UNIVERSITY OF CALIFORNIA,
IRVINE

EFFICIENT REASONING IN GRAPHICAL MODELS

DISSERTATION
submitted in partial satisfaction of the requirements for the degree of
DOCTOR OF PHILOSOPHY
in Information and Computer Science
by

Irina Rish

Dissertation Committee:
Professor Rina Dechter, Chair
Professor Sandra Irani
Professor Padhraic Smyth

1999

The dissertation of Irina Rish is approved
and is acceptable in quality and form
for publication on microfilm:

Committee Chair

University of California, Irvine
1999

To my parents

Contents

List of Tables

List of Figures

Acknowledgments

First, I wish to thank my advisor, Rina Dechter, for her guidance, criticism, and advice during all my years in graduate school. Many thanks to my committee, Sandy Irani and Padhraic Smyth, for their comments and suggestions that helped to improve this manuscript. I especially thank Padhraic for insightful discussions on probabilistic decoding. I also wish to express my gratitude to Susan Moore and Theresa Klonecky, our wonderful graduate student advisors, to Lynn Harris for her kind help in polishing this dissertation, and to all ICS faculty for creating an inspiring environment in our department, especially to our chair Mike Pazzani, Rick Lathrop, Dennis Kibler, David Eppstein, and Mike Dillencourt. My research also benefited from discussions with colleagues outside UCI, especially Toby Walsh, David McAllester, Henry Kautz, Bart Selman, Rich Korf, Robert McEliece, and David MacKay.

I am also grateful to the members of our research group, Dan Frost, Eddie Schwalb, and Kalev Kask, for numerous discussions, comments, and ideas. Dan, thank you for very inspiring and exciting collaboration, and for your encouragement and support. I would also like to thank all my friends and fellow students, Anna Radovic, Igor Cadez, Andrei Tchernychev, Elena Pashenkova, Eugene Gendelman, Alex Kiskachi, Anna Budnyatsky, Lilia Lobanova, Igor Novikov, and many others, for making those years in graduate school enjoyable and unforgettable part of my life, and for being really wonderful friends.

Special thanks to Alex Kozlov, for his friendship and support during my most difficult times, for encouragement and advice that guided me through the last years of graduate school, and for opening to me new horizons of personal growth.

I dedicate this dissertation to my parents. Without their love, encouragement and support, this work would never been completed. I am also thankful to my brothers and best friends Dima and Ilya, who always served me as role models.

This work was partially supported by NSF Grant IRI-9157636, by Air Force Office of Scientific Research Grant, by a Micro-grant from Rockwell International, and by the UCI Regents Dissertation Fellowship.

Curriculum Vitae

IRINA RISH

1999	Ph.D. in Computer Science. University of California, Irvine. Research area: automated reasoning (constraint satisfaction and probabilistic inference). Dissertation: “Efficient reasoning in graphical models”. Advisor: Rina Dechter.
1994	M.S. in Computer Science. University of California, Irvine. GPA 3.97/4.
1991	M.S. in Applied Mathematics. Applied Mathematics Department, Moscow Oil & Gas Gubkin Institute (Moscow, Russia). GPA 4.98/5. MS Thesis: “A rule-based expert system for geophysical applications”.

Abstract of the Dissertation

Efficient Reasoning in Graphical Models
by
Irina Rish
Doctor of Philosophy in Information and Computer Science
University of California, Irvine, 1999
Professor Rina Dechter, Chair

Most artificial intelligence problems are computationally hard (NP-hard). However, in practice, the pessimistic worst-case performance can be improved by exploiting problem structure and by using approximations that trade accuracy for efficiency. Theoretical studies identify tractable problem classes while empirical evaluations shed light on average performance.

This thesis is concerned with efficient algorithms for automated inference in graphical models, such as constraint networks and belief networks. We use a general graph-based algorithmic framework that combines a dynamic-programming approach called *variable elimination* with *conditioning* techniques, such as backtracking search. We investigate the effects of certain problem structures, identify new tractable classes, and propose several structure-exploiting algorithms.

The central idea of this thesis is that efficiency can be gained by reducing the *induced width*, a graph parameter that bounds the complexity of variable elimination. We approach this problem by combining elimination with conditioning, which reduces the graph connectivity; by exploiting hidden structure such as causal independence in belief networks, which allows decomposition of large dependencies into smaller ones; and by using approximation algorithms that bound the size of recorded dependencies. Our empirical studies demonstrate promising results obtained both on randomly generated problems and on realistic domains such as medical diagnosis and probabilistic decoding.

Chapter 1

Introduction and Overview

Automated reasoning is a field of artificial intelligence concerned with answering queries and drawing new conclusions from previously stored knowledge. It includes many areas such as theorem-proving, game playing, propositional satisfiability, constraint satisfaction, planning, scheduling, probabilistic inference and decision-making.

This dissertation is focused on reasoning in graphical frameworks such as constraint and belief networks, where domain knowledge is represented by a graph depicting variables as nodes and dependencies (e.g., propositional clauses, constraints, probabilities, and utilities) as edges. Some reasoning tasks can be formulated as combinatorial optimization or constraint satisfaction problems, while others can be viewed as knowledge compilation, or inference. We approach those tasks using a general graph-based algorithmic framework that combines a dynamic-programming technique called *variable elimination* with *backtracking search*, and investigate the effect of problem structure on the performance of such algorithms.

A popular method for solving combinatorial optimization and constraint satisfaction problems is to search the space of variable assignments. This method can also be viewed as *conditioning*, or *reasoning by assumptions*: a problem is divided into subproblems conditioned on an instantiation of a subset of variables (also called *cutset*). Each subproblem can be solved by any means; if the current subproblem is insoluble, or if more solutions are needed, the algorithm tries a different assignment to the cutset variables, and so on.

An alternative to search algorithms are dynamic-programming techniques also known as *variable-elimination* algorithms which process (eliminate) variables in a certain order and infer new dependencies among the remaining variables. In this thesis we use an algorithmic framework called *bucket elimination* [26, 27] that generalizes and unifies non-serial dynamic programming techniques [7] for various reasoning tasks.

Both search and elimination schemes are used in many areas of reasoning. For instance, a common approach to solving constraint satisfaction problems combines backtracking search with *local consistency* enforcing which is a limited form of variable elimination. *Branch-and-bound* is an example of a search algorithm for solving combinatorial optimization problems. The *tree-clustering* method for belief networks is closely related to variable elimination, while *backward induction*, *value iteration* and *policy iteration* algorithms for decision-theoretic planning employ both elimination and conditioning.

Most reasoning problems are known to be computationally hard (NP-hard). It is believed that the inherent complexity of those problems is associated with the level of interactions among the problems' variables, that is captured, for instance, by the notion of i-consistency [44, 22]. A constraint-satisfaction problem is called i-consistent if any consistent assignment to $i-1$ variables can be extended to any i-th variable without violating any constraint. Consider, for example, a constraint network defined on n variables that is $(n-1)$-consistent, but not n-consistent. Such networks are hard to solve both by search and elimination. Search may encounter all partial solutions of length $n-1$, therefore traversing the complete search tree, while elimination may deduce dependencies of size $O(exp(n-1))$ involving $n-1$ variables. However, the level of i-consistency of a given problem is not known in advance and therefore cannot serve as a complexity bound. Instead, we use a parameter of the problem's *interaction graph*, called *induced width*, which does capture the problem's level of interactions and which can be assessed prior to the algorithm's execution. The induced width

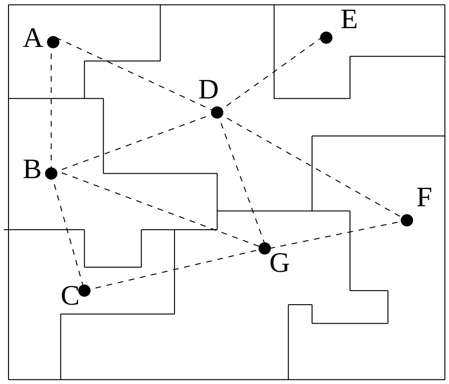


Figure 1.1: An example of a constraint satisfaction problem: map coloring.

describes the size of largest constraint, function, or another dependence created by a variable-elimination algorithm, which corresponds to a largest clique "induced" in the problem's graph.

The central theme of this thesis is to improve the performance of reasoning algorithms by reducing their induced width. We investigate several approaches that include combining elimination with conditioning, exploiting hidden structure such as causal independence, and using approximate algorithms.

The following three subsections present an overview of the reasoning tasks and algorithms addressed in this thesis, and summarize the thesis' contributions.

