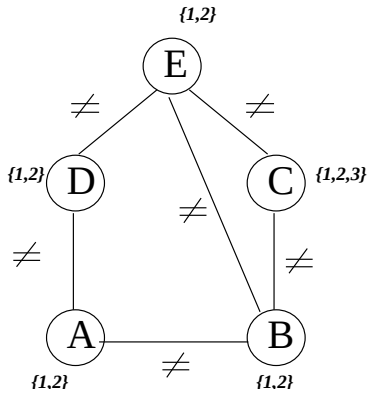
next and then B, yielding the last variable A with two values $\{1, 2\}$.

Subsequently, the reverse process of generating a solution starts at A. Since it has 2 legitimate values, we can select any of those. The value $A = 1$ is selected. The next value satisfying the inequality constraint is $B = 2$, then $D = 2$ (satisfying $D = B$), then $C = 3$ (satisfying $C \neq B$). To assign a value for E we look at the constraint on $EDBC$ which only allows $E = 1$ to extend the current partial solution $A = 1, B = 2, D = 2, C = 3$. yielding a full solution. □

What is the complexity of adaptive-consistency? It is clearly linear in the number of buckets and the time to process each bucket. However, since processing a bucket amounts to solving a constraint-satisfaction subproblem (generating the join of all relations) its complexity is exponential in the number of variables mentioned in a bucket. Conveniently, the number of variables appearing in a bucket along a given ordering, can be obtained using the *induced-width* of the graph along that ordering. The induced-width is an important graph parameter that is instrumental to all bucket-elimination algorithms, and we define it next.

Definition 3.1.7 (Induced graph and induced width) *Given an undirected graph $G = (V, E)$, where $V = \{v_1, ..., v_n\}$ is the set of nodes, E is a set of arcs over V, an* ordered graph *is a pair (G, d), where $d = (v_1, ..., v_n)$ is an ordering of the nodes. The nodes adjacent to v that precede it in the ordering are called its* parents. *The* induced graph *of an ordered graph (G, d) is an ordered graph (G^*, d) where G^* is obtained from G as follows: the nodes of G are processed from last to first (top to bottom) along d. When a node v is processed, all of its parents are connected. The* induced width of an ordered graph, *(G, d), denoted $w^*(d)$, is the maximum number of parents a node has in the the induced ordered graph (G^*, d). The* induced width of a graph, *w^*, is the minimal induced width over all its orderings.*

Example 3.1.8 Generating the induced-graph for $d_1 = E, B, C, D, A$ and $d_2 = A, B, D, C, E$ leads to the two graphs in Figure 3.5. The broken lines are the newly added arcs. The induced width along d_1 and d_2 are 2 and 3 respectively. They suggest different performance bounds for adaptive-consistency because the number of variables in a bucket is



(a)

Ordering d_1

$Bucket(A)$: $A \neq D$, $A \neq B$
$Bucket(D)$: $D \neq E$ || R_{DB}
$Bucket(C)$: $C \neq B$ $C \neq E$
$Bucket(B)$: $B \neq E$ || R^1_{BE}, R^2_{BE}
$Bucket(E)$: || R_E

Ordering d_2
$Bucket(E)$: $E \neq D$, $E \neq C$, $E \neq B$
$Bucket(c)$: $C \neq B$ || R_{DCB}
$Bucket(D)$: $D \neq A$ || $R_{DB}(= D = B)$
$Bucket(B)$: $B \neq A$ || $R_{AB}(= R \neq B)$
$Bucket(A)$: || R_A

Figure 3.3: A modified graph coloring problem

bounded by the number of parents of the corresponding variable in the induced ordered graph which is equal to its induced-width. □

Theorem 3.1.9 *The time and space complexity of Adaptive-Consistency is* $O((r+n)k^{w^*(d)+1})$ *and* $O(n \cdot k^{w^*(d)})$ *respectively, where* n *is the number of variables,* k *is the maximum domain size and* $w^*(d)$ *is the induced-width along the order of processing* d *and* r *is the number of the problems' constraints.*

Proof: Since the total number of input functions plus those generated is bounded by $r + n$ and since the computation in a bucket is $O(r_i k^{w^*(d)+1})$, where r_i is the number of functions in bucket i, the total over all buckets is $O((r + n)k^{w^*(d)+1})$ □

The above analysis suggests that problems having bounded induced width $w^* \leq b$ for some constant b can be solved in polynomial time. In particular, observe that when the graph is cycle-free its width and induced width are 1. Consider, for example, ordering $d = (A, B, C, D, E, F, G)$ for the tree in Figure 3.6. As demonstrated by the schematic execution along d in Figure 3.6, adaptive-consistency generates only unary relationships

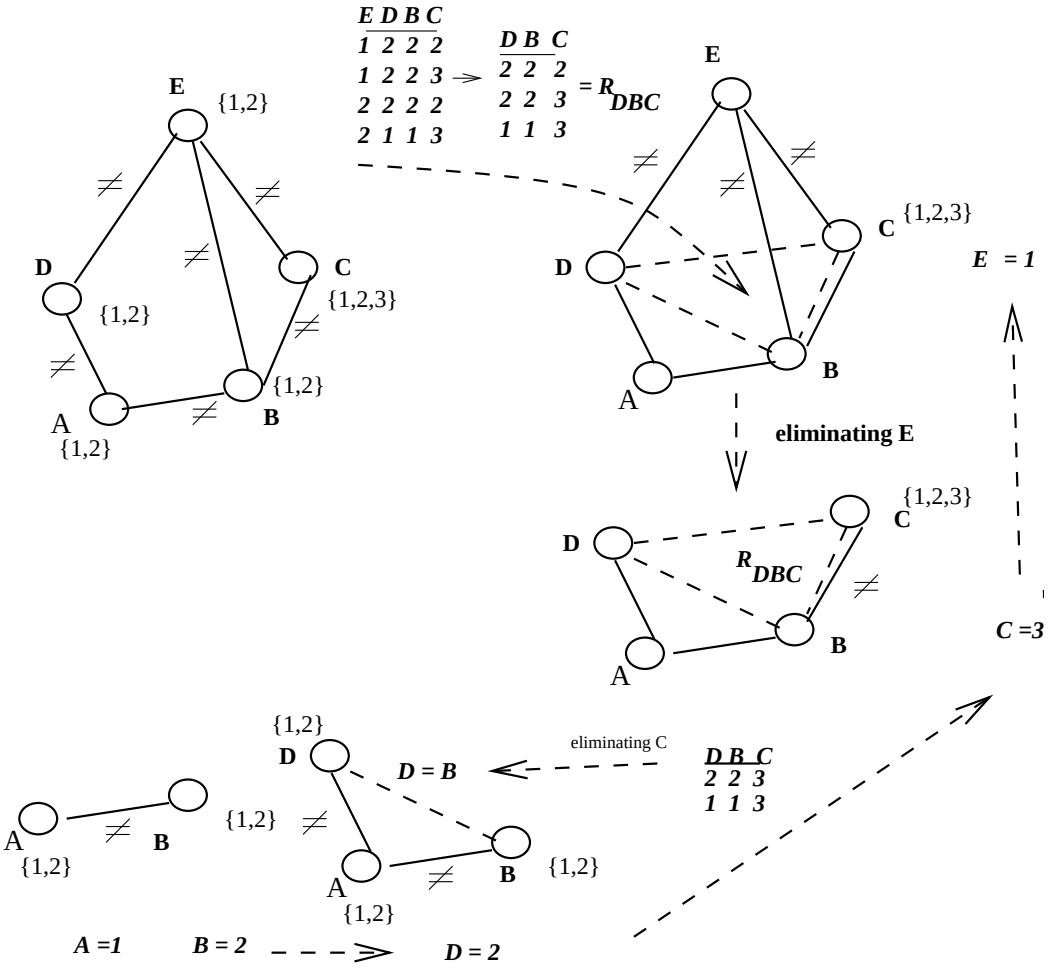


Figure 3.4: A schematic variable-elimination and solution genration process is backtrack-free (comment: change the order of d_2)

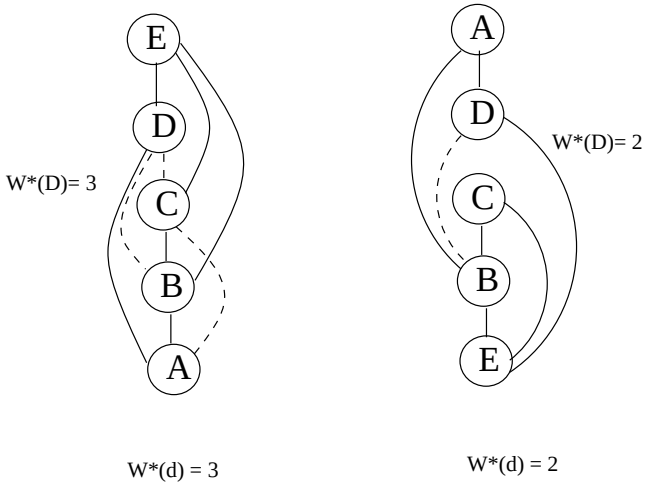


Figure 3.5: The induced width along the orderings: $d_1 = A, B, C, D, E$ and $d_2 = E, B, C, D, A$

in this cycle-free graph.We note that on trees the algorithm can be accomplished in a distributed manner as a message passing algorithm which converges to exact solution. We will come back to this point in a later chapter.

3.2 Bucket elimination for Propositional CNFs

Since propositional CNF formulas, discussed in Chapter 2, are a special case of constraint networks, we might wonder what adaptive consistency looks like when applied to them.

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., $P =$"true" or $P =$"false") by P and $\neg P$ 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

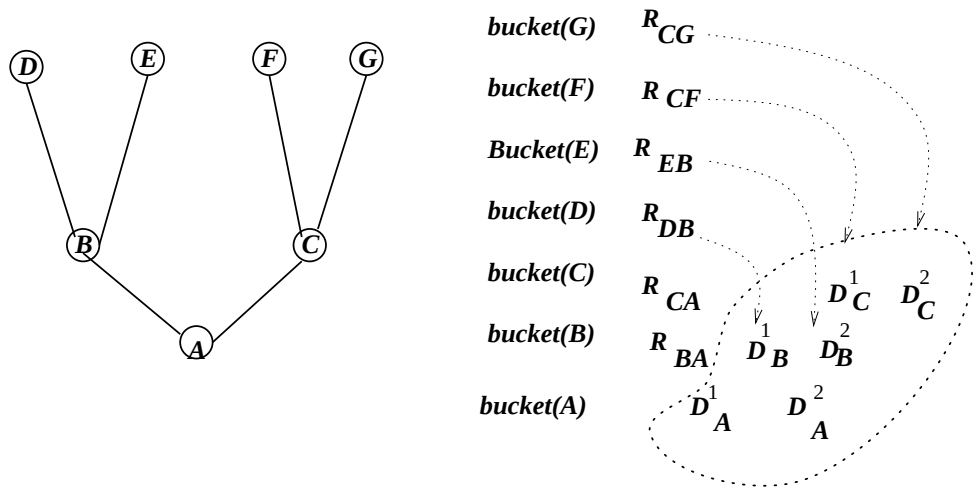


Figure 3.6: Schematic execution of adaptive-consistency on a tree network. D_X denotes unary constraints over X

.

that do not violate any clause in φ. Deciding if a formula is satisfiable is known to be NP-complete [33].

It turns out 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 [27].

Definition 3.2.1 (extended composition) *The extended composition of relation R_{S_1}, ..., R_{S_m} relative to a subset of variables $A \subseteq \bigcup_{i=1}^{m} S_i$, denoted $EC_A(R_{S_1}, \ldots, R_{S_m})$, is defined by*

$$EC_A(R_{S_1}, \ldots, R_{S_m}) = \pi_A(\bowtie_{i=1}^{m} R_{S_i})$$

When the operation of extended composition is applied to m relations, it is called extended m-composition. *If the projection operation is restricted to subsets of size i, it is called* extended (i, m)-composition.

It is not hard to see that extended composition is the operation applied in each bucket by adaptive-consistency. We next show that the notion of resolution is equivalent to extended 2-composition.

Example 3.2.2 Consider the two clauses $\alpha = (P \vee \neg Q \vee \neg O)$ and $\beta = (Q \vee \neg W)$. Now let the relation $R_{PQO} = \{000, 100, 010, 001, 110, 101, 111\}$ be the models of α and the relation $R_{QW} = \{00, 10, 11\}$ be the models of β. Resolving these two clauses over Q generates the resolvent clause $\gamma = res(\alpha, \beta) = (P \vee \neg O \vee \neg W)$. The models of γ are $\{(000, 100, 010, 001, 110, 101, 111\}$. It is easy to see that $EC_{PQW}(R_{PQO}, R_{QW}) = \pi_{RQW}(R_{PQO} \bowtie R_{Qw})$ yields the models of γ. □

Indeed,

Lemma 3.2.3 *The* resolution *operation over two clauses, $(\alpha \vee Q)$ and $(\beta \vee \neg Q)$, results in a clause $(\alpha \vee \beta)$ for which $models(\alpha \vee \beta) = EC_{Q'}(models(\alpha \vee Q), models(\beta \vee \neg Q))$, where Q' is the union of scopes of both clauses excluding Q.* □

Substituting extended decomposition by resolution in adaptive consistency yields a bucket-elimination algorithm for propositional satisfiability which we call *directional resolution (DR).*

We call the output theory (i.e., formula) of directional resolution, denoted $E_d(\varphi)$, the *directional extension* of φ. The following description of the algorithm should look familiar. 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 $bucket_i$. The algorithm processes the buckets in the reverse order of d. Processing of $bucket_i$ means resolving over Q_i all the possible pairs of clauses in the bucket and inserting the resolvents into appropriate lower buckets.

Note that if the bucket contains a unit clause (Q_i or $\neg Q_i$), only unit resolutions are performed. As implied by Theorem 3.1.9, DR is guaranteed to generate a backtrack-free representation along the order of processing. Indeed, as already observed in the above example, once all the buckets are processed, and if the empty clause was not generated, a truth assignment (model) can be assembled in a backtrack-free manner by consulting $E_d(\varphi)$, using the order d.

Theorem 3.2.4 (backtrack-free by DR) *Given a theory φ and an ordering of its variables d, the directional extension $E_d(\varphi)$ generated by DR is backtrack-free along d*

DIRECTIONAL-RESOLUTION (DR)

Input: A *CNF* theory φ, an ordering $d = Q_1, \ldots, Q_n$ of its variables.

Output: A decision of whether φ is satisfiable. If it is, a theory $E_d(\varphi)$, equivalent to φ, else an empty directional extension.

1. **Initialize:** generate an ordered partition of clauses into buckets $bucket_1, \ldots, bucket_n$, where $bucket_i$ contains all clauses whose highest variable is Q_i.
2. **for** $i \leftarrow n$ **downto** 1 process $bucket_i$:
3. **if** there is a unit clause **then** (the instantiation step)
   apply unit-resolution in $bucket_i$ and place the resolvents in their right buckets.
   if the empty clause was generated, theory is not satisfiable.
4. **else** resolve each pair $\{(\alpha \vee Q_i), (\beta \vee \neg Q_i)\} \subseteq bucket_i$.
   if $\gamma = \alpha \vee \beta$ is empty, return $E_d(\varphi) = \{\}$, the theory is not satisfiable
   else determine the index of γ and add it to the appropriate bucket.
5. **return** $E_d(\varphi) \leftarrow \bigcup_i bucket_i$

Figure 3.7: Directional-resolution

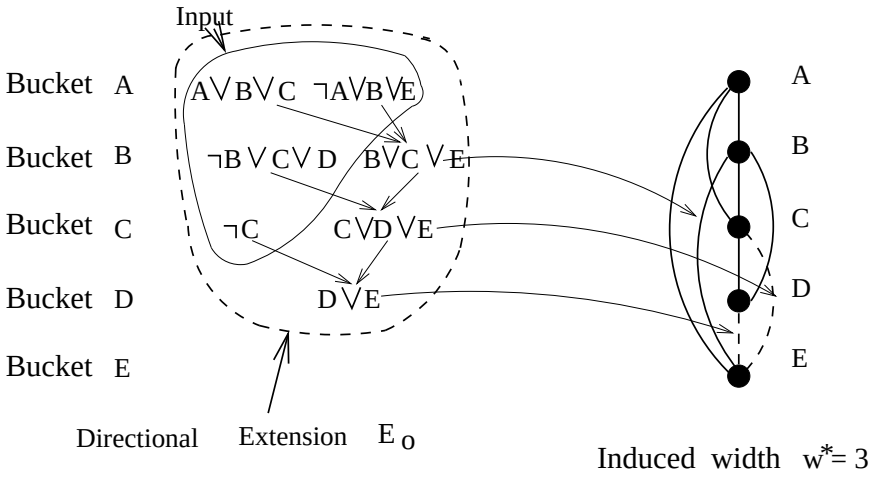


Figure 3.8: A schematic execution of directional resolution using ordering $d = (E, D, C, B, A)$

Example 3.2.5 Given the input theory $\varphi_1 = \{(\neg C), (A \vee B \vee C), (\neg A \vee B \vee E), (\neg B \vee C \vee D)\}$, and an ordering $d = (E, D, C, B, A)$, the theory is partitioned into buckets and processed by directional resolution in reverse order. Resolving over variable A produces a new clause $(B \vee C \vee E)$, which is placed in $bucket_B$. Resolving over B then produces clause $(C \vee D \vee E)$, which is placed in $bucket_C$. Finally, resolving over C produces clause $(D \vee E)$, which is placed in $bucket_D$. Directional resolution now terminates, since no resolution can be performed in $bucket_D$ and $bucket_E$. The output is a non-empty directional extension $E_d(\varphi_1)$. Once the directional extension is available, model generation can begin. There are no clauses in the bucket of E, the first variable in the ordering, and therefore E can also be assigned any value (e.g., $E = 0$). Given $E = 0$, the clause $(D \vee E)$ in $bucket_D$ implies $D = 1$, clause $\neg C$ in $bucket_C$ implies $C = 0$, and clause $(B \vee C \vee E)$ in $bucket_B$, together with the current assignments to C and E, implies $B = 1$. Finally, A can be assigned any value since both clauses in its bucket are satisfied by previous assignments. The initial partitioning into buckets along the ordering d as well as the buckets' contents generated by the algorithm following resolution over each bucket are depicted in Figure 3.8. □

Not surprisingly, the complexity of directional-resolution is exponentially bounded (time and space) in the *induced width* of the theory's interaction graph along the order

of processing. Notice that the graph of theory φ_1 along the ordering d (depicted also in Figure 3.8) has an induced width of 3.

Lemma 3.2.6 *Given a theory φ and an ordering $d = (Q_1, ..., Q_n)$, if Q_i has at most w parents in the induced graph along d, then the bucket of Q_i in $E_d(\varphi)$ contains no more than 3^{w+1} clauses.*

Proof: Given a clause α in the bucket of Q_i, there are three possibilities for each parent P of Q_i: either P appears in α, $\neg P$ appears in α, or neither of them appears in α. Since Q_i also appears in α, either positively or negatively, the number of possible clauses in a bucket is no more than $2 \cdot 3^w < 3^{w+1}$. ∎

Since the number of parents of each variable is bounded by the induced-width along the order of processing we get:

Theorem 3.2.7 *(complexity of DR)*
Given a theory φ and an ordering of its variables d, the time complexity of algorithm DR along d is $O(n \cdot 9^{w^*_d})$, and $E_d(\varphi)$ contains at most $n \cdot 3^{w^*_d+1}$ clauses, where w^*_d is the induced width of φ's interaction graph along d. □

3.3 Bucket elimination for linear inequalities

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

Definition 3.3.1 (Linear elimination) *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.*

DIRECTIONAL-LINEAR-ELIMINATION (φ, d)

Input: A *set of linear inequalities* φ, an ordering $d = x_1, \ldots, x_n$.

Output: A decision of whether φ is satisfiable. If it is, a backtrack-free theory $E_d(\varphi)$.

1. **Initialize:** Partition inequalities into ordered buckets.
2. **for** $i \leftarrow n$ **downto** 1 **do**
3. **if** x_i has one value in its domain **then**
. substitute the value into each inequality in the bucket and put the resulting inequality in the right bucket.
4. **else,for each** pair $\{\alpha, \beta\} \subseteq bucket_i$, compute $\gamma = elim_i(\alpha, \beta)$
if γ has no solutions, return $E_d(\varphi) = \{\}$, "inconsistency"
else add γ to the appropriate lower bucket.
5. **return** $E_d(\varphi) \leftarrow \bigcup_i bucket_i$

Figure 3.9: Fourier Elimination; DLE

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 *Directional Linear Elimination* (abbreviated DLE), which is the well known Fourier elimination algorithm. (see [40]).

As in the case of propositional theories, 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 is summarized in Figure 3.9.

Theorem 3.3.2 *Given a set of linear inequalities φ, algorithm DLE (Fourier elimination) decides the consistency of φ over the Rationals and the Reals, and it generates an equivalent backtrack-free representation.* □

Example 3.3.3 Consider the set of inequalities over the Reals:

$bucket_4$: $5x_4 + 3x_2 - x_1 \leq 5,\ x_4 + x_1 \leq 2,\ -x_4 \leq 0,$
$bucket_3$: $x_3 \leq 5,\ x_1 + x_2 - x_3 \leq -10$
$bucket_2$: $x_1 + 2x_2 \leq 0.$
$bucket_1$:

Figure 3.10: initial buckets

$bucket_4$: $5x_4 + 3x_2 - x_1 \leq 5,\ x_4 + x_1 \leq 2,\ -x_4 \leq 0,$
$bucket_3$: $x_3 \leq 5,\ x_1 + x_2 - x_3 \leq -10$
$bucket_2$: $x_1 + 2x_2 \leq 0 \mid\mid 3x_2 - x_1 \leq 5, x_1 + x_2 \leq -5$
$bucket_1$: $\mid\mid x_1 \leq 2.$

Figure 3.11: final buckets

$$\varphi(x_1, x_2, x_3, x_4) = \{(1)\ 5x_4 + 3x_2 - x_1 \leq 5,\ (2)\ x_4 + x_1 \leq 2,\ (3)\ -x_4 \leq 0,$$
$$(4)\ x_3 \leq 5,\ (5)\ x_1 + x_2 - x_3 \leq -10,\ (6)\ x_1 + 2x_2 \leq 0\}.$$

The initial partitioning into buckets is shown in Figure 3.10. Processing $bucket_4$, which involves applying elimination relative to x_4 over inequalities $\{(1),(3)\}$ and $\{(2),(3)\}$, respectively, results in $3x_2 - x_1 \leq 5$, placed into $bucket_2$, and $x_1 \leq 2$, placed into $bucket_1$. Next, processing the two inequalities $x_3 \leq 5$, and $x_1 + x_2 - x_3 \leq -10$ in $bucket_3$ eliminates x_3, yielding $x_1 + x_2 \leq -5$ placed into $bucket_2$. When processing $bucket_2$, containing $x_1 + 2x_2 \leq 0$, $3x_2 - x_1 \leq 5$, and $x_1 + x_2 \leq -5$, no new inequalities are added. The final set of buckets is displayed in Figure 3.11. □

Once the algorithm is applied, we can generate a solution in a backtrack-free manner as usual. Select a value for x_1 from its domains that satisfies the unary inequalities in $bucket_1$. From there on, after selecting assignments for $x_1, \ldots, x_{i-1}$, select a value for x_i that satisfies all the inequalities in $bucket_i$. This is an easy task, since all the constraints are unary once the values of $x_1, \ldots, x_{i-1}$ are determined.

The complexity of Fourier elimination is not, however, bounded exponentially by the induced width. The reason is that the number of linear inequalities that can be specified over a scope of size i cannot be bounded exponentially by i.

3.4 Summary and bibliography notes

Algorithm Adaptive-consistency was introduced by Dechter [24] as well as its complexity analysis using the concept of *induced-width* as the principle graph-parameter that controls the algorithms's complexity. A similar elimination algorithm was introduced earlier by Seidel [55]. It was observed that these algorithms belong to the class of Dynamic Programming algorithms as presented in [9]. In [25], the connection between Adaptive-Consistency and tree-clustering algorithms was made explicit, as will be shown in Chapter ??.

The observation that pair-wise resolution is the variable-elimination operation for CNFs and adaptive-consistency implies algorithm directional-resolution for CNFs, was presented in [26, 51]. It was observed that the resulting algorithm is the well known David-Putnam algorithm [15]. This correspondence offered bounding the complexity worst-case performance of the directional resolution algorithm exponentially by the induced-width as well.

3.5 Exercises

1. Prove theorem 3.1.9

2. Prove Lemma 3.2.3.

3. Prove that Directional Resolution is polynomial for 2-CNFs.

4. Assume that the input to Directional Resolution is a Horn theory (clauses that have at most one positive literal). Is the algorithm tractable? Analyze its complexity.

5. Define a bucket-elimination algorithm for a) finding all solutions to constraint satisfaction problem, b) for finding all solutions consistent with $X_1 = 0, X_2 = 1$.

6. Consider a tree-like constraint network. Show how would adaptive-consistency be applied and how a solution can be generated [give a specific tree example].

7. Write adaptive-consistency for tree networks as a distributed message-passing algorithms. Apply the algorithm to an arbitrary binary constraint network and discuss the properties of this algorithm

Chapter 4

Bucket-Elimination for Probabilistic Networks

Having investigated bucket-elimination in deterministic constraint networks in the previous chapter, we now present the bucket-elimination algorithm for the three primary queries defined over probabilistic networks: 1) belief-updating or computing posterior marginals (bel) as well as finding the probability of evidence 2) finding the most probable explanation (mpe) and 3) finding the maximum a posteriori hypothesis (map).

