Reasoning with Graphical Models

Rina Dechter

April 27, 2011

Contents

Bibliography 101

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 key idea of the graphical models paradigm is that knowledge can be more efficiently expressed and reasoned about if it is represented as a set of local functions defined over small subsets of variables where the variable interactions induced by these functions are captured by a graph structure. The graph is a powerful abstraction, revealing the independencies and irrelevancies that make it tractable to reason over high dimensional domains [30].

Although queries posed over graphical models are NP-hard, and thus generally intractable, graphical models invite effective algorithms for many graph structures and for many queries, including combinatorial optimization, constraint satisfaction, counting, likelihood computation, and even knowledge compilation. 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.

The goal of this book is to present a unifying treatment of graphical models in a way that goes beyond a commitment to the particular types of knowledge expressed in the model. The study of graphical models has been largely fragmented among several research communities that have and continue to utilize the properties of these models to answer

specific queries about a particular type of knowledge. In chapter two, we will review the various flavors of models, but the focus of this book is on query processing algorithms which exploit graph structures 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 (e.g., variable-elimination, join-tree clustering) have been well studied for over two decades, and their complexity bounds are thoroughly understood. These algorithms are exponentially bounded in both time and space by a graph parameter called *tree-width*. Search-based algorithms can be executed in linear space, and this makes them attractive, but it is only recently that search algorithms with efficient time bounds have emerged. Furthermore, search methods are more naturally poised to exploit the internal structure of the functions themselves, what is often called their *local structure*. Borrowing on ideas from the inference literature, these newer algorithms enable improved performance by flexibly trading off time and space.

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 we refer the readers to the books in the area. First and foremost is the classical book that introduced probabilistic graphical models [30] and a sequence of books that followed amongst which are [28, 21]. In particular note the comprehensive two recent textbooks [1, 23]. For deterministic graphical models of Constraint networks see [13].

1.1 Overview by chapter

Chapter 2 present the reader to the concepts of graphical models, provide definitions and the specific graphical models that we will discuss throughout the book. Chapters 3-6 discuss exact inference algorithms, Chapter 5, provides background in graph theory that will be needed throughout the book. Chapter 7 focuses on exact search schemes. Specifically, chapter 3 describes the bucket-elimination for deterministic networks, chapter 4 focuses on probabilistic networks. Chapter 6 shows how these variable elimination algorithms can be extended to tree-decomposition yielding the jointree and junction tree propagation. Chapter 7 focuses on search

1.2 Mathematical background

The formalization of constraint networks relies upon concepts drawn from the related areas of discrete mathematics, logic, the theory of relational databases and graph theory and probability theory. This section is a summary of the mathematical knowledge needed for understanding the formalization of constraint networks and the analyses presented in subsequent chapters. Here we present the basic notations and definitions for sets, relations, probabilistic functions, operations on relations, on cost functions and graphs. For those readers already familiar with these topics, a skim of this material will suffice to ensure your understanding of notation used in the book.

1.2.1 Sets, domains, and tuples

A *set* is a collection of distinguishable objects, and an object in the collection is called a *member* or an *element* of the set. A set cannot contain the same object more than once, and its elements are not ordered. If an object x is a member of set A, we write $x \in A$; if an object x is not a member of set A, we write $x \notin A$. A set can be defined explicitly, by listing the members of the set, or implicitly, by stating a property satisfied by elements of the set. For example, $A = \{1, 2, 3\}$ and $A = \{x \mid x \text{ an integer and } 1 \leq x \leq 3\}$ both represent the same set with three members, 1, 2 and 3. If each element of a set A is also an element of set B, then we write $A \subseteq B$ and say that A is a *subset* of B. Set A is a *proper subset* of B, written $A \subset B$, if $A \subseteq B$ but $A \neq B$.

Given two sets A and B, we can also define new sets by applying set operations: the *intersection* of two sets A and B is the set $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$, the *union* of two sets A and B is the set $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$, and the *difference* of two sets A and B is the set $A - B = \{x \mid x \in A \text{ and } x \notin B\}$. A set containing no members is called an *empty set* and is denoted $\emptyset$. The number of elements in a set S is called the *size* (or *cardinality*) of the set and is denoted $|S|$. Two sets A and B are *disjoint* if they have no elements in common; that is, if $A \cap B = \emptyset$.

The *domain* of a variable is simply a set that lists all of the possible objects that a variable can denote or all of the possible values that a variable can be assigned. A *k-tuple* (or simply a *tuple*) is a sequence of k not necessarily distinct objects denoted by

$(a_1, \ldots, a_k)$, and an object in the sequence is called a *component*. The *Cartesian product* (or simply the *product*) of a list of domains $D_1, \ldots, D_k$, written $D_1 \times \cdots \times D_k$, is the set of all k-tuples $(a_1, \ldots, a_k)$ such that a_1 is in D_1, a_2 is in D_2, and so on.

Example 1.2.1 Let $D_1 =$ {black, green} and $D_2 =$ {apple juice, coffee, tea}. The Cartesian product $D_1 \times D_2$ is the set of tuples {(black, apple juice), (black, coffee), (black, tea), (green, apple juice), (green, coffee), (green, tea)}. □

1.2.2 Relations

Given a set of variables $X = \{x_1, \ldots, x_k\}$, each associated with a domain $D_1, \ldots, D_k$ respectively, a *relation* R on the set of variables is any subset of the Cartesian product of their domains. The set of variables on which a relation is defined is called the *scope* of the relation, denoted *scope*(R). Each relation that is a subset of some product $D_1 \times \cdots \times D_k$ of k domains is said to have *arity* k. If $k = 1$, 2, or 3, then the relation is called a *unary*, *binary* or *ternary* relation, respectively. If $R = D_1 \times \cdots \times D_k$, then R is called a *universal* relation. We will frequently denote a relation defined on a scope S by R_S.

Example 1.2.2 Let $D_1 =$ {black, green} be the domain of variable x_1 and let $D_2 =$ {apple juice, coffee, tea} be the domain of variable x_2. The set of tuples {(black, coffee), (black, tea), (green, tea)} is a relation on $\{x_1, x_2\}$, since the tuples are a subset of the product of D_1 and D_2. The scope of this relation is $\{x_1, x_2\}$. □

The empty set is another example of a relation.

Representing relations

Relations are sets of tuples defined over the same scope, and, as discussed above for sets, they may be either explicitly or implicitly defined. For example, let R be a relation on the set of variables $\{x_1, x_2\}$, where $D_1 =$ {black, green} and $D_2 =$ {apple juice, coffee, tea}. Then the relation R_1 on the scope $\{x_1, x_2\}$ given by $R_1 =$ {(black, coffee), (black, tea), (green, tea)} and $R_2 = \{(x_1, x_2) \mid x_1 \in D_1, x_2 \in D_2,$ and x_1 is before x_2 in dictionary ordering} both represent the same relation. Using arithmetic expressions we can also write more succinctly, $x_1 \leq x_2$ where $\leq$ is a lexicographic ordering.

Two additional ways to explicitly express a relation make use of tables and (0,1)-matrices. In a table representation each row is a tuple and each column corresponds to one component of the tuple. Each column is identified by the variable associated with that component (in the database community, the names of columns are called *attributes*). The ordering of the columns is inconsequential; two relations that differ only in the ordering of their columns are considered the same.

1.2.3 Propositional Theories

Propositional variables take only two values $\{true, false\}$ or "1" and "0." We denote propositional *variables* by uppercase letters $P, Q, R, \ldots$, propositional literals (i.e., $P, \neg P$) stand for $P =$ "$true$" or $P =$ "$false$," and disjunctions of literals, or *clauses*, are denoted by $\alpha, \beta, \ldots$. A *unit clause* is a clause of size 1. The notation $(\alpha \vee T)$, when $\alpha = (P \vee Q \vee R)$ is shorthand for the disjunction $(P \vee Q \vee R \vee T)$. $\alpha \vee \beta$ denotes the clause whose literal appears in either α or β. The *resolution* operation over two clauses $(\alpha \vee Q)$ and $(\beta \vee \neg Q)$ results in a clause $(\alpha \vee \beta)$, thus eliminating Q. A formula φ in conjunctive normal form (*CNF*) is a set of clauses $\varphi = \{\alpha_1, \ldots, \alpha_t\}$ that denotes their conjunction. The set of *models* or *solutions* of a formula φ is the set of all truth assignments to all its symbols that do not violate any clause. Deciding if a theory is satisfiable is known to be NP-complete [20].

Operations on relations

Having introduced the mathematical notion of a relation, lets now consider operations on relations. First we discuss how general set operations apply to relations, and then we focus on three operations specific to relations: selection, projection and join.

Intersection, union, and difference. Given two relations, R and R', on the same scope, the intersection of R and R', denoted $R \cap R'$, is the relation containing all tuples that are in both R and R'; the union $R \cup R'$ is the relation containing all the tuples that are in either R or R', or both, and; the difference $R - R'$ is the relation containing those tuples that are in R but not in R'. The scope of the resulting relations is the same as the scope of the relations R and R'.

x_1	x_2	x_3
a	b	c
b	b	c
c	b	c
c	b	s

(a) Relation R

x_1	x_2	x_3
b	b	c
c	b	c
c	n	n

(b) Relation R'

x_2	x_3	x_4
a	a	1
b	c	2
b	c	3

(c) Relation R''

Figure 1.1: Three relations.

Example 1.2.3 Let the relations R, R' and R'' be as shown in Figure 1.1. The relations R and R' have the same scopes $\{x_1, x_2, x_3\}$, so the set operations intersection, union, and difference are well-defined for these sets. Since the scope of R'' is not the same, none of these three operations is well-defined for R'' in conjunction with either of the other two relations. Figure 1.2 shows the relations $R \cap R'$, $R \cup R'$ and $R - R'$. □

x_1	x_2	x_3
b	b	c
c	b	c

(a) $R \cap R'$

x_1	x_2	x_3
a	b	c
b	b	c
c	b	c
c	b	s
c	n	n

(b) $R \cup R'$

x_1	x_2	x_3
a	b	c
c	b	s

(b) $R - R'$

Figure 1.2: Example of set operations intersection, union, and difference applied to relations.

Let us now consider operations specific to relations.

Selection. A selection takes a relation R and yields a new relation: the subset of tuples of R, with specified values on specified variables. In a table representation of a relation, selection chooses a subset of the rows. Let R be a relation, let $x_1, \ldots, x_k$ be variables in the scope of R, and let a_i be an element of D_i, the domain of x_i. We use the notation

$\sigma_{x_1=a_1,\ldots,x_k=a_k}(R)$ to denote the selection of those tuples in R that have the value a_1 for variable x_1, the value a_2 for variable x_2, and so on. An alternative and more succinct notation is: if $Y = \{x_1, \ldots, x_k\}$ and $t = (a_1, \ldots, a_k)$, then $\sigma_{Y=t}(R)$. The scope of the resulting relation is the same as the scope of R.

Projection. Projection takes a relation R and yields a new relation that consists of the tuples of R with certain components removed. In a table representation of a relation, projection chooses a subset of the columns. Let R be a relation, and let $Y = \{x_1, \ldots, x_k\}$ be a subset of the variables in the scope of R. We use the notation $\pi_Y(R)$ to denote the projection of R onto Y. That is, the set of tuples obtained by taking in turn each tuple in R and forming from it a smaller tuple, keeping only those components associated with variables in Y. Projection specifies a subset of the variables of a relation, and so the scope of the resulting relation is that subset of variables.

Join. The join operator takes two relations R_S and R_T, and yields a new relation that consists of the tuples of R_S and R_T combined on all their common variables in S and T. For illustration, let R_S be a relation with scope S, and R_T a relation with scope T. A tuple r is in the join of R_S and R_T, denoted $R_S \bowtie R_T$, if it can be constructed according to the following steps: (i) take a tuple s from R_S, (ii) select a tuple t from R_T such that the components of s and t share the variables that R_S and R_T have in common (that is, on the variables in $S \cap T$), and, (iii) form a new tuple r by combining the components of s and t, keeping only one copy of those components corresponding to variables in $S \cap T$. The scope of the resulting relation is the union of the scopes of R and S, that is $S \cup T$. We can see now that a join of two relations with the same scopes is equivalent to the intersection of the two relations.

Example 1.2.4 Let the relations R, R' and R'' be as shown in Figure 1.1. Figure 1.3 shows examples of the selection, projection, and join operations applied to these relations. The relation $\sigma_{x_3=c}(R')$ consists of those tuples in R' that have the value c for variable x_3. The relation $\pi_{\{x_2,x_3\}}(R')$ consists of the tuples in R', each with the component that corresponds to the variable x_1 removed; only the components that correspond to variables x_2 and x_3 are kept. Duplicate entries are removed, since a relation is a set and sets do not contain duplicate objects. The relation $R' \bowtie R''$ consists of tuples that are combinations

of pairs of tuples from R' and R'' which share common variables $\{x_2, x_3\}$. To construct $R' \bowtie R''$, we consider each tuple in R', match it up with all possible tuples in R'' that agree with it on the common variables, and delete duplicate components associated with these variables. For example, the tuple (b, b, c) in R' agrees with the tuples (b, c, 2) and (b, c, 3) in R'', resulting in the tuples (b, b, c, 2) and (b, b, c, 3). Similarly, the tuple (c, b, c) in R' results in the tuples (c, b, c, 2) and (c, b, c, 3). The tuple (c, n, n) in R' does not agree with any tuple in R'' on variables x_2 and x_3, so no additional tuples are added to $R' \bowtie R''$. □

x_1	x_2	x_3
b	b	c
c	b	c

(a) $\sigma_{x_3=c}(R')$

x_2	x_3
b	c
n	n

(b) $\pi_{\{x_2,x_3\}}(R')$

x_1	x_2	x_3	x_4
b	b	c	2
b	b	c	3
c	b	c	2
c	b	c	3

(c) $R' \bowtie R''$

Figure 1.3: Example of selection, projection, and join operations on relations.

1.2.4 Graphs: general concepts

Definition: a *graph* $G = (V, E)$ is a structure which consists of a finite set of *vertices* or *nodes*, $V = \{v_1, ..., v_n\}$ and a set of *edges* or *arcs*, $E = \{e_1, e_2, ...e_l\}$. Each edge e is incident to an unordered pair of vertices $\{u, v\}$ which are not necessarily distinct (as in the case of a loop). Although the vertices are unordered, they will often be written as an ordered pair (u, v). If $e = (u, v) \in E$ we say that e connects u and v and that u and v are *adjacent* or *neighbors*. The degree $d(u)$ of a vertex u in a graph is the number of its adjacent vertices.

A *path* is a sequence of edges $e_1, e_2, ..., e_k$ such that e_i and e_{i+1} share an endpoint. Namely, if $e_i = (u_1, v_1)$ and $e_{i+1} = (u_2, v_2)$, then v_1 and u_2 are the same. It is also convenient to describe a path using its vertices $v_0, v_1, ...v_k$, where $e_i = (v_{i-1}, v_i)$. In this case node v_o is called the *start-vertex* of the path, v_k is called the *end-vertex*, and the

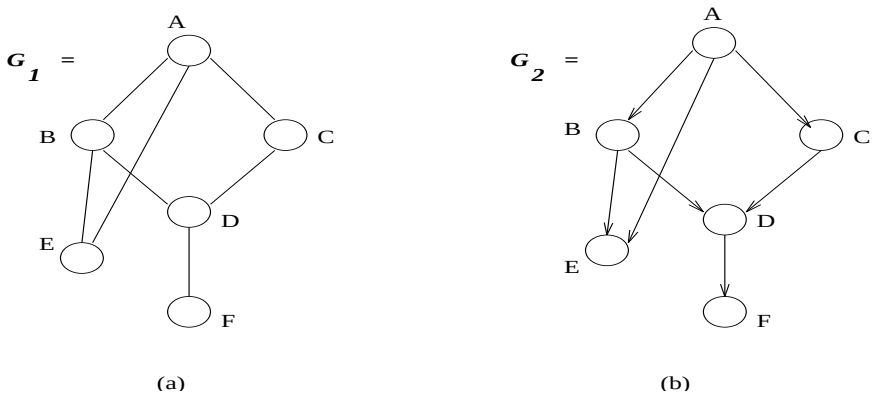


Figure 1.4: Two graphs: (a) undirected and (b) directed.

length of the path is k. A *cycle* is a path whose start and end vertices are the same. A path is *simple* if no vertex appears on it more than once. A cycle is *simple* if no vertex other than the start-end vertex appears more than once and the start-end vertex does not appear elsewhere in the cycle. If for every two vertices u and v in the graph there exists a path from u to v, then the graph is said to be *connected*. An undirected graph with no cycles is called a *tree*. Given a subset of the nodes S in the graph G, a *subgraph* relative to S, denoted G_S, is the graph whose nodes are in S and whose edges, all in G, are incident only to nodes in S. A graph is *complete* if every two nodes is adjacent. A *clique* in a graph is a complete subgraph.