1.1 Automated reasoning: frameworks and tasks

Constraint satisfaction

An example of a constraint satisfaction problem (CSP) is the *map coloring problem*, illustrated in Figure 1.1. Given a fixed set of colors, the task is to color each country on the map so that countries having a common border are assigned different colors. Generally, a constraint satisfaction problem is defined on a *constraint network* $< X, D, C >$, where $X = \{X_1, ..., X_n\}$ is the set of *variables*, associated with a set of finite *domains*, $D = \{D_i, ..., D_n\}$, and a set of *constraints*, $C = \{C_1, ..., C_m\}$. Each

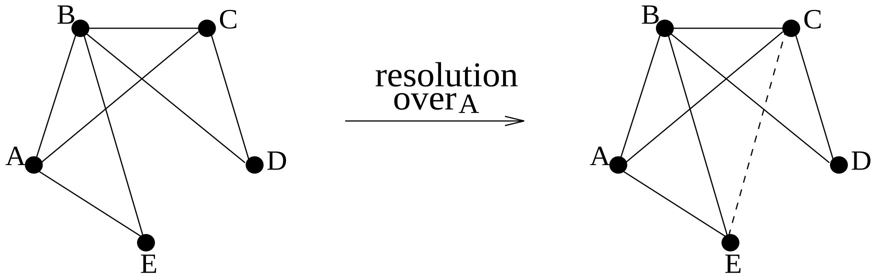


Figure 1.2: The effect of resolution on the interaction graph.

constraint C_i is a relation $R_i \subseteq D_{i_1} \times ... \times D_{i_k}$ defined on a subset of variables $S_i = \{X_{i_1}, ..., X_{i_k}\}$. A constraint network can be associated with an undirected graph, called a *constraint graph*, where the nodes correspond to the variables, and two nodes are connected if and only if they participate in the same constraint. The *constraint satisfaction problem (CSP)* is to find a *solution*, namely a value assignment to all the variables that satisfies all the constraints. If no such assignment exists, the constraint network is *inconsistent*. For example, in the map-coloring problem presented in Figure 1.1, countries correspond to the variables (i.e., $X = \{A, B, C, D, E, F, G\}$) and colors correspond to the domain values (e.g., $D_i = \{red, green, blue\}$). The problem is defined by a set of pairwise inequality constraints between neighboring countries, such as $A \neq B$, $B \neq C$. Constraint networks are widely used in many practical applications such as scheduling, planning, electronic circuit diagnosis, query answering in databases, and line drawings understanding.

Propositional satisfiability (SAT)

A special case of a CSP is *propositional satisfiability problem* (abbreviated SAT). Consider the following example. Assume that you would like to invite your friends Alex, Beki, and Chris to a party. Let A, B, and C denote the propositions "Alex comes", "Beki comes", and "Chris comes", respectively. You know that if Alex comes to the party, Beki will come as well, and that if Chris comes, then Alex will, too. This can be expressed in propositional calculus as $(A \rightarrow B) \wedge (C \rightarrow A)$, or, equivalently, as $(\neg A \vee B)$ $\wedge(\neg C \vee A)$, where disjunctive formulas $(\neg A \vee B)$ and $(\neg C \vee A)$ are called *clauses*. Assume now that Chris came to the party; should you expect to see Beki? Or, in propositional logic, does the propositional *theory* $\varphi = C$ $\wedge(A \rightarrow B)$ $\wedge(C \rightarrow A)$ *entail* B? A common way to answer this query is to assume that Beki will not come and check whether this is a plausible situation (i.e., decide whether $\varphi' = \varphi \wedge \neg B$ is satisfiable). If φ' is unsatisfiable, we conclude that φ entails B. Propositional satisfiability can be defined as a CSP, where propositions correspond to the variables, domains are $\{0, 1\}$, and constraints are represented by clauses (for example, clause $(\neg A \vee B)$ allows all tuples (A, B) except $(A = 1, B = 0)$).

Formally, the propositional satisfiability problem (SAT) is to decide whether a given *cnf theory* has a *model*, i.e. an assignment to its propositions that does not violate any clause. A formula φ in *conjunctive normal form* (cnf) is a conjunction of *clauses* $\alpha_1, ..., \alpha_t$ (denoted as a set $\{\alpha_1, ..., \alpha_t\}$) where a clause is a disjunction of *literals* (propositions and their negations). For instance, $\alpha = (P \vee \neg Q \vee \neg R)$ is a clause, where P, Q, and R are propositions, and P, $\neg Q$, and $\neg R$ are literals. A *resolution* over two clauses $(\alpha \vee Q)$ and $(\beta \vee \neg Q)$ results in a clause $(\alpha \vee \beta)$ (called *resolvent*) thus eliminating proposition Q.

The structure of a propositional theory can be captured by its *interaction graph* (equivalent to the constraint graph in a CSP). The interaction graph of a theory φ, denoted $G(\varphi)$, is an undirected graph that contains a node for each propositional variable and an edge connecting any two nodes representing variables appearing in the same clause. The resolution operation creates new clauses which correspond to new edges in the interaction graph. For example, given the theory $\varphi = \{(A \vee B \vee C)$, $(\neg A \vee B \vee E)$, $(B \vee \neg C \vee D)\}$, resolution over A results into new clause $(B \vee C \vee E)$, thus adding a new edge between nodes E and C. Figure 1.2 shows the interaction graph of φ and its *induced graph*, which corresponds to $\varphi \cup (B \vee C \vee E)$ obtained by resolving over A.

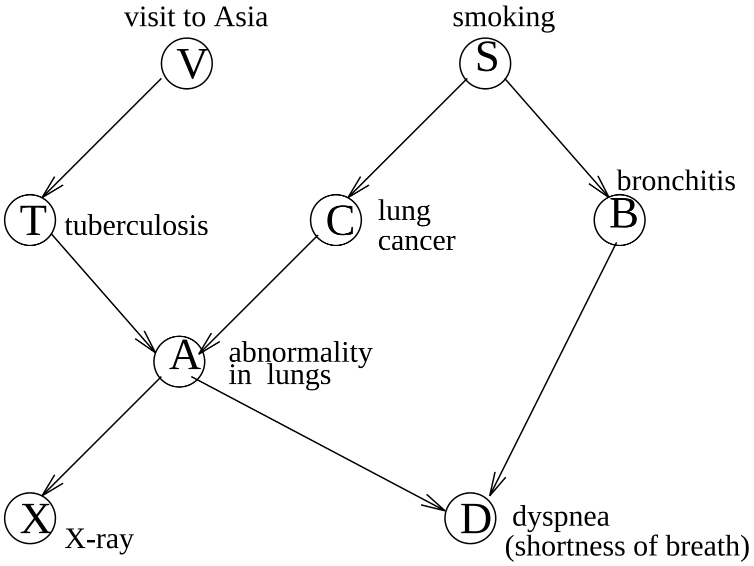


Figure 1.3: An example of a belief network.

Probabilistic inference

Constraint networks offer a convenient *deterministic* framework for a wide variety of reasoning tasks. However, this framework does not include representation of uncertain knowledge. There are various formalisms for modeling uncertainty that include nonmonotonic logic, probabilistic logic, fuzzy logic, and probabilistic constraint networks. We will focus on the popular framework known as *Bayesian*, or *belief networks*. The success of belief networks is not surprising: their clear semantics is based on mathematical formalism of probability theory, the oldest and most developed tool for modeling uncertainty. Belief networks exploit conditional independencies in order to provide a convenient graphical representation of complex probabilistic distributions. Belief networks are used in a variety of applications including medical diagnosis, troubleshooting in computer systems, circuit diagnosis, traffic control, and signal processing.

A *belief network (BN)* is a directed acyclic graph, where the nodes correspond to random variables and the edges denote probabilistic dependencies. Given a directed edge (X, Y), the node X is called a *parent* of Y, while the node Y is called a *child* of X. A node and all its parents are called a *family.* Each node X in the network is associated with the probability function $P(x|pa(x))$, where x and $pa(x)$ denote the values of X and of X's parents, respectively. The network defines a joint distribution over the n variables:

$$P(x_1, ..., x_n) = \prod_{i=1}^{n} P(x_i|pa(x_i)).$$

Frequently, belief networks represent causal dependencies between parents and children and therefore are also called *causal networks.* An example of a simple belief network for medical diagnosis is shown in Figure 1.3. A sample query may be to assess the probability that a patient has tuberculosis given that he has recently visited Asia (which may increase the risk of getting the disease), or to determine the probability of lung cancer given that patient suffers from dyspnea (shortness of breath), but has normal X-ray results. The task of computing the posterior probability of *query* node(s), given observations of some other nodes (*evidence*), is called *belief updating.* Another task is to find a most probable explanation (MPE), namely, a maximum-likelihood assignment to all unobserved nodes. For tasks that involve actions, such as planning and decision-making, there is a *utility function* associated with the outcome of actions. Given a utility function defined on the network's nodes, and a distinguished set of *decision nodes*, we want to find an assignment to the decision nodes that maximizes the expected utility. A generalization of a belief network that includes decision nodes and utility function is known as an *influence diagram* [103, 104].