Recall the definition of Bayesian network in Definition 4.0.1. We start focusing on queries over Bayesian networks first and later show that the algorithms we derive are applicable with minor changes to Markov networks as well.

Definition 4.0.1 (Bayesian networks) *A* Bayesian network (BN) *is a 4-tuple* $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$. $\mathbf{X} = \{X_1, \ldots, X_n\}$ *is a set of ordered variables defined over domains* $\mathbf{D} = \{D_1, \ldots, D_n\}$, *where* $o = (X_1, \ldots, X_n)$ *is an ordering of the variables. The set of functions* $P_G = \{P_1, \ldots, P_n\}$ *consist of conditional probability tables (CPTs for short)* $P_i = \{P(X_i \,|\mathbf{Y}_i)\}$ *where* $\mathbf{Y}_i \subseteq \{X_{i+1}, ..., X_n\}$. *These* P_i *functions can be associated with a directed acyclic graph* G *in which each node represents a variable* X_i *and* $\mathbf{Y}_i = pa\,(X_i)$ *are the parents of* X_i *in* G. *That is, there is a directed arc from each parent variable of* X_i *to* X_i. *The Bayesian network* $\mathcal{B}$ *represents the probability distribution over* $\mathbf{X}$, $P_{\mathcal{B}}(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|x_{pa(X_i)})$. *We define an evidence set* e *as an instantiated subset of the variables.*

4.1 Belief Updating and Probability of Evidence

Belief updating is the primary inference task over Bayesian networks. The task is to determine the posterior probability of singleton variables once new evidence arrives. For instance, if we are interested in likelihood that the sprinkler was on last night (as we were in a Bayesian network example in Chapter 2), then we need to update this likelihood if we observe that the pavement near the sprinkler is slippery. More generally, we are sometime asked to compute the posterior marginals of a subset of variables. Another important query over Bayesian networks, computing the probability of the evidence, namely computing the joint likelihood of a specific assignment to a subset of variables, is tightly related to belief updating. We will show in this chapter how these tasks can be computed by the bucket-elimination scheme.

We will distinguish probabilistic functions (i.e., CPTs) given in the input of the Bayesian networks from probabilistic functions generated during computation. So, as we did so far, by $P(X|Y)$ we denote an input CPT of variable X given its parent set Y, while derived probability functions will be denoted by λs.

4.1.1 Deriving BE-bel

We next present a step-by-step derivation of a general variable-elimination algorithm for belief updating. We will reserve the symbol $P()$ to describe probability function of probability quantities. Sometime it will refer to a specified function, and sometime it will refer to the semantic notion of probabilities. This algorithm is similar to adaptive-consistency, but the join and project operators of adaptive-consistency are replaced, respectively, with the operations of product and summation. We begin with an example and then proceed to describe the general case.

Let $X = x_1$ be an atomic proposition (e.g., pavement = slippery). The problem of belief updating is to compute the conditional probability of x_1 given evidence e, $P(x_1|e)$, and the probability of the evidence's $P(e)$. By Bayes rule we refer to $P(x_1|e) = \frac{P(x_1,e)}{P(e)}$, where $\frac{1}{P(e)}$ is the normalization constant denoted by α. To develop the algorithm, we will use the previous example of a Bayesian network, 2.4.2 (Figure 2.4), and assume the evidence is $g = 1$.

Consider the variables in the ordering $d_1 = A, C, B, F, D, G$. We want to compute $P(A = a|g = 1)$ or $P(A = a, g = 1)$ (note that upper case denotes variable names and lower-case denotes a generic value for the same variable.) By definition

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

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

$$P(a, g = 1) = P(a) \sum_c P(c|a) \sum_b P(b|a) \sum_f P(f|b, c) \sum_d P(d|b, a) \sum_{g=1} P(g|f). \quad (4.1)$$

Carrying the computation from right to left (from G to A), we first compute the rightmost summation, which generates a function over F that we denote by $\lambda_G(F)$, defined by: $\lambda_G(f) = \sum_{g=1} P(g|f)$ and place it as far to the left as possible, yielding

$$P(a, g = 1) = P(a) \sum_c P(c|a) \sum_b P(b|a) \sum_f P(f|b, c)\lambda_G(f) \sum_d P(d|b, a). \quad (4.2)$$

(We index a generated function by the variable that was summed over to create it; for example, we created $\lambda_G(f)$ by summing over G.) Summation removes or eliminates a variable from the calculation.

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

$$P(a, g = 1) = P(a) \sum_c P(c|a) \sum_b P(b|a)\lambda_D(a, b) \sum_f P(f|b, c)\lambda_G(f) \quad (4.3)$$

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

$$P(a, g = 1) = P(a) \sum_c P(c|a) \sum_b P(b|a)\lambda_D(a, b)\lambda_F(b, c) \quad (4.4)$$

Summing over B (generating $\lambda_B(A, C)$), we get

$$P(a, g = 1) = P(a) \sum_c P(c|a)\lambda_B(a, c) \quad (4.5)$$

$bucket_G = \quad P(g|f), g = 1$
$bucket_D = \quad P(d|b, a)$
$bucket_F = \quad P(f|b, c)$
$bucket_B = \quad P(b|a)$
$bucket_C = \quad P(c|a)$
$bucket_A = \quad P(a)$

Figure 4.1: Initial partitioning into buckets using $d_1 = A, C, B, F, D, G$

Finally, summing over C (generating $\lambda_C(A)$), we get

$$P(a, g = 1) = P(a)\lambda_C(a) \tag{4.6}$$

Summing over the values of variable A, we generate $P(g = 1) = \sum_a P(a)\lambda_C(a)$. The answer to the query $P(a|g = 1)$ can be computed by normalizing the last product. Namely, $P(a|g = 1) = \alpha P(a) \cdot \lambda_C(a)$ where $\alpha = \frac{1}{P(g=1)}$.

We can create a bucket-elimination algorithm for this calculation by mimicking the above algebraic manipulation, using *buckets* as the organizational device for the various sums. First, we partition the conditional probability tables ($CPTs$, for short) into buckets relative to the given order, $d_1 = A, C, B, F, D, G$. 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. This is precisely the partition rule we used in the adaptive-consistency algorithm for constraint networks. This results in the initial partitioning given in Figure 6.1. Note that observed variables are also placed in their corresponding bucket.

Initializing the buckets corresponds to deriving the expression in Eq. (4.1). Now we process the buckets from last to first (or top to bottom in the figures), implementing the right to left computation in Eq. (4.1). Processing a bucket amounts to eliminating the variable in the bucket from subsequent computation. $bucket_G$ is processed first. We eliminate G by summing over all values g of G, but since we have observed that $g = 1$, the summation is over a singleton value. The function $\lambda_G(f) = \sum_{g=1} P(g|f) = P(g = 1|f)$, is computed and placed in $bucket_F$. In our calculations above, this corresponds to deriving Eq. (4.2) from Eq. (4.1)). Once we have have created a new function, it is placed in a

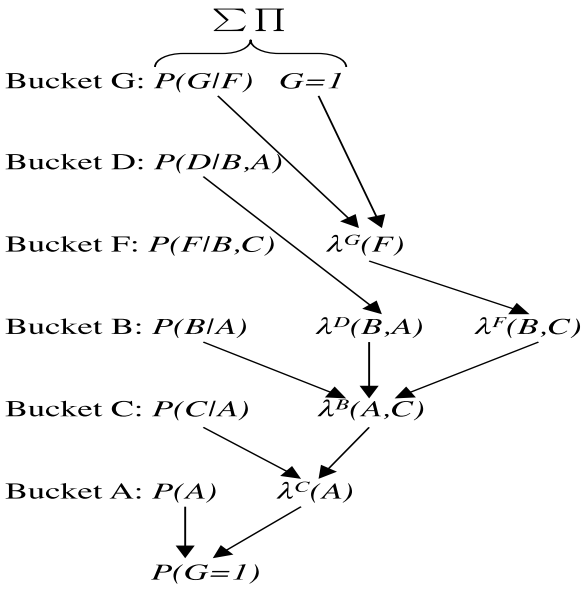


Figure 4.2: Bucket elimination along ordering $d_1 = A, C, B, F, D, G$.

lower bucket in accordance with the same rule we used to partition the original $CPTs$.

Following order d_1, we proceed by processing $bucket_D$, summing over D the product of all the functions that are in its bucket. Since there is a single function, the resulting function is $\lambda_D(b, a) = \sum_d P(d|b, a)$ and it is placed in $bucket_B$. Subsequently, we process the buckets for variables F, B, and C in order, each time summing over the relevant variable and moving the generated function into a lower bucket according to the placement rule. Since the query here is to compute the posterior marginal on A given $g = 1$, and $bucket_A$ contains $P(a)$ and $\lambda_C(a)$, we normalize the product of those two functions to get the answer: $P(a|g = 1) = \alpha \cdot P(a) \cdot \lambda_C(a)$. Figure 4.2 summarizes the flow of this computation.

In this example, the generated λ functions were at most two-dimensional; thus, the complexity of processing each bucket using ordering d_1 is (roughly) time and space quadratic in the domain sizes. But would this also be the case had we used a different variable ordering? Consider ordering $d_2 = A, F, D, C, B, G$. To enforce this ordering in our algebraic calculations we require that the summations remain in order d_2 from right to left, yielding (and we leave it to you to figure how the λ functions are generated):

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

The analogous bucket elimination schematic process for this ordering is shown in Figure 4.3a. As before, we finish by calculating $P(A = a|g = 1) = \alpha P(a)\lambda_F(a)$, where $\alpha = \frac{1}{\sum_a P(a)\lambda_F(a)}$.

We conclude this section with a general derivation of the bucket elimination algorithm for probabilistic networks, called *BE-bel*. As a byproduct, this algorithm yields the probability of the evidence. Consider an ordering of the variables $d = (X_1, ..., X_n)$ and assume we seek $P(X_1|e)$. Note that if we seek the belief for variable X_1 it should initiate the ordering. Later we will see how this requirement can be relaxed. Using the notation $\bar{x}_i = (x_1, ..., x_i)$ and $\bar{x}_i^j = (x_i, x_{i+1}, ..., x_j)$, we want to compute:

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

Separating X_n from the rest of the variables results in (F_n is variable X_n together with its parents):

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

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

where

$$\lambda_n(x_{S_n}) = \sum_{x_n} \prod_{X_i \in F_n} P(x_i, e|x_{pa_i}) \tag{4.7}$$

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

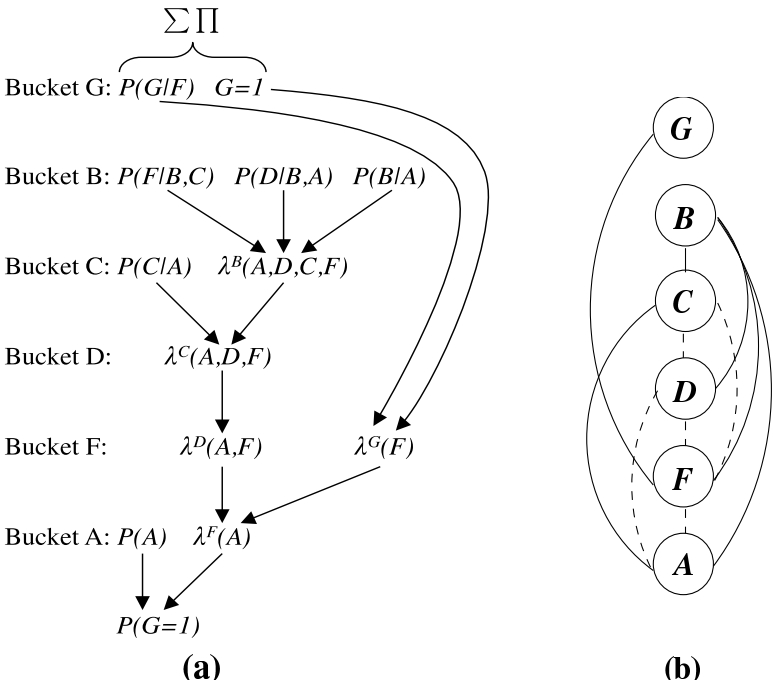


Figure 4.3: The bucket's output when processing along $d_2 = A, F, D, C, B, G$

Thus, the computation performed in bucket X_n is captured by Eq. (4.7). 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, the bucket's functions, include two types of functions the original CPT functions denoted by $P_1, ..P_k$ and the generated messages denoted by λs. For convenience we denote the product of all the original functions placed in $bucket_X$ by ψ_X, $\psi_X = \prod_{P \in bucket_X} P$, where $scope(\psi_X)$ is the union of the scopes of all the original functions in the bucket. The *messages* are the λ functions computed and placed there by other buckets to which we now refer as $\lambda_1, ..., \lambda_j$ and defined over scopes $S_1, ..., S_j$. All these functions in the bucket are multiplied and the bucket's variable is eliminated by summation. The computed function in $bucket_{X_p}$ is λ_{X_p} and is defined over scope $S_p = \cup_i S_i - X_p$ by $\lambda_X = \sum_{X_p} \psi_X \cdot \prod_{\lambda \in bucket_{X_p}} \lambda$. This function is placed in the bucket of its largest-index variable in S_p. Once processing reaches the first bucket, we have all the information to compute the answer which is the product of those functions. If we also *process* the first

bucket we get the probability of the evidence. Algorithm BE-bel is described formally in Figure 4.4. With the above derivation we showed that:

Theorem 4.1.1 *Algorithm BE-Bel applied along any ordering that starts with X_1 computes the belief $P(X_1|e)$. It also computes the probability of evidence $P(e)$ as the inverse of the normalizing constant in the first bucket.* □

The bucket's operations for BE-bel. Processing a bucket requires the two types of operations on the functions in the buckets, combinations and marginalization. The combination operation in this case is a product, which generates a function whose scope is the union of the scopes of the bucket's functions. The marginalization operation, also called elimination, is summation, summing out the bucket's variable. The algorithm often refered to as beeing a *sum-product algorithm*. Note that we refer to the variables that we eliminate in a bucket as the bucket's variable.

Let's look at an example of both of these operations in a potential bucket of B assuming it contains only two functions $P(F|B,C)$, and $P(B|A)$. These functions are displayed in Figure 4.5. To take the product of the functions $P(F|B,C)$ and $P(B|A)$ we create a function over F,B,C,A where for each tuple assignment, the function value is the product of the respective entries in the input functions. To eliminate variable B by summation, we sum the function generated by the product, over all values in D_B. We say that we *sum out* variable B. The computation of both the product and summation operators are depicted in Figure 4.6.

The implementing details of the algorithm to perform these operations might have a significant impact on the performance. In particular, much depends on how the bucket's functions are represented. If, for example, the $CPTs$ are represented as matrices, then we can exploit efficient matrix multiplication algorithms.

4.1.2 Complexity of BE-bel

Although BE-Bel can be applied along any ordering, its complexity varies considerably across different orderings. Using ordering d_1 we recorded λ functions on pairs of variables

ALGORITHM BE-BEL

Input: A belief network $\mathcal{B} =< \mathcal{X}, \mathcal{D}, \mathcal{G}, \mathcal{P} >$, an ordering $d = (\ x_1, \ldots, x_n)$; evidence e
output: The belief $P(x_1|e)$ and probability of evidence $P(e)$

1. Partition the input functions (CPTs) into $bucket_1, \ldots, bucket_n$ as follows: **for** $i \leftarrow n$ **downto** 1, put in $bucket_i$ all unplaced functions mentioning x_i. Put each observed variable in its bucket. Denote by ψ_i the product of input functions in $bucket_i$.
2. **backward: for** $p \leftarrow n$ **downto** 1 **do**
3. **for** all the functions $\psi_{S_0}, \lambda_{S_1}, \ldots, \lambda_{S_j}$ in $bucket_p$ **do**

   If (observed variable) $X_p = x_p$ appears in $bucket_p$,

   assign $X_p = x_p$ to each function in $bucket_p$ and then

   put each resulting function in the bucket of the *closest* variable in its scope.

   else,

4. $S_p \leftarrow scope(\psi_p) \cup \bigcup_{i=0}^{j} scope(\lambda_i) - \{X_p\}$
5. $\lambda_p \leftarrow \sum_{X_p} \psi_p \cdot \prod_{i=1}^{j} \lambda_{S_i}$
6. add λ_p to the bucket of the latest variable in S_p,
8. **return** $P(e) = \alpha = \sum_{X_1} \psi_1 \cdot \prod_{\lambda \in bucket_1} \lambda$

   return: $P(x_1|e) = \frac{1}{\alpha}\psi_1 \cdot \prod_{\lambda \in bucket_1} \lambda$

Figure 4.4: BE-bel: a sum-product bucket-elimination algorithm

B	C	F	$P(F\|B,C)$
false	false	true	0.1
true	false	true	0.9
false	true	true	0.8
true	true	true	0.95

A	B	$P(B\|A)$
Summer	false	0.2
Fall	false	0.6
Winter	false	0.9
Spring	false	0.4

Figure 4.5: Examples of functions in the bucket of B

A	B	C	F	$f(A,B,C,F) = P(F\|B,C) \cdot P(B\|A)$
summer	false	false	true	0.2×0.1 =0.02
summer	false	true	true	0.2×0.8 =0.16
fall	false	false	true	$0.6 \times 0.1 = 0.06$
fall	false	true	true	$0.6 \times 0.8 = 0.46$
winter	false	false	true	$0.9 \times 0.1 = 0.09$
winter	false	true	true	$0.9 \times 0.8 = 0.72$
spring	false	false	true	0.4×0.1 =0.04
spring	false	true	true	0.4×0.8 =0.32
summer	true	false	true	0.8×0.9 =0.72
summer	true	true	true	0.8×0.95 =0.76
fall	true	false	true	0.4×0.9 =0.36
fall	true	true	true	0.4×0.95 =0.38
winter	true	false	true	0.1×0.9 =0.09
winter	true	true	true	0.1×0.95 =0.095
spring	true	false	true	0.6×0.9 =0.42
spring	true	true	true	0.6×0.95 =0.57

A	B	F	$\lambda_C(A,B,F) = \sum_B f(A,B,C,F)$
summer	false	true	0.02+0.72 = 0.74
fall	false	true	0.06 + 0.36 = 0.42
winter	false	true	0.09 +0.09 = 0.18
spring	false	true	0.04 + 0.42 = 0.46
summer	true	true	0.72 + 0.16 = 0.88
fall	true	true	0.46 + 0.38 = 0.84
winter	true	true	0.72 + 0.095 = 0.815
spring	true	true	0.32 + 0.57 = 0.89

Figure 4.6: Processing the functions in the bucket of B

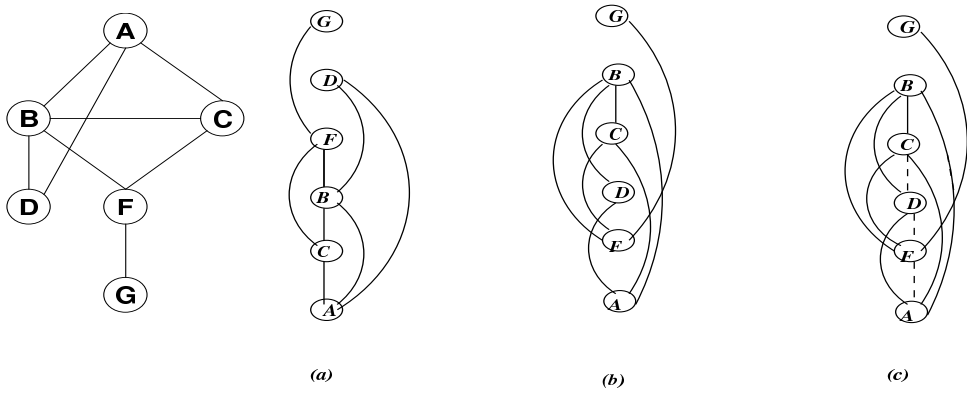


Figure 4.7: Two orderings, d_1 (a) and d_2 (b) of our example moral graph. In (c) the induced graph along ordering d_2

only, while using d_2 we had to record functions on as many as four variables (see $Bucket_C$ in Figure 4.3a). The arity (i.e., the scope size) of the function generated during processing of a bucket equals the number of variables appearing in that processed bucket, excluding the bucket's variable itself. 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 dominated by its largest scope bucket and it is therefore exponential in the size (number of variables) of the bucket having the largest number of variables. The base of the exponent is bounded by a variable's domain size.

Fortunately, as was observed earlier for adaptive-consistency, the bucket sizes can be easily predicted from the elimination process along the ordered graph. Consider the *primal graph* of a given Bayesian network. (see chapter 2) This graph has a node for each variable and any two variables appearing in the same CPT are connected. It is often called *moral graph* in the context of Bayesian networks because it can be obtained by connecting the parents of each node in the Bayesian directed graph and ignoring the arrows. We will use *moral graph* and *primal graph* interchangeably in the context of Bayesian networks. The moral graph of the network in Figure 2.4(a) is given in Figure 2.4(b). If we take this moral graph and impose an ordering on its nodes, the induced-width of the ordered graph of each nodes captures the number of variables which would be processed in that bucket. We demonstrate this next.

Example 4.1.2 Recall the definition of induced graph (Definition 3.1.7). The induced graph of the moral graph in Figure 2.4b, relative to ordering $d_1 = A, C, B, F, D, G$ is depicted in Figure 4.7a. Along this ordering the induced ordered graph was not added any edges over the original graph, since all the earlier neighbors of each node are already connected. The induced width of this graph is 2. Indeed, in this case, the maximum arity of functions recorded by the algorithm is 2. For ordering $d_2 = A, F, D, C, B, G$, the ordered moral graph is depicted in Figure 4.7b and the induced graph is given in Figure 4.7c. In this ordering, the induced width is not the same as the width. For example, the width of C is initially 2, but its induced width is 3. The maximum induced width over all the variables in this ordering is 4 which is the induced-width of this ordering. □

Theorem 4.1.3 (Complexity of BE-bel) *Given a Byaesian network whose moral graph is G, let w^* be its induced width of G along ordering d, k the maximum domain size and r be the number of input $CPTs$. The time complexity of BE-bel is $O(r \cdot k^{w^*+1})$ and its space complexity is $O(n \cdot k^{w^*})$.*

Proof. During BE-bel, each bucket creates a λ function which can be viewed as a message that it sends to a *parent* bucket, down the ordering (recall that we process the variables from last to first). Since to compute this function over w^* variables the algorithm needs to consider all the tuples defined on all the variables in the bucket, whose number is bounded by $w^* + 1$, the time to compute the function is bounded by k^{w^*+1} and it size is bounded by k^{w^*}. For each of these k^{w^*+1} tuple we need to compute its value by considering information from each of the functions in the buckets. If r_i is the number of the bucket's input messages and deg_i is the number of messages it receives from its children, then the computation of the bucket's function is $O((r_i + deg_i + 1)k^{w^*+1})$. Therefore, summing over all the buckets, the algorithm's computation is bounded by

$$\sum_i (r_i + deg_i - 1) \cdot k^{w^*+1}.$$

We can argue that $\sum_i deg_i \leq n$, when n is the number of variables, because only a single function is generated in each bucket, and there are total of n buckets. Therefore 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 bounded by n, the total space complexity is $O(n \cdot k^{w^*})$. □

4.1.3 The impact of Observations

In this section we will see that observations, which are variable assignments, can have two opposing effects on inference. One effect is of universal simplification and applies to any graphical model, while the other introduces complexity but is specific to Bayesian networks.

Evidence removes connectivity

The presence of observed variables, which we call evidence in the Bayesian network context, is inherent to queries over probabilistic networks. From a computational perspective evidence is just an assignments of values to a subset of the variables. It turns out that the presence of such partial assignments can significantly simplify inference algorithms such as bucket-elimination. In fact we will see that this property of variable instantiations, or conditioning, as it is sometime called, is the basis for algorithms that combine search and variable-elimination, to be discussed in future chapters.

We will show now that observed variables can be handled in a way that case reduce computation. Take our belief network example with ordering d_1 and suppose we wish to compute the belief in A, having observed $B = b_0$. 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 $B = b_0$ (see Figure 4.2 and add $B = b_0$ to $bucket_B$). Note that b_0 represent a specific value in the domain of B while b stands for an arbitrary value in its domain.

The processing rule dictates computing $\lambda_B(a, c) = P(b_0|a)\lambda_D(b_0, a)\lambda_F(b_0, c)$. Namely, generating and recording a two-dimensioned function. It would be more effective, however, to apply the assignment b_0 to each function in the bucket *separately* and then put the individual resulting functions into lower buckets. In other words, we can generate $\lambda_1(a) = P(b_0|a)$ and $\lambda_2(a) = \lambda_D(b_0, a)$, each of which has a single variable in its scope and will be placed in bucket A, and $\lambda_F(b_0, c)$, which will be placed in bucket C. By doing so, we avoid increasing the dimensionality of the recorded functions. In order to exploit this feature we introduce a special rule for processing buckets with observations: the observed value is assigned to each function in a bucket, and each function generated by this assignment is moved to the appropriate lower bucket.

Considering now ordering d_2, $bucket_B$ contains $P(b|a), P(d|b,a), P(f|c,b)$, and $B = b_0$ (see Figure 4.3a). The special rule for processing buckets holding observations will place the function $P(b_0|a)$ in $bucket_A$, $P(d|b_0,a)$ in $bucket_D$, and $P(f|c,b_0)$ in $bucket_F$. In subsequent processing only one-dimensional functions will be recorded. We see therefore that the presence of observations often reduces complexity: buckets of observed variables are processed in linear time and their recorded functions do not create functions on new subsets of variables.