A *directed graph (digraph)* is defined similarly to an undirected graph except that the pair of endpoints of an edge is now ordered; the first endpoint is the *start-vertex* of the edge and the second is the *end-vertex*. The edge $e = (u, v)$, also denoted $u \rightarrow v$, is said to be directed from u to v. The *outdegree* of a vertex v is the number of edges which have v as their start-vertex; the *indegree* of v is the number of edges which have v as their end-vertex. The set of nodes that point to node u is its *parents* and is denoted $pa(u)$. Similarly, the set of vertices to which u points is called the set of *child nodes* of u and is denoted $ch(u)$. A *directed path* is a sequence of edges $e_1, e_2, ..., e_k$ such that the end-vertex of e_{i-1} is the start-vertex of e_i. A directed path is a *a directed cycle* if the start-vertex of the path is the same as the end-vertex. A directed graph is *strongly connected* if for every vertex u and every vertex v there is a directed path from u to v. A directed graph is

acyclic if it has no directed cycles. The following example illustrates the above definitions.

Example 1.2.5 The graph G_1 in Figure 1.4a is an undirected graph over vertices $\{A, B, C, D, E, F\}$. The edge $e = (A, B)$ is in the graph while (B, C) is not. The sequence (A, B, D, F) is a path whose start-vertex is A and whose end-vertex is F. The path (A, B, D, C, A) is a simple cycle. The degree of vertex D is 3. The subgraph $\{A, B, E\}$ is a clique. The subgraph $\{A, B, E, D\}$ contains four edges: $\{(A, B), (B, E), (A, E), (B, D)\}$. The graph G_2 in Figure 1.4b is an acyclic directed graph. The indegree of D is 2, and its outdegree is 1; D in G_2 has two parents and one child node. □

Definition 1.2.6 (hypergraph) *A* hypergraph *is a pair $H = (X, S)$, where $S = \{S_1, \ldots, S_t\}$ is a set of subsets of V called* hyperedges.

1.2.5 Background in complexity

Throughout the book we will analayze the complexity of algorithms by determining their asymptotic efficiency. That is, we are concerned with how the running time of an algorithm increases with the size of the input, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs to the algorithm. An excellent comprehensive introduction into the asymptotic analysis of algorithms is given in [39] (Chapters 1 and 2). We briefly summarize from Chapter 2 several relevant concepts used throughout this book:

In general, an algorithm's complexity is its worst-case running time $T(n)$ over all its inputs of a fixed size n. The asymptotic analysis of algorithms use special notation for characterizing the running time. We will use the O-notation. Intuitively, if we say that an algorithm's complexity is $O(f(n))$ we mean that for large enough inputs the number of basic steps of the algorithm as a function of its input size n is bounded below by $c \cdot f(n)$ for some constant c. Formally, the O-notation describes the asymptotic upper bound. For a given function $g(n)$, $O(g(n))$ denotes the set of functions

$$O(g(n)) = \{f(n) : \quad \textit{there exists positive constants } c \textit{ and } n_0 \quad \textit{such that}$$

$$0 \leq f(n) \leq cg(n) \quad \textit{for all } n \geq n_0\}$$

For all values n greater than n_0, the value of $f(n)$ is on or below $g(n)$. To indicate that a function $f(n)$ is a member of $O(g(n))$, we write $f(n) = O(g(n))$.

The asymptotic upper bound provided by O-notation is meant to be tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. o-notation (small "o") is used to denote an upper-bound that is *not* asymptotically tight. Formally:

$$o(g(n)) = \{f(n) : \quad for\ any\ positive\ constant\ c\ and\ n_0\ such\ that$$

$$0 \leq f(n) \leq c \cdot g(n)\ for\ \ all\ n \geq n_0\}$$

The main difference between O-notation and o-notation is that in $f(n) = O(g(n))$, the bound $0 \leq f(n) \leq c \cdot g(n)$ holds for some constant $c > 0$, whereas in $f(n) = o(g(n))$, the bound $0 \leq f(n) \leq c \cdot g(n)$ holds for every constant $c > 0$. Intuitively, in the o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity.

Additional notations commonly used for performance evaluation are Ω-notation which is analogous to O-notation but denotes asymptotic lower bounds which are tight, and ω-notation, where ω to Ω is like o to O. Finally, θ notation is used to denote an asymptotic tight function that is both a lower and an upper bound. In the book we use O-notation primarily, even when the claims can be strengthened using θ notations.

Two additional central concepts we use when discussing complexity is *polynomial complexity* and *exponential complexity*. A *polynomial of degree d* is a function $p(n)$ of the form

$$p(n) = \sum_{i=1}^{d} a_i n^i$$

where the a_i are constants. A polynomial is asymptotically positive iff $a_d > 0$. We say that a function $f(n)$ is polynomially bounded if $f(n) = O(n^k)$ for some constant k.

For a constant a the function $f(n) = a^n$ is an exponential function. The rate of growth of exponential and polynomials can be related by the fact that for all a and b, such that $a > 1$

$$lim_{n \to \inf} \frac{n^b}{a^n} = 0$$

An algorithm is *tractable* if it has polynomial complexity. A class of problems is tractable if there exist a polynomial algorithm to solve it and intractable if it is known that there does not exists a polynomial algorithm for its solution.

NP-complete problems is a class of problems that are believed to be worst-case exponential. Namely, it is believed that for any algorithm that solves a problem in this class, there is some instances for which the algorithm will take an exponential number of steps in the problem size. Up to polynomial differences, the NP-complete problems all have the same complexity. That is, if there is a polynomial time algorithm for any NP-complete problem then this would yield a polynomial time algorithm for every NP-complete problem. *NP-complete problems* have the property that a potential solution to the problem can be verified in polynomial time. The class of *NP-hard problems* are at least as difficult as NP-complete problems. For the NP-hard class, there is no known way to verify a solution in polynomial time.

Constraint satisfaction problems, the focus of this book, are known to be NP-complete, and therefore, general purpose polynomial algorithms are clearly unavailable. The trust of all constraint processing is to develop algorithms that work well in a wide range of problem classes.

An algorithm is *complete* if it is guaranteed to solve the problem it addresses, and is otherwise *incomplete*. For the task of constraint satisfaction a complete algorithm will find a solution if one exists, or otherwise determine that the problem is inconsistent.

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

Chapter 2

Basics of Graphical Models

Our unifying perspective on graphical models is motivated by their various instantiations and associated algorithms. In this section, we will begin by defining the most common types of graphical models and provide examples of each type: these are constraint networks [13], Bayesian networks, Markov networks [30] and cost networks. These examples will be followed by a general framework for describing graphical models.

2.1 Classes of Graphical Models

There are two main divisions among the classes of graphical models. The first of these has to do with the kind information represented by the graph, with the main division being whether the information is deterministic or probabilistic. Constraint networks are, for example, deterministic; an assignment of variables is either valid or it is not. Markov networks and Bayesian networks, on the other hand, represent probabilistic relationships; the nodes represent random variables and the graph as a whole encodes the probability distribution of those random variables. There are Cost networks which represents preferences and which can be grouped with probabilistic networks as they are defined by real-valued functions as well. The second main division among graphical models has to do with how the information is encoded in the graph, with the two main classes being graphical models that have directed edges and graphical models that have undirected edges. For example, Markov networks are probabilistic networks that have undirected edges; the

edges do not encode an explicit directional relationship between the variables. Bayesian networks are similarly probabilistic, but they have directed edges, so they can explicitly encode directional relationships in the graph structure. Cost and constraint networks are primarily un-directional yet some constraints are functional and can be then associated with a directed model.

To make these divisions more concrete, let's take a very simple example of a relationship between two variables. Say we want to represent the relationship $A \vee B$ using a graphical model.

If nodes in this graph are logical binary variables, then the graph represents a constraint network. If we want to be able to control this $A \vee B$ relation and allow it to hold in some cases while we would not impose it in others, we can add another variable C that will be assigned true when the "OR" relation holds and false otherwise. The constraint network between A, B and C can be a complete graph over A, B, C which can be parameterized by the relation $C = A \vee B$. Alternatively, since C is a function of A and B we can use a directed graph that has directed edges from A and B into C. This directed graph mimics vividly the functional nature between the 3 variables which can be possibly useful. Figure 2.1 shows that different graphs.

If, on the other hand, we want to make a probabilistic version of this relationship, 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 produces an output of 1 with high probability $1 - \epsilon$ for some small ϵ. If we specify the prior probability of A as P_A and that of B as P_B, then we can assign the edge AB an appropriate function. Using ϵ_A and ϵ_B determine how noisy the NOISY-OR function is our graphical model could then be written in the following way, and would be a Markov Network:

$$f(A,B) = A(1-B)\cdot P_A(1-P_B)(1-\epsilon_A)+B(1-A)P_B(1-P_A)(1-\epsilon_B)+ABP_AP_B(1-\epsilon_A)(1-\epsilon_B)$$

This network represents the idea that if A is false then it is very likely that B is true and that if B is false then it is very likely that A is true. This relationship does not seem to have an directionality to it, in the same way that the crisp $A \vee B$ did not. The status of A seems to influence B just as naturally as the status of B influences A.

Figure 2.1: A Bayesian network where A and B point to C

If we want to force a directed graph on this relationship, There are two possibilities. The first is that we can arbitrarily pick a direction of influence, say, A influences B and draw a directed edge from A to B. This edge would represent the likelihood of B being true or false, given the value of A. This, however, seems somewhat ad hoc, especially if the relationship is symmetric. Another reasonable thing to do is to introduce a third variable C $NOISY-OR(A,B)$ the has directed edges into it from A and B. The graph that represents this, the Bayesian in Figure 2.1, explicitly suggests that A influences C and B influences C and A and B influence each other only indirectly.

From an algorithmic perspective, the second division, between directed and undirected graphical models, is more salient and received considerable treatment in the literature [30]. Indeed deterministic information seems to be merely a limiting case of nondeterministic information where probability values are limited to 0 and 1. Yet, this book will be focused 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 numerical vs boolean graphical models.

Graphical models include constraint networks [13] defined by relations of allowed tuples, probabilistic networks [30], defined by conditional probability tables over subsets of variables or by a set of potentials, cost networks defined by costs functions. 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.

2.1.1 Constraint Networks

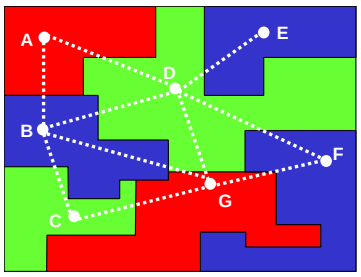
Constraint networks provide a framework for formulating real world problems as satisfying a set of constraints among variables, and they are the simplest and most computationally tractable of the graphical models we will be considering. Problems in scheduling, design, planning and diagnosis are often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

Let's take scheduling as an example: one approach to formulating a scheduling problem as a constraint satisfaction problem is to create a variable for each combination resource and time slice (e.g. the conference room at 3pm on Tuesday). 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 modeled as formal 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 network, 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. This is the most basic question we can ask of the network. 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 2.1.1 (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* $\bowtie$ *simply note that the constraints can be combined to form a new constraint by the join operator. It will serve to unify constraint networks within framework of graphical models. 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*



(a) Graph coloring problem

Figure 2.2: A constraint network example of a map coloring

solutions, $sol(\mathcal{R}) = \bowtie_i R_i$. *The minimal domain of a variable* X *is all its values that participate in any solution. Using relational operations,* $MinDom(X_i) = \pi_{X_i} \bowtie_j R_j$

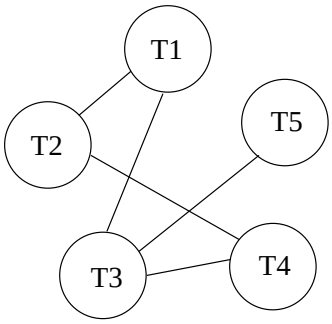
The primary query over a constraint network is deciding if it has a solution. Other relevant queries are enumerating or counting the solutions, or finding the minimal relations over a subset of variables.

Definition 2.1.2 (constraint graph) *The* constraint graph *of a constraint network 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 2.1.3 The map coloring problem in Figure 2.2(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 2.2 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 2.1.4 Constraint networks are also particularly useful for expressing and solving scheduling problems. Consider the problem of scheduling five tasks (T1, T2, T3, T4,



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 2.3: The constraint graph and constraint relations of the scheduling problem in Example 1.

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
- 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 2.3, and the relations expressed by the graph are shown beside the figure. □

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. $\varphi = \alpha \& \beta = (P \vee \neg Q \vee \neg R) \& (R)$ 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 tuples over (A, B) except $(A = 1, B = 0)$.

2.1.2 Cost Networks

In constraint networks, the relations are always 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, each relation is a cost-component, and the sum of these cost-components is the overall combined cost function of the network. The primary task is to find an assignment of the variables such that the combined cost function is optimized (minimized or maximized). Cost networks enable one to express preferences among relations and their assignments and therefore preferences among different solutions.

In the real world, problems are modeled using both constraints and cost functions. The constraints can be expressed explicitly as being of a different type than the cost functions, or they can be included as cost components themselves.

Superficially, optimization over constraint networks and cost networks is different. However, as we show in the following definition, a single definition of optimization over a graphical model can be used whether the underlying relations are boolean or real-values. In fact, more complex extensions, in which constraints are combined not by sums but by products or other algebraic expressions, can be handled using the same definition. In the

constraint community different types of cost networks are distinguished and called *Valued CSPs*.

Definition 2.1.5 (cost network, combinatorial optimization) *A* cost network *is a 4-tuple,* $\langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$*, 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 cost functions* $\mathbf{F} = \{f_1, \ldots, f_r\}$*. Each* f_i *is a real-valued function defined on a subset of variables* $\mathbf{S}_i \subseteq \mathbf{X}$*. The cost components are combined into a global cost function via the* $\sum$ *operator.*

The primary optimization task (which we will represent as a minimization, w.l.o.g) is to find a solution for the global cost function $F = \sum_i f_i$*. Namely, finding a tuple* x *such that* $F(x) = \min_x \sum_i f_i(x)$*. The graph of a cost networks associates a node with a variable and any two variables that are included in a single scope are connected. 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 the Weighted Constraint Satisfaction Problem (WCSP) [7]. These networks extends the classical constraint satisfaction problem formalism with *soft constraints*, that is, positive integer-valued cost functions. Formally,

Definition 2.1.6 (WCSP) *A* Weighted Constraint Satisfaction Problem (wcsp) *is a triplet* $\langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$ *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,* $f_i : D_{X_{i_1}} \times ... \times D_{X_{i_t}} \to \mathbb{N}$*, 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 [5], bioinformatics [11, 42], combinatorial auctions [34, 13] or maximum satisfiability problems [10].

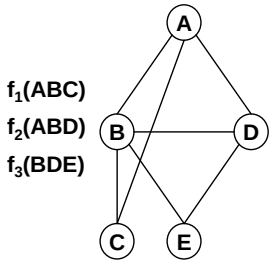
Example 2.1.7 Figure 2.4 shows an example of a WCSP instance with boolean variables. The cost functions are given in Figure 2.4(a), and the associated graph is shown in Figure 2.4(b). Note that a value of ∞ in the cost function denotes a hard constraint (i.e., high penalty). You should check that the minimal cost solution of the problem is 5, which corresponds to the assignment $(A = 0, B = 1, C = 1, D = 0, E = 1)$. □

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 2.4: A cost network.

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.