The *moral graph* of a belief network is obtained by connecting ("marrying") the parents of each node and dropping the directionality of edges. The moral graph parallels the notions of the constraint graph and the interaction graph: they all have the property that a pair of nodes is connected in the graph if the corresponding variables belong to the same dependence, which can be a constraint, a clause, or a conditional probability function.

1.2 Reasoning algorithms

The two general approaches to reasoning include divide-and-conquer *conditioning* strategies such as search and dynamic-programming techniques such as *variable-elimination*. Conditioning, or reasoning by assumptions, splits a problem into subproblems by instantiating a subset of variables, while variable elimination transforms a problem into an equivalent one, replacing the eliminated variable by new dependencies deduced on the remaining variables. The next subsections elaborate on those two approaches.

1.2.1 Conditioning and search

An example of conditioning (search) is *backtracking*, or *depth-first search* algorithm, a common technique for solving constraint satisfaction problems. It processes the variables in some order, instantiating each variable if it has a value consistent with previous assignments. If there is no such value (a situation called a *dead-end*), the algorithm *backtracks* to the previous variable (hence the name) and tries an alternative assignment. If no consistent assignment is found, the algorithm backtracks again, and so on. The algorithm explores the search space in a systematic way until it either finds a solution, or concludes that no solution exists. The search space can be represented by a *search tree*, which is traversed in a depth-first manner. For example, the search tree in Figure 1.4 is traversed when searching for a model of $\varphi = (\neg A \vee B) \wedge (\neg C \vee A) \wedge \neg B \wedge C$. The tree nodes correspond to variables, while its branches represent value assignments. Dead-end nodes are crossed out. Clearly, φ is inconsistent, since every leaf, i.e., every partial assignment, is a dead-end.

There are many advanced backtracking algorithms for solving CSPs that improve the basic scheme by using "smart" variable- and value-ordering heuristics ([14], [51]). More efficient backtracking mechanisms, such as *backjumping* [53, 21, 92], constraint preprocessing (e.g., *arc-consistency*, *forward checking* [59]), or learning (recording constraints) [21, 48, 3], are available.

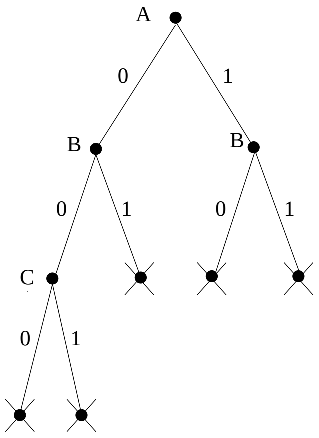


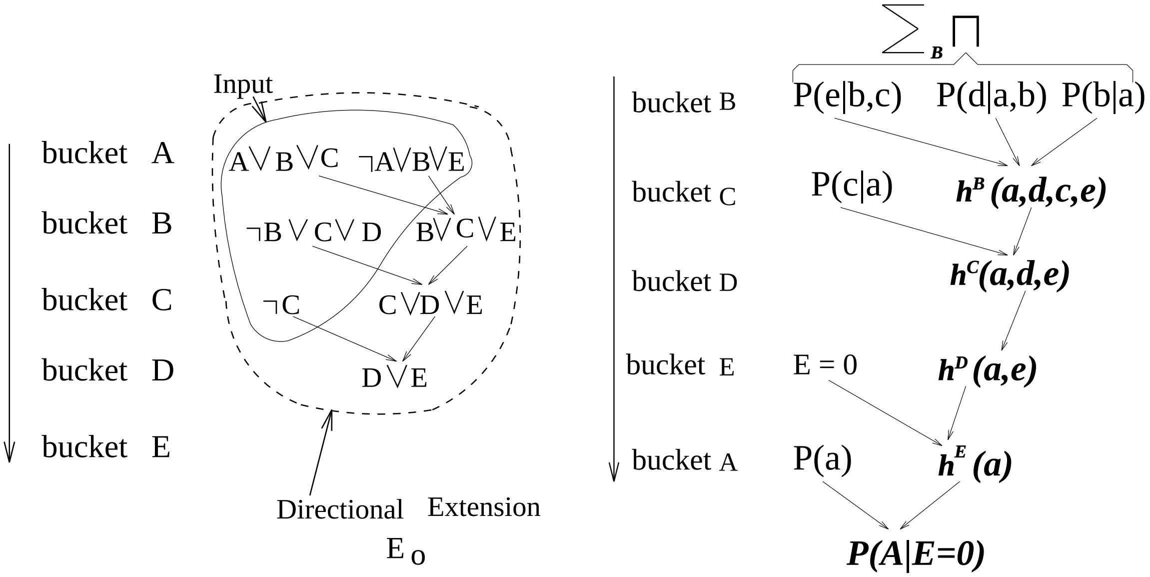
Figure 1.4: Backtracking search tree.

The worst-case time complexity of backtracking is exponential in the number of variables, while its space requirements are linear. However, the time complexity of backtracking varies considerably from one instance to another. Usually, the average performance of backtracking is much lower than its worst-case bound, and is affected by rare, but exceptionally hard instances; as noted by Donald E. Knuth in 1975, "great discrepancies in execution time are characteristic of backtrack programs." Long-tail exponential-family empirical distributions (e.g., lognormal, Weibull) observed in recent studies [50, 96] summarize such observations in a concise way.

1.2.2 Variable elimination

We next discuss the *bucket-elimination* framework [26, 27], that provides a unifying view of variable-elimination algorithms for a variety of reasoning tasks.

A bucket-elimination algorithm accepts as an input an ordered set of variables and a set of dependencies, such as propositional clauses, constraints, probability or utility functions. Each variable is then associated with a *bucket* constructed as follows: all the dependencies defined on variable X_i but not on higher-index variables are placed into the bucket of X_i, denoted $bucket_i$. Once the buckets are created, the algorithm processes them from last to first. It computes new dependencies, applying



(a) (b)

Figure 1.5: (a) A trace of algorithm *directional resolution*; (b) a trace of algorithm *elim-bel*.

an *elimination operator* to all the dependencies in the bucket. The new dependencies, summarizing the effect of X_i on the rest of the problem, are placed in the appropriate lower buckets. The elimination operation depends on the task. In propositional satisfiability, the bucket of X_i is processed by resolving all possible pairs of clauses over X_i; in belief updating, eliminating a variable X_i is equivalent to computing $\sum_{X_i} F$, where F is the product of all functions in $bucket_i$, and summation is over all values of X_i. The bucket-elimination algorithm terminates when all buckets are processed, or when some stopping criterion is satisfied. For example, for SAT, a theory is declared inconsistent as soon as an empty clause is generated.

To demonstrate the bucket-elimination approach, we consider two examples, one for belief updating, and one for SAT (Figure 1.5). Figure 1.5a illustrates the bucket-elimination algorithm for propositional satisfiability, called *directional resolution (DR)*. The algorithm is applied to the theory $\varphi = \{(\neg C), (A \vee B \vee C), (\neg A \vee B \vee E), (\neg B \vee C \vee D)\}$ along the ordering $o = (E, D, C, B, A)$. The theory is partitioned

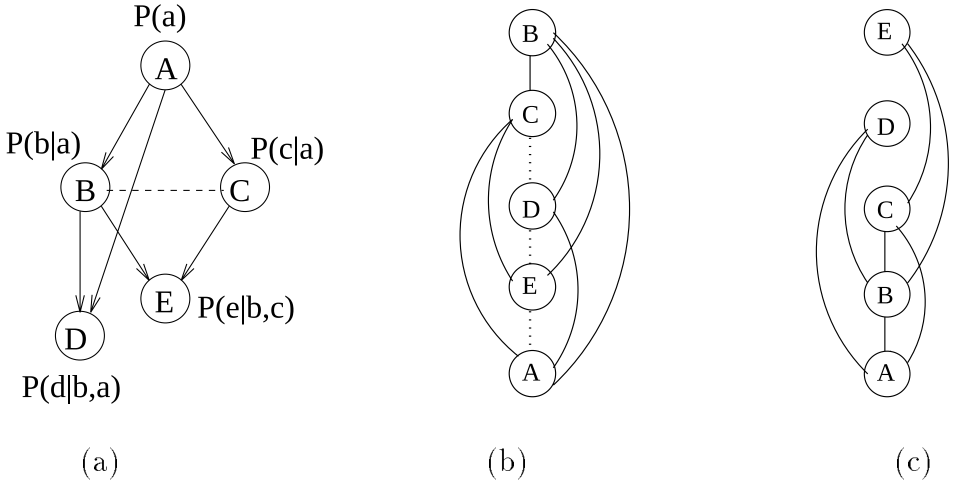


Figure 1.6: (a) A belief network, (b) its induced graph along $o = (A, E, D, C, B)$, and (c) its induced graph along $o = (A, B, C, D, E)$.