Alternatively, we could just preprocess all the functions in which B appears and assign it the value b_0. This will reduce those functions scope and remove variable B altogether. We can then apply BE to the resulting pre-processed problem. Both methods will lead to an identical performance, but using an explicit rule for observations during BE allows for a more general and dynamic treatment. It can later be generalized by replacing observations by more general constraints (see xxx).

In order to see the implication of the observation rule computationally, we can modify the way we manipulate the ordered graph and will not add arcs among parents of observed variables when computing the induced graph which will permit a tighter bound on complexity. Adding arcs unnecessarily will results in loose complexity bounds. To capture this refinement we use the notion of *conditional induced graph*.

Definition 4.1.4 (conditional induced-width) *Given a graph G, the conditional induced graph relative to ordering d and evidence variables E, denoted $w^*_d(E)$, is generated, processing the ordered graph from last to first, by connecting the earlier neighbors of unobserved nodes only. The* conditional induced width *is the width of the conditional induced graph, disregarding observed nodes.*

For example, in Figure 4.8(a,b) we show the ordered moral graph and the induced ordered moral graph of the graph in Figure 2.4. In Figure 4.8(c) the arcs connected to the observed node B are marked by broken lines and are disregarded, resulting in the conditional induced-graph given in Figure 4.8(d). Modifying the complexity in Theorem 4.1.3, we get that,

Theorem 4.1.5 *Given a Bayesian network having n variables, algorithm BE-bel when*

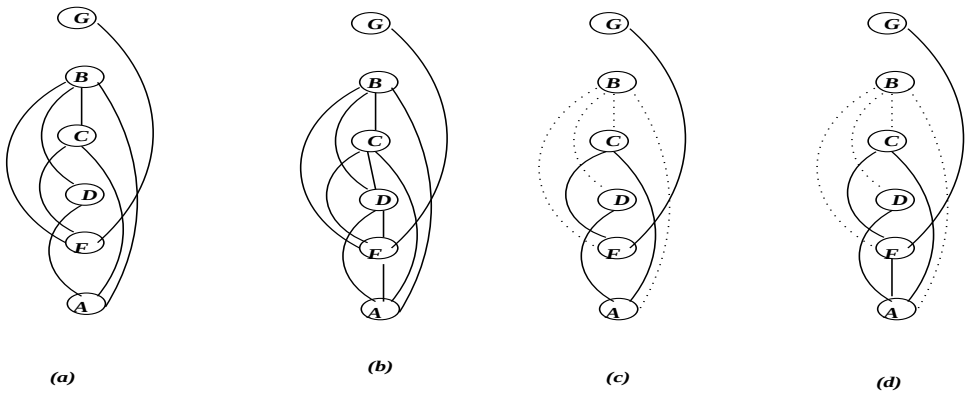


Figure 4.8: Adjusted induced graph relative to observing B

using ordering d and evidence $E = e$, is time and space exponential in the conditional induced width $w_d^*(E)$ of the network's ordered moral graph. Specifically, The time complexity is $O(r \cdot k^{w_d^*(E)+1})$ and its space complexity is $O(n \cdot k^{w_d^*(E)})$. □

It is easy to see that the conditional induced-width is the induced-width obtained by removing the evidence variables altogether. This means that the impact of evidence can be accounted for before we commit to a particular variable orderings. Namely, we can remove evidence nodes and their incident edges from the moral graph, and only then compute an ordering whose induced-width is small.

Evidence creating connectivity; relevant subnetworks

We saw that observation can simplify computation. But, in Bayesian networks observation can also complicate inference. Belief-updating for Bayesian networks can be restricted to a subset of the network if the evidence and the query variables are concentrated in a certain part of the graph. When there is no evidence, computing the belief of a variable depends only on its non-descendant portion. The portion of the network which is relevant to a query grows as we seek to determine the belief based on a larger evidence set. This is because Bayesian networks functions convey local l probability distributions, and summation over all arguments values of a probability function is the constant 1. If we identify the portion of the network that is irrelevant, we can skip some buckets. For example, if we use a *topological ordering* from root to leaves along the directed acyclic

graph, (where parents precede their child nodes), and assuming that the queried variable is the first in the ordering, we can identify skippable buckets dynamically during processing.

Proposition 4.1.6 *Given a Bayesian network and a topological ordering $X_1, ..., X_n$, that begins a query variable X_1, algorithm BE-bel, computing $P(x_1|e)$, can skip a bucket if during processing the bucket contains no evidence variables and no newly computed messages.*

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 neither evidence nor new functions in the bucket the summation operation $\sum_x P(x|pa(X))$ will yield the constant 1. □

Example 4.1.7 Consider again the belief network whose acyclic graph is given in Figure 2.4(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 in the ordered processing are not skippable. □

Alternatively, we can consider pruning the non-relevant portion of the Bayesian network in advance, before committing to any processing ordering. The relevant subnetwork is called *ancestral subnetwork* and is defined recursively as follows.

Definition 4.1.8 (ancestral graph) *Given a Bayesian network's directed graph $G = (X, E)$, and a query involving variables S, (including the evidence variables), the ancestral graph of G, G_{anc}, relative to $S \subseteq X$, includes all variables in S and if a node is in G_{anc}, its parents are also in G_{anc}.*

Example 4.1.9 Continuing with the example from Figure 2.4(a), and assuming we want to assess the belief in A given evidence on F, the relevant ordered moral graph in Figure 2.4(a)(c) should be modified by deleting nodes D and G. The resulting graph has nodes A, B, C and F only. □

Theorem 4.1.10 *Given a Bayesian $\mathcal{B} =< \mathcal{X}, \mathcal{D}, \mathcal{G}, \mathcal{P} >$ and a query $P(Y|e)$, when $Y \subseteq X$ and $E \subseteq X$ is the evidence variable set, we can compute $P(Y|e)$ by considering only the ancestral Bayesian network relative to $Y \cup E$.*

Proof: The proof is left as an exercise. ∎

4.2 Bucket elimination for optimization tasks

Having examined the task of belief-updating using the algorithmic framework of bucket elimination, we will now focus on another primary query we often have related to a Bayesian network, that is, finding the most probable explanation (mpe) for the evidence. Belief-updating answers the question "what is the likelihood of a variable given the observed data?" Answering that question, however, is often not enough; we want to be able to find the most likely explanation for the data we encounter. This is an optimization problem, and while we pose the problem here on a probabilistic network, it is a problem that is representative of optimization tasks on many types of graphical model.

The most probable explanation (mpe) task appears in numerous applications. Examples range from diagnosis and design of probabilistic codes to haplotype recognition in the context of family trees, and medical and circuit diagnosis. 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 which would correspond to a *map* query. However the mpe is close enough and is often used in applications (see [1] for more examples). Finally the queries of mpe/map (see Chapter 2) drive most of the learning algorithms for graphical model [39]. Our focus here is on algorithms for answering such queries on a given graphical model.

4.2.1 A Bucket-Elimination Algorithm for mpe

Given a Bayesian network $\mathcal{B} =< X, D, G, \mathcal{P} >$, the mpe task seeks an assignmnet to all the variables excluding the evidence variables that has the maximal probability given the evidence. Namely the task is to find a full instantiation $\overline{x}^0$ such that $P(\overline{x}^0) = \max_x P(\overline{x}, e)$, where denoting $\overline{x} = (x_1, ..., x_n)$, $P(\overline{x}, e) = \prod_i P(x_i, e|x_{pa_i})$. (remember the x_{pa_i} is the assignments to x restricted to the variables in the parent set of X_i.) Let e be a set of observations on a subset of the variables E. Given a variable ordering $d = X_1, ..., X_n$, we can accomplish this task by performing maximization operation, variable by variable, along the ordering from last to first (i.e., right to left), migrating to the left all CPTs that

do not mention the maximizing variable. We will derive this algorithm in a similar way to that in which we derived BE-bel. Using the notation defined earlier for operations on functions, our goal is to find M, s.t.

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

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

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

where

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

and S_n are the variables in the scopes of the local functions having X_n in their scope and ch_n are the child variables of X_n. Clearly, the algebraic manipulation of the above expressions is the same as the algebraic manipulation for belief updating where summation is replaced by maximization. Consequently, the bucket-elimination procedure *BE-mpe* is identical to BE-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 functions, which in this case can be regarded as cost functions denoted $h_1, ..., h_j$ and defined over some scopes and then eliminate the bucket's variable by maximization. If we distinguish again between the original function in the bucket denoted ψ_p and the messages which in this case will be denoted by h, the generated function in the bucket of X_p is $h_p : S_p \rightarrow R$, $h_p = \max_{X_p} \psi_p \cdot \prod_{i=1}^{j} h_i$, where $S_p = scope(\psi_p) \cup \cup_i scope(h_i) - X_p$. The function generated by a bucket is placed in the bucket of its largest-index variable in S_p. If the function is a constant, we can place it directly in the first bucket; constant functions are not necessary to determine the exact mpe value.

We define the function $x^o : D_{S_P - X_p} \rightarrow D_{X_p}$ by $x_p^o(x_{S_P - X_p}) = argmax_{x_p} h_p(x_{S_p})$, which provides the optimizing assignment to X_p given the assignment to the variables in the function's scope. This function can be recorded and placed in the bucket of X_p [1].

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

The bucket-processing continues with the next variable, proceeding from the last to the first variable. Once all buckets are processed, the *mpe* value, M, can be extracted as the maximizing product of functions in the first bucket. At this point we know the mpe value but we have not generated an optimizing tuple. This requires a forward phase, which was not needed when we computed beliefs or the probability of evidence. The algorithm initiates a *forward phase* to compute an *mpe* tuple by assigning the variables along the ordering from X_1 to X_n, consulting the information recorded in each bucket. Specifically, the value x_i is selected to maximize the product in $bucket_i$ given the partial assignment $x = (x_1, ..., x_{i-1})$. The algorithm is presented in Figure 4.14. Observed variables are handled as in BE-bel.

Example 4.2.1 Consider again the belief network in Figure 2.4(a). Given the ordering $d = A, C, B, F, D, G$ and the evidence $G = 1$, we process variables from last to first once partitioning the conditional probability functions into buckets, as was shown in Figure 4.1. 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. In this case it is just $G^o(f) = 1$, namely, the value of G yielding the maximum cost extension into G is the constant $G = 1$. 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 functions: $P(f|b, c)$ 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 process $bucket_B$, 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 process C, (i.e., 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. A schematic computation is the same as in Figure 4.2 where λ is simply replaced by h. □

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

Algorithm BE-mpe

Input: A belief network $\mathcal{B} =< X, D, G, \mathcal{P} >$, where $\mathcal{P} = \{P_1, ..., P_n\}$; an ordering of the variables, $d = X_1, ..., X_n$; observations e.

Output: The most probable assignment given the evidence.

1. **Initialize:** Generate an ordered partition of the conditional probability function, $bucket_1$, ..., $bucket_n$, where $bucket_i$ contains all functions whose highest variable is X_i. Put each observed variable in its bucket. Let ψ_i be the input function in a bucket and let h_i be the messages in the bucket.

2. **Backward:** For $p \leftarrow n$ downto 1, do
for all the functions $h_1, h_2, ..., h_j$ in $bucket_p$, do

- **If** (observed variable) $bucket_p$ contains $X_p = x_p$, assign $X_p = x_p$ to each function and put each in appropriate bucket.
- **else**, $S_p \leftarrow \bigcup_{i=1}^{j} scope(h_i) \cup scope(\psi_p) - \{X_p\}$. Generate functions $h_p \Leftarrow \max_{X_p} \psi_p \cdot \Pi_{i=1}^{j} h_i$ Add h_p to the bucket of the largest-index variable in S_p.

3. **Forward:**

- Generate the mpe cost by maximizing over X_1, the product in $bucket_1$.
- (generate an mpe tuple)
  For $i = 1$ to n along d do: Given $\overline{x}_{i-1} = (x_1, ..., x_{i-1})$ Choose $x_i = argmax_{X_i} \psi_i \cdot \Pi_{\{h_j \in \ bucket_i\}} h_j(\overline{x}_{i-1})$

Figure 4.9: Algorithm *BE-mpe*

As in the case of belief updating, the complexity of BE-mpe is bounded exponentially by the dimension of the recorded functions, and those functions are bounded by the induced width $w_d^*(E)$ of the ordered moral graph conditioned on the evidence variables, E.

Theorem 4.2.2 *Algorithm BE-mpe is complete for the mpe task. Its time and space complexity are* $O(r \cdot k^{w_d^*(E)+1})$ *and* $O(n \cdot k^{w_d^*(E)})$*, respectively, where* n *is the number of variables,* k *bound the domain size and* $w_d^*(E)$ *is the induced width of the ordered moral graph along* d*, conditioned on the evidence* E*.* □

4.2.2 An Elimination Algorithm for Map

The maximum a'posteriori hypothesis (map)[2] task is a generalization of both mpe and belief updating. It asks for the maximal probability associated with a *subset of hypothesis variables* and is likewise widely applicable especially for diagnosis tasks. Belief updating is the special case where the hypothesis variables are just single variables. The mpe query is the special case when the hypothesis variables are all the unobserved variables. We will see that since it is a mixture of the previous two tasks, in its bucket-elimination algorithm some of the variables are eliminated by summation while others by maximization.

Given a Bayesian network, a subset of hypothesized variables $A = \{A_1, ..., A_k\}$, and some evidence e, the problem is to find an assignment to A having maximum probability given the evidence compared with all other assignments to A. Namely, the task is to find $a^o = argmax_{a_1,...,a_k} P(a_1, ..., a_k, e)$ (see also Definition 2.4.3). So, we wish to compute $\max_{\bar{a}_k} P(a_1, ..., a_k, e) = \max_{\bar{a}_k} \sum_{\bar{x}_{k+1}^n} \prod_{i=1}^n P(x_i, e|x_{pa_i})$ where $\overline{x} = (a_1, ..., a_k, x_{k+1}, ..., x_n)$. Algorithm *BE-map* in Figure 4.10 considers only orderings in which the hypothesized variables start the ordering because summation should be applied first to the subset of variables which are in $X - A$, and subsequently maximization is applied to the variables in A. Since summation and maximization cannot be permuted we have to be restricted in the orderings (more on this shortly). Like BE-mpe, the algorithm has a backward phase and a forward phase, but the forward phase extends to the hypothesized variables only. Because only restricted orderings are allowed, the algorithm may be forced to have far

[2]sometime map is meant to refer to the mpe, and the map task is called marginal map.

higher induced-width than would otherwise be allowed. We can partially alleviate this orderings restriction and allow maximization and summations to be interleaved as long as some constraints are obeyed. (We will leave this as an exercise.)

Theorem 4.2.3 *Algorithm BE-map is complete for the map task for orderings started by the hypothesis variables. Its time and space complexity are are* $O(r \cdot k^{w_d^*(E)+1})$ *and* $O(n \cdot k^{w_d^*(E)})$*, respectively, where* n *is the number of variables in graph,* k *bounds the domain size and* $w_d^*(E)$ *is the conditioned induced width of its moral graph along* d*. (prove as an exercise.)* □

4.3 Bucket-elimination for Markov Random Fields

Recalling Definition **??** of a Markov network (presented below for convenience), it is easy to see that the bucket-elimination algorithms we presented for Bayesian networks are immediately applicable to all the main queries over Markov networks. We only need to do is replace the input conditional probabilities through which a Bayesian network is specified by the collection of local functions denoted by $\psi(.)$ which are also know as potentials or factors. The query of computing posterior marginals is accomplished by BE-bel, computing mpe and map are accomplished by BE-mpe and BE-map respectively. The task of computing Z, the partition function, is identical algorithmically to the task of computing the probability of the evidence. Therefore BE-evidence is applicable.

If the Markov network is specified by the exponential representation, then the combination function is summation rather than multiplication, but everything else stays the same. This is also demonstrated in the following subsection. In an exercise we ask that you will apply, BE-evidence to the Markov network in Figure 2.5.

Definition 4.3.1 (Markov Networks) *A Markov network is a graphical model* $\mathcal{M} = \langle \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{H}, \prod \rangle$ *where* $\boldsymbol{H}= \{\psi_1, \ldots, \psi_m\}$ *is a set of potential functions where each potential* ψ_i *is a non-negative real-valued function defined over a scope of variables* $\boldsymbol{S}_i$*. The Markov network represents a global joint distribution over the variables* $\boldsymbol{X}$ *given by:*

$$P(\boldsymbol{x}) = \frac{1}{Z} \prod_{i=1}^{m} \psi_i(\boldsymbol{x}) \quad , \quad Z = \sum_{\boldsymbol{x}} \prod_{i=1}^{m} \psi_i(\boldsymbol{x})$$

Algorithm BE-map

Input: A Bayesian network $\mathcal{B} =< X, D, G, \mathcal{P} > P = \{P_1, ..., P_n\}$; a subset of hypothesis variables $A = \{A_1, ..., A_k\}$; an ordering of the variables, d, in which the A's are first in the ordering; observations e. ψ_i is the input function in the bucket of X_i.

Output: A most probable assignment $A = a$.

1. **Initialize:** Generate an ordered partition of the conditional probability functions, $bucket_1$, ..., $bucket_n$, where $bucket_i$ contains all functions whose highest variable is X_i.

2. **Backwards** For $p \leftarrow n$ downto 1, do

for all the message functions $\beta_1, \beta_2, ..., \beta_j$ in $bucket_p$ and for ψ_p do

- **If** (observed variable) $bucket_p$ contains the observation $X_p = x_p$, assign $X_p = x_p$ to each β_i and ψ_p and put each in appropriate bucket.
- **else**, $S_p \leftarrow scope(\psi_p) \cup \bigcup_{i=1}^{j} scope(\beta_i) - \{X_p\}$. If X_p is not in A, then $\beta_p \Leftarrow \sum_{X_p} \psi_p \cdot \Pi_{i=1}^{j} \beta_i$;
  else, $(X_p \in A)$, $\beta_p \Leftarrow \max_{X_p} \psi_p \cdot \prod_{i=1}^{j} \beta_i$
  Place β_p in the bucket of the largest-index variable in S_p.

3. **Forward:** Assign values, in the ordering $d = A_1, ..., A_k$, using the information recorded in each bucket in a similar way to the forward pass in BE-mpe.

Figure 4.10: Algorithm *BE-map*

where the normalizing constant Z is referred to as the partition function.

4.4 Cost Networks and Dynamic Programming

As we mentioned at the outset, bucket-elimination algorithms are variations of a very well known class of optimization algorithms known as *Dynamic Programming* [7, 9]. Here we make the connection explicit, observing that BE-mpe is a dynamic programming scheme with some simple transformations.

That BE-mpe is dynamic programming becomes apparent once we transform the mpe's cost function, which has a product combination operator, into the traditional additive combination operator 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 $C_i = -logP_i$.

The general dynamic programming algorithm is defined over *cost networks* (see discussion in Section 2.3). As we showed earlier a *cost network* is a tuple $\mathcal{C} =< X, D, C, \sum >$, where $X = \{X_1, ..., X_n\}$ are variables over domains $D = \{D_1, ..., D_n\}$, C is a set of real-valued cost functions $C_1, ..., C_l$, defined over scopes $C_i : \bowtie_{j=1}^{r} D_{scop(C_j)} \rightarrow R^+$. The task is to find an assignment to the variables that minimizes $\sum_i C_i$.

A straightforward elimination process similar to that of BE-mpe, (where the product is replaced by summation and maximization by minimization) yields the non-serial dynamic programming algorithm [9]. The algorithm, called *BE-opt*, is given in Figure 4.11. A schematic execution of our example along ordering $d = G, A, F, D, C, B$ is depicted in Figure 4.12. And, not surprisingly, we can show that

Theorem 4.4.1 *Given a cost network $\mathcal{C} =< X, D, C, \sum >$, BE-opt is complete for finding an optimal cost solution. Its time and space complexity are $O(r \cdot k^{w_d^*(E)+1})$ and $O(n \cdot k^{w_d^*(E)})$, respectively, where n is the number of variables in graph, k bounds the domain size and $w_d^*(E)$ is the conditioned induced width of its moral graph along d.* □

Algorithm BE-opt

Input: A cost network $C = \{C_1, ..., C_l\}$; ordering d; assignment e over variables E.

Output: A minimal cost assignment.

1. **Initialize:** Partition the cost components into buckets. Define ψ_i as the sum of the input cost functions in bucket X_i.

2. **Process buckets** from $p \leftarrow n$ downto 1

For ψ_p and the cost messages $h_1, h_2, ..., h_j$ in $bucket_p$, do:

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

3. **Forward:** Assign minimizing values in ordering d, consulting functions in each bucket, as in BE-mpe

Figure 4.11: Dynamic programming as BE-opt

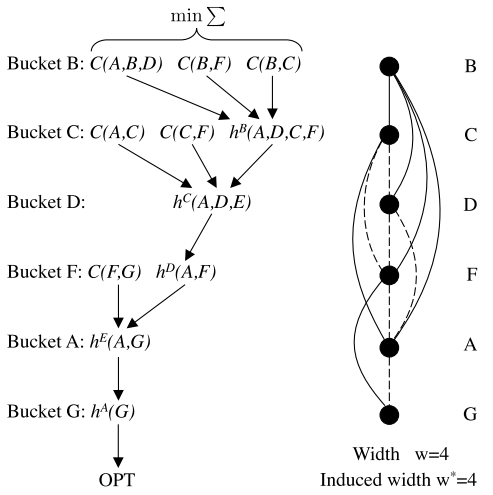


Figure 4.12: Schematic execution of BE-Opt

Consulting again the various classes of cost networks elaborated on in Section 2.3, algorithm, BE-opt is applicable to all including weighted-csps, max-csps and max-sat.

4.5 Mixed Networks

[3]

The last class of graphical models we will address is mixed network which were discussed and defined in Section 2.5. To refresh, these models allows the explicit representation of both probabilistic information and constraints. Therefor we assume that the mixed network is defined by a pair of a Bayesian network and a constraint network. This pair express a conditional probability distribution over all the variables which requires that all the assignments that have non-zero probability will satisfy all the constraints.

We will focus only on the task that is unique for this graphical model, the *constraint probability evaluation* (CPE) which can also stand for *CNF probability evaluation.* For example, if we have both a Bayesian network and a cnf formula φ, we want to compute $P(x \in \varphi)$.

Given a mixed network $\mathcal{M}_{(\mathcal{B},\varphi)}$, where φ is a CNF formula defined on a subset of variables Q, the CPE task is to compute: (remember $moels(\varphi)$ are the assignments to all variables satisfying φ. $\overline{x}_Q$ is the restriction of $\overline{x}$ to the variables in Q.

$$P_{\mathcal{B}}(\varphi) = \sum_{\bar{x}_Q \in models(\varphi)} P(\bar{x}_Q).$$

Using the belief network product form we get:

$$P(\varphi) = \sum_{\{\bar{x}|\bar{x}_Q \in models(\varphi)\}} \prod_{i=1}^{n} P(x_i|x_{pa_i}).$$

We assume that X_n is one of the CNF variables, and we separate the summation over X_n and the rest of the variables $\mathbf{X} - \{X_n\}$ as usual. We denote by γ_n the set of all clauses that are defined on X_n and by β_n all the rest of the clauses. The scope of γ_n is denoted by Q_n, and we define $S_n = \mathbf{X} - Q_n$ and S_n is the set of all variables in the scopes of CPTs

[3]* can be skipped on a first reading

and clauses that are defined over X_n. We get:

$$P(\varphi) = \sum_{\{\bar{x}_{n-1}|\bar{x}_{S_n} \in models(\beta_n)\}} \sum_{\{x_n|\bar{x}_{Q_n} \in models(\gamma_n)\}} \prod_{i=1}^{n} P(x_i|x_{pa_i}).$$

Denoting by t_n the set of indices of functions in the product that *do not* mention X_n and by $l_n = \{1, \ldots, n\} \setminus t_n$ we get:

$$P(\varphi) = \sum_{\{\bar{x}_{n-1}|\bar{x}_{S_n} \in models(\beta_n)\}} \prod_{j \in t_n} P_j \cdot \sum_{\{x_n|\bar{x}_{Q_n} \in models(\gamma_n)\}} \prod_{j \in l_n} P_j.$$

Therefore:

$$P(\varphi) = \sum_{\{\bar{x}_{n-1}|\bar{x}_{S_n} \in models(\beta_n)\}} (\prod_{j \in t_n} P_j) \cdot \lambda^{X_n},$$

where λ^{X_n} is defined over $S_n - \{X_n\}$, by

$$\lambda^{X_n} = \sum_{\{x_n|\bar{x}_{Q_n} \in models(\gamma_n)\}} \prod_{j \in l_n} P_j. \tag{4.8}$$

The case of observed variables When X_n is observed, that is constrained by a literal, the summation operation reduces to assigning the observed value to each of its CPTs *and* to each of the relevant clauses. In this case Equation (4.8) becomes (assume $X_n = x_n$ and $P_{=x_n}$ is the function instantiated by assigning x_n to X_n):

$$\lambda^{x_n} = \prod_{j \in l_n} P_{j=x_n}, \quad if\ \bar{x}_{Q_n} \in m(\gamma_n \wedge (X_n = x_n)). \tag{4.9}$$

Otherwise, $\lambda^{x_n} = 0$. Since $\bar{x}_{Q_n}$ satisfies $\gamma_n \wedge (X_n = x_n)$ only if $\bar{x}_{Q_n - X_n}$ satisfies $\gamma^{x_n} = resolve(\gamma_n, (X_n = x_n))$, we get:

$$\lambda^{x_n} = \prod_{j \in l_n} P_{j=x_n} \quad if\ \bar{x}_{Q_n - X_n} \in m(\gamma_n^{x_n}). \tag{4.10}$$

Therefore, we can extend the case of observed variable in a natural way: CPTs are assigned the observed value as usual while clauses are individually resolved with the unit clause $(X_n = x_n)$, and both are moved to appropriate lower buckets.