Definition 2.1.8 (MAX-CSP) *A* MAX-CSP *is a WCSP* $\langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$ *with all penalty costs equal to* 1. *Namely,* $\forall f_i \in \mathbf{F}$, $f_i : D_{X_{i_1}} \times ... \times D_{X_{i_t}} \rightarrow \{0, 1\}$, *where scope*$(f_i) = S_i = \{X_{i_1}, ..., X_{i_t}\}$.

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.

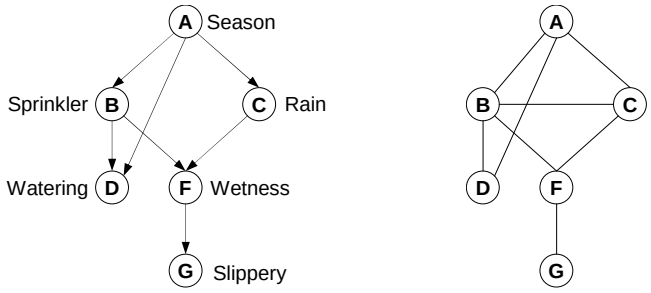
2.1.3 Probability Networks

As mentioned previously, Bayesian networks and Markov networks are the two primary formalisms for expressing probabilistic information via graphical models. A **Bayesian network** [30] 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 between the respective variables, and this influence is quantified by 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 2.1.9 (Bayesian networks) *A* Bayesian network (BN) *is a 4-tuple* $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$. $\mathbf{X} = \{X_1, \ldots, X_n\}$ *is a set of ordered variables defined over domains* $\mathbf{D} = \{D_1, \ldots, D_n\}$, *where* $o = \{X_1, \ldots, X_n\}$. *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 *define an associated 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. *A Bayesian network represents a probability distribution over* $\mathbf{X}$, $P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i | x_{pa(X_i)})$. *where* x_S *is the projection of a tuple* x, $x = (x_1, \ldots, x_n)$ *over a subset* S. *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 yield a valid probabilistic network; that is, they represent a valid joint probability distribution.

In addition to the directed graph G associated with a Bayesian network, it is often useful to refer to another associated graph called the *moral graph* which is undirected. The moral graph can be obtained from G by connecting the parents of each child node and removing the arrows from all the edges. Therefore, the moral graph for a Bayesian network has a node for each variable and any two nodes are connected by an edge if they both appear in the scope of a single relation.



(a) Directed acyclic graph (b) Moral graph

Figure 2.5: Belief network $P(g, f, c, b, a) = P(g|f)P(f|c, b)P(d|a, b)P(c|1)P(b|a)P(a)$

Example 2.1.10 [30] Figure 2.5(a) is a Bayesian network over six variables, and Figure 2.5(b) shows the corresponding moral graph. The example expresses the causal relationship between variables "season" (A), "the automatic sprinkler system is "on"' (B), "wheather it rains or 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 2.6.

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 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 2.1.11 (queries over Bayesian networks) *Let $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$ be a Bayesian network. Given evidence e and letting x denote a tuple over all variables, the primary queries over Bayesian networks are to find the following quantities:*

1. **Marginals:** *The posterior marginals (or beliefs) of $X_i = x_i$ are $bel(x_i) = P(x_i|e)$ for each x_i not in e.*

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	false	0.9
Fall	false	0.6
Winter	false	0.1
Spring	false	0.7

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

F	G	$P(G\|F)$
false	false	0.9
true	false	0

Figure 2.6: Possible CPTs that accompany our example

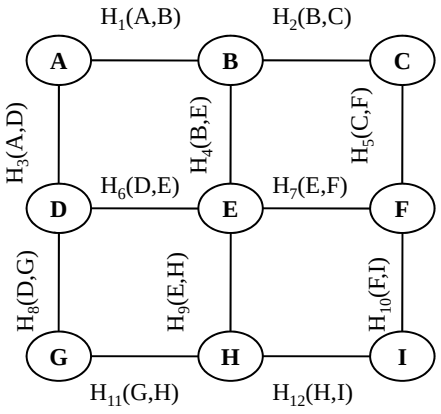
2. **Probability of evidence:** *The probability of the evidence $P(e)$ given the probability distribution defined by $\mathcal{P}$.*

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

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.*

These queries are applicable to a variety of applications such as situation assessment, diagnosis, probabilistic decoding and linkage analysis, to name a few.

Markov networks also called *Markov Random Fields (MRF)*, Markov networks are undirected probabilistic graphical models very similar to Bayesian networks. Unlike



(a)

D	E	H_6(D,E)
0	0	20.2
0	1	12
1	0	23.4
1	1	11.7

(b)

Figure 2.7: (a) An example 3×3 square Grid Markov network (ising model) and (b) An example potential $H_6(D, E)$

Bayesian networks, however, they are defined over an undirected graph. Moreover, whereas the relations in Bayesian networks are conditional probability tables of children given their parents, in Markov networks the relations, called compatibility functions or potentials, can be defined over any subset of variables in the graph that form a clique.

Definition 2.1.12 (Markov Networks) *A Markov network is a graphical model* $\mathcal{T} = \langle \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{H} \rangle$ *where* $\boldsymbol{H} = \{H_1, \ldots, H_m\}$ *is a set of potential functions where each potential* H_i *is a non-negative real-valued function defined over variables* $\boldsymbol{S}_i$*. The Markov network represents a joint distribution over the variables* $\boldsymbol{X}$ *given by:*

$$P(\boldsymbol{x}) = \frac{1}{Z} \prod_{i=1}^{m} H_i(\boldsymbol{x}) \quad \textit{where} \quad Z = \sum_{\boldsymbol{x} \in \boldsymbol{X}} \prod_{i=1}^{m} H_i(\boldsymbol{x})$$

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

The primary queries over Markov networks are computing the posterior marginal distribution over all variables $X_i \in \mathbf{X}$ and finding the partition function.

Example 2.1.13 Figure 2.7 shows a 3×3 square grid Markov network with 9 variables $\{A, B, C, D, E, F, G, H, I\}$. The twelve potentials are: $H_1(A, B)$, $H_2(B, C)$, $H_3(A, D)$, $H_4(B, E)$, $H_5(C, F)$, $H_6(C, D)$, $H_7(D, E)$, $H_8(D, G)$, $H_9(E, H)$, $H_{10}(F, I)$, $H_{11}(G, H)$

and
$H_{12}(H, I)$. The Markov network represents the probability distribution formed by taking a product of these twelve functions and then normalizing. Namely,

$$P(A, B, \ldots, I) = \frac{1}{Z} \prod_{i=1}^{12} H_i$$

where Z is the partition function. □

2.1.4 Mixed networks

As we have seen that graphical models can accommodate both probabilistic and deterministic information. Probabilistic information typically associates a strictly positive number with an assignment of variables, quantifying our expectation that the assignment may be realized. The deterministic information has a different semantics, annotating assignments with binary values, either *valid* or *invalid*. In this section, we introduce the mixed network, a graphical model which allows for both probabilistic information and deterministic constraints and which provides a coherent meaning to the combination.

Definition 2.1.14 (mixed networks) *Given a belief network $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{G}, \mathbf{P} \rangle$ that expresses the joint probability $P_{\mathcal{B}}$ and given a constraint network $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ that expresses a set of solutions ρ, a mixed network based on $\mathcal{B}$ and $\mathcal{R}$ denoted $\mathcal{M}_{(\mathcal{B},\mathcal{R})} = \langle \mathbf{X}, \mathbf{D}, \mathbf{G}, \mathbf{P}, \mathbf{C} \rangle$ is created from the respective components of the constraint network and the belief network as follows: the variables $\mathbf{X}$ and their domains are shared, (we could allow non-common variables and take the union), and the relationships include the CPTs in $\mathbf{P}$ and the constraints in $\mathbf{C}$. The mixed network expresses the conditional probability $P_{\mathcal{M}}(\mathbf{X})$:*

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

Clearly, $P_{\mathcal{B}}(\bar{x} \mid \bar{x} \in \rho) = \frac{P_{\mathcal{B}}(\bar{x})}{P_{\mathcal{B}}(\bar{x} \in \rho)}$. When clarity is not compromised, we will abbreviate $\langle \mathbf{X}, \mathbf{D}, \mathbf{G}, \mathbf{P}, \mathbf{C} \rangle$ to $\langle \mathbf{X}, \mathbf{P}, \mathbf{C} \rangle$.

Belief updating, MPE and MAP queries can be extended to mixed networks straightforwardly. 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 $P_{\mathcal{B}}(\bar{x} \in \rho(\mathcal{R}))$. We will call this *a Constraint Probability Evaluation (CPE)*. Note that evidence is a special type of constraint. We will elaborate on this next.

Definition 2.1.15 (queries on mixed networks) *We have the following 2 new queries:*

- CPE: *Given a mixed network $\mathcal{M}_{(\mathcal{B},\mathcal{R})}$, where the belief network is defined over variables $\mathbf{X} = \{X_1, ..., X_n\}$ and the constraint portion is a either a set of relational constraints over a set of subsets $Q = \{Q_1, ...Q_r\}$, where $Q_i \subseteq \mathbf{X}$, the* constraint*, Probability Evaluation (CPE) task is to find the probability $P_{\mathcal{B}}(\bar{x} \in \rho(\mathcal{R}))$. If R is a CNF expression, 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 $P(X|\varphi) = \alpha \cdot P(X \wedge \varphi)$ 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 $((X = x) \wedge \varphi)$. 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.*

Popular queries over Mixed un-directional networks are the following.

Definition 2.1.16 (The Weighted Counting Task) *Given a mixed network $\mathcal{M} = \langle \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{F}, \boldsymbol{C} \rangle$, the weighted counting task is to compute the normalization constant given by:*

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

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

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

We will refer to Z as **weighted counts**.

Definition 2.1.17 (Marginal task) *Given a mixed network $\mathcal{M} = \langle \boldsymbol{X}, \boldsymbol{D}, \boldsymbol{F}, \boldsymbol{C} \rangle$, the marginal task is to compute the marginal distribution at each variable. Namely, for each variable X_i and $x_i \in \boldsymbol{D}_i$, compute:*

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

To be able to use the constraint portion of the mixed network more effectively, for the remainder of the thesis, we require that all zero probabilities in the mixed network are also represented as constraints. It is easy to define such a network as we show below. The new constraints are redundant though.

Definition 2.1.18 (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.

2.2 General Graphical models

In the previous section we defined and explored several classes of graphical models, encompassing the different kinds of information and the various queries that we might want

to represent. Often, these models are treated separately, but we have intentionally attempted to describe them in a common language and to draw connections between them. In this section, we will go a step beyond this and provide a general formulation of graphical models and reasoning problems that unifies the previously described models and tasks into a single framework.

To do this, we will define a graphical model as a collection of function over a set of variables that conveys probabilistic, deterministic or preferential information and whose structure is captured by a graph.

Definition 2.2.1 (graphical model) *A* graphical model $\mathcal{M}$ *is a 4-tuple,* $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes \rangle$, *where:*

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

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

Definition 2.2.2 (a reasoning problem) *A* reasoning problem *over a graphical model* $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes \rangle$ *is defined by a marginalization operator* $\Downarrow_{\mathbf{Y}}$ *and a set of subsets of* $\mathbf{X}$ *that are of interest. It is therefore a triplet,* $\mathcal{P} = \langle \mathcal{M}, \Downarrow_{\mathbf{Y}}, \{\mathbf{Z}_1, \ldots, \mathbf{Z}_t\} \rangle$, *where* $\mathbf{Z} = \{\mathbf{Z}_1, \ldots, \mathbf{Z}_t\}$ *is a set of subsets of variables of* $\mathbf{X}$. *If* $\mathbf{S}$ *is the scope of function* f *and* $\mathbf{Y} \subseteq \mathbf{X}$, *then* $\Downarrow_{\mathbf{Y}} f \in \{\overset{max}{\mathbf{S}-\mathbf{Y}} f, \overset{min}{\mathbf{S}-\mathbf{Y}} f, \Pi_{\mathbf{Y}} f, \overset{\sum}{\mathbf{S}-\mathbf{Y}} f\}$ *is a marginalization operator. The reasoning problem* $\mathcal{P}$ *can be written as a vector-valued function over the scopes* $\mathbf{Z}_1, \ldots, \mathbf{Z}_t$ *where the goal is to compute* $\mathcal{P}_{\mathbf{Z}_1,\ldots,\mathbf{Z}_t}(\mathcal{M}) = (\Downarrow_{\mathbf{Z}_1} \otimes_{i=1}^{r} f_i, \ldots, \Downarrow_{\mathbf{Z}_t} \otimes_{i=1}^{r} f_i)$. r *is the number of functions.*

We will focus often on reasoning problems defined by $\mathbf{Z} = \{\emptyset\}$. The marginalization operator is sometimes called *elimination* operator because it removes some arguments

[1] The combination operator can also be defined axiomatically [37].

from the scope of the input function. Specifically, $\Downarrow_{\mathbf{Y}} f$ is a function whose scope is $\mathbf{Y}$. It therefore removes variables $\mathbf{S} - \mathbf{Y}$ from $\mathbf{S} = scope(f)$. 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.

We will now go back and indicate how each of the framework mentioned earlier fits the general graphical model definition.

Constraint satisfaction is a reasoning problem $\mathcal{P} = \langle \mathcal{R}, \Pi, \mathbf{Z} \rangle$, where $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \bowtie \rangle$ is a constraint network, the marginalization operator $\Downarrow_{\mathbf{Y}}$ is the projection operator $\Pi_{\mathbf{Y}}$, and $\mathbf{Z} = \{\emptyset\}$. Therefore, the task is to evaluate $\mathcal{P}_{\{\emptyset\}} = \Downarrow_{\emptyset} \otimes_i f_i = \Pi_{\emptyset}(\bowtie_i f_i)$. This corresponds to enumerating all the solutions of the constraint network.

If we want to count the number of solutions instead, 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 combination operator be a product over the cost-based representation of the relations and 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.

Finding a minimum solution for a *cost network* can be expressed by letting the marginalization operator be *minimization*. Therefore, the task is to evaluate $\Downarrow_{\emptyset} \otimes_i f_i = \min_X \sum_i f_i$. Naturally, Max-CSP can be defined in the same way.

In a *Bayesian network* the combination operator $\otimes = \prod$ is the product operator as in $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$, where $P_G = \{P_1, \ldots, P_n\}$, are the set of functions F. The query of finding the probability of the evidence is defined by $Z = \{\emptyset\}$. An MPE reasoning task is defining by letting $\Downarrow_{\emptyset} \otimes_i f_i = \max_X \prod_i P_i$, i.e., by letting the marginalization operator be maximization and letting be $\mathbf{Z} = \{\emptyset\}$. Given evidence e, the *belief updating* task can be formulated using the marginalization operator $\Downarrow_{\mathbf{Y}} = \sum_{\mathbf{X}-\mathbf{Y}}$, and $\mathbf{Z}_i = \{X_i\}$. Namely, $\forall X_i, \Downarrow_{X_i} \otimes_k f_k = \sum_{\{X - X_i | E=e\}} \prod_k P_k$.

Definition 2.2.3 (Flat functions) *If a function in a graphical model assigns the value "0" to some tuple in it scope, then that function implicitly expresses a constraint, namely, that the tuple assigned a zero is never consistent. Therefore, for each function f_i in a graphical model we can define an associated* flat *constraint R_i which includes all tuples in the domain of f_i that are not assigned the value "0". In this way, a general graphical model*

can be associated with a flattened version, i.e., a constraint network. In the following chapters, when we refer to the constraint network of a general graphical model, we will be referring to the flattened version of the model. When all tuples are consistent in a graphical model's flat constraint network, we say that the graphical model strictly positive.