into buckets, as shown in Figure 1.5a, and processed by directional resolution in the reverse order. Resolving over variable A produces a new clause $(B \vee C \vee E)$, which is placed in the bucket of its highest-index variable, B. Resolving over B results into the new clause $(C \vee D \vee E)$ placed in the bucket of C. Finally, resolving over C produces the clause $(D \vee E)$, which is placed in the bucket of D. The buckets of D and E do not produce any new clauses. The theory is declared satisfiable since no empty clause was generated. The output of the algorithm is the collection of the clauses in all buckets, called a *directional extension* of φ. As it will be shown later, any model of φ can be found by consulting the directional extension in a backtrack-free manner.

Another example (Figure 1.5b) demonstrates the bucket-elimination algorithm for belief updating, called *elim-bel* [23], applied to the network in Figure 1.6a, given the query variable A, the ordering $o = (A, E, D, C, B)$, and evidence $E = 0$ (for illustration, we selected an arbitrary ordering which is not necessarily the most efficient one). The updated belief in A is computed as follows (upper-case letters denote variables, while lower-case letters denote their values):

$$P(a|e=0) = \sum_{E=0,d,c,b} P(a,b,d,c,e) = \sum_{E=0,d,c,b} P(a)P(c|a)P(e|b,c)P(d|a,b)P(b|a) =$$

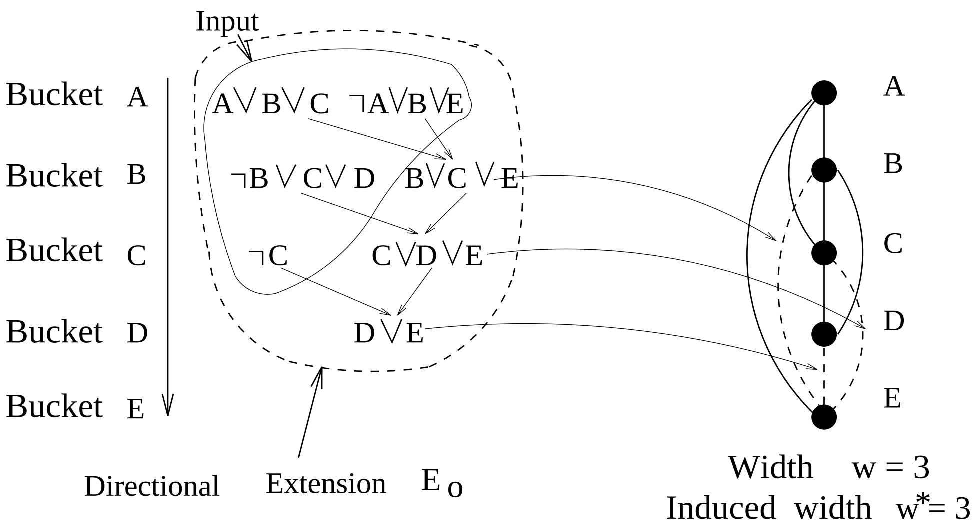


Figure 1.7: The effect of algorithm DR on the interaction graph of theory φ_2 along the ordering $o = (E, D, C, B, A)$.

$$\sum_{E=0}\sum_{d}\sum_{c} P(c|a) \sum_{b} P(e|b,c)P(d|a,b)P(b|a).$$

The bucket-elimination algorithm computes this sum from right to left using the buckets, as shown below:

1. bucket B: $h^B(a,d,c,e) = \sum_b P(e|b,c)P(d|a,b)P(b|a)$
2. bucket C: $h^C(a,d,e) = \sum_c P(c|a)h^B(a,d,c,e)$
3. bucket D: $h^D(a,e) = \sum_D h^C(a,d,e)$
4. bucket E: $h^E(a) = h^D(a, E = 0)$
5. bucket A: $Bel(a) = P(a|E = 0) = \alpha P(a)h^E(a)$,

where α is a normalizing constant. A schematic trace of the algorithm is also shown in Figure 1.5b.

Bucket-elimination algorithms record new dependencies in a form of conditional probability tables or relational tables (constraints). The number of variables in a dependence, k, is called the *arity* of the dependence. The table representation may require enumerating $O(exp(k))$ tuples. Therefore, the time and space complexity of bucket elimination is bounded by $O(m \cdot exp(k))$, where k is in the arity of largest dependence recorded, and m is the number of such dependencies.

The new dependencies are associated with new edges in the corresponding constraint graph, interaction graph, or moral graph. An important property of bucket-elimination algorithms is that their performance can be predicted using a graph parameter called *induced width* [33] (also known as *tree-width* [2]). This parameter describes the largest clique created in the problem's graph during the algorithm's execution. The induced width is defined as follows. Given a graph G, the *width* of X_i along ordering o is the number of X_i's neighbors preceding X_i in o. The *width of the graph* along o, denoted w_o, is the maximum width along o. The *induced graph* of G along o is obtained by connecting the preceding neighbors of each X_i, for i from n to 1. The induced width along o, denoted w^*_o, is the width of the induced graph along o, while the induced width w^* is the minimum induced width along any ordering. For example, Figures 1.6b and 1.6c depict the induced graphs (induced edges are shown as dashed lines) of the moral graph in Figure 1.6a along the orderings $o = (A, E, D, C, B)$ and $o' = (A, B, C, D, E)$, respectively. Clearly, $w^*_o = 4$ and $w^*_{o'} = 2$.

Figure 1.7 demonstrates how processing theory φ in Figure 1.5a by bucket elimination generates directional extension of φ containing new clauses that correspond to new edges in the interaction graph. Resolving over A creates the clause $(B \vee C \vee E)$, which corresponds to an induced edge between the nodes B and E. Similarly, resolving over B creates the clause $(C \vee D \vee E)$, which induces an edge between C and E. In this example both the width and the induced width equal 3.

It can be shown that the induced width $w^*_o(X_i)$ of node X_i bounds the number of arguments of any function computed in $bucket_i$. Consequently, the *number* of clauses (e.g., resolvents) defined on the variables in $bucket_i$ (or the *size* of a table representing a new probabilistic function) is $O(exp(w^*_o))$, where o is the elimination ordering. Therefore, the complexity of bucket-elimination algorithms is time and space exponential in w^*_o. Clearly, the induced width will vary with the variable ordering. Although finding a minimum-w^* ordering is NP-hard [2], good heuristic algorithms are available [7, 22, 97].

1.3 Thesis overview and results

The central theme of this thesis is that efficiency can be gained by reducing the induced width of variable-elimination algorithms. We investigated three approaches:

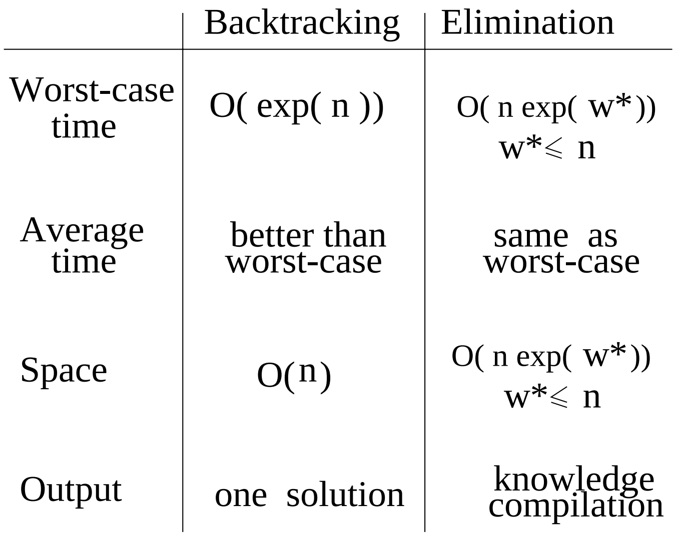
1. Combining elimination with backtracking search into hybrid schemes that use conditioning to reduce the induced width (Chapter 2).
2. Exploiting specific problem structures, such as causal independence in belief networks, that allows decomposition of large dependencies into smaller ones (Chapter 3).
3. Using approximation schemes, such as *mini-buckets*, that bound the size of recorded dependencies (Chapter 4).