Therefore, in the bucket of X_n we should compute λ^{X_n}. We need to place all CPTs and clauses mentioning X_n and then compute the function in Equation (4.8). The computation

Algorithm 1: ELIM-CPE

Input: A belief network $\mathcal{B} = \{P_1, ..., P_n\}$; a CNF formula on k propositions $\varphi = \{\alpha_1, ...\alpha_m\}$ defined over k propositions; an ordering of the variables, $d = \{X_1, \ldots, X_n\}$.

Output: The belief $P(\varphi)$.

1 Place buckets with unit clauses last in the ordering (to be processed first).

`// Initialize`

Partition $\mathcal{B}$ and φ into $bucket_1, \ldots, bucket_n$, where $bucket_i$ contains all the CPTs and clauses whose highest variable is X_i.

Put each observed variable into its appropriate bucket. (We denote probabilistic functions by λs and clauses by αs).

2 **for** $p \leftarrow n$ **downto** 1 **do** `// Backward`

  Let $\lambda_1, \ldots, \lambda_j$ be the functions and $\alpha_1, \ldots, \alpha_r$ be the clauses in $bucket_p$

  `Process-bucket`$_p(\sum, (\lambda_1, \ldots, \lambda_j), (\alpha_1, \ldots, \alpha_r))$

3 **return** $P(\varphi)$ as the result of processing $bucket_1$.

of the rest of the expression proceeds with X_{n-1} in the same manner. This yields algorithm *Elim-cpe* (described in Algorithm 1 and Procedure `Process-bucket`$_p$). The elimination operation is summation for the current query. Thus, for every ordering of the propositions, once all the CPTs and clauses are partitioned, we process the buckets from last to first, in each applying the following operation. Let $\lambda_1, ...\lambda_t$ be the probabilistic functions in bucket P and $\alpha_1, ...\alpha_r$ be the clauses. The algorithm computes a new function λ^P over $S_p = scope(\lambda_1, ...\lambda_t) \cup scope(\alpha_1, ...\alpha_r) - \{X_p\}$ defined by:

$$\lambda^P = \sum_{\{x_p | \bar{x}_Q \in models(\alpha_1, ..., \alpha_r)\}} \prod_j \lambda_j$$

Example 4.5.1 Consider the belief network in Figure 4.13, which is similar to the one in Figure 2.4, and the query $\varphi = (B \vee C) \wedge (G \vee D) \wedge (\neg D \vee \neg B)$. The initial partitioning into buckets along the ordering $d = A, C, B, D, F, G$, as well as the output buckets are given in Figure 4.14. The initial partitioning can be gleaned from Figure 4.15. We compute:

In bucket G: $\lambda^G(f, d) = \sum_{\{g | g \vee d = true\}} P(g|f)$

In bucket F: $\lambda^F(b, c, d) = \sum_f P(f|b, c)\lambda^G(f, d)$

Procedure `Process-bucket`$_p(\sum, (\lambda_1, \ldots, \lambda_j), (\alpha_1, \ldots, \alpha_r))$

if *bucket*$_p$ *contains evidence* $X_p = x_p$ **then**

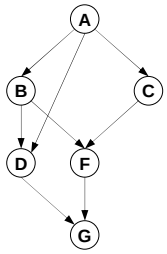
1. Assign $X_p = x_p$ to each λ_i and put each resulting function in the bucket of its latest variable

2. Resolve each α_i with the unit clause, put non-tautology resolvents in the buckets of their latest variable and **move any bucket with unit clause to top of processing**

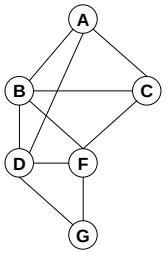
else

Generate $\lambda^p = \Downarrow_{\{x_p | \bar{x}_{U_p} \in models(\alpha_1, \ldots, \alpha_r)\}} \prod_{i=1}^{j} \lambda_i$

Add λ^p to the bucket of the latest variable in S_p, where $S_p = \bigcup_{i=1}^{j} scope(\lambda_i)) \bigcup_{i=1}^{r} scope(\alpha_i) - \{X_p\}$



(a) Directed acyclic graph



(b) Moral graph

Figure 4.13: Belief network

In bucket D: $\lambda^D(a, b, c) = \sum_{\{d | \neg d \vee \neg b = true\}} P(d|a, b)\lambda^F(b, c, d)$
In bucket B: $\lambda^B(a, c) = \sum_{\{b | b \vee c = true\}} P(b|a)\lambda^D(a, b, c)\lambda^F(b, c)$
In bucket C: $\lambda^C(a) = \sum_c P(c|a)\lambda^B(a, c)$
In bucket A: $\lambda^A = \sum_a P(a)\lambda^C(a)$
$P(\varphi) = \lambda^A$. □

For example $\lambda^G(f, d = 0) = P(g = 1|f)$, because if $D = 0$ g must get the value "1", while $\lambda^G(f, d = 1) = P(g = 0|f) + P(g = 1|f)$. In summary,

Theorem 4.5.2 (Correctness and Completeness) *Algorithm Elim-cpe is sound and complete for the CPE task.*

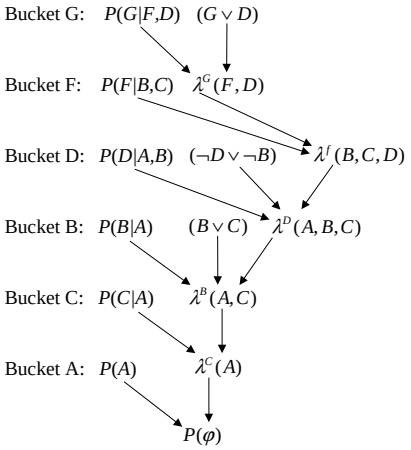


Figure 4.14: Execution of ELIM-CPE

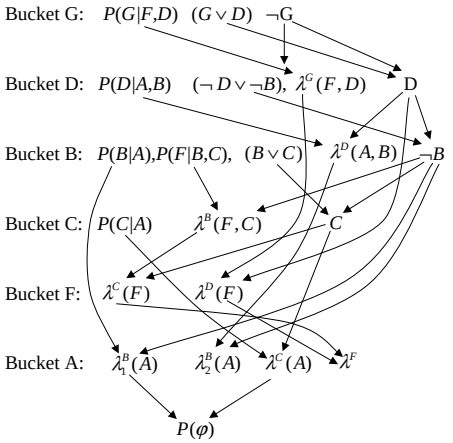


Figure 4.15: Execution of ELIM-CPE (evidence $\neg G$)

Notice that algorithm Elim-cpe also includes a unit resolution step whenever possible (see Procedure `Process-bucket`$_p$) and a dynamic reordering of the buckets that prefers processing buckets that include unit clauses. This may have a significant impact on efficiency because treating observations (namely unit clauses) in a special way can avoid creating new dependencies as we already observed.

Example 4.5.3 Let's now extend the example by adding $\neg G$ to the query. This will place $\neg G$ in the bucket of G. When processing bucket G, unit resolution creates the unit clause D, which is then placed in bucket D. Next, processing bucket F creates a probabilistic function on the two variables B and C. Processing bucket D that now contains a unit clause will assign the value D to the CPT in that bucket and apply unit resolution, generating the unit clause $\neg B$ that is placed in bucket B. Subsequently, in bucket B we can apply unit resolution again, generating C placed in bucket C, and so on. In other words, aside from bucket F, we were able to process all buckets as observed buckets, by propagating the observations. (See Figure 4.15.) To incorporate dynamic variable ordering, after processing bucket G, we move bucket D to the top of the processing list (since it has a unit clause). Then, following its processing, we process bucket B and then bucket C, then F, and finally A. □

Since unit resolution increases the number of buckets having unit clauses, and since

those are processed in linear time, it can improve performance substantially. Such buckets can be identified a priori by applying unit resolution on the CNF formula or arc-consistency on the constraint expression. In general, any level of resolution can be applied in each bucket. This can yield stronger CNF expressions in each bucket and may help improve the computation of the probabilistic functions.

4.6 The General Bucket Elimination

We now summarize and generalize the bucket elimination algorithm using the two operators of *combination* and *marginalization*. As defined in Chapter 2, the general task can be defined over a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes \rangle$, where: $X = \{X_1, ..., X_n\}$ is a set of variables having domain of values $D = \{D_1, ..., D_n\}$ and $F = \{f_1, ..., f_k\}$ is a set of functions, where each f_i is defined over $S_i = scope(f_i)$. Given a function h and given $Y \subseteq scope(h)$, the (generalized) projection operator $\Downarrow_Y h$ 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 $\otimes_j f_j$ defined over $U = \cup_j scope(f_j)$, $\otimes_{j=1}^k f_j \in \{\Pi_{j=1}^k f_j, \ \sum_{j=1}^k f_j, \ \bowtie_j f_j\}$. All queries require computing $\Downarrow_Y \otimes_{i=1}^n f_i$. Such problems can be solved by a general bucket-elimination algorithm stated in Figure 4.16. For example, BE-bel is obtained when $\Downarrow_Y = \sum_{S-Y}$ and $\otimes_j = \Pi_j$, BE-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 and directional resolution can be shown to be expressible in terms of such operators.

4.7 Combining Elimination and Conditioning

In this section we temporarily departure from inference algorithm and have a glimpse into a search scheme via the *cutset-conditioning* idea. Search methods will be presented in details later on. It is natural to mention the idea of search now by focusing on the notion of observations which simplify inference (section 4.1.3). Specifically, we observed that when some variables are assigned, connectivity in the graph reduces yielding saving in computation. The magnitude of saving is reflected through the *conditioned induced-graph*.

Cutset-conditioning is a scheme that exploits this property in a systematic way. We

Algorithm General bucket-elimination (GBE)

Input: A set of functions $F = \{f_1, ..., f_n\}$, an ordering of the variables, $d = X_1, ..., X_n$; $Y \subseteq$.

Output: A new compiled set of functions

from which the query $\Downarrow_Y \otimes_{i=1}^{n} f_i$ 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. An input function in each bucket ψ_i.

2. **Backward:** For $p \leftarrow n$ downto 1, do

for all the functions ψ_p, $\lambda_1, \lambda_2, ..., \lambda_j$ in $bucket_p$, do

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

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

Figure 4.16: Algorithm *bucket-elimination*

ALGORITHM VEC-EVIDENCE

Input: A belief network $\mathcal{B} =< \mathcal{X}, \mathcal{D}, \mathcal{G}, \mathcal{P} >$, an ordering $d = (x_1, \ldots, x_n)$; evidence e over E, a subset C of conditioned variables;

output: The probability of evidence $P(e)$

Initialize: $\lambda = 0$.

1. For every assignment $C = c$, do
   - $\lambda_1 \leftarrow$ The output of BE-bel with $c \cup e$ as observations.
   - $\lambda \leftarrow \lambda + \lambda_1$. (update the sum).
2. **Return** $P(e) = \alpha \cdot \lambda$ (α is a normalization constant.)

Figure 4.17: Algorithm *vec-evidence*

can select a subset of variables, which we call *cutset*, assign them values (i.e., condition on them), solve the conditioned problem by an inference algorithm such as BE, and repeat this process for all assignments to the cutset. The cutset can be explored by brute-force enumeration of all its assignments. We give a highlevel description of the algorithm, called *Variable elimination and Conditioning* or VEC for short in Figure 4.17.

However, the enumeration process can be done more efficiently using a search over the tree of all variables assignments. In particular, using depth-first-search (dfs) the assignments to the cutset variables can be enumerated in linear space only.

In the extreme, if we condition on all the variables (namely the custet is the whole set) we will get a brute-force dfs search that traverses the full search-tree and accumulates the appropriate sums of probabilities. For example, we can compute expression 4.11 below for the probability of evidence in the network of Figure 2.4 by traversing the search-tree in Figure 4.18 along the ordering, from first variable to last variable.

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

$$= P(a) \sum_{c} P(c|a) \sum_{b} P(b|a) \sum_{f} P(f|b,c) \sum_{d} P(d|b,a) \sum_{g} P(g|f), \tag{4.11}$$

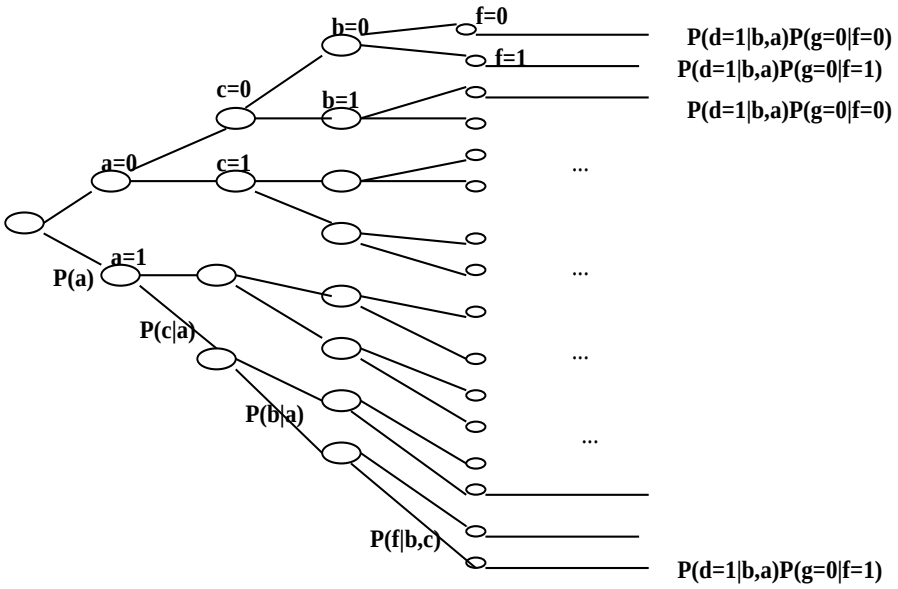


Figure 4.18: probability tree

The idea of cutset-conditioning search however, is to limit search to only a subset of variables: to the cutset variables. Let C be a cutset $C \subseteq X$, and $V = X - C$. (as usual, we denote by v an assignment to V and by c an assignment to C.) Obviously,

$$P(e) = \sum_x P(x, e) = \sum_{c \in D_C} \sum_{v \in D_V} P(c, v, e) = \sum_{c,v} \prod_i P(x_i | x_{pa_i}, c, v, e)$$

For every $C = c$, we can compute $\sum_v P(v, c, e)$ using Bucket-Elimination over variables V, while treating the conditioned variables, C, as observed variables. This basic computation will be enumerated for all value combinations of the conditioned variables, and the sum is accumulated (see the algorithm in Figure 4.17).

Clearly,

Theorem 4.7.1 *Algorithm VEC-bel is sound and complete for computing the belief of the first variable and the probability of evidence.*

Given an assignment $C = c$, the time and space complexity of computing the conditioned probability by BE is bounded exponentially by the induced-width of the ordered moral graph along ordering d conditioned for both observed (E) and conditioned (C)

nodes, (namely when both C and E are removed from the graph). This parameter is denoted by $w_d^*(E \cup C)$.

Theorem 4.7.2 (complexity) *Given a set of conditioning variables, C, the space complexity of algorithm VEC-bel is $O(n \cdot k^{(w_d^*(C \cup E))}$, while its time complexity is $O(n \cdot k^{(w_d^*(E \cup C)+|C|)})$.* □

Proof: see exercises ∎

If, removing the variables $E \cup C$ yields a cycle-free graph, we call C a cycle-cutset. Recall that in this case the conditioned induced-width is 1, and therefore the remaining subproblem can be solved efficiently by BE. This specific case yields the cycle-cutset algorithm [18], or loop-cutset algorithm [47] which we will discuss in detail later.

Definition 4.7.3 (cycle-cutset) *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.*

Theorem 4.7.2 calls for a secondary optimization task on graphs:

Definition 4.7.4 (Find a minimal w-cutset) *Given a graph $G = (V, E)$ and a constant r, find a smallest subset of nodes C_w, such that $G\prime$, the subgraph of G restricted to $V - C_w$, has induced-width less than or equal r.*

Finding a minimal r-cutset, for any r is known to be hard.

4.8 Summary and Bibliographical Notes

In the last two chapters, we showed how the bucket-elimination framework can be used to unify variable-elimination algorithms for both deterministic and probabilistic graphical models The chapters described the bucket-elimination framework which unifies variable-elimination algorithms deterministic and probabilistic reasoning as well as for optimization tasks. The algorithms take advantage of the structure of the graph. Most bucket-elimination algorithms[4] are time and space exponential in the induced-width of the underlying dependency primal graph of the problem.

[4]all, except Fourier algorithm.

The chapter is based on Dechter's Bucket-elimination algorithm that appeared in [20] and [21]. Among the early variable elimination algorithms we find the peeling algorithm for genetic trees [14], Zhang and Poole's VE1 algorithm [64] and SPI algorithm by D'Ambrosio et.al., [50] 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 [54].

In [49] the connection between optimization and constraint satisfaction and its relationship to dynamic programming is explicated. In the work of [44, 57] and later in [12] an axiomatic framework that characterize tasks that can be solved polynomially over hyper-trees, is introduced. functions, and satisfy a certain set of axioms.

The axiomatic framework [57] 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 [25] and here, tree-clustering and bucket elimination schemes are closely related, tasks that fall within the axiomatic framework [57] can be accomplished by bucket elimination algorithms as well. In [12] 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.

4.9 Exercises

1. Consider the Markov network in Figure 2.5.

   (a) Show schematically how you compute the partition function of this network using BE-evidence.

   (b) Assume that you use exponential representation. Show what will be the specification of this network using exponential notation

   (c) Compute th empe using the exponential representation

   (d) How would you compute the partition function with that representation assuming that variable E is assigned the value 0.

2. Consider the Bayesian network in Figure 2.4(a). What porion of the Bayesian network do you need to consider to compute the belief given evidence 1. $A = 0$, 2. $D = 1$ 3. $G = 0$, 4. $D = 1, G = 0$.

3. Prove Theorem 4.1.10.

4. Perform the numerical computation fully of the example of 4.2.1. Assume the evidence is $G = 1$. Find the mpe value and the corresponding assignment along the order of variables specified.

5. Prove the completeness of Algorithm BE-map. Namely prove Theorem **??**.

6. In Algorithm BE-map (Figure 4.10), the ordering of the variables is restricted so that the hypothesis variables $A \subseteq X$ appear first in the ordering and therefore process last by BE-mao. Devise a more general constraints of the feasible orderings for which the algorithm would still be complete. Namely for which it will provide the exact map value and map assignment.

7. . Cost networks are defined by ... define a bucket elimination algorithm that finds the optimal solution for a const network. Prove the correctness of your algorithm and analyze its complexity.

8. (conditioning) Consider the following belief network and assume we want to compute the probability of $Z = a$. Describe your algorithm and analyze its complexity.

9. A question on irrelevant subnetworks

10. Prove the complexity of Algorithm $VEC - evidence$ (see Theorem 4.7.2).

Chapter 5

The Graphs of Graphical Models

As we saw, and as we will see throughout the book, the structure of graphical models can be described by graphs that capture dependencies and independencies in the knowledge-base. These graphs are useful because they convey information regarding the interaction between variables and allow efficient query processing. In this chapter we provide a general overview of the graph properties that are relevant to the algorithms that we encounter. We will focus on a graph parameter called *induced-width* or *treewidth* and cycle-cutset decomposition that captures well the complexity of reasoning algorithms for graphical models.

Although we already assumed familiarity with the notion of a graph, we take the opportunity to define it formally now.

Definition 5.0.1 (Directed and undirected graphs) *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.*

A graphical model can be represented by a *primal graph*.

Definition 5.0.2 (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 absence of an arc between two nodes indicates that there is no direct function specified between the corresponding variables. We observed earlier primal graphs for a variety of graphical models depicting both constraints and probabilistic functions.

The primal graph (also called moral graph for Bayesian networks) is an effective way to capture the structure of the knowledge. In particular, graph separation is a sound way to capture conditional independencies relative to probability distributions over directed and undirected graphical models. In the context of probabilistic graphical models, primal graphs are also called i-maps (independence maps[47]). In the context of relational databases [42] primal graphs capture the notion of embedded multi-valued dependencies (EMVDs).

All advanced algorithms for graphical models exploit their graphical structure. Besides the primal graph, other graph depictions include hyper-graphs, dual graphs and factor graphs.

5.1 Types of graphs

Although the arcs in a primal graph are defined over pairs of variables in a function scopes, they are well defined for functions whose scopes is larger than two. It's arcs do not provide a one to one correspondence with scopes. Hypergraphs and dual graphs are such representations:

Definition 5.1.1 (hypergraph) *A hypergraph is a pair* $\mathcal{H} = (V, S)$ *where* $V = \{v_1, .., v_n\}$ *is a set of nodes and* $S = \{S_1, ..., S_l\}$, $S_i \subseteq V$, *is a set of subsets of* V *called hyperedges.*

In the *hypergraph* representation of a graphical model, nodes represent the variables, and *hyperarcs* (drawn as regions) are the scopes of functions. They group those variables that belong to the same scope.

A related representation is the *dual graph*. Unlike a hypergraph, the dual graph is just a regular graph structure; it represents each function scope by a node and associates a labeled arc with any two nodes whose scopes share variables. The arcs are labeled by the shared variables.

Definition 5.1.2 (a dual graph) *A hypergraph $\mathcal{H} = (V, S)$ can be mapped to a* dual graph *denoted $\mathcal{H}^{dual}$. The nodes of the dual graph are the hyperedges (or edges if we have a graph), and a pair of such nodes is connected if they share vertices in V. The arc that connects two such nodes is labeled by the shared vertices. Formally, given a hypergraph $\mathcal{H} = (V, S)$, $\mathcal{H}^{dual} = (S, E)$ where $S = \{S_1, ..., S_l\}$ are edges in $\mathcal{H}$, and $(S_i, S_j) \in E$ iff $S_i \cap S_j \neq \emptyset$.*

Definition 5.1.3 (A primal graph of a hypergraph) *A primal graph of a hypergraph $\mathcal{H} = (V, S)$ has V as its set of nodes, and any two nodes are connected if they appear in the same hyperedge.*

Note that if all the functions in the graphical model are binary, then its hypergraph is identical to its primal graph.

Graphical models and hypergraphs

Any graphical model $\mathcal{R} =< X, D, G, \mathcal{F} >$, $F = \{f_{S_1}, ..., f_{S_t}\}$ can be associated with a hypergraph $\mathcal{H}_{\mathcal{R}} = (X, H)$, where X is the set of nodes (variables), and H is the scopes of the functions in $\mathcal{F}$, namely $H = \{S_1, ..., S_l\}$. Therefore, the dual graph of the hypergraph of a graphical model associates a node with each function's scope and an arc for each two nodes sharing variables.

Example 5.1.4 Figure 5.1 depicts the *hypergraph* (a), the *primal graph* (b) and the *dual graph* (c) representations of a graphical model with variables A, B, C, D, E, F and with functions on the scopes (ABC), (AEF), (CDE) and (ACE). The specific functions are irrelevant to the current discussion; they can be arbitrary relations over domains of $\{0, 1\}$, such as $C = A \vee B$, $F = A \vee E$, CPTs or cost functions. The meaning of graph (d) will be described shortly. □

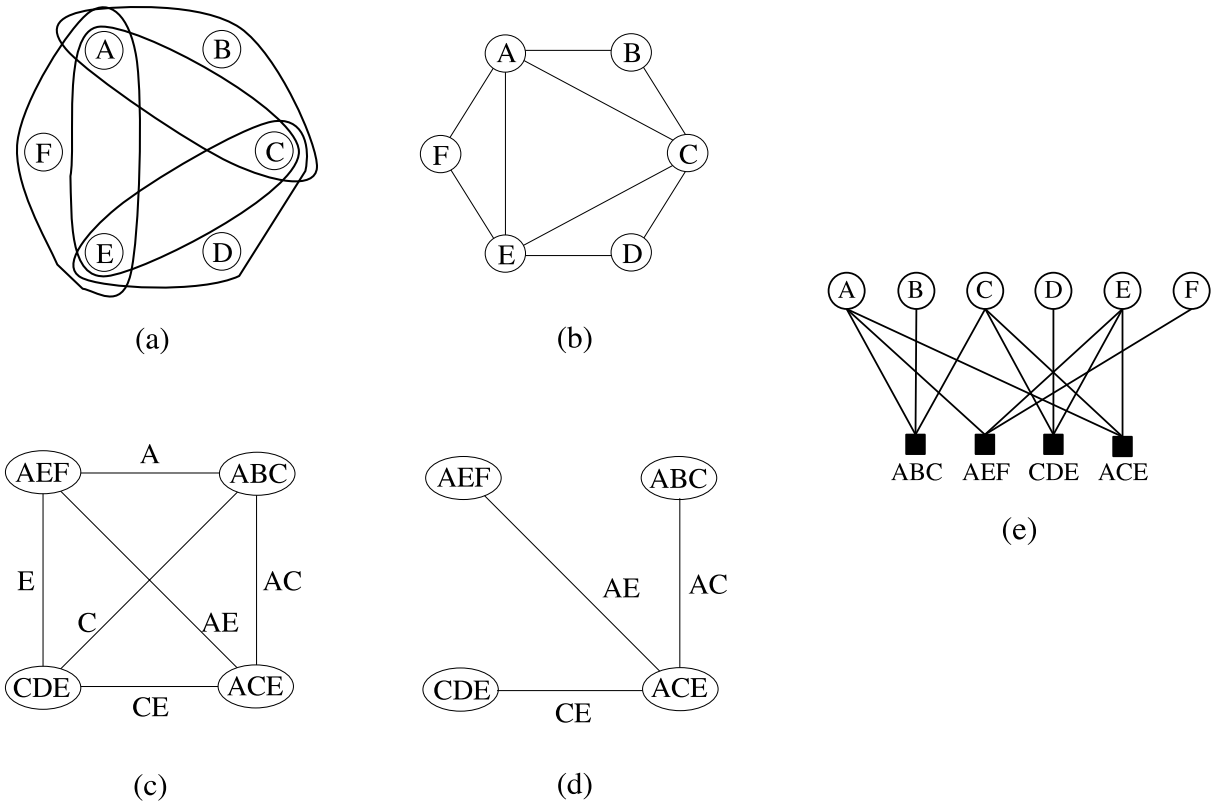


Figure 5.1: (a)Hyper, (b)Primal, (c)Dual and (d)Join-tree of a graphical model having scopes ABC, AEF, CDE and ACE. (e) the factor graph

A factor graph is also a popular graphical depiction of a graphical model.