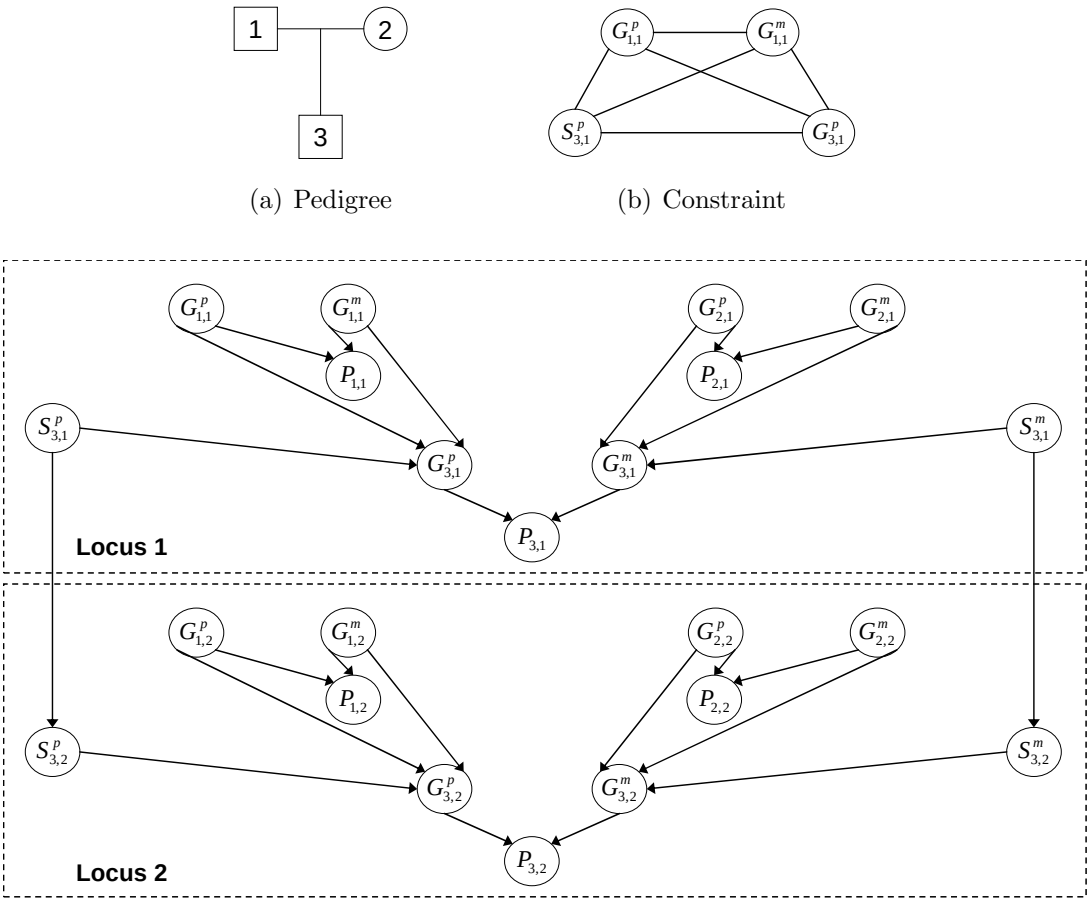
2.3 Example of Real applications and Benchmarks

2.3.1 Linkage analysis

We describe next the problem of genetic linkage analysis [29], which is usually formulated as a belief network, but can be represented as a mixed network to leverage the deterministic information abundantly present.

Human cells contain 23 pairs of chromosomes, which are sequences of DNA containing the genetic makeup of an individual and are inherited from a persons parents. Each pair consists of one chromosome inherited from the persons father and one from their mother. Locations on these chromosomes are referred to as loci (singular: locus). A locus which has a specific function is known as a gene. These functions, which can be a result of a combination of multiple genes, can determine a persons blood type, hair color, or their susceptibility to a disease. The actual state of the genes is called the genotype and the observable outcome of the genotype (e.g., trait) is called the phenotype. A genetic marker is a locus with a known DNA sequence which can be found in each person in the general population. These markers are used to help locate disease genes.

Each parent contains 23 *pairs* of chromosomes, however they each pass on 23 chromosomes, each resulting from a combination of their own pair, resulting in the child having 23 pairs. It is possible for the transferred copy to be entirely a duplicate of the chromosome from the parents father or from the parents mother (the offsprings grandfather or grandmother), however more likely it contains non-overlapping sequences from both. The locations on the chromosome where the sequences switch between the two chromosomes of the parents are known as crossover or recombination events. The recombination frequency, θ, (also called the recombination fraction) between two consecutive markers is defined as the probability of a recombination event occurring between them. This



(a) Pedigree (b) Constraint

(c) Mixed network

Figure 2.8: Genetic linkage analysis

frequency is related to the physical distance between them.

Figure 2.8(a) shows the simplest pedigree, with two parents (denoted by 1 and 2) and an offspring (denoted by 3). Square nodes indicate males and circles indicate females. Figure 2.8(c) shows the usual belief network that models this small pedigree for two particular loci (locations on the chromosome). There are three types of variables, as follows. The G variables are the genotypes (the values are the specific alleles, namely the forms in which the gene may occur on the specific locus), the P variables are the phenotypes (the observable characteristics). Typically these are evidence variables, and for the purpose of the graphical model they take as value the specific unordered pair of

alleles measured for the individual. The S variables are selectors (taking values 0 or 1) are auxiliary variables representing gene flow in the pedigree. The upper script p stands for paternal, and the m for maternal. The first subscript number indicates the individual (the number from the pedigree in 2.8(a)), and the second subscript number indicates the locus. The interactions between all these variables are indicated by the arcs in Figure 2.8(c).

Due to the genetic inheritance laws, many of these relationships are actually deterministic. For example, the value of a selector variable determines the genotype variable. Formally, if a is the father and b is the mother of x, then:

$$G^p_{x,j} = \begin{cases} G^p_{a,j}, & if \quad S^p_{x,j} = 0 \\ G^m_{a,j}, & if \quad S^p_{x,j} = 1 \end{cases}$$

and,

$$G^m_{x,j} = \begin{cases} G^p_{b,j}, & if \quad S^m_{x,j} = 0 \\ G^m_{b,j}, & if \quad S^m_{x,j} = 1 \end{cases}$$

The CPTs defined above are deterministic and functional, and can be captured by a constraint, whose constraint graph is depicted graphically in Figure 2.8(b). The only real probabilistic information is defined by the CPTs between selector variables and the prior probabilities of the founders, namely the individuals having no parents in the pedigree. Figure 2.8c provides description across 2 markers of the 3-member family. The transition probabilities between selectors in different loci capturing the probability of recombination is given by:

$$P(S^p_{x,j}|S^p_{x,j-1}) = \begin{cases} 1-\theta, & \theta \\ \theta, & 1-\theta \end{cases}$$

Figure 2.9 provides the mixed network formulation of a founder variable (top of Figure), on the bottom left we have the probabilistic subnetwork that consists of three independent variables and on the right there is a constraint subnetwork. Figure **??** describes the 3 member family formulation as a mixed network.

Genetic linkage analysis is an example of a belief network that contains many deterministic or functional relations that can be exploited as constraints. The typical reasoning

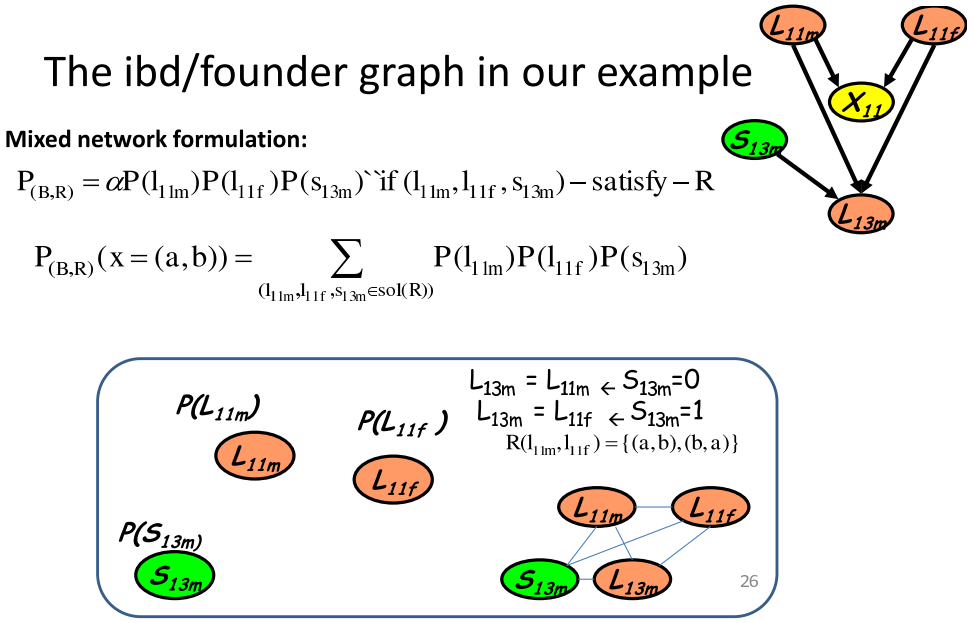


Figure 2.9: A non-founder mixed network: **Figure should be redone**

tasks are equivalent either to computing the probability of the evidence, or to maximum probable explanation (e.g., haplotyping).

2.3.2 Probabilistic decoding

The purpose of *channel coding* is to provide reliable communication through a noisy channel. Transmission errors can be reduced by adding redundancy to the information source. For example, a *systematic error-correcting code* [26] maps a vector of K *information bits* $u = (u_1, ..., u_K), u_i \in \{0, 1\}$, into an N-bit *codeword* $c = (u, x)$, adding $N - K$ *code bits* $x = (x_1, ..., x_{N-K}), x_j \in \{0, 1\}$. The *code rate* $R = K/N$ is the fraction of the information bits relative to the total number of transmitted bits. A broad class of systematic codes includes linear block codes. Given a binary-valued *generator matrix* G, an *(N,K) linear block code* is defined so that the codeword $c = (u, x)$ satisfies $c = uG$, assuming summation modulo 2. The bits x_i are also called the *parity-check* bits. For example, the generator matrix

```
1 0 0 0 1 1 0
```

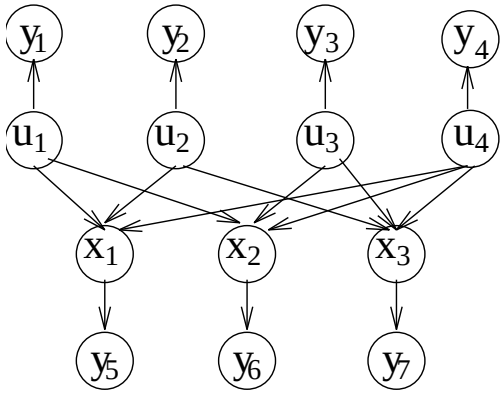


Figure 2.10: Belief network for a (7,4) Hamming code

```
G =  0 1 0 0 1 0 1
     0 0 1 0 0 1 1
     0 0 0 1 1 1 1
```

defines a (7,4) *Hamming code*.

The codeword $c = (u, x)$, also called the *channel input*, is transmitted through a noisy channel. A commonly used Additive White Gaussian Noise (AWGN) channel model assumes that independent Gaussian noise with variance σ^2 is added to each transmitted bit, producing a *real-valued channel output* y. Given y, the decoding task is to restore the input information vector u [24, 26, 9].

The decoding problem can be formulated as a probabilistic inference task over a Bayesian network [26]. For example, a (7,4) Hamming code can be represented by the belief network in Figure 2.10, where the bits of u, x, and y vectors correspond to the nodes, the parent set for each node x_i is defined by non-zero entries in the $(K + i)$th column of G, and the (deterministic) conditional probability function $P(x_i|pa_i)$ equals 1 if $x_i = u_{j_1} \oplus ... \oplus u_{j_p}$ and 0 otherwise, where $\oplus$ is addition modulo 2. Each output bit y_j has exactly one parent, the corresponding channel input bit. The conditional density function $P(y_j|x_j)$ is a Gaussian (normal) distribution $N(x_j; \sigma)$, where the mean equals the value of the transmitted bit, and σ^2 is the noise variance.

The probabilistic decoding task can be formulated in two ways. Given the observed output y, the task of *bit-wise* probabilistic decoding is to find the most probable value of

each *information bit*, namely:

$$u_k^* = \arg \max_{u_k \in \{0,1\}} P(u_k|y), \ for\ 1 \leq k \leq K.$$

The *block-wise* decoding task is to find a maximum a posteriori (maximum-likelihood) *information sequence*

$$u' = \arg \max_u P(u|y).$$

Block-wise decoding is sometimes formulated as finding a most probable explanation (MPE) assignment (u', x') to the codeword bits, namely, finding

$$(u', x') = \arg \max_{(u,x)} P(u, x|y).$$

Accordingly, bit-wise decoding, which requests the posterior probabilities for each information bit, can be solved by belief updating algorithms, while the block-wise decoding translates to the MAP or MPE tasks, respectively.

In the coding community, decoding error is measured by the *bit error rate (BER)*, defined as the average percentage of incorrectly decoded bits over multiple transmitted words (blocks). It was proven by Shannon [36] that, given the noise variance σ^2, and a fixed code rate $R = K/N$, there is a theoretical limit (called *Shannon's limit*) on the smallest achievable BER, no matter which code is used. Unfortunately, Shannon's proof is non-constructive, leaving open the problem of finding an optimal code that achieves this limit. In addition, it is known that low-error (i.e., high-performance) codes tend to be long [33], and thus intractable for exact (optimal) decoding algorithms [26]. [26].

Linear block codes can be represented by four-layer belief networks having K nodes in each layer (see Figure 2.11). The two outer layers represent the channel output $y = (y^u, y^x)$, where y^u and y^x result from transmitting the input vectors u and x, respectively. The input nodes are binary (0/1), while the output nodes are real-valued.

2.3.3 Networks from optimization and constraints

SPOT5 benchmark contains a collection of large real scheduling problems for the daily management of Earth observing satellites [7]. These problems can be described as follows: Given a set P of photographs which can be taken the next day from at least one of the

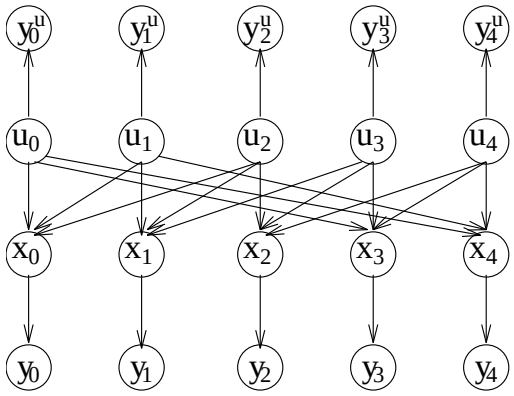


Figure 2.11: Belief network for structured (10,5) block code with P=3.

three instruments, w.r.t. the satellite trajectory; Given, for each photograph, a weight expressing its importance; Given a set of imperative constraints: non overlapping and minimal transition time between two successive photographs on the same instrument, limitation on the instantaneous data flow through the satellite telemetry; The goal is to find an admissible subset P0 of P which maximizes the sum of the weights of the photographs in P0 when all imperative constraints are satisfied.

The can be modeled as a weighted CSP by:

- Associating a variable Xi with each photograph pi 2 P;
- Associating with X_i a domain D_i to express the different ways of achieving p_i and adding to D_i a special value, called rejection value, to express the possibility of not selecting the photograph pi;
- Associating with every X_i an unary constraint forbidding the rejection value, with a valuation equal to the weight of p_i;
- Translating as imperative constraints (binary or ternary) the constraints of non overlapping and minimal transition time between two (or three) photographs on the same instrument, and of limitation on the instantaneous data flow. Each imperative constraint is defined over a subset of two or three photographs and for each value combination of its scope variables it associates a high penalty cost (106) if the corresponding photographs cannot be taken simultaneously, on the same instrument.

The task is to compute: $\min_X \sum_{i=1}^{r} f_i$, where r is the number of unary, binary and ternary cost functions in the problem.