The following three subsections summarize the contributions along each of those lines.

1.3.1 Hybrid algorithms for SAT (Chapter 2)

Chapter 2 compares backtracking search with the variable-elimination algorithm *directional resolution (DR)* for propositional satisfiability. Backtracking search and variable-elimination algorithms have distinct properties, summarized in Figure 1.8. As noted, the time complexity of backtracking is worst-case exponential in the number of variables, n, while algorithm bucket-elimination is time and space exponential in w^*, $w^* \leq n$, where w^* is the induced width along the given variable ordering. However, while the average performance of backtracking is often much better than its worst-case bound, the average complexity of elimination is close to its worst-case. In terms of space complexity, backtracking is linear in n, while the elimination algorithms require $O(n \cdot exp(w^*))$ space as well.

Because of their average-case performance and efficient memory use, backtracking algorithms are more popular than variable elimination for finding one solution (e.g., constraint satisfaction and optimization), while variable elimination is more suitable



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

Figure 1.8: Conditioning versus elimination.

for *knowledge compilation* tasks. However, even when only one solution is required, variable elimination can still be more efficient than backtracking for problems with low induced width. Indeed, we will provide experimental data demonstrating that on low-width problems elimination algorithms sometimes outperform backtracking by several orders of magnitude.

The complementary properties of backtracking and elimination, summarized in Figure 1.8, call for hybrid algorithms that exploit the advantages of both techniques. One approach is to reduce the amount of search by preprocessing it with some variable elimination. Another approach is to alternate between both methods, using conditioning to reduce a problem to a collection of subproblems that have smaller w^*, and thus can be tractable for variable elimination. For example, we may require that the induced width of an eliminated variables does not exceed b. As demonstrated in Figure 1.9 for $b = 2$, we condition on variable B since $w^*(B) > 2$, and resolve over the rest of variables having $w^* \leq 2$. Based on these ideas, Chapter 2 presents a family of parameterized hybrid algorithms allowing a flexible control between variable elimination and search.

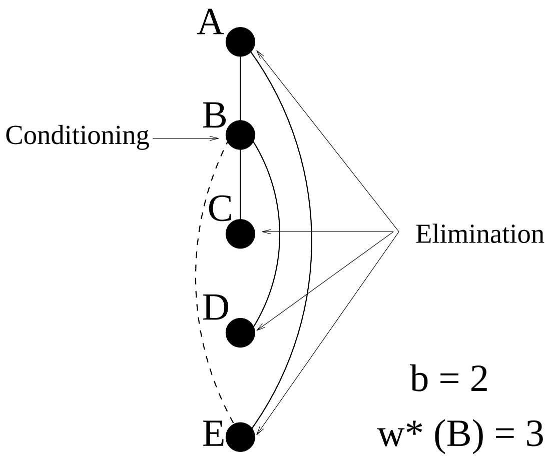


Figure 1.9: The idea of DCDR.

Contributions

The original resolution-based Davis-Putnam algorithm [19] is "revived" (in the form of a bucket-elimination algorithm called directional resolution (DR)) by analyzing its complexity, identifying tractable classes, and performing empirical studies. DR is compared against the backtracking-based Davis-Putnam-Logeman-Loveland Procedure [18] (herein called *DP*), which is one of the most effective complete satisfiability algorithms known to date. In summary,

1. We showed that the time and space complexity of DR is $O(n \cdot exp(w^*))$, where n is the number of variables and w^* is the induced width of the problem's interaction graph.

2. Our empirical studies confirm that DR is impractical for theories having large w^* such as uniform random 3-cnfs; however, on low-w^* problems, such as *k-tree-embeddings* [1], DR is very efficient and outperforms DP by several orders of magnitude.

3. We emphasized complementary properties of backtracking search and resolution, and proposed two parametric families of hybrid algorithms, BDR-DP(i)

and DCDR(b), that combine both approaches and that coincide with DR or DP at the extreme values of their parameters. Empirical evaluation demonstrates that the hybrid algorithms can be more efficient than both DR and DP.

1.3.2 Exploiting causal independence (Chapter 3)

In probabilistic networks, the specification of conditional probability tables (CPTs) is exponential in the family size, which may be large. For example, dozens of different diseases may cause the same symptom, such as fever. In such cases, even knowledge representation is difficult, and therefore, simplifying assumptions about the nature of probabilistic dependencies are required. The focus of Chapter 3 is on *causal independence* assumption [63, 111, 114] which reduces the CPT representation from exponential to linear in the family size, and which can be exploited to speed-up inference.

Causal independence assumes that several causes contribute independently to a common effect. For example, a burglary alarm can be turned on by a burglary or by an earthquake. Assessing the conditional probability of alarm given every possible combination of its causes is not straightforward either from statistical data, or from our beliefs, since the causal mechanisms are unrelated to each other (belong to different "frames of knowledge" [89]). Namely, the probability of an alarm not turning on in case of a burglary depends on burglar's skills, which are unrelated to earthquakes. Thus, we may assume independence of causal mechanisms, and specify them separately. Generally, a causally-independent probabilistic relation between a set of causes $c_1, ..., c_n$ and an effect e can be decomposed into a noisy transformation of each cause c_i into a *hidden* variable u_i, and a deterministic function $e = u_1 * ... * u_n$, where $*$ is a commutative and associative binary operator, such as logical OR, logical AND, addition, multiplication, etc. Figure 1.10 demonstrates such decomposition.

Chapter 3 investigates how causal independence can be further exploited to improve inference algorithms, building upon the approaches of [63, 84] and [114]. We

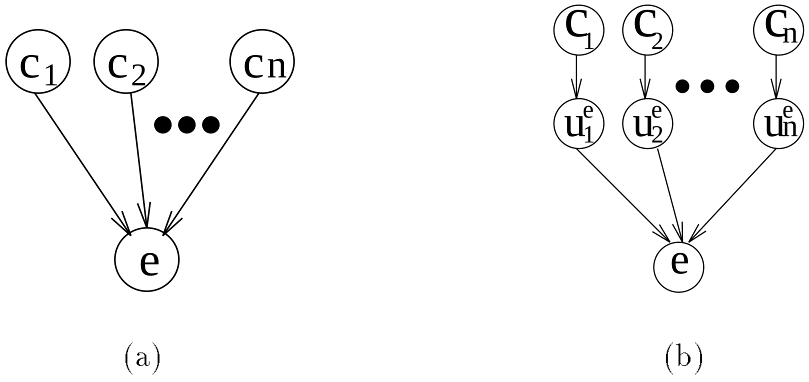


Figure 1.10: (a) a belief network and (b) its decomposition using causal independence.

show that the "effective" induced width of algorithms exploiting causal independence can be significantly reduced. For example, exploiting causal independence in polytrees with an arbitrary large family of size m reduces the effective induced width from m to 2. Given a network, the anticipated computational benefits can be evaluated in advance and contrasted with those of general-purpose "causally-blind" algorithms.

Contributions

We investigated the impact of causal independence on several probabilistic tasks, such as belief updating, finding a most probable explanation (MPE), finding a maximum a posteriori hypothesis (MAP), and finding the maximum expected utility (MEU) decision. Specifically,

1. We explicated the relationship between the previously proposed approaches, such as network transformations and variable-elimination algorithm VE1 [114], using the bucket-elimination framework. We showed that the ordering restrictions implied by algorithm VE1 may sometimes lead to an unnecessary (exponential) complexity increase, and proposed a more general variable-elimination scheme, called *ci-elim-bel*, that avoids VE1's drawback by allowing any variable ordering.

2. We presented bucket-elimination algorithms that exploit causal independence

for the tasks of belief updating, MPE, and MEU. We also showed that, generally, causal independence cannot be exploited for finding the MPE.

3. We analyzed the complexity of the above "causally-informed" algorithms, and showed a significant potential reduction of their "effective" induced width, up to the induced width of the (unmoral) input network (note that the induced width of standard "causally-blind" elimination algorithms is computed on the moral network).
4. Finally, we showed how constraint-propagation techniques can be used for evidence propagation in causally-independent networks.