Definition 5.1.5 (factor graph) *Given a graphical model and its hypergraph $H = (V, S)$ defined by the functions scopes, the factor graph has function nodes and variable nodes. Each scope is associated with a function node and it is connected to all the variable nodes appearing in the function.*

Figure 5.1e depicts the factor graph of the hypergraph in part (a).

We have seen that there is a tight relationship between the complexity of inference algorithms such as bucket elimination and the graph concept called induced width. All inference algorithms are time and space exponential in the induced-width along the order

of processing. This motivates finding an ordering with a smallest induced width, a task known to be hard [4]. However, useful greedy heuristics algorithms are available and we will briefly review these in the next few paragraphs [23, 5, 59].

5.2 The induced width

Definition 5.2.1 (width) *Given an undirected graph $G = (V, E)$, an* ordered graph *is a pair (G, d), where $V = \{v_1, ..., v_n\}$ is the set of nodes, E is a set of arcs over V, and $d = (v_1, ..., v_n)$ is an ordering of the nodes. The nodes adjacent to v that precede it in the ordering are called its* parents. *The* width of a node *in an ordered graph is its number of parents. The* width of an ordering *d of G, denoted $w_d(G)$ (or w_d for short) is the maximum width over all nodes. The* width of a graph *is the minimum width over all the orderings of the graph.*

Example 5.2.2 Figure 5.2 presents a graph G over six nodes, along with three orderings of the graph: $d_1 = (F, E, D, C, B, A)$, its reversed ordering $d_2 = (A, B, C, D, E, F)$, and $d_3 = (F, D, C, B, A, E)$. Note that we depict the orderings from bottom to top, so that the first node is at the bottom of the figure and the last node is at the top. The arcs of the graph are depicted by the solid lines. The parents of A along d_1 are $\{B, C, E\}$. The width of A along d_1 is 3, the width of C along d_1 is 1, and the width of A along d_3 is 2. The width of these three orderings are: $w_{d_1} = 3$, $w_{d_2} = 2$, and $w_{d_3} = 2$. The width of graph G is 2. □

Definition 5.2.3 (induced width) *The* induced width of an ordered graph *(G, d), denoted $w^*{}_d$), is the width of the* induced ordered graph *along d obtained as follows: nodes are processed from last to first; when node v is processed, all its parents are connected. The* **induced width** *of a graph, denoted by w^*, is the minimal induced width over all its orderings. Formally*

$$w^*(G) = \min_{d \in orderings} w^*{}_d(G)$$

Example 5.2.4 Consider again Figure 5.2. For each ordering d, (G, d) is the graph depicted without the broken edges, while (G^*, d) is the corresponding induced graph that

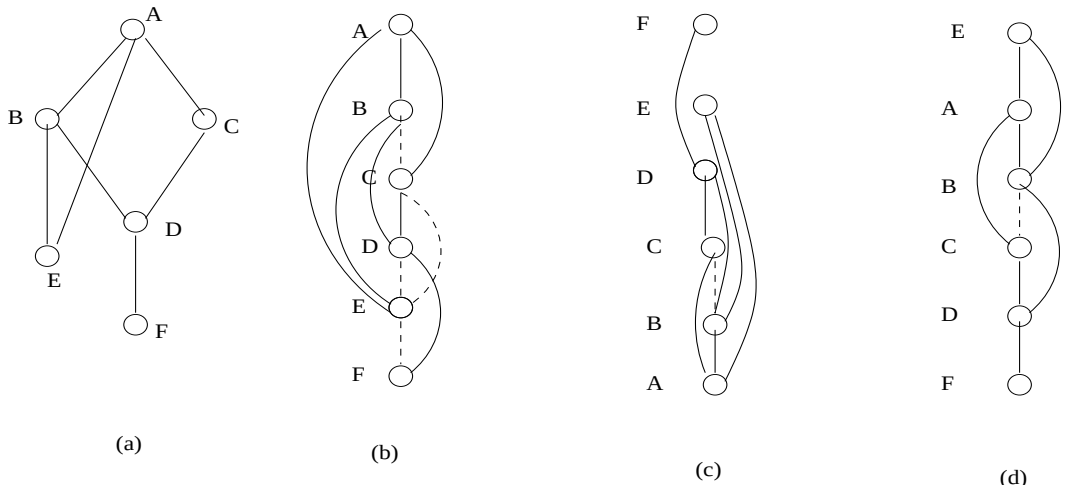


Figure 5.2: (a) Graph G, and three orderings of the graph; (b) $d_1 = (F, E, D, C, B, A)$, (c) $d_2 = (A, B, C, D, E, F)$, and (d) $d_3 = (F, D, C, B, A, E)$. Broken lines indicate edges added in the induced graph of each ordering.

includes the broken edges. We see that the induced width of B along d_1 is 3, and that the overall induced width of this ordered graph is 3. The induced widths of the graph along orderings d_2 and d_3 both remain 2, and, therefore, the induced width of the graph G is 2. □

Trees.

A rather important observation is that a graph is a tree (has no cycles) iff it has an ordering whose width is 1. The reason a width-1 graph cannot have a cycle is because, for any ordering, at least one node on a cycle would have two parents, thus contradicting presumption of having a width-1 ordering. 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, according to the rooted tree, every parent node precedes its child nodes, has a width of 1. Notice that 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 5.2.5 *A graph is a tree iff it has width and induced width of 1.* □

Finding a minimum-width ordering of a graph can be accomplished by the greedy algorithm *min-width* (see Figure 5.3). The algorithm orders variables from last to first as follows: in the first step, a variable with minimum degree is selected and placed 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 5.2(c) could have been generated by a min-width ordering.

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 V 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 5.3: The min-width (MW) ordering procedure

Proposition 5.2.6 *[31] Algorithm min-width* (MW) *finds a minimum width ordering of a graph and its complexity is $O(|E|)$ when E are the edges in the graph.* □

Though finding the min-width ordering of a graph is easy, finding the minimum *induced width* of a graph is hard (NP-complete [4]). Nevertheless, deciding whether there exists an ordering whose induced width is less than a constant k, takes $O(n^k)$ time [52]. The property follows from the fact that a graph having a treewidth bounded by k must have a seperator of size bounded by $k + 1$ such that the resulting components created when the separator is removed each has a tree-width of size k. Enumerating all subgraph of size k in order to consider them as appropriate separators is $O(n^k)$. For details see [52] and also [60].

A decent greedy algorithm, obtained by a small modification to the min-width algorithm, is the *min-induced-width* (MIW) algorithm (Figure 5.4). 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 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 5.2(c) could have been generated by a min-induced-width ordering of G. In this case, it so happens that the algorithm achieves w^*, the minimum induced width of the graph.

Another variation yields a greedy algorithm known as *min-fill*. It uses the *min-fill set*, that is, the number of edges needed to be filled so that the node's parent set be fully connected, as an ordering criterion. This *min-fill* heuristic described in Figure 5.5, was demonstrated empirically to be somewhat superior to min-induced-width algorithm [38]. The ordered graph in Figure 5.2(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 5.4: The min-induced-width (MIW) procedure

What is the complexity of MIW and MIN-Fill? It is easy to see that their complexity is bounded by $O(n^3)$.

The notions of width and induced width and their relationships to various graph parameters, have been studied extensively and will be briefly discuss next.

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 5.5: The min-fill (MIN-FILL) procedure

5.3 Chordal graphs

For some special graphs such as chordal graphs, computing the induced-width 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 5.2(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) is facilitated by using yet another ordering procedure called the *max-cardinality ordering* [61]. 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. Ordering d_2 in Figure 5.2(c) is a max-cardinality ordering.

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 identify the maximal size of a clique). Notice that there are at most n maximal 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 5.3.1 *If G is the induced graph of a graph G, along some ordering d, then G is chordal.* □

Proof: One way to show this is to realize that the ordering d can be realized by a max-cardinality ordering of G. ∎

Example 5.3.2 We see again that G in Figure 5.2(a) is not chordal since the parents of A are not connected in the max-cardinality ordering in Figure 5.2(d). If we connect B and C, the resulting induced graph is chordal. □

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 5.6: The max-cardinality (MC) ordering ordering procedure

Proposition 5.3.3 *[61] Given a graph $G = (V, E)$ the complexity of max-cardinality search is $O(n + m)$ when $|V| = n$ and $|E| = m$.*

Definition 5.3.4 (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. A partial k-tree is a k-tree having some of its arcs removed. Namely it will clique of size smaller than k.*

5.4 From linear orders to tree orders

5.4.1 Elimination trees

We can associate with any ordering with a partial order that can be captured by a tree rather than by a chain. This tree is called an *elimination tree* and is related to the notion of a *pseudo-tree* defined subsequently. Given an ordering d of a graph G, we can designate as a parent of node n its closest earlier neighbor in the induced order graph graph (G^*, d). Fromally,

Definition 5.4.1 (elimination trees) *Given an ordered induced graph (G^*, d), its elimination tree has the nodes of G and every node has a directed arc from its preceding neighbor in the induced graph (G^*, d) that appears latest in the order. The root of the tree is the first node in the ordering. The height (or depth) of an ordering is the height (or depth) of its elimination tree.*

Example 5.4.2 consider the graph in Figure 5.7(a). An elimination tree for the ordering $d = (1, 2, 3, 4, 7, 5, 6)$ is depicted in Figure 5.7(b). □

5.4.2 Pseudo Trees

It turns out that elimination trees of a graph are a special case of what is known as a *pseudo-trees* of a graph. As we will show, a pseudo-tree includes *depth-first search (dfs)* spanning trees of a graph as a special case.

Definition 5.4.3 (pseudo tree, extended graph) *[32] Given an undirected graph $G = (V, E)$, a directed rooted tree $T = (V, E')$ defined on all its nodes is a* pseudo tree *if any arc of G which is not included in E' is a back-arc in T, namely it connects a node in T to an ancestor in T. The arcs in E' may not all be included in E. Given a pseudo tree T of G, the* extended graph *of G relative to T includes also the arcs in G that are not in T. Namely the extended graph is defined as $G^T = (V, E \cup E')$.*

It is well known [28] that a spanning tree of G that is generated by a DFS traversal of the graph, has all its non-tree arcs as back-arcs.

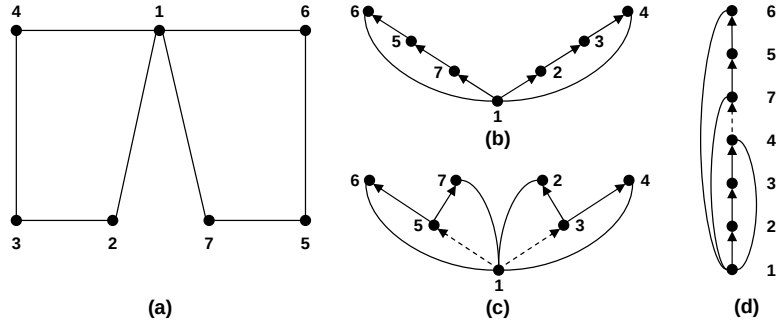


Figure 5.7: (a) A graph; (b) a DFS tree $\mathcal{T}_1$; (c) a pseudo tree $\mathcal{T}_2$; (d) a chain pseudo tree $\mathcal{T}_3$

Clearly, a tree generated by a dfs traversal of a graph is a pseudo-tree whose arcs are all included in G.

Example 5.4.4 Consider the graph G displayed in Figure 5.7(a). Ordering $d_1 = (1, 2, 3, 4, 7, 5, 6)$ is a DFS ordering of a DFS tree T_1 having the smallest DFS tree depth of 3 (Figure 5.7(b)). The tree T_2 in Figure 5.7(c) is a pseudo tree and has a tree depth of 2 only. The two tree-arcs (1,3) and (1,5) are not in G. The tree T_3 in Figure 5.7(d), is a chain. The extended graphs G^{T_1}, G^{T_2} and G^{T_3} are presented in Figure 5.7(b),(c),(d) when we ignore directionality and include the dotted arcs. The tree T_1 is also an elimination tree along the order $d = (1, 2, 3, 4, 7, 5, 6)$. T_2 is an elimination tree along $d = (1, 3, 4, 2, 5, 7, 6)$. However the chain T_3 is not an elimination tree because 7 is not connected to 4 in the induced graph along that ordering. □

Proposition 5.4.5 *An elimination-tree is a pseudo-tree.*

Proof: We have to show that a non-tree arcs in the elimination-tree are back-arcs (left as exercise) ■

Generating low height elimination trees It is desirable to get orderings that have a small height of their pseudo-tree. Finding a minimal height ordering is also known to be hard. We can use a DFS tree traversal of the induced-width of a graph to generate a small tree. The tree height is dependant on the initial ordering used. Note that every

elimination-tree is a pseudo-tree. Generating an elimination-tree is easy since they are defined constructively, and they are also pseudo-trees.

Proposition 5.4.6 *1. Any DFS tree of the induced graph (G^*, d) is a pseudo-tree. 2. There exists a DFS ordering of the induced-graph that is an elimination-tree.*

Proof: Exercise ∎

Another greedy algorithm called hypergraph decomposition aims to generate small hight elimination orderings and is sometime also used to generate small induced-width. It is based on the recursive decomposition of the dual hypergraph and it uses the notion of hypergraph separator.

Definition 5.4.7 (hypergraph separators) *Given a dual hypergraph $\mathcal{H} = (\mathbf{V}, \mathbf{E})$ of a graphical model, a* hypergraph separator decomposition *is a pair $\langle \mathcal{H}, \mathcal{S} \rangle$ where: $\mathcal{S} \subset \mathbf{E}$, and the removal of $\mathcal{S}$ separates $\mathcal{H}$ into more than 2 disconnected components;*

The algorithm works by partitioned the hypergraph into two balanced (roughly equal-sized) parts, while minimizing the number of hyperedges across. A small number of crossing edges translates into a small number of variables shared between the two sets of functions. Each part is then recursively partitioned in the same fashion, until they contain a single vertex. This yields a tree of hypergraph separators where each separator corresponds to a subset of variables chained together.

Since the above hypergraph partitioning heuristic can select the separations in many ways it is a non-deterministic algorithm (see software `hMeTiS`[1].) and the height and induced width of the resulting elimination tree may vary significantly from one run to the next.

In Table5.1 we illustrate the induced width and depth of the elimination tree obtained with the hypergraph and min-fill heuristics for 10 Bayesian networks graphs from the Bayesian Networks Repository[2] and 10 constraint networks graphs derived from the SPOT5 benchmark [8]. It was generally observed that the min-fill heuristic generates lower induced width elimination trees, while the hypergraph heuristic produces much smaller height elimination trees.

[1]Available at: http://www-users.cs.umn.edu/karypis/metis/hmetis
[2]Available at: http://www.cs.huji.ac.il/labs/compbio/Repository

Network	hypergraph		min-fill		Network	hypergraph		min-fill	
	width	depth	width	depth		width	depth	width	depth
barley	7	13	7	23	spot_5	47	152	39	204
diabetes	7	16	4	77	spot_28	108	138	79	199
link	21	40	15	53	spot_29	16	23	14	42
mildew	5	9	4	13	spot_42	36	48	33	87
munin1	12	17	12	29	spot_54	12	16	11	33
munin2	9	16	9	32	spot_404	19	26	19	42
munin3	9	15	9	30	spot_408	47	52	35	97
munin4	9	18	9	30	spot_503	11	20	9	39
water	11	16	10	15	spot_505	29	42	23	74
pigs	11	20	11	26	spot_507	70	122	59	160

Table 5.1: Bayesian Networks Repository (left); SPOT5 benchmarks (right).

5.5 Tree-decompositions

Another important parameter that is highly related to the induced-width called *treewidth*. It aims at capturing how close a graph is to a tree by embedding the graph in a tree of clusters, yielding a *tree-decomposition*.

Definition 5.5.1 (tree decomposition) *A* tree decomposition *of a hypergraph* $H = (X, S)$ *where* $S = \{S_1, \ldots, S_t\}$*, is a tree* $T = (V, E)$ *where* V *is the set of nodes, also called "clusters", and* E *is the set of edges, together with a labeling function* χ *that associates with each vertex* $v \in V$ *a set (a cluster)* $\chi(v) \subseteq X$ *satisfying:*

1. *For each* $S_i \in S$ *there exists a vertex* $v \in V$ *such that* $S_i \subseteq \chi(v)$*;*
2. (running intersection property) *For each* $X_i \in X$*, the set* $\{v \in V \mid X_i \in \chi(v)\}$ *induces a connected subtree of* T*.*

Definition 5.5.2 (treewidth, pathwidth) *The* treewidth *of a tree decomposition of a graph is the size of its largest cluster minus 1. Namely* $tw(T) = \max_v |\chi(v)| - 1$*. The*

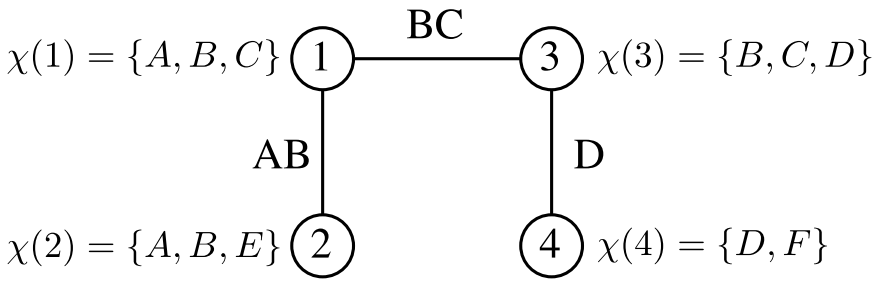


Figure 5.8: Tree decomposition example

treewidth *of a hypergraph is the minimum treewidth over all possible tree-decompositions. The* **pathwidth** *is the treewidth over the restricted class of chain decompositions (namely when T is a chain.)*

Example 5.5.3 We will look at two graph examples. First consider the hypergraph in Figure 5.9(a) where $\mathcal{H} = (V = \{A, B, C, D, E, F\}, S = \{AEF, ABC, CDE, ACE\}$. The tree in Figure 5.9(d) is a depiction of a tree-decomposition, where each hyperedge is its own cluster. You can verify that the property of connectedness is satisfied. In this case we did not need to combine different hyperedges into a single cluster. The actual hypergraph is already a (hyper)tree. Consider now the second example in Figure 5.2(a). In this case we have a simple graph. A tree-decomposition is given by the tree whose nodes are $V = \{1, 2, 3, 4\}$ and edges are $E = \{(1, 2), (1, 3), (3, 4)$ and $\chi(1) = \{A, B, C\}$, $\chi(2) = \{A, B, E\}$, $\chi(3) = \{B, C, D\}$. $\chi(4) = \{D, F\}$. See Figure 5.8 □

How can we construct a tree decomposition? Interestingly, we can show that the set maximal cliques of an induced-graph provides a tree decomposition of the graph. Namely, the maximal cliques of the induced graph can be connected into a tree structure that satisfies the running intersection property (property 2 of definition 5.5.1). This is because an induced-graph is a chordal graph. All that remains is to show then is that the maximal cliques of a chordal graph provide a tree-decomposition.

Proposition 5.5.4 *Given a hypergaph $H = (X, S)$ whose primal graph G is chordal, the maximal cliques of G constitute a tree-decomposition.*

Proof: Let's use a maximal cardinality order d of the chordal graph. It is easy to identify the maximal cliques and enumerate them going from the last to the first node. Let this order be $C = \{C_1, ...C_r\}$. Then the tree $T = (C, E)$, is built as follows. For every j connect C_j to C_k, where $k < j$ iff clique C_k shares a maximum number of variables with C_j. Clearly property 1 of the tree decomposition definition is satisfied by the tree T since ever $S_i \in H$ appears as a clique in the primal graph G and is therefore subsumed in one of the maximal cliques. property 2, the running intersection property, is also satisfied. This is less straightforward and we leave it to the reader to explore (see literature on chordal graphs). ■

Consequently, a popular way of generating a tree-decomposition of a hypergraph is to embed it in an induced-graph that is chordal (i.e., select an ordering and generate the induced graph along that ordering) and then select the maximal cliques and connect them into a tree. By definition, the induced width of that ordering and the treewidth of that ordering are identical, both coincide with the size of the maximal clique - 1. Consequently the minimum over all orderings, of the induced-width and the treewidth are identical as well. Indeed, the induced-width and the tree-width for any graph are the same parameter. (For mode see [25]). For various relationships between these and other graph parameters see [4, 34, 13].

Example 5.5.5 Consider the hypergraph $\mathcal{H} = (V, S)$ where $V = \{1, 2, ..., 13\}$ and $S = \{\{1, 2, 3, 4, 5\}, \{3, 6, 9, 12\}, \{12, 13\}, \{5, 7, 11\}, \{8, 9, 10, 11\}, \{10, 13\}$. The primal graph of this hypergraph is given in Figure 5.9(a). The hypergraph is given in 5.9(b), and its dual graph is depicted in Figure 5.9(c). The reader is encouraged to compute the induced width along two orderings and speculate what the treewidth of this hypergraph might be. Suggest a tree-decomposition. □

Definition 5.5.6 (hypertree) *A hypergraph whose primal graph is chordal, and whose maximal ciques correspond to the hyperedges is a hypertree*

Definition 5.5.7 (connectedness, join-trees, hypertrees and acyclic networks) *Given a dual graph of a hypergraph, an arc subgraph of the dual graph satisfies the* connectedness *property iff for each two nodes that share a variable, there is at least one path*

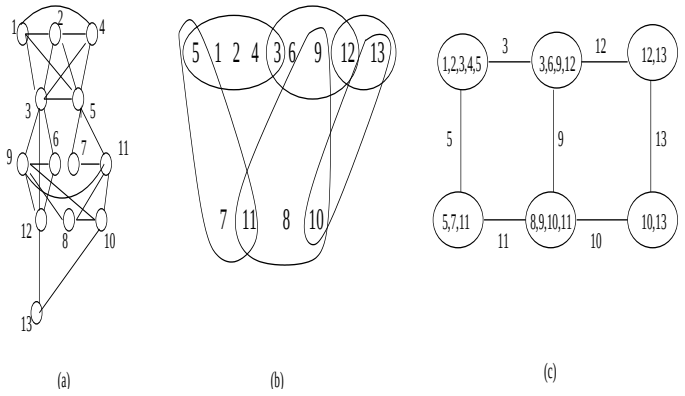


Figure 5.9: A hypergraph represented by (a) a primal graph, (b) a hypergraph, and (c) a dual graph.

of labeled arcs, each containing the shared variables. An arc subgraph of the dual graph that satisfies the connectedness property is called a join-graph. *A join-graph that is a tree is called a* join-tree. *A hyper-tree is also called an* acyclic graph.

It can be shown that

Proposition 5.5.8 *A hypergraph whose dual-graph has a join-tree is a* hypertree.

Example 5.5.9 Considering again Figure 5.1 We see that the arc between (AEF) and (ABC) in Figure 5.1(c) is redundant because variable A also appears along the alternative path $(ABC) - AC - (ACE) - AE - (AEF)$. A consistent assignment to A is thereby ensured by these constraints even if the constraint between AEF and ABC is removed. Likewise, the arcs labeled E and C are also redundant, and their removal yields the graph in 5.1(d). We also see that the join-tree in Figure 5.1(d) satisfies the connectedness property. The hypergraph in Figure 5.1(a) has a join-tree and is therefore a hypertree. □

5.6 The cycle-cutset and w-cutset schemes

We descibed earlier the treewidth parameter as a measure capturing how close is a graph to a tree as reflected via a tree of clusters of nodes. Here we will focus on alternative way of transforming a graph into a tree called *cycle-cutset* and more generally, $w-cutset$,

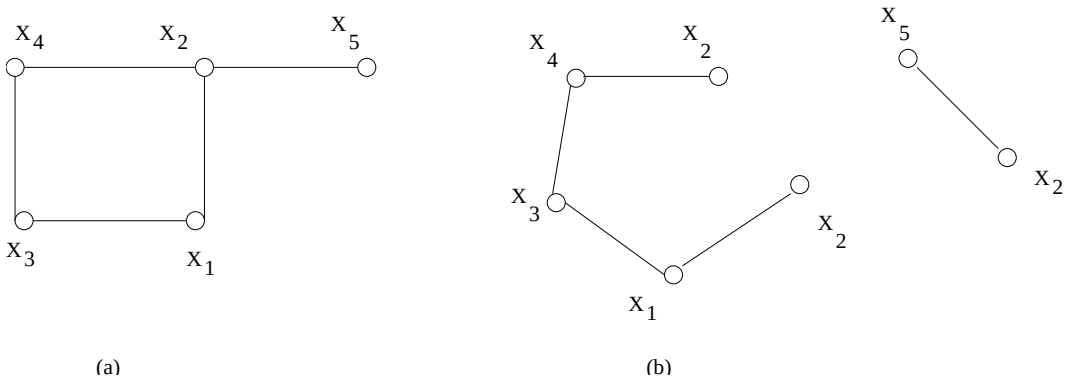


Figure 5.10: An instantiated variable cuts its own cycles.

decomposition. A cutset is a set of nodes that can be removed from the graph. We showed at the end of Chapter 4 that we can consider conditioning on a subset of the variables, namely, assigning them values and solve the rest of the problem as if its graph does not include the cutset. Cutset-decomposition is a class of algorithms that we will focus on in later chapters. The graph parameter that controls its promise and complexity is w if we use a $w - cutset$ decomposition.