2.4 Bibliographical notes

2.5 Exercises

Chapter 3

Inference: Bucket-elimination for Deterministic Networks

NOTE: What is a scheme? Is this a technical term? A scheme is another loose word for an algorithm but hint on being more general. It may be a principle for many algorithms

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. 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 understand in the context of constraint networks. Therefore, this chapter will introduce bucket elimination in its application to constraint networks. In the next chapter, we will apply this scheme to probabilistic inference 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, this target query is whether or not an input 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 3.1. The task is to assign a color (green or red) to each node in the graph so that adjacent nodes will have different colors. Here is one way to solve this problem: consider node E first. It can be

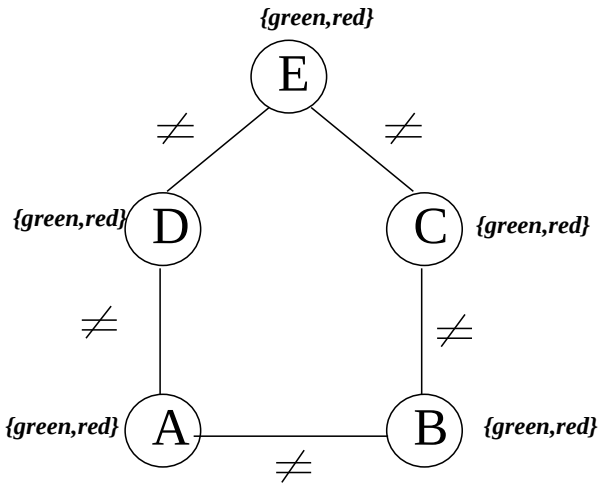


Figure 3.1: A graph coloring example

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. 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 E and C on the other variables in the network, we can ignore them 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 [15] 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 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. 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 D's bucket, and so on. The initial partitioning of the constraints is depicted in Figure 3.2a. In general, each constraint is placed in the bucket of its latest variable.

After this initialization step, the buckets are processed from last to first. This means that we compute the constraint that the bucket-variable induce 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 3.2b. 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 constraint is inferred on all of the rest of the participating variables. It can be shown that once all the buckets are processed, and if no inconsistencies were discovered, a solution can be generated in a *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 proceeding to the last. It is guaranteed that this process will continue until all the variables are assigned a value from their respective domains, thus yielding a *solution* to the problem.

In general, the input to a bucket elimination algorithm is a graphical model and a query. The graphical model is specified by a collection of functions over subsets of variables. The algorithm partitions these functions into buckets, with each bucket indexed by a single variable. The partition depends only on the ordering of the variables and their scopes: a bucket contains a function if the function has the bucket's variable as an argument and if the function has no later variable as an argument. Once the partitioning is completed, buckets are processed from last to first. When a specific bucket, say $bucket_X$, is processed, an "elimination procedure" is applied to the functions in $bucket_X$ that results in a new function that does not have X as an argument. We say that the function does not "mention" X. This function summarizes the "effect" of X on the remainder of the

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

(a)

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

(b)

Figure 3.2: A schematic execution of adaptive-consistency

problem and is placed in on the unprocessed buckets, according to the function placement rule.

3.1 The case of Constraint Networks

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

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 3.3: Adaptive-Consistency as a bucket-elimination algorithm

Using these operations, Adaptive-consistency can be specified as in Figure 3.3. 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.

The adaptive-consistency algorithm specifies that it returns a "backtrack-free" network along the ordering d. A common approach to finding a solution in a constraint satisfaction problem is to perform backtracking search, that is, to assign values to the variables in a certain order, 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, backtracking search will return to a previous variable, change its value, and proceed again along the ordering. We say that a network is *backtrack-free* along an ordering d of its variables if it is guaranteed that a dead-end will never occur.

Theorem 3.1.2 (Correctness and Completeness of Adapative-consistency) *[15] 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 generates an equivalent representation that is backtrack-free along the input variable ordering.* □

Proof: see exercises ∎

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 using 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 here.

Definition 3.1.3 (induced-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, *denoted $w(d)$, is the maximum width over all nodes. The* width of a graph *is the minimum width over all the orderings of the graph.*

Example 3.1.4 Consider the graph coloring problem depicted in Figure 3.4 (domains are 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 3.5, 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 E's bucket 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 the bucket of C (see Figure 3.5 for details). Next, the algorithm processes C's bucket which contains $C \neq A$ and the new constraint R_{DCB}. (For the explicit constraint see Figure 3.5). Joining these two constraints and projecting out D yields a constraint R_{DB} that is placed in the bucket of D, and so on.

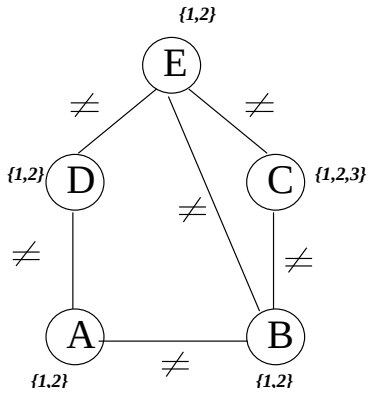
Applying adaptive-consistency along ordering d_1 generates a different set of constraints. Note that while ordering d_1 generates only binary constraints, ordering d_2 generates a ternary constraint.

A couple of notes on our presentation of bucket-elimination: notice 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. So the constraint is redundant. Also, the constraint R_{BE}, recorded when processing bucket C, is the universal constraint and should therefore not be recorded at all; we chose to display it only to illustrate the general case. □

An alternative graphical illustration of the algorithm's performance along d_2 is given in Figure 3.5. The figure shows, through the changing graph, how constraints are generated in ordering $d_2 = A, B, D, C, E$, and how a solution is created in the reverse order.

Generating the induced-graph along the orderings $d_1 = E, B, C, D, A$ and $d_2 = A, B, D, C, E$ leads to the two graphs in Figure 3.6. The broken arcs 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 bounded by the number of parents of the corresponding variable in the induced ordered graph which is equal to its induced-width.

Theorem 3.1.5 *The time and space complexity of Adaptive-Consistency is* $O(n \cdot (2k)^{w^*(d)+1})$ *and* $O(n \cdot k^{w^*(d)})$ *respectively, where* n *is the number of variables,* k *bounds the domain*



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 3.4: A modified graph coloring problem

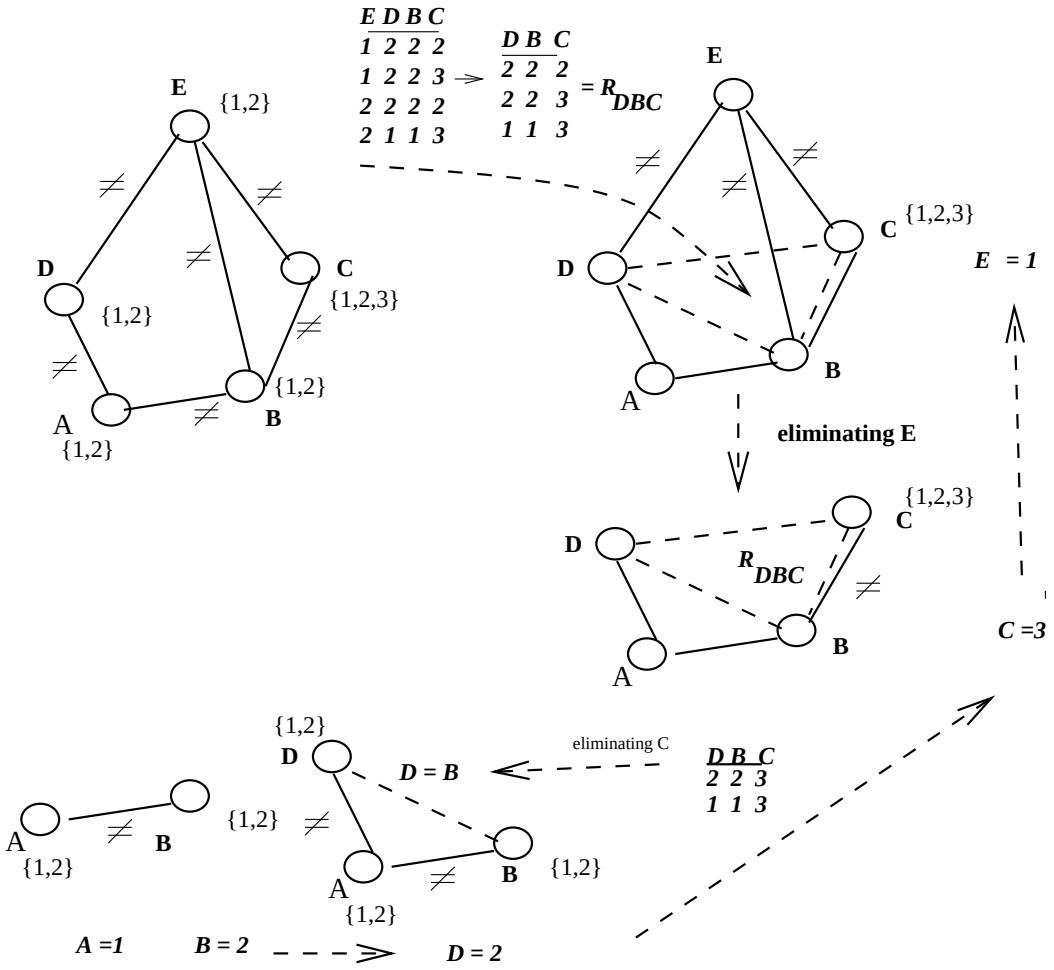


Figure 3.5: A schematic variable-elimination and solution genration process is backtrack-free (comment: change the order of d_2)

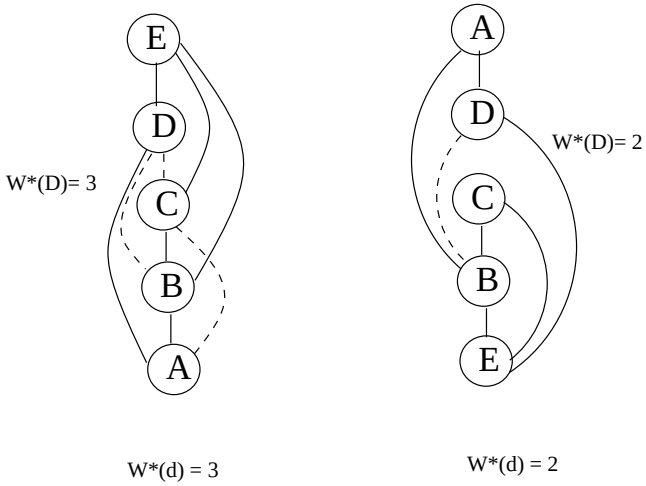


Figure 3.6: The induced width along the orderings: $d_1 = A, B, C, D, E$ and $d_2 = E, B, C, D, A$

size and $w^*(d)$ is the induced-width along the order of processing d. If r is the number of the problems' constraints, the complexity can also be bounded by $O((r+n)k^{w^*(d)+1})$.

Proof: The number of constraints (relations) in each bucket will increase to at most $2^{w^*(d)+1}$ relations, because there are at most $w^*(d)+1$ variables in a bucket. Therefore testing that many constraints over all $O((k)^{w^*(d)+1})$ tuples yields the overall complexity of $O(n \cdot (2k)^{w^*(d)+1})$. Alternatively, since the total number of input functions plus those generated is bounded by $2r$ 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 Figure 3.7. As demonstrated by the schematic execution along d in Figure 3.7, adaptive-consistency generates only unary relationships in this cycle-free graph.

It is interesting to 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 later in Chapter **??**.

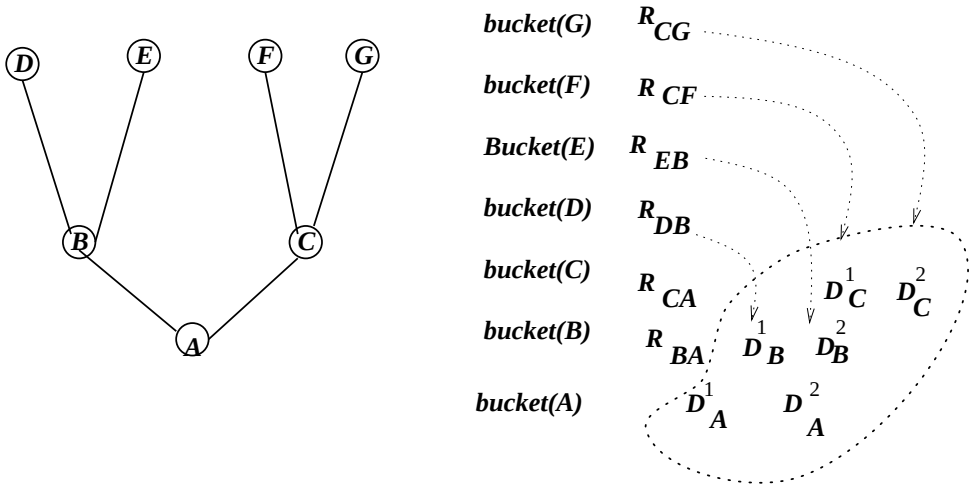


Figure 3.7: Schematic execution of adaptive-consistency on a tree network. D_X denotes unary constraints over X

.

3.2 Bucket elimination for Propositional CNFs

Since propositional CNF formulas, discussed in Chapter 2, are 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 that do not violate any clause in φ. Deciding if a theory is satisfiable is known to be NP-complete [20].

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 [18].

Definition 3.2.1 (extended composition) *The extended composition of relation* R_{S_1}, $\ldots$, 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 operator 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.

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* $(\alpha \vee \beta)$ *satisfying* $models(\alpha \vee \beta) = EC_{Q'}(models(\alpha \vee Q), models(\beta \vee \neg Q))$, *where* Q' *is the union of scopes of both clauses excluding* Q. □

Replacing 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 of Directional Resolution, $E_d(\varphi)$, the *directional extension* of φ. Given an ordering $d = (Q_1, ..., Q_n)$, all the clauses containing Q_i that do not contain any symbol higher in the ordering are placed in the bucket of Q_i, denoted $bucket_i$. 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.

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 3.8: Directional-resolution

Note that if the bucket contains a unit clause (Q_i or $\neg Q_i$), only unit resolutions are performed. As implied by Theorem 3.1.5, 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.

Example 3.2.4 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 3.9. □

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 3.9 has an induced width of 3.

Lemma 3.2.5 *Given a theory φ and an ordering $d = (Q_1, ..., Q_n)$, if Q_i has at most k parents in the induced graph along d, then the bucket of Q_i in $E_d(\varphi)$ contains no more than 3^{k+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

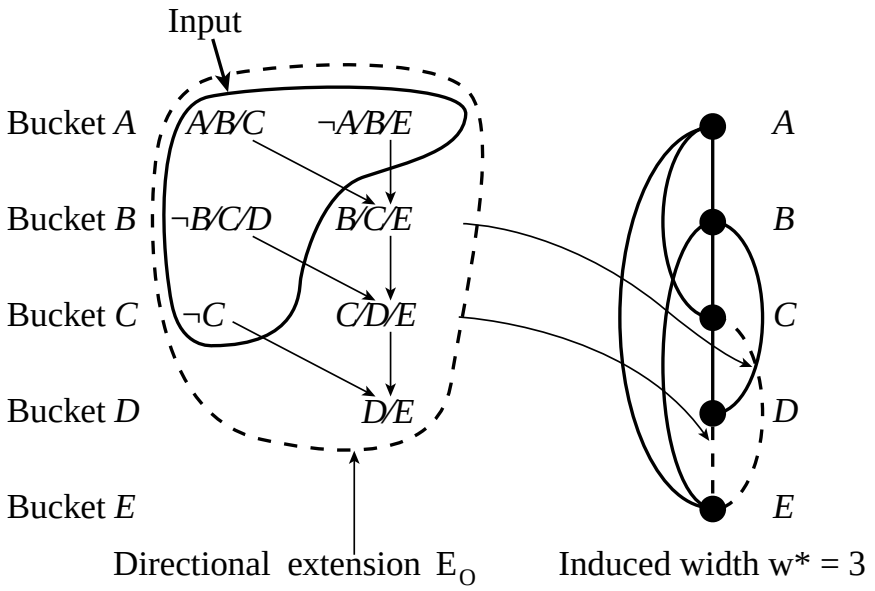


Figure 3.9: A schematic execution of directional resolution using ordering $d = (E, D, C, B, A)$

Q_i also appears in α, either positively or negatively, the number of possible clauses in a bucket is no more than $2 \cdot 3^k < 3^{k+1}$. ∎

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

Theorem 3.2.6 *(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 *Let $\alpha = \sum_{i=1}^{(r-1)} a_i x_i + a_r x_r \leq c$, and $\beta = \sum_{i=1}^{(r-1)} b_i x_i + b_r x_r \leq d$. Then $elim_r(\alpha, \beta)$ is applicable only if a_r and b_r have opposite signs, in which case $elim_r(\alpha, \beta) = \sum_{i=1}^{r-1}(-a_i \frac{b_r}{a_r} + b_i)x_i \leq -\frac{b_r}{a_r}c + d$. If a_r and b_r have the same sign the elimination implicitly generates the universal constraint.*

It is possible to show that the pair-wise join-project operation applied in a bucket can be accomplished by *linear elimination* as defined above. Applying adaptive-consistency to linear constraints and processing each pair of relevant inequalities in a bucket by linear elimination yields a bucket elimination algorithm *Directional Linear Elimination* (abbreviated *DLE*), which is the well known Fourier elimination algorithm. (see [**?**]).