1.3.3 Approximate inference (Chapter 4)

Another way of coping with computational complexity is to look for *approximate* rather than exact solutions. Although approximation within a given error bound is known to be NP-hard [86, 98], there are approximation strategies that work well in practice. One approach advocates *anytime algorithms*. These algorithms can be interrupted at any time, producing the best solution found thus far [20, 8]. Another approach is to identify problem classes that can be solved approximately within given error bounds, thus applying the idea of tractability to approximation.

Chapter 4 presents the general framework of *mini-bucket* approximations that trade accuracy for efficiency in those cases when computational resources are bounded. This class of mini-bucket algorithms imports the idea of *local inference* from constraint networks to probabilistic reasoning and combinatorial optimization. *Local inference* algorithms like *i-consistency* [44, 22] bound the computational complexity by restricting the arity of recorded dependencies to i. Known special cases are *arc-consistency* ($i = 2$) and *path-consistency* ($i = 3$) [78, 43, 22]. Indeed, the recent success of constraint-processing algorithms can be attributed primarily to this class of algorithms, especially when combined with backtracking search [28, 29]. The idea, demonstrated in Figure 1.11, shows that while exact algorithms may record arbitrarily

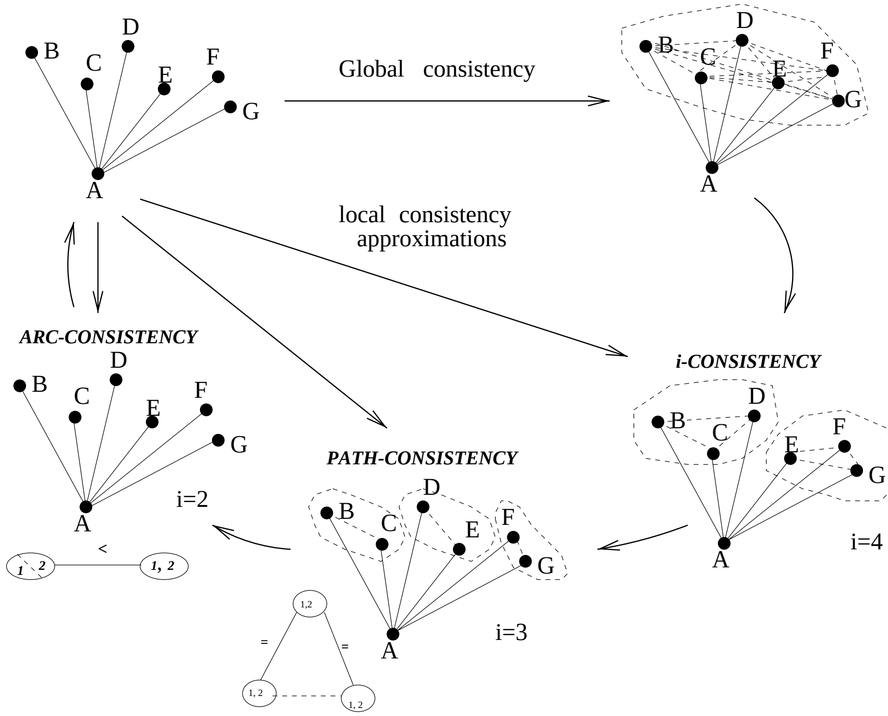


Figure 1.11: From global to local consistency: algorithm i-consistency and its particular cases path-consistency (i=3) and arc-consistency (i=2).

large constraints, i-consistency algorithms decide consistency of smaller subproblems, recording constraints of size i or less.

Contributions

The mini-bucket algorithms for probabilistic tasks of belief updating, finding the most probable explanation, finding the maximum a posteriori hypothesis, and for optimization tasks are presented and analyzed. We identify regions of completeness and demonstrate promising empirical results obtained both on randomly generated networks and on realistic domains such as medical diagnosis and probabilistic decoding. For example, for noisy-OR random networks and for CPCS networks, we often computed an accurate solution in cases when the exact algorithm was much slower, or infeasible. For probabilistic decoding, we obtained preliminary results that demonstrated the advantages of the mini-bucket scheme over the state-of-the-art iterative belief propagation decoding algorithm on problems having low induced width.

Theoretical bounds on the complexity of mini-bucket algorithms allow us to predict in advance, using both memory considerations and the problem's graph, the suitability of the algorithm's parameters for given networks.

1.3.4 Organization of this thesis

This chapter provided an overview of automated reasoning tasks addressed in this thesis and summarized the main ideas and contributions of our work. Chapters 2,3 and 4 will present the main results. Those chapters are minor modifications of the articles currently submitted to journals. Therefore, we apologize for possible minor repetitions (e.g., some figures and definitions in the introduction were borrowed from the corresponding chapters). Chapter 5 concludes this thesis by summarizing the results and outlining the directions for future work.

Chapter 2

Hybrid Algorithms for SAT

2.1 Introduction

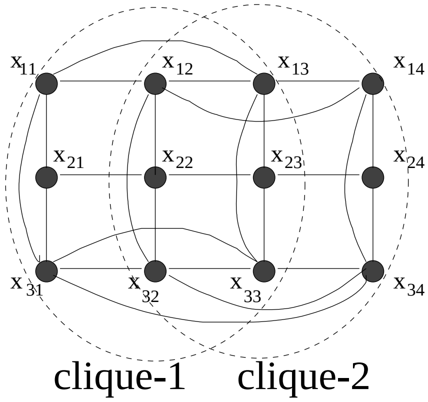
Propositional satisfiability (SAT) is a prototypical example of an NP-complete problem; any NP-complete problem is reducible to SAT in polynomial time [12]. Since many practical applications such as planning, scheduling, and diagnosis can be formulated as propositional satisfiability, finding algorithms with good average performance has been a focus of extensive research for many years [102, 15, 49, 70, 71, 4]. In this chapter, we consider *complete* SAT algorithms that can always determine satisfiability as opposed to incomplete *local search* techniques [102, 101]. The two most widely used complete techniques are backtracking search (e.g., the Davis-Putnam Procedure [18]) and resolution (e.g., Directional Resolution [19, 35]). We compare both approaches theoretically and empirically, suggesting several ways of combining them into more effective hybrid algorithms.

In 1960, Davis and Putnam presented a resolution algorithm for deciding propositional satisfiability (the Davis-Putnam algorithm [19]). They proved that a restricted amount of resolution performed along some ordering of the propositions in a propositional theory is sufficient for deciding satisfiability. However, this algorithm has received limited attention and analyses of its performance have emphasized its worst-case exponential behavior [52, 57], while overlooking its virtues. It was quickly overshadowed by the Davis-Putnam Procedure, introduced in 1962 by Davis, Logemann, and Loveland [18]. They proposed a minor syntactic modification of the original algorithm: the resolution rule was replaced by a splitting rule in order to avoid an exponential memory explosion. However, this modification changed the nature of the algorithm and transformed it into a backtracking scheme. Most of the work on propositional satisfiability quotes the backtracking version [58, 82]. We will refer to the original Davis-Putnam algorithm as *DP-resolution*, or *directional resolution (DR)*[1], and to its later modification as *DP-backtracking*, or *DP* (also called DPLL in the SAT community).

Our evaluation has a substantial empirical component. A common approach used in the empirical SAT community is to test algorithms on randomly generated problems, such as uniform random k-SAT [82]. However, these benchmarks often fail to simulate realistic problems. On the other hand, "real-life" benchmarks are often available only on an instance-by-instance basis without any knowledge of underlying distributions which makes the empirical results hard to generalize. An alternative approach is to use *structured* random problem generators inspired by the properties of some realistic domains. For example, Figure 2.1 illustrates the *unit commitment* problem of scheduling a set of n power generating units over T hours (here $n = 3$ and $T = 4$). The state of unit i at time t ("up" or "down") is specified by the value of boolean variable x_{it} (0 or 1), while the minimum up- and down-time constraints specify how long a unit must stay in a particular state before it can be switched. The corresponding constraint graph can be embedded in a chain of cliques where each clique includes the variables within the given number of time slices determined by the up- and down-time constraints. These clique-chain structures are common in many temporal domains that possess the *Markov property* (the future is independent

---

[1] A similar approach known as "ordered resolution" can be viewed as a more sophisticated first order version of directional resolution [38].



Unit #	Min Up Time	Min Down Time
1	3	2
2	2	1
3	4	1

Figure 2.1: An example of a "temporal chain": the unit commitment problem for 3 units over 4 hours.

of the past given the present). Another example of structured domain is circuit diagnosis. In [40] it was shown that circuit-diagnosis benchmarks can be embedded in a tree of cliques, where the clique sizes are substantially smaller than the overall number of variables. In general, one can imagine a variety of real-life domains having such structure that is captured by *k-tree-embeddings* [1] used in our random problem generators.