Definition 5.6.1 (cycle-cutset,w-cutset) *Given a graph G, a subset of nodes is called a w-cutset iff when removed from the graph the resulting graph has an induced-width less than or equal to w. A minimal w-cutset of a graph has a smallest size among all w-cutsets of the graph. A cycle-cutset is a 1-cutset of a graph.*

A cycle-cutset is known by the name a *feedback vertex set* and it is known that finding the minimal such set is NP-complete [33]. However, we can always settle for approximations, provided by greedy schemes. Cutset-decomposition schemes call for a new optimization task on graphs:

Definition 5.6.2 (finding a minimal w-cutset) *Given a graph $G = (V, E)$ and a constant w, find a smallest subset of nodes U, such that when removed, the resulting graph has induced-width less than or equal w.*

Finding a minimal w-cutset is hard, but various greedy heuristic algorithms were investigated. In particular several greedy algorithms for the special case of cycle-cutset can

be found [2]. The general task of finding a minimal w-cutset was addressed in recent papers [29, 10]. Note that verifying that a given subset of nodes is a w-cutset can be accomplished in polynomial time (linear in the number of nodes), by deleting the candidate cutset from the graph and verifying that the remaining graph has an induced width bounded by w [4].

It can be shown that the size of the smallest cycle-cutset (1-cutset), c_1^* and the smallest induced width, w^*, obey the inequality $c_1^* \geq w^* - 1$. Therefore, $1 + c_1^* \geq w^*$. In general, if c_w^* is the size of a minimal w-cutset then,

Theorem 5.6.3

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

Proof: exercise ∎

5.7 Summary and Bibliographical Notes

5.8 Exrecises

1. Show what is the factor graph of the hypergraph in Figure 5.1?
2. Analyze the complexity of the the greedy algorithms, MW, MIW, MF, MC, appearing in this chapter.

Chapter 6

Tree-Clustering Schemes

In this chapter, we take the bucket elimination algorithm a step further. We will show that bucket-elimination can be viewed as an algorithm that send messages along a tree (the bucket-tree). The algorithm can then be augmented with a second set of messages passed from bottom to top, yielding a message-passing schemes that belongs to the class of *cluster tree elimination* algorithms.

These latter methods have received different names in different research areas, such as join-tree clustering or junction-tree algorithms, clique-tree clustering and hyper-tree decompositions. We will refer to all these as cluster-tree processing schemes over *tree-decompositions*. Our algorithms are applicable to a general reasoning problem described by $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes, \Downarrow \rangle$, where the first 4 elements identify the graphical model and the fifth identifies the reasoning task (see definition 2.1.2). However, we will assume the specific probabilistic networks when developing the algorithms and the reader can just make the appropriate generalization.

6.1 Bucket-Tree Elimination

The bucket-elimination algorithm, *BE-bel* (see Figure 4.4) for belief updating is designed to compute the belief of the first node in a given ordering given all the evidence, and the probability of evidence. However, it is often desirable to answer the belief query for each and every variable in the network. A brute-force approach will require running *BE-Bel* n

times, each time with a different variable at the start of the ordering. We will show next that this is unnecessary. By viewing bucket-elimination as a message passing algorithm along a rooted *bucket-tree*, we can augment it with a second message passing phase in the opposite direction, from root to leaves, achieving the same goal.

Example 6.1.1 Consider the Bayesian network defined over the directed acyclic graph (DAG) in Figure 2.4(a). Figure 6.1a shows the initial buckets along ordering $d = (A, B, C, D, F, G)$ and the messages, labeled by λ, that will be passed by BE from top to bottom. Figure 6.1b depicts the same computation as message-passing along a tree which we will refer to as a *bucket-tree*. Notice that in the figure the ordering is displayed from the bottom up (A, the first variable, is at the bottom and G, the last one, is at the top), and the messages are passed top down. This computation results in the belief in A, $bel(A) = P(A|G = 1)$, and consulted only functions that reside in the bucket of A. What if we now want to compute $bel(D)$? We can start the algorithm using a new ordering such as (D, A, B, C, F, G). Alternatively, rather then doing all the computations from scratch using a different variable ordering whose first variable is D, we can take the bucket tree and re-orient the edges to make D the root of the tree. Reorienting the tree so that D is the root, requires reversing only 2 edges, (B, D) and (A, B), suggesting that we only need to recompute messages from node A to B and from B to D. We can think about a new virtual partial order expressed as $(\{D, A, B\}, C, F, G)$, namely, collapsing the buckets of B, A and D into a single bucket and therefore ignoring their internal order. By definition, we can compute the belief in D by the expression

$$bel(d) = \alpha \sum_a \sum_b P(a) \cdot p(b|a) \cdot P(d|a, b) \cdot \lambda_{C \to B}(b) \tag{6.1}$$

Likewise, we can also compute the belief in B by

$$bel(b) = \alpha \sum_a \sum_d P(a) \cdot p(b|a) \cdot P(d|a, b) \cdot \lambda_{C \to B}(b) \tag{6.2}$$

This computation can be carried over the bucket-tree, whose downward messages were already passed, in 3 steps. The first executed in bucket A, where the function $P(A)$ is moved to $bucket_B$, the second is executed by $bucket_B$, computing a function (a product) that is moved to $bucket_D$. The final computation is carried in $bucket_D$. Denoting the

new reverse messages by π, a new $\pi_{A\to B}(a) = P(A)$, is passed from $bucket_A$ to $bucket_B$. Then, an intermediate function is computed in $bucket_B$, to be sent to $bucket_D$, using the messages received from $bucket_C$ and $bucket_A$ and its own function,

$$\pi_{B\to D}(a,b) = p(b|a) \cdot \pi_{A\to B}(a) \cdot \lambda_{C\to B}(b)$$

Finally the belief is computed in $bucket_D$ using its current function content by

$$bel(d) = \alpha \sum_{a,b} P(d|a,b) \cdot \pi_{B\to D}(a,b). \tag{6.3}$$

This accomplishes the computation of the algebraic expression in Equation 6.1. You can see some of these messages depicted in Figure 6.2a. The belief in B can also be computed in $bucket_D$. However, if we want each bucket to compute its own belief, $bucket_D$ can send $P(D|A,B)$ to $bucket_B$ and the computation of Equation 6.2 can be carried out there, autonomously. □

The example generalizes. We can compute the belief for every variable by a second message passing from the root to the leaves along the bucket-tree, such that at termination the belief for each variable can be computed locally, n each bucket, consulting only the functions in its own bucket.

In the following we will describe the idea of message passing along the bucket-tree assuming sum-product networks. However, the description is applicable to general reasoning task when $\otimes$ replaces $\prod$ and $\Downarrow$, the $\sum$.

Let $\mathcal{M}$ be a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod, \sum \rangle$ and d an ordering of its variables $X_1, ..., X_n$. Let $B_{X_1}, ..., B_{X_n}$ denote a set of buckets, one for each variable. We will use B_{X_i} and B_i interchangeably. Each bucket B_i contains those functions in F whose latest variable in ordering d is X_i (i.e., according to the bucket-partitioning rule). As before, we denote by ψ_i the product of functions in B_i. A bucket-tree of a model $\mathcal{M} =, X, D, F, \prod >$ along d has buckets as its nodes. Bucket B_X is connected to bucket B_Y if the function generated in bucket B_X by BE is placed in B_Y. The variables of B_X are those appearing in the scopes of any of its new or old functions. Therefore, in a bucket-tree, every vertex B_X other than the root has one parent vertex B_Y and possibly several child vertices $B_{Z_1}, ..., B_{Z_t}$.

The structure of the bucket-tree can also be extracted from the induced-ordered graph of $\mathcal{M}$ along d using the following definition. (For details consult Chapter 5.)

Definition 6.1.2 (bucket-tree, separator, eliminator) *Let $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod \rangle$ be a graphical model whose primal graph is G, and let $d = (X_1, ..., X_n)$ be an ordering of its variables. Let (G^*, d) be the induced graph along d of G.*

- *The bucket tree has the buckets denoted $\{B_i\}_{i=1}^n$ as its nodes. Each bucket* contains *a set of functions and a set of variables. The functions are those placed in the bucket according to the bucket partitioning rule. ψ_i denotes the combined input functions in B_i. The set of variables in B_i is X_i and all its parents in (G^*, d), denoted $scope(B_i)$. Each vertex B_i points to B_j (or, B_j is the parent of B_i) if X_j is the closest neighbor of X_i that appear before it in (G^*, d).*

- *If B_j is the parent of B_i in the bucket-tree, then the separator of X_i and X_j, $sep(i, j) = scope(B_I) \cap scope(B_J)$.*

- *Given a directed edge (B_i, B_j) in the bucket-tree, $elim(i, j)$ is the set of variables in B_i and not in B_j, namely $elim(B_i, B_j) = scope(B_i) - sep(B_i, B_j)$. We will call this set "the eliminator from B_i to B_j."*

Algorithm *bucket-tree-elimination (BTE)* presented in Figure 6.3 includes the two message passing phases along the bucket-tree. Notice that we always assume that the input may contain also a set of evidence nodes because this is typical to a large class of problems in probabilistic networks. Given an ordering of the variables, the first step of the algorithm generates the bucket-tree by partitioning the functions into buckets and connecting the buckets into a tree. The subsequent *top-down* phase is identical to general bucket-elimination. The *bottom-up* messages are defined as follows. The messages sent from the root up to the leaves will be denoted by π. The message from B_j to a child B_i is generated by multiplying all the bucket's function ψ_j by the π messages from its parent bucket and all the λ messages from its *other* child buckets and marginalizing (e.g., summing) over the eliminator from B_j to B_i. By construction, downward messages are generated by eliminating a single variable. Upward messages, on the other hand, may be generated by eliminating zero, one or more variables.

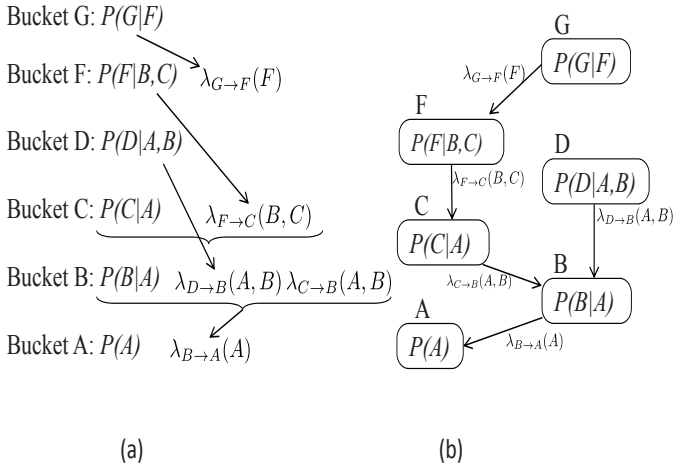


Figure 6.1: Execution of BE along the bucket-tree

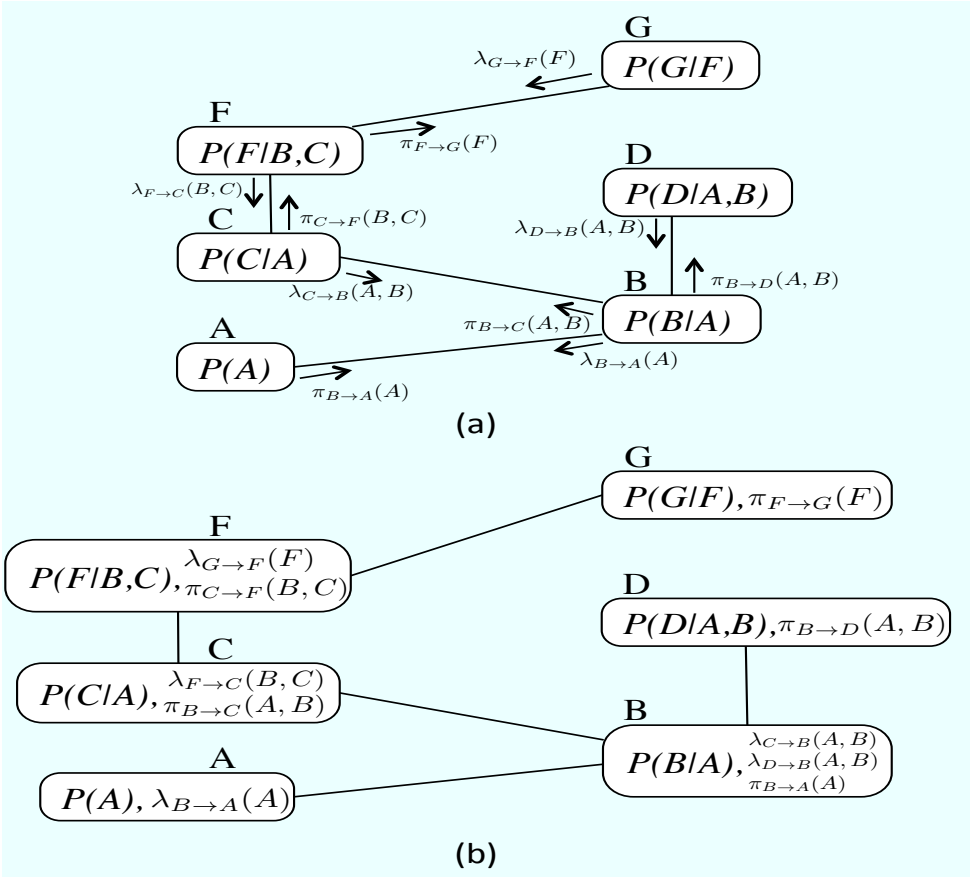


Figure 6.2: Propagation of π's and λ's along the bucket-tree (a), the augmented output bucket-tree (b)

When BTE terminates, each output bucket B'_i contains the $\pi_{j\to i}$ it received from its parent B_j, its own function ψ_j and the $\lambda_{k\to i}$ messages sent from each child B_k. Then, each bucket can compute its belief over all the variables in its bucket, by combining all the functions in a bucket as specified in step 3 of BTE. It can then compute also the belief on single variables and the probability of evidence. See the procedure in Figure 6.11.

ALGORITHM BUCKET-TREE ELIMINATION (BTE)

Input: A problem $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod \rangle$, ordering d. $X = \{X_1, ..., X_n\}$ and $F = \{f_1, ..., f_r\}$ Evidence $E = e$.

Output: Augmented buckets $\{B'_i\}$, containing the original functions and all the π and λ functions received from neighbors in the bucket-tree.

1. **Pre-processing:** Partition functions to the ordered buckets as usual and generate the bucket-tree.
2. **Top-down phase:** λ messages (BE) **do**
   for $i = n$ to 1, in reverse order of d process bucket B_i:
   The message $\lambda_{i\to j}$ from B_i to its parent B_j, is:
   $\lambda_{i\to j} \Leftarrow \sum_{elim(i,j)} \psi_i \cdot \prod_{k\in child(i))} \lambda_{k\to i}$
   endfor
3. **bottom-up phase:** π **messages**
   for $j = 1$ to n, process bucket B_j **do**:
   B_j takes $\pi_{k\to j}$ received from its parent B_k, and computes a message $\pi_{j\to i}$ for each child bucket B_i by
   $\pi_{j\to i} \Leftarrow \sum_{elim(j,i)} \pi_{k\to j} \cdot \psi_j \cdot \prod_{r\neq i} \lambda_{r\to j}$
   endfor
4. **Output:** and answering singleton queries (e.g., deriving beliefs).
   Output augmented buckets $B'_1, ..., B'_n$, where each B'_i contains the original bucket functions and the λ and π messages it received.

Figure 6.3: Algorithm Bucket-Tree Elimination

COMPUTING MARGINAL BELIEFS

Input: a bucket-tree processed by BTE with augmented buckets: $B'_1, ..., B'_n$

output: beliefs of each variable, bucket and probability of evidence.

$bel(B_i) \Leftarrow \prod_{f \in B'_i} f$

$bel(x_i) \Leftarrow \sum_{B_i - \{X_i\}} \prod_{f \in B'_i} f$

P(evidence) $\Leftarrow \sum_{B_i} \cdot \prod_{f \in B'_i} f$

Figure 6.4: Query answering

Example 6.1.3 Figure 6.2(a) shows the complete execution of BTE along the bucket-tree. The π and λ messages are placed on the outgoing upward arcs. The π functions in the bottom-up phase are computed as follows (the first 3 were demonstrated earlier):

$\pi_{A \to B}(a) = P(a)$

$\pi_{B \to C}(c, a) = P(b|a)\lambda_{D \to B}(a, b)\pi_{A \to B}(a)$

$\pi_{B \to D}(a, b) = P(b|a)\lambda_{C \to B}(a, b)\pi_{A \to B}(a, b)$

$\pi_{C \to F}(c, b) = \sum_a P(c|a)\pi_{B \to C}(a, b)$

$\pi_{F \to G}(f) = \sum_{b,c} P(f|b, c)\pi_{C \to F}(c, b)$

The actual output (the augmented buckets) are shown in Figure 6.2(b). □

Extending the view of the above algorithms for any reasoning task over graphical models, we can show that when BTE terminates we have in each bucket all the information needed to answer any automated reasoning task on the variables appearing in that bucket. In particular, we do not need to look outside a bucket to answer a belief query. We call this property "explicitness". It is sometime referred to also as *minimality* or *decomposability* [45].

Definition 6.1.4 (explicit function and explicit sub-model) *Given a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod \rangle$, and reasoning tasks defined by marginalization $\sum$ and given a subset of variables Y, $Y \subseteq \mathbf{X}$, we define $\mathcal{M}_Y$, the explicit function of $\mathcal{M}$ over Y:*

$$\mathcal{M}_Y = \sum_{X - Y} \prod_{f \in F} f, \tag{6.4}$$

We denote by F_Y any set of functions whose scopes are subsumed in Y over the same domains and ranges as the functions in $\mathbf{F}$. *We say that (Y, F_Y) is an explicit submodel of M iff*

$$\prod_{f \in F_Y} f = \mathcal{M}_Y \tag{6.5}$$

Theorem 6.1.5 (Completeness of BTE) *Given $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod, \sum \rangle$ and evidence $E = e$, when algorithm BTE terminates, each output bucket is explicit relative to its variables. Namely, for each B_i, $\Pi_{f \in B'_i} f = \mathcal{M}_{scope(B_i)}$.*

The correctness of BTE will be shown as a special case from the correctness of a larger class of tree propagation algorithms that we present in the following section. We next address the complexity of BTE, comparing it, in particular, with n executions of BE.

Theorem 6.1.6 (Complexity of BTE) *Let w^* be the induced width of (G^*, d) where G is the primal graph of $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod, \sum \rangle$, r be the number of functions in* $\mathbf{F}$ *and k be the maximum domain size. The time complexity of BTE is $O(r \cdot deg \cdot k^{w^*+1})$, where deg is the maximum degree of a node in the bucket-tree. The space complexity of BTE is $O(n \cdot k^{w^*})$.*

Proof: As we previously showed, the downward λ messages take $O(r \cdot k^{w^*+1})$ steps. This simply is the complexity of BE. The upward messages per bucket are computed for each of its child nodes. Each such message takes $O(r_i \cdot k^{w*+1})$ steps,where r_i is the number of functions in bucket B_i, yielding a time complexity per upward bucket of $O(r_i \cdot deg \cdot k^{w*+1})$. Summing over all buckets we get complexity of $O(r \cdot deg \cdot k^{w^*+1})$. Since the size of each downward message is k^{w^*} we get space complexity of $O(n \cdot k^{w^*})$. $\square$.

The complexity of BTE can be improved to be $O(rk^{w^*+1})$ time and $O(nk^{w^*+1})$ space [37], (do as an exercise).

In theory the speedup expected from running BTE vs. running BE n times is at most n. This may seem insignificant compared with the exponential complexity in w^*, however it can be very significant in practice, especially when n is large. Beyond the saving in computation, the bucket-tree provides an architecture for distributed computation of the algorithm when each bucket is implemented by a different cpu.

Asunchronous Bucket-tree propagation

Algorithm *BTE* can also be described without committing to a particular schedule when viewing the bucket-tree as an undirected and by unifying the up and down messages into a single message-type denote by λ. In this case each bucket receives a λ message from each of its neighbors and each sends a λ message to every neighbor. This distributed algorithm, called Bucket Tree Propagation or *BTP*, is written for a single bucket described in Figure 6.5. It is easy to see that the algorithm is correct. It sends at most 2 messages on each edge in the tree (computation starts from the leaves of the tree.

Theorem 6.1.7 (Completeness of BTP) *Algorithm BTP terminates generating explicit buckets.*

Proof: The proof of *BTP* correctness follows from the correctness of *BTE*. All we need to show is that at termination the buckets' content in *BTE* and *BTP* are the same. (Prove as an exercise.) ∎

BUCKET-TREE PROPAGATION (BTP)

Input: A problem $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod \rangle$, ordering d. $X = \{X_1, ..., X_n\}$ and $F = \{f_1, ..., f_r\}$, $E = e$. An ordering d and a corresponding bucket-tree structure, in which for each node X_i, its bucket B_i and its neighboring buckets are well defined.

Output: Explicit buckets.

1. **for** bucket B_i **do:**
2. **for** each neighbor bucket B_j **do**,
   once all messages from all other neighbors were received, **do**
   compute and send to B_j the message
   $\lambda_{i \to j} \Leftarrow \sum_{elim(i,j)} \psi_i \cdot (\prod_{k \neq j} \lambda_{k \to i})$
3. **endfor**

Figure 6.5: Algorithm Bucket-tree propagation (BTP)

6.2 From bucket-trees to cluster-trees

Algorithms BTE and its synchronous version BTP are special cases of a class of algorithms that operates over a tree-decomposition of the graphical model. A tree-decomposition takes a graphical model and embeds it in a tree where each node in the tree is a cluster of variables and functions. The decomposition allows message-passing between the clusters in a manner similar to BTE and BTP.

Some graphical models are inherently tree-structured. Namely, their input functions already has a dependency structure that can be captured by a tree-like graph (with no cycles). Such graphical models are called *Acyclic Graphical models* [42]. We will describe acyclic models first and then show how a tree-decomposition can impose a tree structure on non-acyclic graphical models as well.

6.2.1 Acyclic graphical models

As we know, a graphical model can be associated with a dual graph which provides an alternative view of the graphical model. In this view each function resides in its own node can be viewed as meta variables and arcs indicate equality constraints between shared variables. So, if a graphical model's dual graph happens to be a tree, it can be solved in linear time using a BTE-like message-passing algorithm over the dual graph.

Proposition 6.2.1 *BTE is linear for graphical model having a dual graph that is a tree. (prove as an exercise.)*

The situation is even nicer because, in some cases the dual graph seems to not be a tree, but it is. Some of its arcs can be removed while not violating the original independency relationships that is captured by the graph. Such arcs can be viewed as redundant. Namely, they express a dependency between two nodes that is already captured by an alternative path. Alternatively we can say that arcs are redundant if their removal does not violate its *i-mapness*. Namely conditional independence through the graph separation properties is maintained (see [47]).

Example 6.2.2 Refer back to Figure 5.1. We see that the arc between (AEF) and (ABC) in Figure 5.1(c) expresses redundant dependency because variable A also appears

along the alternative path $(ABC) - AC - (ACE) - AE - (AEF)$. In other words, a dependency between AEF and ABC relative to A is maintained through the path even if they are not directly connected. Likewise, the arcs labeled E and C are also redundant. Their removal yields the tree in 5.1(d) which we call a join-tree. This reduced dual graph in 5.1(d) satisfies a property called connectedness which we already defined in Chapter 5, but will include here again for convenience. □

We next define the *connectedness* property. We say that a graph G' is an *arc subgraph* of graph G if it contains the same set of nodes as G and a subset of its arcs.

Definition 6.2.3 (connectedness, join-trees) *(see also 5.5.1). Given a dual graph of a hypergraph, an arc subgraph of the dual graph satisfies the* connectedness *property iff for each two nodes that share a variable, there is at least one path of labeled arcs of the dual graph such that each contains the shared variables. An arc subgraph of the dual graph that satisfies the connectedness property is called a* join-graph. *A join-graph that is a tree is called a* join-tree.

We can now formally define acyclic graphical models.

Definition 6.2.4 (acyclic networks) *A graphical model , $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \rangle$, whose dual graph is a tree or has a join-tree is called an* acyclic graphical model.

Example 6.2.5 Considering again the graphs in Figure 5.1, we can see that the join-tree in Figure 5.1(d) satisfies the connectedness property. The model is therefore, acyclic. □

Theorem 6.2.6 *Given an acyclic graphical model, algorithm BTE generates the explicit sub-model (definition 6.1.4) for scopes of every input function, in linear time and space. (See proof in chapter's Appendix.)*

Now that we have established that acyclic graphical models can be solved efficiently, all that remains is to transform a general graphical model into an acyclic one. This task is facilitated by algorithm tree-decomposition.

6.2.2 Tree-decomposition and cluster-tree elimination

We now define *tree-decompositions* of graphical models which facilitate message propagation along a tree of clusters. We will show that a bucket-tree is a special case of tree-decomposition. (See also Chapter 5).