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 3.10.

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:

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

The initial partitioning into buckets is shown in Figure 3.11. 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 3.12. □

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 3.10: Fourier Elimination; DLE

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

Figure 3.11: initial buckets

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

Figure 3.12: final buckets

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.

Chapter 4

Inference: Bucket-Elimination for Probablistic Networks

Having investigated bucket-elimination in the 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 as well as finding the probability of evidence 2) finding the most probable explanation (mpe) and 3) finding the maximum a posteriori hypothesis (map).

4.1 Belief Assessment and Probability of Evidence

Belief updating is the primary inference task over belief networks. The task is to update the posterior probability of singleton variables once new evidence arrives. For instance, if we are interested in likelihood that the sprinkler was last night (as we were in a belief network example in Chapter 2), then will 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. The second primary query over belief networks, computing the probability of the evidence, is tightly related to belief updating; namely, it is to update the joint likelihood of a specific subset of variable assignments. We will show in this chapter how these tasks can be computed by the bucket-elimination scheme. We will use the following notations:

Notation 4.1.1 (elimination functions) *Given a function h defined over scope S (a*

subset of variables and an $X \in S$, the functions $(min_X h)$, $(max_X h)$, $(mean_X h)$, and $(\sum_X h)$ are defined over $U = S - \{X\}$ as follows: (NOTE: the $U = u$ notation is really confusing in the context. We were just thinking of U as a set and now we're making it a random variable without an warning). for every assignment $U = u$, $(min_X h)(u) = \min_x h(u, x)$, $(max_X h)(u) = \max_x h(u, x)$, $(\sum_X h)(u) = \sum_x h(u, x)$. $mean_X h(u) = \frac{1}{|X|} \sum_x h(u, x)$. (NOTE: there should probably be a better way to write this than $h(u, x)$ which makes it seem like the function h is of 2 variables rather than $len(S)$ variables. Maybe: $h(u, x_1, \ldots, x_n)$?) Given a set of functions $h_1, ..., h_j$ defined over the subsets $S_1, ..., S_j$, the product function $(\Pi_j h_j)$ and $\sum_j h_j$ are defined over the scope $U = \cup_j S_j$ as follows. For every $U = u$, $(\Pi_j h_j)(u) = \Pi_j h_j(u_{S_j})$, and $(\sum_j h_j)(u) = \sum_j h_j(u_{S_j})$.

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 P_rs or λs. Same goes for the specific parameters $P(x|y)$ or $Pr(x|y)$ for specific values in the domains of the respective variables X and Y.

4.1.1 Deriving BE-bel

We next present a step by step derivation of a general variable-elimination algorithm for belief updating. This algorithm is similar to adaptive-consistency, but the join or 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 is to compute both the conditional probability of x_1 given evidence e, $Pr(x_1|e)$, and the probability of the evidence $Pr(e)$. By Bayes rule $P(x_1|e) = \alpha \cdot P(x_1, e)$, where α is the normalization constant $\frac{1}{P(e)}$.

To develop the algorithm in a specific case, we will use a previous example of belief networks, 2.1.10 (Figure 2.5), and assume the evidence is $g = 1$.

Consider the variables in the order $d_1 = A, C, B, F, D, G$. We want to compute $Pr(A =$

$a|g = 1)$ or $Pr(A = a, g = 1)$. By definition

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

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

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

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

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

(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.

As shown, for emphasis we index a function generated by the variable (in our case G) which is summed out. By summation we remove or eliminate a variable. 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

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

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

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

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

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

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

$$P(a)\lambda_C(a) \quad (4.6)$$

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

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

The answer to the query $Pr(a|g = 1)$ can be computed by normalizing the last product. Namely, $Pr(A = a|g = 1) = \alpha P(a) \cdot \lambda_C(a)$ where $\alpha = \frac{1}{P_r(g=1)}$ and $P_r(g = 1) = \sum_a P(a)\lambda_C(a)$ is the probability of the evidence $g = 1$.

We can create a bucket-elimination algorithm for this same 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 precisely the partition rule we used in the adaptive-consistency algorithm for constraint networks. This results in the initial partitioning given in Figure 4.1. 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 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 a lower bucket in accordance with the same rule we used to partition in the original $CPTs$.

Following order d_1, we proceed by processing $bucket_D$, summing over D all the functions that are in its bucket. The resulting function $\lambda_D(b, a) = \sum_d P(d|b, a)$ is placed in $bucket_B$. Subsequently, we process the buckets for variables F, B, and C in order, each

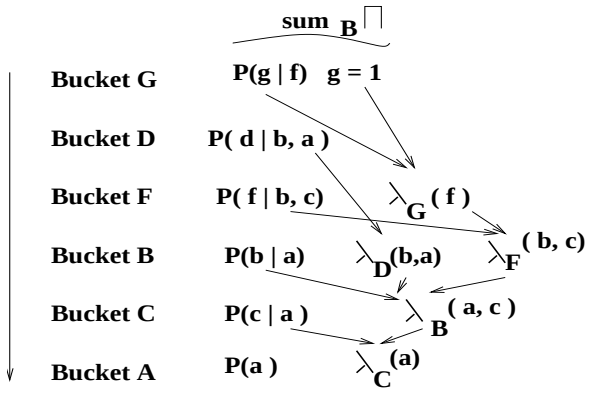


Figure 4.2: Bucket elimination along ordering $d_1 = A, C, B, F, D, G$.

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 that 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 the algorithm using ordering d_1 is (roughly) time and space quadratic in the domain sizes. But is this also the case if we use 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:

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

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

We conclude this section with a general derivation of the bucket elimination algorithm

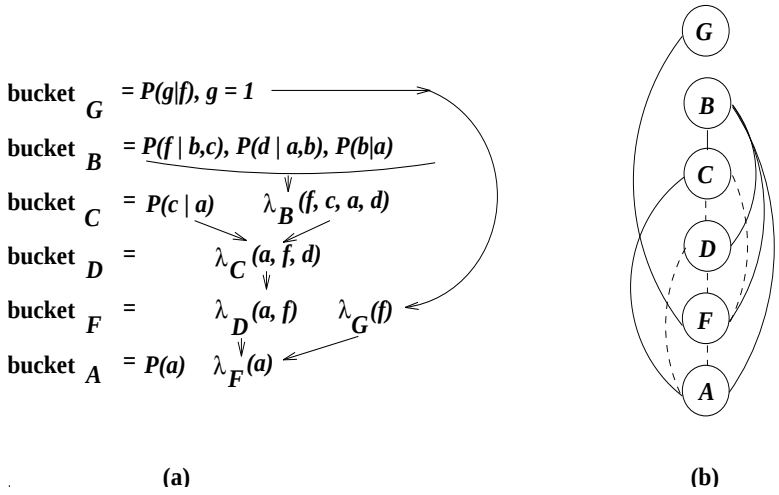


Figure 4.3: The bucket's output when processing along $d_2 = A, F, D, C, B, G$

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 $P(X_1|e)$. Using the notation $\bar{x}_i = (x_1, ..., x_i)$ and $\bar{x}_i^j = (x_i, x_{i+1}, ..., x_j)$, where F_i is the family of variable X_i, we want to compute:

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

Separating X_n from the rest of the variables results in:

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

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

where

$$\lambda_n(x_{U_n}) = \sum_{x_n} P(x_n, e|x_{pa_n}) \Pi_{X_i \in ch_n} P(x_i, e|x_{pa_i}) \tag{4.7}$$

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

Thus, the computation performed in bucket X_n is captured by Eq. (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, all the bucket's functions, which we now refer to uniformly as $\lambda_1, ..., \lambda_j$ and defined over scopes $S_1, ..., S_j$ are multiplied and the bucket's variable is eliminated by summation. The computed function is $\lambda_p : U_p \rightarrow R$, $\lambda_p = \sum_{X_p} \Pi_{i=1}^{j} \lambda_i$, where $U_p = \cup_i S_i - X_p$. This function is placed in the bucket of its largest-index variable in U_p. Once all the buckets but the first are processed, the answer is available in the first bucket. If we also process the first bucket we get the probability of the evidence. Algorithm BE-bel is described formally in Figure 4.4. We conclude:

Theorem 4.1.2 *When algorithm BE-Bel is applied along any ordering that starts with X_1 it computes the belief $Pr(X_1|e)$. It also computes the probability of evidence $Pr(e)$ as the inverse of the normalizing constant in the first bucket.* □

The bucket's operations for BE-Bel

Processing a bucket requires two types of operations on the functions in those buckets; one type is a combination operation, in this case product, which generates a function whose scope is the union of the scopes of the bucket's functions. The other type is an elimination operation, in this case marginalization, which sums out the bucket's variable. Let's look at an example of both these operations in one of the buckets used previously. Consider the computation performed when processing the bucket of variable B along ordering d_2. The bucket will include three functions: $P(F|B, C)$, $P(D|A, B)$ 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 of F, B, C, A where for each assignment the function value is the product of the respective entries in the input functions. To eliminate variable C by summation, or marginalize over C, we sum the function generated by the product over all values in $C's$ domain. Both the product and summation are depicted in Figure 4.6.

In general, the exact algorithm used to perform these operations can have a significant impact on the performance. In particular, much depends on how the $CPTs$; if, for

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

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

3. **Return:** $Pr(x_1|e) = \frac{1}{\alpha} \Pi_j \lambda_1 j(x_1)$ (where the λ_{1j} are in $bucket_1$), $P_r(e) = \alpha = \sum_{x_1} \Pi \lambda_{1j}$ is the normalizing constant.

Figure 4.4: Algorithm *BE-bel*

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

Figure 4.5: Processing the functions in the bucket of B

example, the $CPTs$ are represented as matrixes, then we can exploit efficient matrix multiplication algorithms.

4.1.2 Complexity

We saw that although BE-Bel can be applied along any ordering, its complexity varies considerably across different orderings. Using ordering d_1 we recorded functions on pairs of variables only, while using d_2 we had to record functions on four variables (see $Bucket_C$ in Figure 4.3a). The arity of the function generated during processing in 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 exponential in the size (number of variables) of the largest bucket. The base of the exponent is a bound on a variable's domain size.

Fortunately, as was observed earlier for adaptive-consistency, the bucket sizes can be easily predicted from an order associated with the elimination process. Consider the *moral graph* of a given belief network. This graph has a node for each variable and any two variables appearing in the same CPT are connected. The moral graph of the network

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	$f_C(A,B,F) = \sum_C f(A,B,C,F)$
summer	false	true	0.02+0.16 = 0.18
fall	false	true	0.06 + 0.46 = 0.52
winter	false	true	0.09 +0.72 = 0.81
spring	false	true	0.04 + 0.32 = 0.36
summer	true	true	0.72 + 0.76 = 1.48
fall	true	true	0.36 + 0.38 = 0.74
winter	true	true	0.09 + 0.95 = 1.04
spring	true	true	0.42 + 0.57 = 0.99

Figure 4.6: Processing the functions in the bucket of B

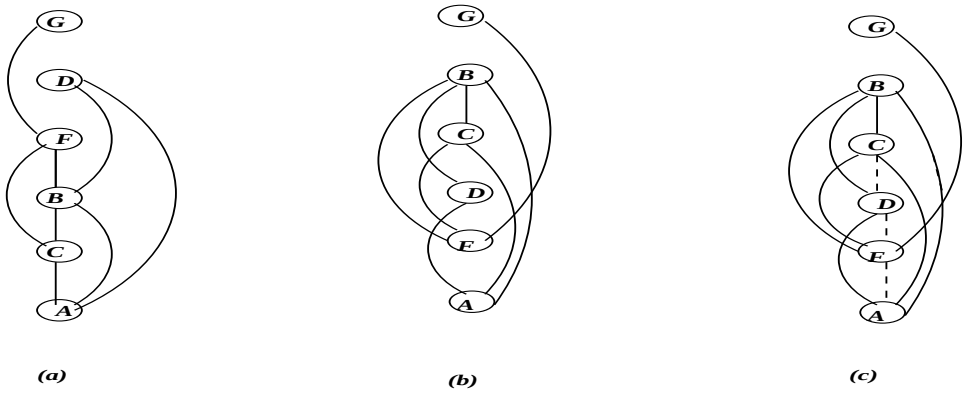


Figure 4.7: Two orderings of the moral graph of our example problem

in Figure 2.5(a) is given in Figure 2.5(b). Let us take this moral graph and impose an ordering on its nodes. Figures 4.7a and 4.7b depict the ordered moral graph using the two orderings $d_1 = A, C, B, F, D, G$ and $d_2 = A, F, D, C, B, G$. As before, the induced-width of the ordered graph of each nodes captures the number of variables which would be processed in that bucket.

Example 4.1.3 The induced moral graph of Figure 2.5b, relative to ordering $d_1 = A, C, B, F, D, G$ is depicted in Figure 4.7a. In this case, the ordered graph and its induced ordered graph are identical, since all the earlier neighbors of each node are already connected. The maximum induced width is 2. Indeed, in this case, the maximum arity of functions recorded by the elimination algorithms 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 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. □

Theorem 4.1.4 (Complexity of BE-bel) *Let w^* be the induced width of G along ordering d and k the maximum domain size of a variable. Let 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 sends a λ message to its parent and since it computes a function defined on all the variables in the bucket, the number of which is bounded by w^*, the size of the computed function is exponential in w^*. Recording the generated

lambda function requires consulting all the original functions in its generating bucket, r_{X_i} for X_i's bucket, and also all the messages received from its children, which is bounded by deg_i. Therefore, summing over all the buckets, the algorithm's computation is bounded by

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

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

4.1.3 Handling Observations by Conditioning

Observed variables (evidence) needs to be handled in a special way when processing of buckets. Take our belief network example with ordering d_1: 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). (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 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. Avoiding this increased dimensionality is the basis of the cutset property of conditioning, a property we will discuss later. Therefore 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.

Note that if bucket B had been last in the ordering, as in d_2, the virtue of conditioning on B could have been exploited earlier. During its processing along ordering d_2, $bucket_B$

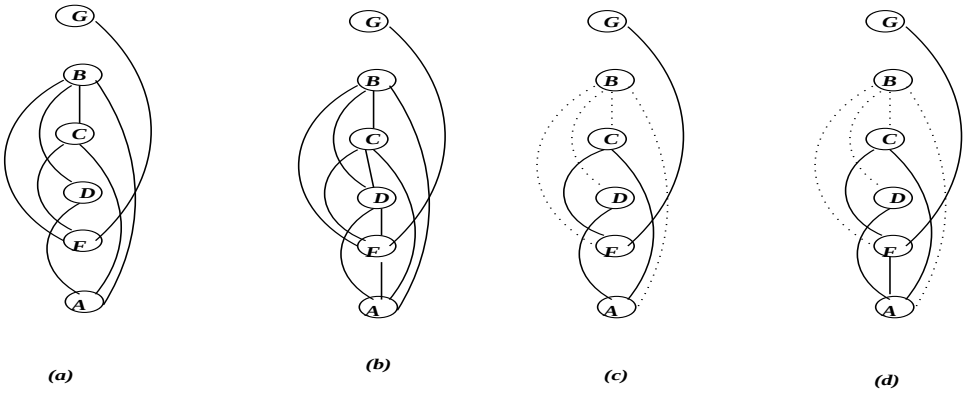


Figure 4.8: Adjusted induced graph relative to observing B

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. Thus, we see 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. Therefore, we do not add arcs between the parents of observed variables when computing the induced graph.

To capture this refinement we use the notion of *adjusted induced graph* which is defined recursively.