Our empirical studies of SAT algorithms confirm previous results: DR is very inefficient when dealing with unstructured uniform random problems. However, on structured problems such as k-tree embeddings having bounded *induced width*, directional resolution outperforms DP-backtracking by several orders of magnitude. The induced width (denoted w^*) is a graph parameter that describes the size of the largest clique created in the problem's interaction graph during inference. We show that the worst-case time and space complexity of DR is $O(n \cdot exp(w^*))$, where n is the number of variables. We also identify tractable problem classes based on a more refined syntactic parameter, called *diversity*.

Since the induced width is often smaller than the number of propositional variables, n, DR's worst-case bound is generally better than $O(exp(n))$, the worst-case time bound for DP. In practice, however, DP-backtracking – one of the best complete

	Backtracking	Resolution
Worst-case time	$O(\exp(n))$	$O(n \exp(w^*))$ $w^* \leqslant n$
Average time	better than worst-case	same as worst-case
Space	$O(n)$	$O(n \exp(w^*))$ $w^* \leqslant n$
Output	one solution	knowledge compilation

Figure 2.2: Comparison between backtracking and resolution.

SAT algorithms available – is often much more efficient than its worst-case bound. It demonstrates "great discrepancies in execution time" (D.E. Knuth), encountering rare but exceptionally hard problems [109]. Recent studies suggest that the empirical performance of backtracking algorithms can be modeled by long-tail exponential-family distributions, such as lognormal and Weibull [50, 96]. The average complexity of algorithm DR, on the other hand, is close to its worst-case [31]. It is important to note that the space complexity of DP is $O(n)$, while DR is space-exponential in w^*. Another difference is that in addition to deciding satisfiability and finding a solution (a model), directional resolution also generates an equivalent theory that allows finding each model in linear time (and finding all models in time linear in the number of models), and thus can be viewed as a knowledge-compilation algorithm.

The complementary characteristics of backtracking and resolution (Figure 2.2) call for hybrid algorithms. We present two hybrid schemes, both using control parameters that restrict the amount of resolution by bounding the resolvent size, either in a preprocessing phase or dynamically during search. These parameters allow time/space trade-offs that can be adjusted to the given problem structure and to the computational resources. Empirical studies demonstrate the advantages of these flexible hybrid schemes over both extremes, backtracking and resolution.

The rest of this chapter is organized as follows. Section 2.2 provides necessary definitions. Section 2.3 describes directional resolution (DR), our version of the original Davis-Putnam algorithm expressed within the *bucket-elimination* framework. Section 2.4 discusses the complexity of DR and identifies tractable classes, while Section 2.5 focuses on DP-backtracking. Empirical comparison of DR and DP is presented in Section 2.6. Section 2.7 introduces the two hybrid schemes, BDR-DP and DCDR, and empirically evaluates their effectiveness. Related work and conclusions are discussed in Sections 2.8 and 2.9. Proofs of theorems are given in the Appendix A.

2.2 Definitions and preliminaries

We denote propositional variables, or propositions, by uppercase letters, e.g. P, Q, R, propositional literals (propositions or their negations, such as P and $\neg P$) by lowercase letters, e.g., p, q, r, and disjunctions of literals, or *clauses*, by the letters of the Greek alphabet, e.g., α, β, γ. For instance, $\alpha = (P \vee Q \vee R)$ is a clause. We will sometimes denote the clause $(P \vee Q \vee R)$ by $\{P, Q, R\}$. A *unit clause* is a clause with only one literal. A clause is *positive* if it contains only positive literals and is *negative* if it contains only negative literals. The notation $(\alpha \vee T)$ is used as shorthand for $(P \vee Q \vee R \vee T)$, while $\alpha \vee \beta$ refers to the clause whose literals appear in either α or β. A clause β is *subsumed* by a clause α if β's literals include all α's literals. A clause is a *tautology*, if for some proposition Q the clause includes both Q and $\neg Q$. A *propositional theory* φ in conjunctive normal form (*cnf*) is represented as a set $\{\alpha_1, ..., \alpha_t\}$ denoting the conjunction of clauses $\alpha_1, ..., \alpha_t$. A *k-cnf theory* contains only clauses of length k or less. A propositional cnf theory φ defined on a set of n variables Q_1,...,Q_n is often called simply "a theory φ".

The set of *models* of a theory φ is the set of all truth assignments to its variables

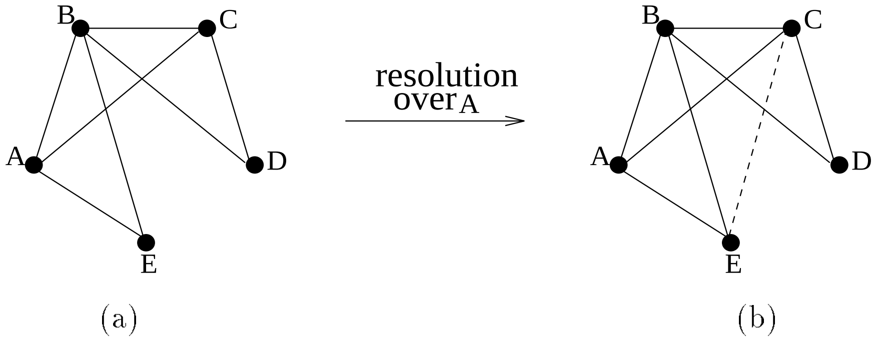


Figure 2.3: (a) The interaction graph of theory $\varphi_1 = \{(\neg C), (A \vee B \vee C), (\neg A \vee B \vee E), (\neg B \vee C \vee D)\}$, and (b) the effect of resolution over A on that graph.

that satisfy φ. A clause α is *entailed* by φ (denoted $\varphi \models \alpha$), if and only if α is true in all models of φ. A propositional satisfiability problem (SAT) is to decide whether a given cnf theory has a model. A SAT problem defined on k-cnfs is called a *k-SAT* problem.

The structure of a propositional theory can be described by an *interaction graph.* The interaction graph of a propositional theory φ, denoted $G(\varphi)$, is an undirected graph that contains a node for each propositional variable and an edge for each pair of nodes that correspond to variables appearing in the same clause. For example, the interaction graph of theory $\varphi_1 = \{(\neg C), (A \vee B \vee C), (\neg A \vee B \vee E), (\neg B \vee C \vee D)\}$ is shown in Figure 2.3a.

One commonly used approach to satisfiability testing is based on the *resolution* operation. Resolution over two clauses $(\alpha \vee Q)$ and $(\beta \vee \neg Q)$ results in a clause $(\alpha \vee \beta)$ (called *resolvent*) eliminating variable Q. The interaction graph of a theory processed by resolution should be augmented with new edges reflecting the added resolvents. For example, resolution over variable A in φ_1 generates a new clause $(B \vee C \vee E)$, so the graph of the resulting theory has an edge between nodes E and C as shown in Figure 2.3b. Resolution with a unit clause is called *unit resolution. Unit propagation* is an algorithm that applies unit resolution to a given cnf theory until no new clauses can be deduced.

```
Directional Resolution: DR
Input: A cnf theory $\varphi$, $o = Q_1, ..., Q_n$.
Output: The decision of whether $\varphi$ is satisfiable.
If it is, the directional extension $E_o(\varphi)$ equivalent to $\varphi$.
1. Initialize: generate a partition of clauses, $bucket_1, ..., bucket_n$,
where $bucket_i$ contains all the clauses whose highest literal is $Q_i$.
2. For $i = n$ to 1 do:
        If there is a unit clause in $bucket_i$,
        do unit resolution in $bucket_i$
        else resolve each pair $\{(\alpha \vee Q_i), (\beta \vee \neg Q_i)\} \subseteq bucket_i$.
        If $\gamma = \alpha \vee \beta$ is empty, return "$\varphi$ is unsatisfiable"
        else add $\gamma$ to the bucket of its highest variable.
3. Return "$\varphi$ is satisfiable" and $E_o(\varphi) = \bigcup_i bucket_i$.
```

Figure 2.4: Algorithm Directional Resolution (DR).