Definition 6.2.7 (tree-decomposition, cluster tree) *Let* $\mathcal{M} =< X, D, F, \prod, \sum >$ *be a graphical model. A* tree-decomposition *of* $\mathcal{M}$ *is a triple* $< T, \chi, \psi >$*, where* $T = (V, E)$ *is a tree, and* χ *and* ψ *are labeling functions which associate with each vertex* $v \in V$ *two sets,* $\chi(v) \subseteq X$ *and* $\psi(v) \subseteq F$ *satisfying:*

1. *For each function* $f_i \in F$*, there is* exactly *one vertex* $v \in V$ *such that* $f_i \in \psi(v)$*, and scope*$(f_i) \subseteq \chi(v)$*.*
2. *For each variable* $X_i \in X$*, the set* $\{v \in V | X_i \in \chi(v)\}$ *induces a connected subtree of* T*. This is also called the running intersection property.*

We will often refer to a node and its functions as a *cluster* and use the term *tree-decomposition* and *cluster tree* interchangeably.

Definition 6.2.8 (treewidth, separator-width, eliminator) *The* treewidth *[4] of a tree-decomposition* $< T, \chi, \psi >$ *is* $max_{v \in V} |\chi(v)|$ *minus 1. Given two adjacent vertices* u *and* v *of a tree-decomposition, the* separator *of* u *and* v *is* $sep(u, v) = \chi(u) \cap \chi(v)$*, and the* eliminator *of* u *with respect to* v *is* $elim(u, v) = \chi(u) - \chi(v)$*. The separator-width is the maximum over all separators.*

Example 6.2.9 Consider the Bayes network in Figure 2.4a. You can verify that any of the trees in Figure 6.6 describe a partition of variables into clusters. We can now place each input function into a cluster that contains its scopes, yielding a legitimate tree-decomposition. For example, Figure 6.6c shows a cluster-tree decomposition with two vertices, and labeling $\chi(1) = \{G, F\}$ and $\chi(2) = \{A, B, C, D, F\}$. Any function with scope $\{G\}$ must be placed in vertex 1 because vertex 1 is the only vertex that contains variable G (placing a function having G in its scope in another vertex will force us to add variable G to that vertex as well). Any function with scope $\{A, B, C, D\}$ or one of its subsets must be placed in vertex 2, and any function with scope $\{F\}$ can be placed either in vertex 1 or 2. □

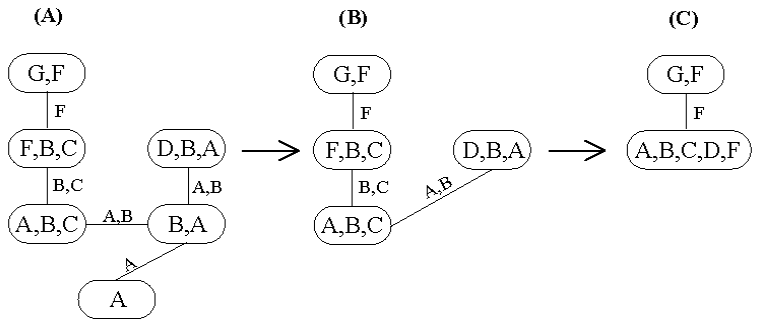


Figure 6.6: From a bucket-tree to join-tree to a super-bucket-tree

Consider now the tree-decomposition at Figure 6.6a. We see that for some nodes $sep(u, v) = \chi(u)$. That is, all the variables in vertex u belong to an adjacent vertex v. In this case the number of clusters in the tree-decomposition can be reduced by merging vertex u into v without increasing the tree-width of the tree-decomposition. This is accomplished by moving from Figure 6.6a to Figure 6.6b.

Definition 6.2.10 (Minimal tree-decomposition) *A tree-decomposition is minimal if* $sep(u, v) \subset \chi(u)$ *and* $sep(u, v) \subset \chi(v)$.

We will next show that:

Theorem 6.2.11 *A bucket-tree of a graphical model* $\mathcal{M}$*, is a tree-decomposition of* $\mathcal{M}$*. (for a proof see the chapter's appendix.)*

We will show now that a tree-decomposition facilitates a message-passing scheme, called *Cluster-Tree Elimination (CTE)*, that is similar to BTE and BTP. The algorithm, is presented in Figure 6.7. Like in BTP, each vertex of the tree sends a function to each of its neighbors. All the functions in a vertex u and all the messages received by u from all its neighbors other than a specific vertex v to which u's message is directed, are combined using the combination operator (e.g., product). The combined function is marginalized over the separator of vertices u and v using the marginalization operator and the projected function is then sent from u to v.

CLUSTER-TREE ELIMINATION (CTE)

Input: A tree decomposition $< T, \chi, \psi >$ for a problem $M =< X, D, F, \prod\} >$, $X = \{X_1, ..., X_n\}$, $F = \{f_1, ..., f_r\}$. Evidence $E = e$, $\psi_u = \prod_{f \in \psi(u)} f$

Output: An augmented tree-decomposition whose clusters are all model explicit. Namely, a tree decomposition $< T, \chi, \bar{\psi} >$ where for $u \in T$, $\chi(u), \bar{\psi}(u)$ is model explicit.

1. **Initialize:** Let $m_{u \to v}$ denote the message sent from vertex u to vertex v.
2. **Compute messages:**
   for every edge (u, v) in the tree, **do**
   If vertex u has received messages from all adjacent vertices other than v, then compute (and send to v)
   $m_{u \to v} \Leftarrow \sum_{sep(u,v)} \psi_u \cdot \prod_{r \in neighbor(u), r \neq v} m_{r \to u}$
   . **endfor**

   Note: functions whose scopes do not contain any separator variable do not need to be combined and can be directly passed on to the receiving vertex.
3. **Return:** The explicit tree-decomposition $< T, \chi, \bar{\psi} >$, where $\bar{\psi}(v) \Leftarrow \psi(v) \cup_{u \in neighbor(v)} \{m_{u \to v}\}$.

Figure 6.7: Algorithm Cluster-Tree Elimination (CTE)

Vertex activation can be asynchronous and convergence is guaranteed. If processing is performed from leaves to root and back, convergence is guaranteed after two passes, where only one message is sent on each edge in each direction. If the tree contains m edges, then a total of $2m$ messages will be sent.

Example 6.2.12 Consider again the graphical model whose primal graph appears in Figure 2.4(a). Assume that all functions are on pairs of variables (you can think of this as a Markov network). Two tree-decompositions are given in Figure 6.8a and 6.8b. For the tree-decomposition in 6.8b we show the propagated messages explicitly in Figure 6.8c. Since cluster 1 contains only one function, the message from cluster 1 to 2 is the marginalization of f_{FG} over the separator between cluster 1 and 2, which is variable F. The message $m_{2\to 3}$ from cluster 2 to cluster 3 combines the functions in cluster 2 with the message $m_{1\to 2}$, and projects over the separator between cluster 2 and 3, yielding $\{B, C\}$, and so on. □

Once all vertices have received messages from all their neighbors we have the explicit clusters and therefore an answer to any singleton marginal query (e.g., beliefs) and a host of other reasoning tasks can be accomplished over the output an explicit tree in linear time.

6.2.3 Generating tree-decompositions

We have already established that a bucket-tree built along a given ordering is a tree-decomposition, where each node is a bucket, whose functions are those assigned to it by the initial bucket-partitioning as described in Chapters 3 and 4. The variables of a bucket are itself and its earlier neighbors (or parents) in the induced-graph. These variables are members in its clique in the chordal induced graph (see Chapter 5).

It turns out that most tree-decompositions can be obtained by merging adjacent buckets, and more generally, adjacent clusters. This process is justified by the following proposition:

Proposition 6.2.13 *If T is a tree-decomposition, then any tree obtained by merging adjacent clusters is also a tree-decomposition.*

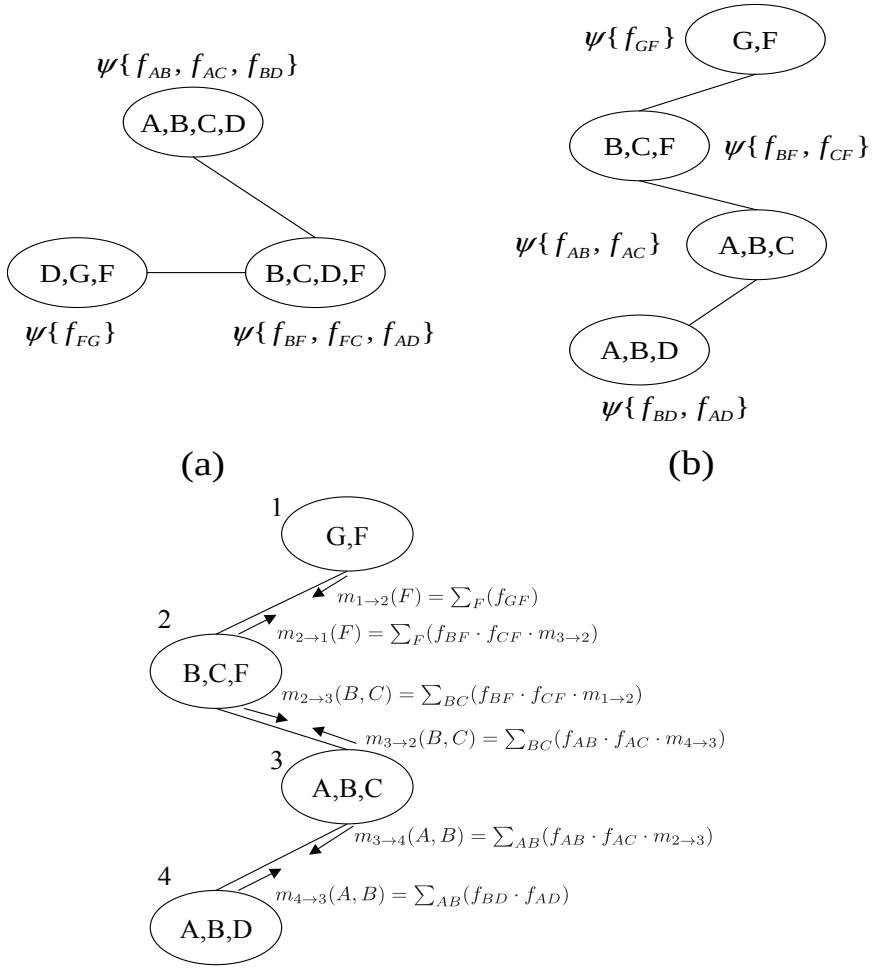


Figure 6.8: Two tree-decompositions of a graphical model

Proof: Do as an exercise.

A special class of tree-decompositions called *join-trees* can be obtained by merging subsumed buckets into their containing buckets. Join-trees can be generated directly based on the induced-graph by selecting as nodes for the tree only those buckets that are associated with maximal cliques.

Join-tree clustering (JTC) for generating join-tree decompositions, is described in Figure 6.9. The algorithm simply generates an induced chordal graph, identify its maximal cliques as the candidate cluster-nodes and connect them in a tree structure. This process determines the variables associated with the nodes. Subsequently, functions are partitions into the clusters appropriately.

JOIN-TREE CLUSTERING (JTC)

Input: A graphical model $\mathcal{M} =< X, D, F, \prod\} >$, $X = \{X_1, ..., X_n\}$, $F = \{f_1, ..., f_r\}$. Its scopes $S = S_1, ..., S_r$ and its primal graph is $G = (X, E)$.

Output: A join-tree decomposition $< T, \chi, \psi >$ for $\mathcal{M}$

1. Select a variable ordering, $d = (X_1, ..., X_n)$.
2. **Triangulation** (create the induced graph along d and call it G^*):
   for $j = n$ to 1 by -1 do
   $E \leftarrow E \cup \{(i, k) |\ (i, j) \in E, (k, j) \in E\}$
3. **Create a join-tree of the induced graph** (G^*, d):
   a. Identify all maximal cliques in the chordal graph.
   Let $C = \{C_1, ..., C_t\}$ be all such cliques,
   created going from last variable to first in the ordering.
   b. Create a tree-structure T over the cliques:
   Connect each C_i to a C_j $(j < i)$ with whom it shares largest subset of variables.
4. Place each input function in one cluster-node whose variables contain its scope.
5. Return a tree-decomposition $< T, \chi, \psi >$, where $T = (C, E)$,
$\chi(u)$ is the clique variables and $\psi(u)$ are the functions placed in node u.

Figure 6.9: Join-tree clustering

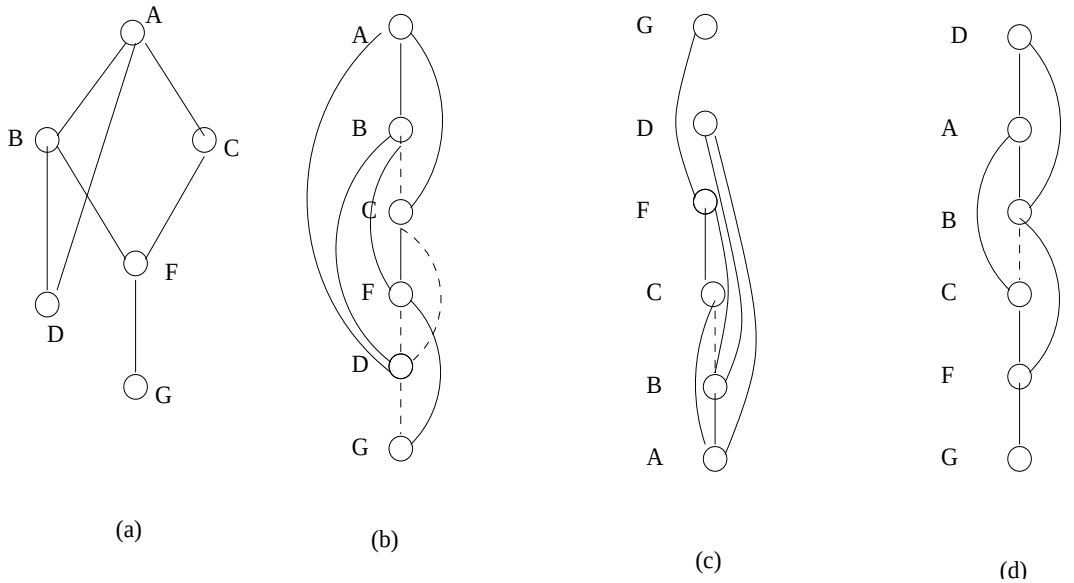


Figure 6.10: A graph (a) and three of its induced graphs (b), (c) and (d).

Example 6.2.14 Consider the graph in Figure 6.10(a) (this example builds upon the same graph and orderings as in Figure 5.2, and we repeat it for convenience). Consider the ordering $d_1 = (G, D, F, C, B, A)$ in Figure 6.10(b). Performing the triangulation step of join-tree-clustering connects parents recursively from the last variable to the first, creating the induced-ordered graph by adding the new (broken) edges of Figure 6.10(b). The maximal cliques of this induced graph are: $Q_1 = \{A, B, C, D\}$, $Q_2 = \{B, C, F, D\}$ and $Q_3 = \{F, D, G\}$. Alternatively, if ordering $d_2 = (A, B, C, F, D, G)$ in Figure 6.10(c) is used, the induced graph generated has only one added edge. The cliques in this case are: $Q_1 = \{G, F\}$, $Q_2 = \{A, B, D\}$, $Q_3 = \{B, C, F\}$ and $Q_4 = \{A, B, C\}$. Yet, another example is given in Figure 6.10(d). The corresponding join-trees of orderings d_1 and d_2 are depicted in the earlier decompositions observed in Figure 6.8(a) and (b), respectively.

□

6.3 Properties of CTE

Algorithm CTE takes as an input a tree-decomposition of a graphical model, and evidence and outputs an explicit tree-decomposition. In this section we will prove the algorithm's soundness and completeness and analyze its complexity.

6.3.1 Correctness of CTE

The correctness of CTE can be shown in two steps. First extending theorem 6.2.6 to show that CTE can solve acyclic graphical models, and then, since a tree-decomposition transforms a graphical model into an acyclic one, the argument follows. Instead of taking this route, we will next state the correctness based on the general properties of the *combine* and *marginalize* operators in order to emphasize the broad applicability of this algorithm. The theorem articulates the properties which are required for correctness. The proof can be found in [43].

Theorem 6.3.1 (soundness and Completeness) *Given a graphical model and reasoning tasks $\mathcal{M} =< X, D, F, \bigotimes, \Downarrow_Y >$, and assuming that the combination operator $\bigotimes_i$ and the marginalization operator $\Downarrow_Y$ satisfy the following properties [58]):*

1. *Order of marginalization does not matter:*
   $\Downarrow_{X-\{X_i\}} (\Downarrow_{X-\{X_j\}} f(X)) = \Downarrow_{X-\{X_j\}} (\Downarrow_{X-\{X_i\}} f(X))$
2. *Commutativity:* $f \bigotimes g = g \bigotimes f$
3. *Associativity:* $f \bigotimes (g \bigotimes h) = (f \bigotimes g) \bigotimes h$
4. *Restricted distributivity:*
   $\Downarrow_{X-\{X_k\}} [f(X - \{X_k\}) \bigotimes g(X)] = f(X - \{X_k\}) \bigotimes \Downarrow_{X-\{X_k\}} g(X)$

Algorithm CTE is sound and complete. Namely it is guaranteed to transform a tree-decomposition into an explicit one. Namely, for every node v, given the messages generated m_{uv} for all $(u, v) \in T$, then

$$M_{\chi(u)} = \psi_u \cdot \prod_{\{j|(j,u)\in E\}} m_{j\to u}$$

□

proof: See appendix.

Clearly, since we know that we get correct explicit functions in any two adjacent nodes in the tree-decomposition, their projection on the separators must be consistent and therefore identical. IN the case of probabilities this means that the marginal probability on the separator variables can be computed in either one of the edge clusters.

Corollary 6.3.2 (pairwise consistency) *For any edge $(u,v) \in T$ in a tree-decomposition $< T, \chi, \psi >$ ($T = (V,E)$ for a graphical model $M =< X, D, F, \prod\} >$, when CTE terminates with the set of messages $m_{u\to v}, m_{v\to u}$ for any eadge in T, then*

$$\Downarrow_{sep(u,v)} \psi_v \cdot \prod_{j|(j,v)\in E} m_{j\to v} = \Downarrow_{sep(u,v)} \psi_u \cdot \prod_{\{j|(j,u)\in E\}} m_{j\to u}.$$

6.3.2 Complexity of CTE

Algorithm *CTE* can be subtly varied to influence its time and space complexities. The description in Figure 6.7 seems to imply an implementation whose time and space complexity are the same. Namely, that the space complexity must also be exponential in w^*. Indeed, if we compute the message in equation 6.7 in a brute-force manner, recording the *combined function* first, and subsequently marginalizing over the separator, we will have space complexity exponential in w^*.

However, we can interleave the combination and marginalization operations, and thereby make the space complexity identical to the size of the sent message, as follows.

GENERATE MESSAGES

1. initialize: for all a_{sep} $m_{u\to v}(a_{sep}) \leftarrow 0$.
2. **for** every assignment a to $\chi(u)$, **do**
3. $m_{u\to v}(a_{sep}) \Leftarrow m_{u\to v}(a_{sep}) + \psi_u(a) \cdot \prod_{\{j|(j,u)\in T, j\neq v\}} m_{j\to u}(a)$
4. **end for**

Figure 6.11: Generate messages

In words, for each assignment a to the variables in $\chi(u)$, we compute the combined functional value, and accumulate the marginalization value on the separator, *sep*, updating the message function m_{sep}. With this modification we now can state (and then prove) the general complexity of CTE

Theorem 6.3.3 (Complexity of CTE) *Given a graphical model* $\mathcal{M} =< X, D, F, \prod, \sum >$ *and its* tree-decomposition $< T, \chi, \psi >$, *where* $T = (V, E)$, *where* N *is the number of vertices in* V, w *its tree-width, sep its maximum separator size,* r *the number of functions in* F, deg *the maximum degree in* T, *and* k *the maximum domain size of a variable. The time complexity of* CTE *is* $O((r + N) \cdot deg \cdot k^{w+1})$ *and its space complexity is* $O(N \cdot k^{sep})$.

Trading space for time in CTE

As we noted earlier, given any tree-decomposition we can generate new tree-decompositions by merging adjacent clusters. Even though time complexity will increase, this process can generate smaller separators. Therefore, while the resulting CTE time will increase the required space, will decrease.

Example 6.3.4 Consider the tree-decompositions in Figure 6.7. For the first two decompositions CTE will have time exponential in 3 and space complexity exponential in 2. The third yields time exponential in 5 but space exponential in 1. □

6.4 Belief Updating, Constraint Satisfaction and Optimization

In this last section of this chapter we will provide more details on algorithms tailored to specific graphical models such as Bayesian networks and constraint networks.

6.4.1 Belief updating and probability of evidence

Applying algorithm CTE to Bayesian networks by specializing combination to *product* and the marginalization operators to *summation*, yields an algorithm that computes the

explicit clusters for a given tree-decomposition. In this case the explicit functions are the marginal probability distribution over the cluster's variables. Therefore, when the algorithm terminates the marginals can be obtained by the product over all functions in the corresponding clusters. From these clusters one can also compute the probability of evidence or the posterior beliefs over singleton variables. We will refer to this specialized algorithm denoted CTE-BU and it is described in Figure 6.13. The algorithm pays a special attention to the processing of observed variables since the presence of evidence is a central aspect in belief updating. When a cluster sends a message to a neighbor, the message contains a single *combined* function and *individual* functions that do not share variables with the relevant eliminator. All the non-individual functions are *combined* in a product and summed over the eliminator.

Example 6.4.1 Figure 6.12 describes a belief network (a) and a join-tree decomposition for it (b). Figure 6.12(c) shows the trace of running *CTE-BU*. In this case no individual functions appear between any of the clusters. Figure 6.12(d) shows the explicit output tree-decomposition. If we want to compute the probability of evidence $P(G = g_e)$, we can pick cluster 4, for example, and compute

$$P(G = g_e) = \sum_{e,f,g=g_e} P(g|e,f) \cdot h_{3\to 4}(e,f)$$

and if we wish to compute the belief for variable B for example we can use the second or the first bucket.

$$P(B|g_e) = \alpha \cdot \sum_{a,c} P(a) \cdot p(b|a) \cdot p(c|a,b) \cdot h_{2\to 1}(b,c)$$

where α is the normalizing constant that is 1 divided by the probability of evidence. □

Pearl's Belief Propagation over Polytrees

It is clear that whenever we have a graphical model that is already a real tree (The functions have only 2 variables in their scope), its treewidth or induced-width is 1. Since it is a special case of acyclic networks a message-passing algorithm like CTE, it is efficient and can be accomplished in linear time and space. Another special acyclic graphical models

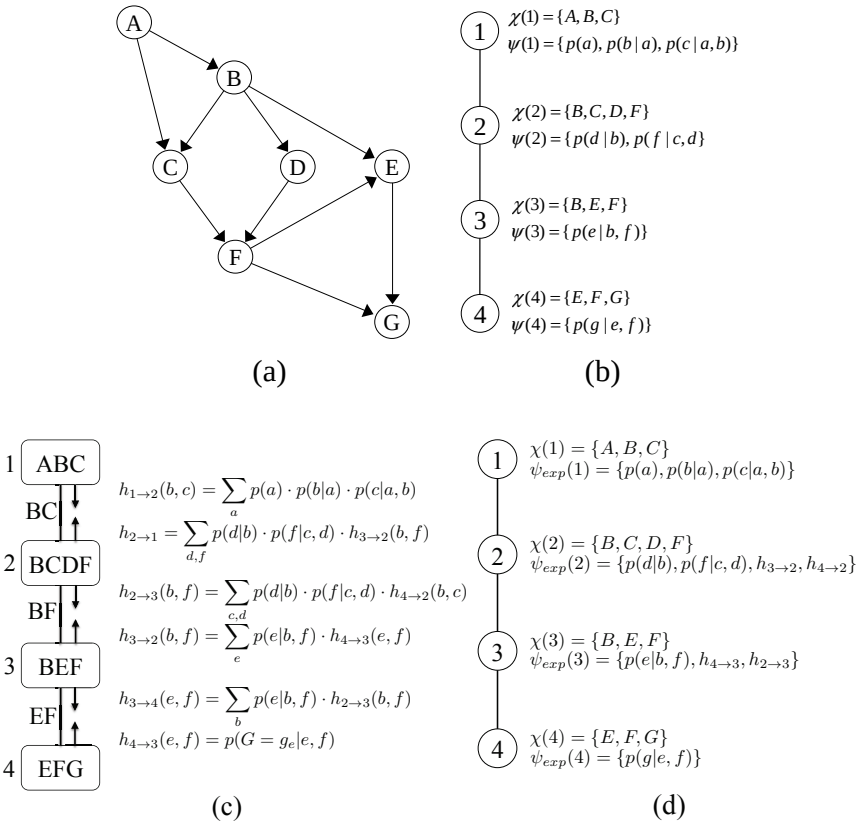


Figure 6.12: [Execution of CTE-BU]: *a*) A belief network; *b*) A join-tree decomposition; *c*)Execution of CTE-BU; no individual functions appear in this case (d) the explicit tree-decomposition

which are not strictly speaking real trees, are *polytrees*. This case deserves attention for historical reasons; it was recognized by Pearl [47] as a generalization of trees on which his belief propagation algorithm was defined and shown to be sound and complete. It also gives rise to an iterative approximation algorithm over general networks, known as *Iterative BP* or *loopy BP* [63].

Definition 6.4.2 (polytree) *A polytree is a directed acyclic graph whose underlying undirected graph has no cycles (see Figure 6.14(a)).*

It is easy to see that the dual graph of a polytree is a tree, and thus yields an acyclic problem that has a join-tree decomposition where each family (i.e., a child node and

Algorithm **CTE for Belief-Updating (CTE-BU)**

Input: A tree decomposition $< T, \chi, \psi >$, $T = (V, E)$ for $BN =< X, D, G, P >$. Evidence variables $var(e)$ evidence e.