Definition 4.1.5 *Given a graph G and an ordering d and given a set of observed nodes E, the adjusted induced graph relative to d and E is generated (processing the ordered graph from last to first) by connecting the earlier neighbors of unobserved nodes only. The* adjusted induced width *is the width of the adjusted 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 2.5. In Figure 4.8(c) the arcs connected to the observed nodes are marked by broken lines, resulting in the adjusted induced-graph given in Figure 4.8(d). In summary,

Theorem 4.1.6 *Given a belief network having n variables, algorithm BE-bel when using ordering d and evidence e, is (time and space) exponential in the adjusted induced width $w^*(d,e)$ of the network's ordered moral graph.* □

4.1.4 Relevant subnetworks

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

Proposition 4.1.7 *Given a Bayesian network and a topological ordering $X_1, ..., X_n$, that is initiated by 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 variable and no newly computed function.*

Proof: If topological ordering is used, each bucket of a variable X contains initially at most one function, $P(X|pa(X))$. Clearly, if there is neither evidence nor new functions in the bucket summation, $\sum_x P(x|pa(X))$ will yield the constant 1. □

Example 4.1.8 Consider again the belief network whose acyclic graph is given in Figure 2.5(a) and the ordering $d_1 = A, C, B, F, D, G$. Assume we want to update the belief in variable A given evidence on F. Obviously the buckets of G and D can be skipped and processing should start with $bucket_F$. Once $bucket_F$ is processed, the remaining buckets are not skippable. □

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

Definition 4.1.9 *Given an acyclic graph and an ordering o that starts with the queried variable, and given evidence e, the marking process proceeds as follows.*

- *Initial marking: an evidence node is marked and any node having a child appearing earlier in o (namely violate the "parent preceding child rule"), is marked.*

- *Secondary marking: Processing the nodes from last to first in o, if a node X is marked, mark all its earlier neighbors.*

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

Theorem 4.1.10 *Let $\mathcal{P} =< X, D, G, P >$ be a Bayesian network, an ordering $o = X_1, ..., X_n$ and e set of evidence. Then $Pr(x_1|e)$ can be obtained by applying BE-bel over the marked subnetwork relative to evidence e and ordering o, denoted $M(\mathcal{P}|e, o)$.*

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

Corollary 4.1.11 *The complexity of algorithm BE-bel along ordering o given evidence e is exponential in the adjusted induced width of the marked ordered moral subgraph.* □

Finally, another possible approach is to prune the network before committing to any order of processing, as follows. Given a set of query nodes and a set of evidence nodes, we can remove from the network any leaf node that is not queried and is not part of the evidence and we can do this node removal recursively until no node can be removed.

Once we have a restricted Bayesian network, we can answer the query over the restricted network. (You should prove that this process is sound).

Theorem 4.1.12 *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 *is a subset of evidence variables, we can compute* $P(Y|e)$ *over the reduced Bayesian network obtained by recursively removing leaf nodes that are not in* $Y \cup E$ *and their incident edges.*

Proof: The proof is left as an exercise.

4.2 Bucket elimination for optimization tasks

Having examined the task of belief-updating in the framework of the bucket elimination, we will now focus on another primary query we often have of a belief network, that is, finding the most probable explanation for the evidence. Belief-updating answers the question "what is the likelihood of a given explanation for 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, then, 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 any type of graphical model.

4.2.1 An Elimination Algorithm for mpe

Given a Bayesian network $\mathcal{B} =< X, D, G, \mathcal{P} >$, the mpe task is to find x^0 such that $P(x^0) = \max_x Pr(x, e)$, where, by definition, $Pr(x, e) = \max_x \Pi_i P(x_i, e|x_{pa_i})$. Let $x = (x_1, ..., x_n)$ and e be a set of observations on subsets of the variables. Given a variable ordering $d = X_1, ..., X_n$, we can accomplish this task by performing the maximization operation along the ordering from right to left, migrating to the left all components 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.

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

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

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

where

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

and U_n are the variables appearing in components defined over X_n. Clearly, the algebraic manipulation of the above expressions is the same as the algebraic manipulation for belief assessment where summation is replaced by maximization. Consequently, the bucket-elimination procedure *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 matrices, which in this case are cost functions denoted $h_1, ..., h_j$ and defined over subsets $S_1, ..., S_j$, and then eliminate the bucket's variable by maximization. The generated function in the bucket of X_p is $h_p : U_p \rightarrow \ R$, $h_p = \max_{X_p} \Pi_{i=1}^{j} h_i$, where $U_p = \cup_i S_i - X_p$. The function obtained by processing a bucket is placed in the bucket of its largest-index variable in U_p. If the function is constant, we can place it in the preceding bucket as usual or directly place it in the first bucket; constant functions are not necessary to determine the exact mpe value.

We define the function $x_p^o(u) = argmax_{X_p} h_p(u)$, which provides the optimizing value of the bucket variable given its family assignments; this function can be recorded and placed in the bucket of X_p [1].

The procedure continues recursively, processing the bucket of the next variable, proceeding from the last to the first variable. Once all buckets are processed, the *mpe* value, M, can be extracted as the maximizing product of functions in the first bucket.

At this point we know the mpe value but we still did not generate an optimizing tuple. This requires a forward phase, which was not needed when we computed the posterior marginal. The algorithm initiates this *forwards phase* to compute an *mpe* tuple by assigning values along the ordering from X_1 to X_n, consulting the information recorded in each bucket. Specifically, once the partial assignment $x = (x_1, ..., x_{i-1})$ is determined, the value of X_i appended to this tuple is $x_i^o(x)$, where x^o is the function recorded in the

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

backward phase. Alternatively, if the functions x^o were not recorded in the backwards phase, the value x_i of X_i is selected to maximize the product in $bucket_i$ given the partial assignment x. 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 2.5(a). Given the ordering $d = A, C, B, F, D, G$ and the evidence $g = 1$, we process variables from last to first after partitioning the conditional probability matrices into buckets, as 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 $G^o(f) = 1$. Process $bucket_D$ by computing $h_D(b, a) = \max_d P(d|b, a)$ and put the result in $bucket_B$. Bucket F, next to be processed, now contains two matrices: $P(f|b, c)$ and $h_G(f)$. Compute $h_F(b, c) = \max_f p(f|b, c) \cdot h_G(f)$, and place the resulting function in $bucket_B$. To eliminate B, we record the function $h_B(a, c) = \max_b P(b|a) \cdot h_D(b, a) \cdot h_F(b, c)$ and place it in $bucket_C$. To eliminate C, we compute $h_C(a) = \max_c P(c|a) \cdot h_B(a, c)$ and place it in $bucket_A$. Finally, the *mpe* value given in $bucket_A$, $M = \max_a P(a) \cdot h_C(a)$, is determined. Next the mpe tuple is generated by going forward through the buckets. First, the value a^0 satisfying $a^0 = argmax_a P(a)h_C(a)$ is selected. Subsequently the value of C, $c^0 = argmax_c P(c|a^0)h_B(a^0, c)$ is determined. Next $b^0 = argmax_b P(b|a^0)h_D(b, a^0)h_F(b, c^0)$ is selected, and so on. A schematic computation is provided by Figure 4.2 where λ is simply replaced by h. (As an exercise, explicitly derive the h functions in this example.) □

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

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.

Theorem 4.2.2 *Algorithm BE-mpe is complete for the mpe task. Its time and space complexity is $O(n \cdot exp(w^*(d, e)))$, where n is the number of variables and $w^*(d, e)$ is the adjusted induced width of the ordered moral graph.*

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.
1. **Initialize:** Generate an ordered partition of the conditional probability matrices, $bucket_1$, ..., $bucket_n$, where $bucket_i$ contains all matrices whose highest variable is X_i. Put each observed variable in its bucket. Let $S_1, ..., S_j$ be the subset of variables in the processed bucket on which matrices (new or old) are defined.
2. **Backward:** For $p \leftarrow n$ downto 1, do
for all the matrices $h_1, h_2, ..., h_j$ in $bucket_p$, do

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

3. **Forward:** The mpe value is obtained by maximizing over X_1, the product in $bucket_1$.
An mpe tuple is obtained by assigning values in the ordering d consulting recorded functions in each bucket as follows.
Given the assignment $x = (x_1, ..., x_{i-1})$ choose $x_i = x_i^o(x)$ (x_i^o is in $bucket_i$), or Choose $x_i = argmax_{X_i} \Pi_{\{h_j \in \ bucket_i | \ x=(x_1,...,x_{i-1})\}} h_j$

Figure 4.9: Algorithm *BE-mpe*

4.2.2 An Elimination Algorithm for MAP

The map task is a generalization of both mpe and belief assessment. It asks for the maximal belief associated with a *subset of unobserved hypothesis variables* and the associated tuple and is likewise widely applicable especially for diagnosis tasks. Since it is a mixture of the previous two tasks, some of the variables are eliminated by summation, 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 the hypothesized variables that maximizes their probability given the evidence, namely to find $a^o = argmax_{a_1,...,a_k} Pr(a_1, ..., a_k, e)$. So, we wish to compute $\max_{\bar{a}_k} Pr(a_1, ..., a_k, e) = \max_{\bar{a}_k} \sum_{\bar{x}_{k+1}^n} \Pi_{i=1}^n P(x_i, e|x_{pa_i})$ where $x = (a_1, ..., a_k, x_{k+1}, ..., x_n)$. Algorithm *BE-map* in Figure 4.10 considers only orderings in which the hypothesized variables start the ordering. Like BE-mpe, it has a backward phase and a forward phase, but the forward phase extends to the hypothesized variables only. Because of the strict restriction on the legitimate orderings, the algorithm may be forced to have far higher induced-width than it would otherwise allow. To alleviate this problem, the maximizations and summations can be interleaved to allow more efficient orderings, as long as some constraints are obeyed. For the sake of simplicity, we will leave the discussion of the flexible ordering criteria to the exercises. The proof of BE-map will also be left as an exercise.

Theorem 4.2.3 *Algorithm BE-map is complete for the map task for orderings where the hypothesis variables are at the beginning of the sequence. Its complexity is* $O(n \cdot exp(w^*(d, e))$*, where* n *is the number of variables in the relevant marked graph and* $w^*(d, e)$ *is the adjusted induced width of its moral graph.*

4.3 Cost Networks and Dynamic Programming

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

That BE-mpe is dynamic programming becomes apparent once we transform the mpe's cost function, which has a product function, into the traditional additive function using the

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.
Output: A most probable assignment $A = a$.
1. **Initialize:** Generate an ordered partition of the conditional probability matrices, $bucket_1$, ..., $bucket_n$, where $bucket_i$ contains all matrices whose highest variable is X_i.
2. **Backwards** For $p \leftarrow n$ downto 1, do
for all the matrices $\beta_1, \beta_2, ..., \beta_j$ in $bucket_p$, do

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

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

Figure 4.10: Algorithm *BE-map*

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

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

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

Figure 4.11: Dynamic programming as BE-opt

log function. For example, $P(a, b, c, d, f, g) = P(a)P(b|a)P(c|a)P(f|b, c)P(d|a, b)P(g|f)$ becomes $C(a, b, c, d, e) = -logP = C(a)+C(b, a)+C(c, a)+C(f, b, c)+C(d, a, b)+C(g, f)$ where each $C_i = -logP_i$.

Indeed, the general dynamic programming algorithm is defined over *cost networks*. 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 are real-valued cost functions $C_1, ..., C_l$. defined over subsets $S_i = \{X_{i_1}, ..., X_{i_r}\}$, $C_i : \bowtie_{j=1}^{r} D_{ij} \rightarrow R^+$. The *cost graph* of a *cost network* has a node for each variable and connects nodes denoting variables appearing in the same cost component. The task is to find an assignment to the variables that minimizes $\sum_i C_i$.

A straightforward elimination process similar to that of BE-mpe, (where the product is replaced by summation and maximization by minimization) yields the non-serial dynamic programming algorithm [6]. 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. Clearly,

Theorem 4.3.1 *Given a cost network, BE-opt generates a representation from which the optimal solution can be generated in linear time by a greedy procedure. The algorithm's*

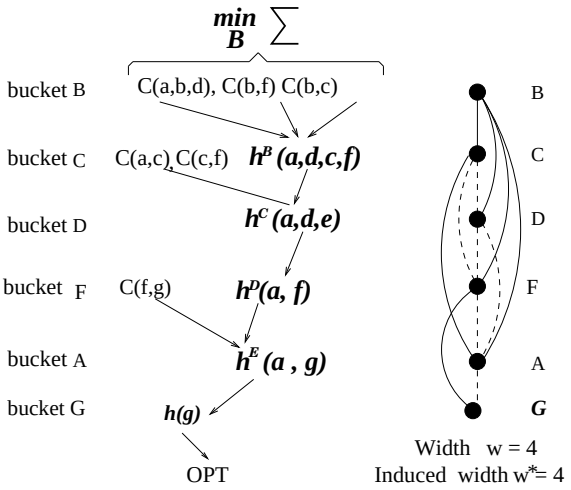


Figure 4.12: Schematic execution of BE-opt

complexity is time and space exponential in the cost-graph's adjusted induced-width. □

4.4 Mixed Networks

We will focus on the CPE task of computing $P(\varphi)$ where φ is the constraint or CNF formula. A number of related tasks can be easily derived by changing the appropriate operator (e.g. using maximization for maximum probable explanation - MPE, or summation and maximization for maximum a posteriori hypothesis - MAP). The results in this section are based for the most part on the work in [14].

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:

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

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 $\mathbf{X} \setminus \{X_n\}$. 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, we define $S_n = \mathbf{X} \setminus Q_n$ and U_n is the set of all variables in the scopes of CPTs 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 $U_n - \{X_n\}$, by

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

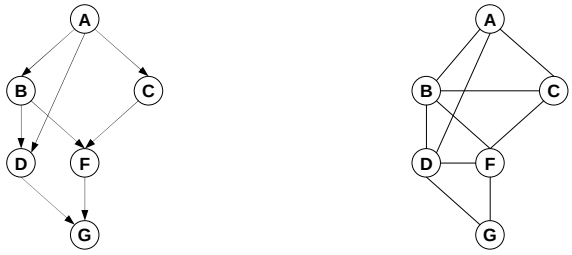
The case of observed variables When X_n is observed, or constrained by a literal, the summation operation reduces to assigning the observed value to each of its CPTs *and* to each of the relevant clauses. In this case Equation (4.8) becomes (assume $X_n = x_n$ and $P_{=x_n}$ is the function instantiated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



(a) Directed acyclic graph (b) Moral graph

Figure 4.13: Belief network

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 of the rest of the expression proceeds with X_{n-1} in the same manner. This yields algorithm *Elim-CPE* (described in Algorithm **??** and Procedure `Process-bucket`$_p$). The elimination operation is denoted by the general operator symbol $\Downarrow$ that instantiates to summation for the current query. Thus, for every ordering of the propositions, once all the CPTs and clauses are partitioned (each clause and CPT is placed in the bucket of the latest variable in their scope), 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 over scopes $S_1, ..., S_t$ and $\alpha_1, ...\alpha_r$ be the clauses over scopes $Q_1, ..., Q_r$. The algorithm computes a new function λ^P over $U_p = S \cup Q - \{X_p\}$ where $S = \cup_i S_i$, and $Q = \cup_j Q_j$, defined by:

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

Example 4.4.1 Consider the belief network in Figure 4.13, which is similar to the one in Figure 2.5, 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. We compute:
In bucket G: $\lambda^G(f, d) = \sum_{\{g|g \vee d = true\}} P(g|f)$
In bucket F: $\lambda^F(b, c, d) = \sum_f P(f|b, c)\lambda^G(f, d)$
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: $\lambda^C(a) = \sum_c P(c|a)\lambda^B(a, c)$

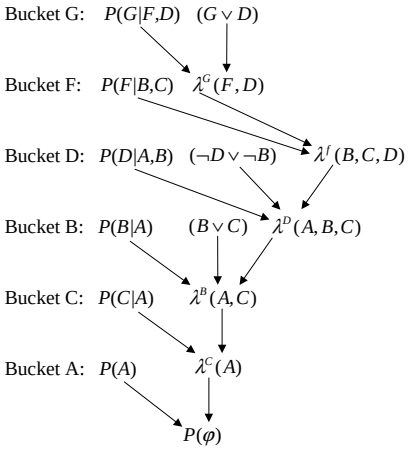


Figure 4.14: Execution of ELIM-CPE

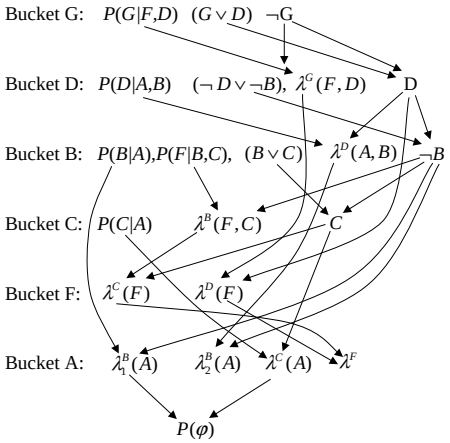


Figure 4.15: Execution of ELIM-CPE (evidence $\neg G$)

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

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) specially can avoid creating new dependencies. In fact, there exists a spectrum of feasible bounded inference schemes that can be applied to the clauses in the buckets and can enhance efficiency considerably.