Output: An explicit tree-decomposition, where each cluster represent its posterior marginals.

• **Compute messages: For** every node u in T, once u has received messages from all neighbors but v, compute

1. **Process observed variables:**
   For each node $u \in T$ assign relevant evidence to $\psi(u)$
2. **Compute the combined function:**

$$m_{(u \to v)} = \sum_{elim(u,v)} \psi_u \cdot \prod_{f \in cluster(u), f \neq m_{v \to u}} f.$$

• **Compute** $P(X_i, e)$**:** For every $X_i \in X$ let u be a vertex in T such that $X_i \in \chi(u)$. Compute $P(X_i, e) = \sum_{\chi(u)-\{X_i\}} (\psi_u \cdot \prod_{f \in cluster(u)} f)$. Compute $bel(X_i) = \alpha P(X_i, e)$

Figure 6.13: Algorithm Cluster-Tree-Elimination for Belief Updating (CTE-BU)

its parents) constitute the variables of a single node u in the decomposition. Namely $\chi(u) = \{X\} \cup pa(X)$, and $\psi(u) = \{P(X|pa(X))\}$. Note, that the separators in this tree-decomposition, called *polytree decomposition*, are all singleton variables. In summary,

Proposition 6.4.3 *A polytree has a tree dual graph and it is therefore an acyclic graphical model.*

If we direct the edges of the polytree-decomposition from a cluster of a parent node X to the cluster of its child, and if we call the messages sent by CTE from child clusters to parent clusters as λ messages and the reverse ones as π messages, we get the original Pearl's belief propagation messages. (Note how similar this is to the π and λ messages of BTE along this ordering.)

Definition 6.4.4 (belief propagation (BP)) *Given a ploytree and a directed dual graph which is a poly-tree decomposition, the belief propagation algorithm (BP) is algorithm CTE, whose messages down the polytree-decomposition are called λ and up are called π.*

The nice feature of BP message-passing scheme is that the messages can be given a meaningful interpretation using probabilistic entities. Pearl calls the π messages predictive messages and the λ messages, diagnostic messages (see [47]).

Example 6.4.5 Consider the polytree given in Figure 6.14(a). An ordering along which we can run the bucket-tree algorithm is given in (b), a directed polytree decomposition is given in Figure 6.14(c) along with the π and λ messages. The explicit output tree-decomposition is given in (d). Once the propagation terminates, beliefs can be computed in each cluster. □

Theorem 6.4.6 *Given a polytree network and its dual graph, algorithms CTE applied along its join-tree is time linear in the network's size and its space complexity is linear in the number of variables and their domains.* □

Proof: Do as an exercise ∎

The case of HMM

[To complete]

6.4.2 Constraint Satisfaction

Algorithm CTE for Constraints

Algorithm CTE for constraint networks can be obtained straightforwardly by using the join operation for combination and the relational project for marginalization. The explicit algorithm is given for completeness sake in Figure 6.15. It generates an *explicit* representation of the constraints in each node. This makes it possible to answer most relevant queries locally, by consulting the constraints inside each of these nodes only. This property of explicitness was called *minimality* and *decomposability* in [45] in the context of constraints.

Once a tree-decomposition is available the algorithm process the decomposition as usual. Node u takes all the constraints in $\psi(u)$ and all the constraint messages received

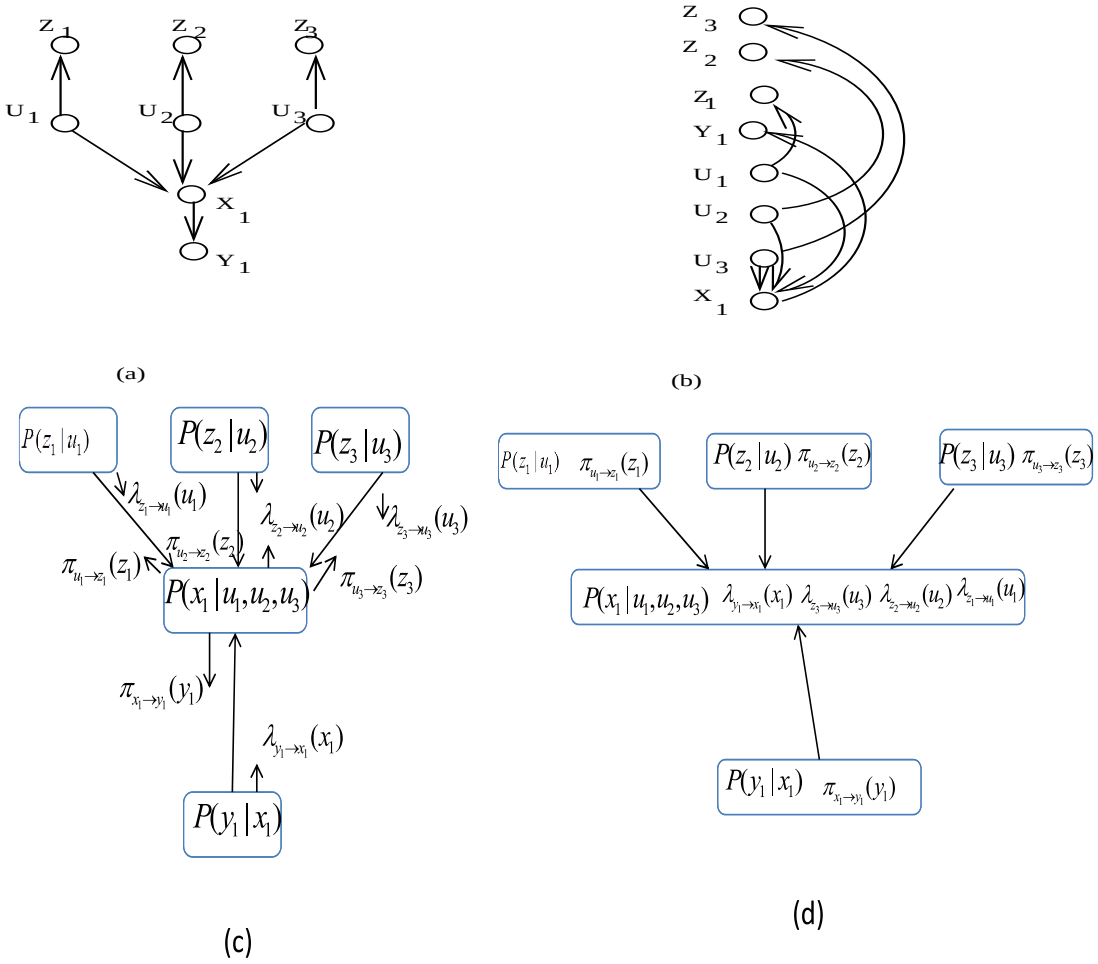


Figure 6.14: (a) A polytree and (b) a legal processing ordering, (c) a polytree decomposition and messages (d) the explicit decomposition

CLUSTER TREE-ELIMINATION (CTE-CONSTRAINTS)

Input: A tree decomposition $< T, \chi, \psi >$ for a constraint network $\mathcal{R} =< X, D, C >$.

Output: An explicit tree-decomposition where each node is minimal.

Compute messages: for every edge (u, v) in the tree, do

- Let $m_{u \to v}$ denote the message sent from u to v.
  - Define, $cluster_v(u) = \psi(u) \cup \{m_{i \to u} | (i, u) \in T,\ i \neq v\}$
  - Compute and send to v:

$$m_{u \to v} \leftarrow \pi_{sep(u,v)}(\bowtie_{R_i \in cluster_v(u)} R_i) \tag{6.6}$$

endfor

Return: A tree-decomposition augmented with constraint messages. For every node $u \in T$, return the decomposable (explicit) subproblem $\psi'(u) = \psi(u) \cup \{m_{(i \to u)} | (i, u) \in T\}$.
If ψ' is consistent for every node the problem is consistent.
Generate a solution in a backtrack-free manner.

Figure 6.15: Algorithm cluster-tree elimination (CTE)

by u from all adjacent nodes, and generates their join projected on the various separators with its neighbors. The resulting constraint-message is sent to each neighboring node.

As noted in the general case, the generation of the message does not necessary require recording the join of all the relations to be then projected on the separators. Rather, we can have a more careful implementation that uses memory exponential in the separator only, as discussed earlier. Another point to note is that in the case of constraints we do not need to be careful regarding the exclusion of the message sent from u to v when v computes the message it sends to u.

For each node, the augmented set of constraints can be shown to be a *minimal subnetwork* relative to the input constraint problem $\mathcal{R}$. Minimality is explicitness for constraints.

Definition 6.4.7 (minimal subproblem, a decomposable network) *Given a constraint problem $\mathcal{R} = (X, D, C)$ and a subset of variables $Y \subseteq X$, a subproblem over Y, $\mathcal{R}_Y = (Y, D_Y, C_Y)$, is decomposable relative to $\mathcal{R}$ iff $sol(R_Y) = \pi_Y sol(\mathcal{R})$ where $sol(\mathcal{R}) = \bowtie_{\mathcal{R}\in\mathcal{R}} \mathcal{R}$ is the set of all solutions of network $\mathcal{R}$. A network of constraints is decomposable if each of its subnetworks, is minimal.*

Example 6.4.8 Consider the constraint network ...[to complete] □

Theorem 6.4.9 *Given a tree-decomposition $< T, \chi, \psi >$ for a constraint network $\mathcal{R} =< X, D, C = \{C_1, ..., C_r\} >$, where R_i is the relation defining C_i over a scope S_i, processed by algorithm* CTE-CONS. *Then at termination the constraints in each node constitute a minimal network. Namely, for each node u, $\bowtie_{R\in cluster(u)} R = \pi_{\chi(u)}(\bowtie_{R\in\mathcal{R}} R)$.*

Proof: The proof follows immediately from the correctness of CTE. All we need is to use relational join for the combined operator and the project for the marginalized operators. We leave the details as an exercise. ∎

The minimality property is powerful. Once we have a minimal subnetwork which is not empty we immediately know that the network is consistent. Moreover the resulting tree-decomposition whose clusters are minimal can be shown to be backtrack-free along orderings consistent with the tree structure. This means that we can generate a solution in any order along the tree and we are guaranteed to not have any dead-ends. Therefore solution generation is linear in the output network. (exercise: prove the CTE-CONS generates a backtrack-free ordering along some variable orderings.)

6.4.3 Optimization

Another popular type of CTE algorithm can be applied to optimization tasks. If for example we want to solve the MPE task over Bayesian networks, we know that we can do so by the bucket elimination algorithm when in each bucket we use the max-product combination operators. Extending that into a CTE algorithm the algorithm (called often the max-product algorithm) will compute, not only the maximum cost of the problem, but also for every tuple of a node $a_{\chi(u)}$ in a tree-decomposition, it generates the maximum cost

(e.g., probability) over any full assignment extending it. For that, all we need is to replace the marginalization operator in CTE-BU of summation by maximization. Alternatively, for minimization over arbitrary cost functions in a cost-network we can use the min-product CTE algorithm. Finally, if in our cost network the combination operator is summation, we can use min-sum or max-sum based CTE.

In all these cases, at termination we do not have only the optimal costs, but we know what is *the optimal cost to go* associated with each node in the tree-decomposition.

[more examples: to complete]

Join-project, Sum-product and Max-product

In the past two subsections we have looked at algorithms for solving the tasks of computing the probability of evidence or marginals in probabilistic networks and of answering constraint satisfaction tasks over constraint networks. The first is often referred to as the *sum-product* algorithm and the second as the *join-project*. The join-project algorithm can be accomplished using Boolean operators. If we express the relations as $\{0, 1\}$ cost functions("0" for inconsistent tuple and "1" for a consistent tuple). Combining functions by a Boolean product operators and marginalizing using Boolean summation will yield an identical algorithm to CTE-constraints (show this as an exercise). Moreover, if we use regular summation for the marginalization operator and a regular product operator for combination over the (0,1) representation of relation, then the CTE sum-product algorithm computes the number of solutions of the constraint network.

Definition 6.4.10 (counts of partial solutions) *Given a $\mathcal{R} =< X, D, C >$, and given a partial value assignment a_S over scope S, $S \subseteq X$, we denote by $count(a_S)$, the number of full solutions of $\mathcal{R}$ that extend the partial assignment a_S. Namely, $count(a_S) = |\{t \in sol(\mathcal{R}) | \sqcup_{\mathcal{S}} = \dashv_S\}|$. The function $count_S$ associates each tuple over a scope S with its count relative the network $\mathcal{R}$.*

Theorem 6.4.11 *Given a tree-decomposition $< T, \chi, \psi >$ for a constraint network $\mathcal{R} =< X, D, C = \{C_1, ..., C_r\} >$, where each C_i is represented as a zero-one function. Then when the sum-product CTE terminates, for each u and each assignment a_u over its scope $\chi(u)$, $\prod_{f \in \psi'(u)} f = count_{\chi(u)}$.*

Proof: Do as an exercise ∎

6.5 Summary and Bibliographical Notes

Join-tree clustering was introduced in constraint processing by Dechter and Pearl [25] and in probabilistic networks by Spigelhalter et. al [41]. Both methods are based on the characterization by relational-database researchers that acyclic-databases have an underlying tree-structure, called join-tree, that allows polynomial query processing using join-project operations and easy identification procedures [6, 42, 61]. In both constraint networks and belief networks, it was observed that the complexity of compiling any knowledge-base into an acyclic one is exponential in the cluster size, which is characterized by the induced-width or tree-width. At the same time, variable-elimination algorithms developed in [9, 55] and [24] (e.g., adaptive-consistency and bucket-elimination) were also observed to be governed by the same complexity graph-parameter. In [24, 25] the connection between induced-width and tree-width was recognized through the work of [4] on tree-width and k-trees and partial k-trees, which was made explicit later in [30]. The similarity between variable-elimination and tree-clustering from the constraint perspective was analyzed [25]. Independently of this investigation, the tree-width parameter was undergoing intensive investigation in the theoretic-graph-community. It characterizes the best embedding of a graph or a hypergraph in a hypertree. Various connections between hypertrees, chordal graphs and k-trees were made by Arnborg and his colleagues [4, 52]. They showed that finding the smallest tree-width of a graph is NP-complete, but deciding if the graph has a tree-width below a certain constant k is polynomial in k. A recent analysis shows that this task can be accomplished in $O(n \cdot f(k))$ where $f(k)$ is a very bad exponential function of k [13].

6.6 Appenix for proofs

Theorem 6.6.1 *6.2.6 Given an acyclic graphical model, algorithm BTE generates the explicit sub-model (definition xxx) for scopes of every input function, in linear time and space.*

Proof: Correctness follows from the correctness of BTE (Theorem 6.1.5) stating that BTE generates explicit buckets therefore, all we need to focus here is on the complexity claim. The algorithm is linear because there exists an ordering for which each function resides alone in its bucket, and for which BTE generates messages that are subsumed by the original functions' scopes. Clearly such messages (at most n in each direction) can be generated in time and space bounded by the functions' sizes (i.e., number of tuples in the domain of each input function). A desired ordering can be generated by processing leaf nodes along the join-tree of the acyclic model, imposing a partial. We can show (exercise) that the ordering generated facilitates messages whose scopes are subsumed by the original function scopes. This implies a linear complexity. ∎

Theorem 6.6.2 *6.2.11 A bucket tree of a graphical model $\mathcal{M}$, is a tree-decomposition of $\mathcal{M}$.*

Proof: Given bucket tree $T = (V, E)$ of $\mathcal{M}$, whose nodes are maped to buckets, we need to show how the tree can be associated with mappings χ and ψ that satisfy the that conditions of Definition 6.2.7. In other words, the tree structure T in tree-decomposition $< T, \chi, \psi >$ is the bucket tree structure, where each B_i corresponds to a vertex in V. If a bucket B_i has a parent (i.e., is connected to) bucket B_j, there is an edge $(B_i, B_j) \in E$. Labeling $\chi(B_i)$ is defined to be the union of the scopes of new and old functions in B_i during processing by BTE, and labeling $\psi(B_i)$ is defined to be the set of functions in the initial partition in B_i. With these definitions, condition 1 of Definition 6.2.7 is satisfied because each function is placed into exactly one bucket and condition 2 of Definition 6.2.7 is also satisfied because labeling $\chi(B_i)$ is the union of scopes of all functions in B_i. Condition 4 of Definition 6.2.7 is trivially satisfied since there is exactly one bucket for each variable.

Finally we need to prove the connectedness property. Lets assume that there is a variable X_k with respect to which the connectedness property is violated. This means that there must be (at least) two disjoint subtrees, T_1 and T_2, of T, such that each vertex in both subtrees contains X_k, and there is no edge between a vertex in T_1 and T_2. Let B_I be a vertex in T_1 such that X_i is the earliest relative to ordering d, and B_j a vertex in T_2 such that X_j is the earliest in ordering d. Since T_1 and T_2 are disjoint, it must be that

$X_i \neq X_j$. However, this is impossible since this would mean that there are two buckets that eliminate variable X_k. □

Theorem 6.6.3 (Complexity of CTE) *6.3.3*

Given a graphical model $\mathcal{M} =< X, D, F, \prod, \sum >$ *and its* tree-decomposition $< T, \chi, \psi >$, *where* $T = (V, E)$, *where* N *is the number of vertices in* V, w *its tree-width, sep its maximum separator size,* r *the number of functions in* F, *deg the maximum degree in* T, *and* k *the maximum domain size of a variable. The time complexity of CTE is* $O((r + N) \cdot deg \cdot k^{w+1})$ *and its space complexity is* $O(N \cdot k^{sep})$.

Proof. The time complexity of processing a vertex u in tree T is $deg_u \cdot (|\psi(u)| + deg_u - 1) \cdot k^{|\chi(u)|}$, where deg_u is the degree of u, because vertex u has to send out deg_u messages, each being a combination of $(|\psi(u)| + deg_u - 1)$ functions, and requiring the enumeration of $k^{|\chi(u)|}$ combinations of values. The time complexity of CTE is

$$Time(CTE) = \sum_u deg_u \cdot (|\psi(u)| + deg_u - 1) \cdot k^{|\chi(u)|}$$

By bounding the first occurrence of deg_u by deg and $|\chi(u)|$ by $w + 1$, we get

$$Time(CTE) \leq deg \cdot k^{w+1} \cdot \sum_u (|\psi(u)| + deg_u - 1)$$

Since $\sum_u |\psi(u)| = r$ we can write

$$Time(CTE) \leq deg \cdot k^{w+1} \cdot (r + N)$$

$$= O((r + N) \cdot deg \cdot k^{w+1})$$

For each edge CTE will record two functions. Since the number of edges is bounded by N and the size of each function we record is bounded by k^{sep}. Based on the above discussion this space complexity is also attainable.

If the cluster-tree is minimal (for any u and v, $sep(u, v) \subset \chi(u)$ and $sep(u, v) \subset \chi(v)$), then we can bound the number of vertices N by n. Assuming $r \geq n$, the time complexity of CTE applied to a minimal tree-decomposition is $O(deg \cdot r \cdot k^{w+1})$. □

Theorem 6.6.4 (soundness and Completeness) *Given a graphical model and reasoning tasks* $\mathcal{M} =< X, D, F, \bigotimes, \Downarrow_Y >$, *and assuming that the combination operator* $\bigotimes_i$ *and the marginalization operator* $\Downarrow_Y$ *satisfy the following properties [58]):*

1. *Order of marginalization does not matter:*
   $\Downarrow_{X-\{X_i\}} (\Downarrow_{X-\{X_j\}} f(X)) = \Downarrow_{X-\{X_j\}} (\Downarrow_{X-\{X_i\}} f(X))$

2. *Commutativity:* $f \bigotimes g = g \bigotimes f$

3. *Associativity:* $f \bigotimes (g \bigotimes h) = (f \bigotimes g) \bigotimes h$

4. *Restricted distributivity:*
   $\Downarrow_{X-\{X_k\}} [f(X - \{X_k\}) \bigotimes g(X)] = f(X - \{X_k\}) \bigotimes \Downarrow_{X-\{X_k\}} g(X)$

Algorithm CTE is sound and complete. Namely it is guaranteed to transform a tree-decomposition into an explicit one. Namely, for every node v, *given the messages generated* m_{uv} *for all* $(u, v) \in T$, *then*

$$M_{\chi(u)} = \psi_u \cdot \prod_{\{j|(j,u)\in E\}} m_{j\to u}$$

Proof:

Using the four properties of combination and marginalization operators, the claim can be proved by induction on the depth of the tree. Specifically, we will show that for every node u, CTE generates the explicit representation for that node. Namely for every node u, $\bigotimes_{f\in\bar{\psi}(u)} f = \mathcal{M}_{\chi(u)}$.

Let $< T, \chi, \psi >$, where $T = (V, E)$ be a cluster-tree decomposition for $\mathcal{M}$, and let root it in vertex $v \in V$. We can create a partial order of the vertices of T along the rooted tree. We denote by $T_u = (V_u, E_u)$ the subtree rooted at vertex u and define by $\chi(T_u)$ all the variables associated with nodes appearing in T_u, namely: $\chi(T_u) = \bigcup_{v\in V_u} \chi(v)$.

Since commutativity permits combining the functions in any order, we select an ordering of the nodes in T $d(j) \in V, j = 1, ..., |V|$ where a vertex in the rooted tree T precedes its children in the ordering, and thus the first vertex is the root of the tree is v. We denote by $\bar{\psi}_u = \bigotimes_{f\in\psi(u)} f$, the combination of all the input functions in node u. Because

of associativity and commutativity clearly:

$$\forall\ u\ \ \mathcal{M}_{\chi(u)} = \Downarrow_{\chi(u)} \bigotimes_{j=1}^{|V|} \bar{\psi}_{d(j)}.$$

Let u be a node having the parent w in the rooted tree T, and define $elim(u) = \chi(u) - sep(u,w)$ and $elim(T_u) = \bigcup_{v \in V_u} elim(v)$ (We will show that $elim(T_u)$ is the set of variables that are eliminated by CTE in the subtree rooted at u). Because of the connectedness property, variables in $elim(T_u)$, appear only in the subtree rooted at u. In other words, $elim(T_u) \bigcap \{X_i | X_i \in V - \chi(T_u)\} = \emptyset$. Consequently, we can marginalize ($\downarrow$) over such variables earlier in the process of deriving $\mathcal{M}_{\chi(u)}$ (remember that $\Downarrow_{Z_i}$ means marginalizing over $X - Z_i$). If $X_i \notin \chi(u)$ and if $X_i \in elim(d(k))$ for some k, then, the marginalization eliminating X_i can be applied to $\bigotimes_{j=k}^{|V|} \bar{\psi}_{d(j)}$ instead of to $\bigotimes_{j=1}^{|V|} \bar{\psi}_{d(j)}$. This is safe to do, because as shown above, if a variable X_i belongs to $elim(d(k))$, then it cannot be part of any $\bar{\psi}_{d(j)}$, $j < k$. We can therefore derive $\mathcal{M}_{\chi(u)}$ as follows:

$$\mathcal{M}_{\chi(u)} = \Downarrow_{\chi(u)} \bigotimes_{j=1}^{|V|} \bar{\psi}_{d(j)} = \tag{6.7}$$

(because of the tree structure)

$$= \Downarrow_{\chi(u)} [\bigotimes_{j=1}^{d(k-1)} \bar{\psi}_{d(j)} \Downarrow_{(X - elim(d(k)))} \bigotimes_{j=k}^{|V|} \bar{\psi}_{d(j)}] = \tag{6.8}$$

$$= \Downarrow_{\chi(u)} \bigotimes_{j=1}^{d(k-1)} \bar{\psi}_{d(j)} \otimes F_T(d(k)) \tag{6.9}$$

where, for $u = d(k)$ $F_T(u) = \Downarrow_{(X - elim(u))} \bigotimes_{j=k}^{|V|} \bar{\psi}_{d(j)}$. We now assert that due to properties 1-4, $F_T(u)$ obeys a recursive definition relative to the tree-decomposition and that this recursive definition is identical to the messages computed by CTE and sent from u to its parent w. This is articulated by the following proposition and will conclude the proof.

Proposition 6.6.5 *The functions $F_T(u)$ defined above relative to the rooted tree-decomposition T, obey the following recursive definition. Let $ch(u)$ be the set of children of u in the rooted tree T.*

- *If* $ch(u) = \emptyset$ *(vertex* u *is a leaf vertex), then* $F_T(u) = \Downarrow_{(X-elim(u))} \bar{\psi}_u$.
- *Otherwise,* $F_T(u) = \Downarrow_{(X-elim(u))} \bar{\psi}_u \otimes \bigotimes_{w \in ch(u)} F_T(w)$.

It is easy to see that the messages computed by CTE up the tree-decomposition along T *are the* $F_T(u)$ *functions. Namely, For every node* u *and its parent* w, $F_T(u) = m_{u \to w}$, *and in particular at the root node* v $F_T(v) = \mathcal{M}_{\chi(v)}$ *which is identical to the message* v *can send to its (empty parent).*

■

Bibliography

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

[2] Bar-Yehuda R A. Becker and D. Geiger. Random algorithms for the loop-cutset problem. In *Uncertainty in AI (UAI'99)*, pages 81–89, 1999.

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

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

[5] 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.

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

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

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

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

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

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

[12] 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.

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

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

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

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

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

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

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

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

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

[22] 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.

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

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

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

[26] 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.

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

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

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

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

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

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

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

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

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

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

[37] 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.

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

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

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

[41] 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.

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

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

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

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

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

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

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

[49] 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.

[50] 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.

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

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

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

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

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

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

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

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

[59] 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.

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

[61] 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.

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

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

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