Example 4.4.3 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,

Procedure Process-bucket-REL$_p$($\Downarrow$, ($\lambda_1, \ldots, \lambda_j$),($R_1, \ldots, R_r$))

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. Apply arc-consistency (or any constraint propagation) over the constraints in the bucket. Put the resulting constraints in the buckets of their latest variable and **move any bucket with single domain to top of processing**

else

Generate $\lambda^p = \sum_{\{x_p | \bar{x}_{U_p} \in \bowtie_j R_j\}} \prod_{i=1}^{j} \lambda_i$

Add λ^p to the bucket of the latest variable in U_p, where

$U_p = \bigcup_{i=1}^{j} S_i \bigcup_{i=1}^{r} Q_i - \{X_p\}$

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 λ products. We will discuss some more details later on.

4.5 The general bucket elimination

In the following paragraphs we 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 $\{D_1, ..., D_n\}$. and

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

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

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

Figure 4.16: Algorithm *bucket-elimination*

$F = \{f_1, ..., f_k\}$ is a set of functions, where each f_i is defined over a scope $S_i \subseteq X$. Given a function h defined over scope $S \subseteq X$, and given $Y \subseteq S$, the (generalized) projection operator $\Downarrow_Y f$ is defined by enumeration as $\Downarrow_Y h \in \{max_{S-Y}h, min_{S-Y}h, \Pi_{S-Y}h, \sum_{S-Y} h\}$ and the (generalized) combination operator $\otimes_j f_j$ is defined over $U = \cup_j S_j$. $\otimes_{j=1}^k f_j \in \{\Pi_{j=1}^k f_j, \ \sum_{j=1}^k f_j, \ \bowtie_j f_j\}$.

The problem is to compute

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

such problems can be solved by the bucket-elimination algorithm, stated using this general form 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, directional resolution as well as BE-meu can be shown to be expressible in terms of such operators.

4.6 Summary

In the last two chapters, we have seen how the bucket-elimination framework can be used to unify variable elimination algorithms on graphical models for both deterministic and probabilistic reasoning. The chapter describes the bucket-elimination framework which unifies variable elimination algorithms appearing for deterministic and probabilistic reasoning as well as for optimization tasks. In this framework, the algorithms exploit the structure of the relevant network without conscious effort on the part of the designer. Most bucket-elimination algorithms[2] are time and space exponential in the induced-width of the underlying dependency graph of the problem.

4.7 Chapter Notes

Among the early bucket elimination algorithms we find the peeling algorithm for genetic trees [8], Zhang and Poole's VE1 algorithm [43] which is identical to elim-bel, SPI algorithm by D'Ambrosio et.al., [32] 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 [35]. We also made explicit the observation that bucket elimination algorithms resemble tree-clustering methods, an observation that was made earlier in the context of constraint satisfaction tasks [16].

The observation that a variety of tasks allow efficient algorithms of hyper-trees and therefore can benefit from a tree-clustering approach was recognized by several works in the last decade. In [31] the connection between optimization and constraint satisfaction and its relationship to dynamic programming is explicated. In the work of [27, 37] and later in [7] an axiomatic framework that characterize tasks that can be solved polynomially over hyper-trees, is introduced. Such tasks can be described using *combination* and *projection* operators over real-valued functions, and satisfy a certain set of axioms. The axiomatic framework [37] was shown to capture optimization tasks, inference problems in probabilistic reasoning, as well as constraint satisfaction. Indeed, the tasks considered in

[2] all, except Fourier algorithm.

this paper can be expressed using operators obeying those axioms and therefore their solution by tree-clustering methods follows. Since, as shown in [16] and here, tree-clustering and bucket elimination schemes are closely related, tasks that fall within the axiomatic framework [37] can be accomplished by bucket elimination algorithms as well. In [7] a different axiomatic scheme is presented using semi-ring structures showing that impotent semi-rings characterize the applicability of constraint propagation algorithms. Most of the tasks considered here do not belong to this class.

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

The work we show here also fits into the framework developed by Arnborg and Proskourowski [3, 2]. They present table-based reductions for various NP-hard graph problems such as the independent-set problem, network reliability, vertex cover, graph k-colorability, and Hamilton circuits. Here and elsewhere [17, 12] we extend the approach to a different set of problems.

Tatman and Schachter [41] have published an algorithm for the general influence diagram that is a variation of elim-meu. Kjaerulff's algorithm [22] can be viewed as a variation of elim-meu tailored to dynamic probabilistic networks.

Chapter 5

The Graphs of Graphical Models

As we saw, and as we will see throughout the book, graphical models structure can be described by graphs that capture dependencies and independencies in the knowledge-base. The graph is useful because it conveys information regarding the interaction between different variables and can allow efficient query processing. In this chapter we provide general overview of graph properties that will be used. We will focus on graph parameter called *induced-width* or *tree-width* that captures the complexity of reasoning algorithms for graphical models.

A graphical model can be represented by a graph called a *primal graph* where each node represents a variable and the arcs connect all nodes whose variables are included in a function scope. The absence of an arc between two nodes indicates that there is no direct function – the one specified in the input – between the corresponding variables. We observed earlier primal graphs for a variety of graphical models, depicting constraints and probabilistic functions.

Definition 5.0.1 (primal graph) *The* primal graph *of a graphical model is an undirected graph that has variables as its vertices and an edge connects any two variables that appear in the scope of the same function.*

The primal graph (also called moral graph for Bayesian networks) is an effective way to capture the structure of the knowledge as expressed by the graphical model. In particular, graph separation is a sound way to capture conditional independencies (and therefore called i-maps [30]) relative to probability distributions over directed and undirected

graphical models. They also capture the notion of *embeded multi-valued dependencies (EMVDs)* in relational databases [25]. All advanced algorithms for graphical models exploit the graphical structure. Additional graph representations that are used are the hyper=graphs, dual graphs as well as factor graphs in the context of Markov networks.

5.1 Dual graphs and hypergraphs

While primal graph of a graphical model for binary and non-binary functions is well defined, a hypergraph representation more accurately maintains the association between arcs and functions scopes.

Definition 5.1.1 (hypergraph) *A hypergraph is a structure $\mathcal{H} = (V, S)$ that consists of vertices $V = \{v_1, .., v_n\}$ and a set of subsets of these vertices $S = \{S_1, ..., S_l\}$, $S_i \subseteq V$, called hyperedges. The hyperedges differ from regular edges in that they "connect" (or are defined over) any number of variables.*

In the *hypergraph* representation of a graphical model, nodes represent the variables, and *hyperarcs* (drawn as regions) are the scopes of functions. The hyperarcs group those variables that belong to the same scope. A related representation is the *dual graph*. A *dual graph* represents each function scope by a node and associates a labeled arc with any two nodes whose *function scopes* share variables. The arcs are labeled by the shared variables.

Example 5.1.2 Figure 5.1 depicts the *hypergraph* (a), the *primal* (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. □

As we already observed there is a tight relationship between the complexity of inference algorithms such as bucket elimination and the graph concept called induced width, also known as *treewidth*. 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 [3]. However, useful greedy heuristics algorithms are available as we briefly show in the next few paragraphs [13, 4, 38].

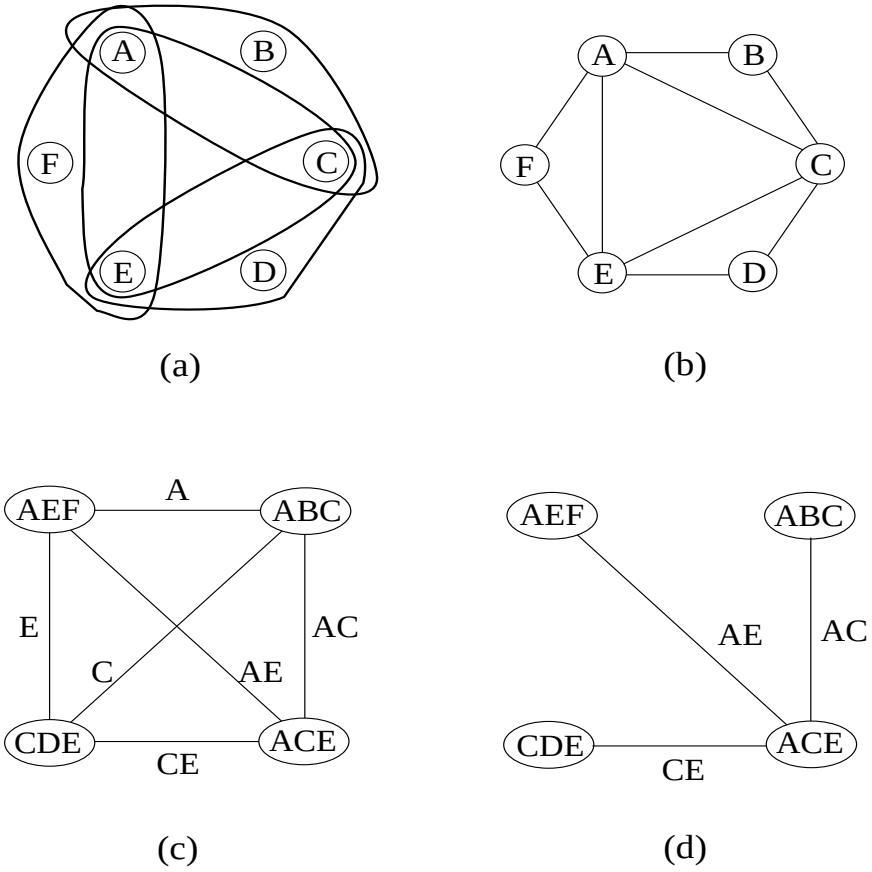


Figure 5.1: (a)Hyper, (b)Primal, (c)Dual and (d)Join-tree constraint graphs of a CSP.

5.1.1 The induced 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, denoted $w(d)$, is the maximum width over all nodes. The *width of a graph* is the minimum width over all the orderings of the graph.

Example 5.1.3 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

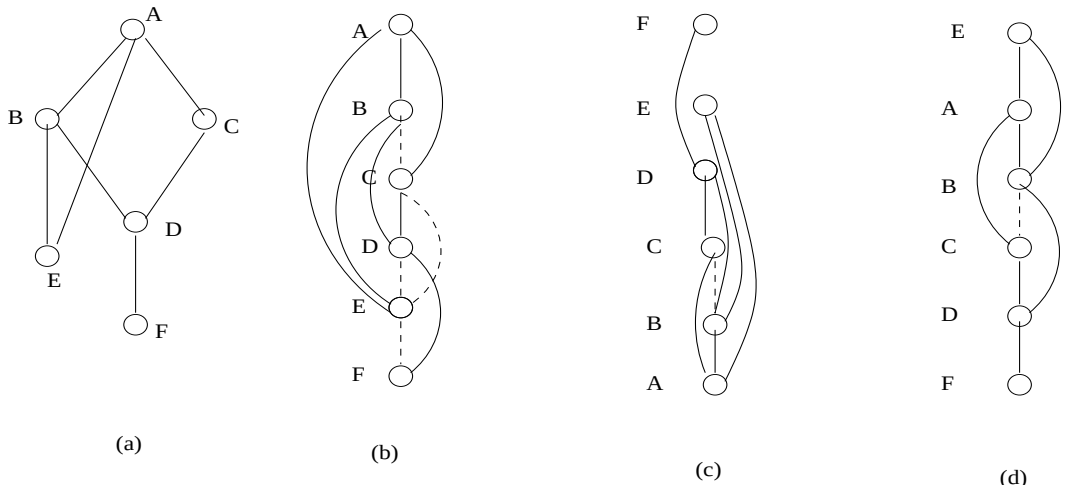


Figure 5.2: (a) Graph G, and three orderings of the graph; (b) $d_1 = (F, E, D, C, B, A)$, (c) $d_2 = (A, B, C, D, E, F)$, and (d) $d_3 = (F, D, C, B, A, E)$. Broken lines indicate edges added in the induced graph of each ordering.

$d_3 = (F, D, C, B, A, E)$. 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. □

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 width of the induced ordered graph (G^*, d). The *induced width of a graph*, w^*, is the minimal induced width over all its orderings.

Example 5.1.4 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 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. □

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

Proposition 5.1.5 *A graph is a tree iff it has both 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 number of neighbors is selected and put last in the ordering. The variable and all its adjacent edges are then eliminated from the original graph, and selection of the next variable continues recursively with the remaining graph. Ordering d_2 of G in Figure 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 G with smallest degree.
3. put r in position j and $G \leftarrow G - r$.
(Delete from V node r and from E all its adjacent edges)
4. **endfor**

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

Proposition 5.1.6 *[19] Algorithm min-width* (MW) *finds a minimum width ordering of a graph.*

Though finding the min-width ordering of a graph is easy, finding the minimum *induced width* of a graph is hard (NP-complete [3]). Nevertheless, deciding whether there exists an ordering whose induced width is less than a constant k, takes $O(n^k)$ time.

A decent greedy algorithm, obtained by a small modification to the min-width algorithm, is the *min-induced-width* (MIW) algorithm (Figure 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 the minimum induced width of the graph, w^*.

Another variation yields a greedy algorithm known as *min-fill*. Rather than order the nodes in order of their min-degree, it uses the *min-fill set*, that is, the number of edges needed to be filled so that 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 [**?**]. 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

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

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

5.2 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 in chordal graphs) is facilitated by using yet another ordering procedure called the *max-cardinality ordering* [40]. A *max-cardinality ordering* of a graph orders the vertices from *first to last* according to the following rule: the first node is chosen arbitrarily. From this point on, a node that is connected to a maximal number of already ordered vertices is selected, and so on. (See Figure 5.6.)

A max-cardinality ordering can be used to identify chordal graphs. Namely, a graph

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

Proposition 5.2.1 *If G^* is the induced graph of a graph G, along some ordering, then G^* is chordal.* □

Example 5.2.2 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

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.

Bibliography

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

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

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

[4] A. Becker and D. Geiger. A sufficiently fast algorithm for finding close to optimal junction trees. In *Uncertainty in AI (UAI'96)*, pages 81–89, 1996.

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

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

[7] 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.

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

[9] R. McEliece D. C. MacKay and J. Cheng. Turbo decoding as an instance of pearl's "belief propagation" algorithm. 1996.

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

[11] 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.

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

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

[14] R. Dechter and D. Larkin. Hybrid processing of belief and constraints. *Proceeding of Uncertainty in Artificial Intelligence (UAI01)*, pages 112–119, 2001.

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

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

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

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

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

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

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

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

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

[24] F. R. Kschischang and B.H. Frey. Iterative decoding of compound codes by probability propagation in graphical models. *submitted*, 1996.

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

[26] R.J. McEliece, D.J.C. MacKay, and J.-F.Cheng. Turbo decoding as an instance of Pearl's belief propagation algorithm. *To appear in IEEE J. Selected Areas in Communication*, 1997.

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

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

[29] Jurg Ott. *Analysis of Human Genetics*. Cambridge University Press, 1999.

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

[31] 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.

[32] 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.

[33] R.G.Gallager. A simple derivation of the coding theorem and some applications. *IEEE Trans. Information Theory*, IT-11:3–18, 1965.

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

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

[36] C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423,623–656, 1948.

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

[38] 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.

[39] C.E. Leiserson T. H. Cormen and R.L. Rivest. In *Introduction to algorithms*. The MIT Press, 1990.

[40] 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.

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

[42] 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.

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