Propositional satisfiability is a special case of *constraint satisfaction problem (CSP)*. CSP is defined on a *constraint network* $< X, D, C >$, where $X = \{X_1, ..., X_n\}$ is the set of *variables*, associated with a set of finite *domains*, $D = \{D_i, ..., D_n\}$, and a set of *constraints*, $C = \{C_1, ..., C_m\}$. Each constraint C_i is a relation $R_i \subseteq D_{i_1} \times ... \times D_{i_k}$ defined on a subset of variables $S_i = \{X_{i_1}, ..., X_{i_k}\}$. A constraint network can be associated with an undirected *constraint graph* where nodes correspond to variables and two nodes are connected if and only if they participate in the same constraint. The *constraint satisfaction problem (CSP)* is to find a value assignment to all the variables (called a *solution*) that is consistent with all the constraints. If no such assignment exists, the network is *inconsistent*. A constraint network is *binary* if each constraint is defined on at most two variables.

2.3 Directional Resolution (DR)

DP-resolution [19] is an ordering-based resolution algorithm that can be described as follows. Given an arbitrary ordering of the propositional variables, we assign to each clause the index of its highest literal in the ordering. Then resolution is applied only to clauses having the same index and only on their highest literal. The result of this restriction is a systematic elimination of literals from the set of clauses that are candidates for future resolution. The original DP-resolution also includes two additional steps, one forcing unit resolution whenever possible, and one assigning values to *all-positive* and *all-negative* variables. An all-positive (all-negative) variable is a variable that appears only positively (negatively) in a given theory, so that assigning such a variable the value "true" ("false") is equivalent to deleting all relevant clauses from the theory. There are other intermediate steps that can be introduced between the basic steps of eliminating the highest indexed variable, such as deleting subsumed clauses. Albeit, we will focus on the ordered elimination step and refer to auxiliary steps only when necessary. We are interested not only in deciding satisfiability but in the set of clauses accumulated by this process constituting an equivalent theory with useful computational features. Algorithm *directional resolution* (DR), the core of DP-resolution, is presented in Figure 2.4. This algorithm can be described using the notion of *buckets*, which define an ordered partitioning of clauses in φ, as follows. Given an ordering $o = (Q_1 , ..., Q_n)$ of the variables in φ, all the clauses containing Q_i that do not contain any symbol higher in the ordering are placed in $bucket_i$. The algorithm processes the buckets in a reverse order of o, from Q_n to Q_1. Processing $bucket_i$ involves resolving over Q_i all possible pairs of clauses in that bucket. Each resolvent is added to the bucket of its highest variable Q_j (clearly, $j < i$). Note that if the bucket contains a unit clause (Q_i or $\neg Q_i$), only unit resolutions are performed. Clearly, a useful dynamic-order heuristic (not included in our current implementation) is to processes next a bucket with a unit clause. The output theory, $E_o(\varphi)$, is called the *directional extension* of φ along o. As shown by Davis and Putnam [19],

```
**find-model** $(E_o(\varphi), o)$
**Input:** A *directional extension* $E_o(\varphi)$, $o = Q_1, ..., Q_n$.
**Output:** A model of $\varphi$.
1. **For** $i = 1$ to N
        $Q_i \leftarrow$ a value $q_i$ consistent with the assignment to
        $Q_1, ..., Q_{i-1}$ and with all the clauses in $bucket_i$.
2. Return $Q_1 = q_1, ..., Q_n = q_n$.
```

Figure 2.5: Algorithm *find-model*.

the algorithm finds a satisfying assignment to a given theory if and only if there exists one. Namely,

Theorem 1: *[19] Algorithm DR is sound and complete.* □

A model of a theory φ can be easily found by consulting $E_o(\varphi)$ using a simple model-generating procedure *find-model* in Figure 2.5. Formally,

Theorem 2: (model generation)
Given $E_o(\varphi)$ of a satisfiable theory φ, the procedure find-model *generates a model of φ backtrack-free, in time $O(|E_o(\varphi)|)$.* □

Example 1: 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 $o = (E, D, C, B, A)$, the theory is partitioned into buckets and processed by directional resolution in reverse order[2]. 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

---

[2] For illustration, we selected an arbitrary ordering which is not the most efficient one. Variable ordering heuristics will be discussed in Section 2.4.3.

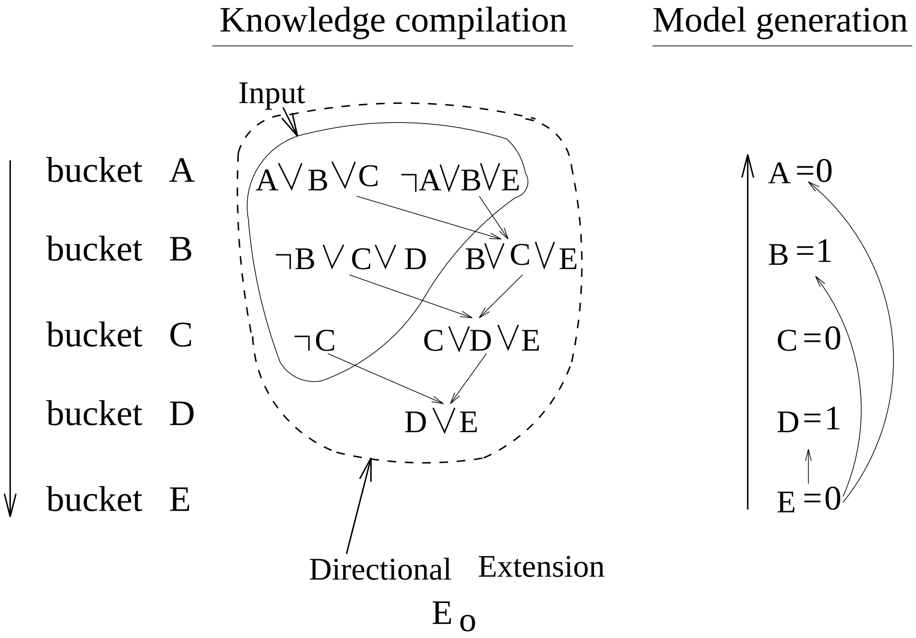


Figure 2.6: A trace of algorithm DR on the theory $\varphi_1 = \{(\neg C), (A \vee B \vee C), (\neg A \vee B \vee E), (\neg B \vee C \vee D)\}$ along the ordering $o = (E, D, C, B, A)$.

output is a non-empty directional extension $E_o(\varphi_1)$. Once the directional extension is available, model generation begins. There are no clauses in the bucket of E, the first variable in the ordering, and therefore E can 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.

As stated in Theorem 2, given a directional extension, a model can be generated in linear time. Once $E_o(\varphi)$ is compiled, determining the entailment of a single literal requires checking the bucket of that literal first. If the literal appears there as a unit clause, it is entailed; if not, its negation is added to the appropriate bucket and the algorithm resumes from that bucket. If the empty clause is generated, the literal is entailed.

2.4 Complexity and tractability

Clearly, the effectiveness of algorithm DR depends on the the size of its output theory $E_o(\varphi)$.

Theorem 3: (complexity)
Given a theory φ *and an ordering* o*, the complexity of algorithm DR is* $O(n|E_o(\varphi)|^2)$ *where* n *is the number of variables.* $\square$

The size of the directional extension and therefore the complexity of directional resolution is worst-case exponential in the number of variables. However, there are identifiable cases when the size of $E_o(\varphi)$ is bounded, yielding tractable problem classes. The order of variable processing has a particularly significant effect on the size of the directional extension. Consider the following two examples:

Example 2: Let $\varphi_2 = \{(B \vee A), (C \vee \neg A), (D \vee A), (E \vee \neg A)\}$. Given the ordering $o_1 = (E, B, C, D, A)$, all clauses are initially placed in $bucket(A)$. Applying DR along the (reverse) ordering, we get: $bucket(D) = \{(C \vee D), (D \vee E)\}$, $bucket(C) = \{(B \vee C)\}$, $bucket(B) = \{(B \vee E)\}$. In contrast, the directional extension along ordering $o_2 = (A, B, C, D, E)$ is identical to the input theory φ_2 since each bucket contains at most one clause.

Example 3: Consider the theory $\varphi_3 = \{(\neg A \vee B), (A \vee \neg C), (\neg B \vee D), (C \vee D \vee E)\}$. The directional extensions of φ_3 along ordering $o_1 = (A, B, C, D, E)$ and $o_2 = (D, E, C, B, A)$ are $E_{o_1}(\varphi_3) = \varphi_3$ and $E_{o_2}(\varphi_3) = \varphi_3 \cup \{(B \vee \neg C), (\neg C \vee D), (E \vee D)\}$, respectively.

In example 2, variable A appears in all clauses. Therefore, it can potentially generate new clauses when resolved upon, unless it is processed last (i.e., it appears first in the ordering), as in o_2. This shows that the interactions among variables can affect the performance of the algorithm and should be consulted for producing preferred orderings. In example 3, on the other hand, all the symbols have the same

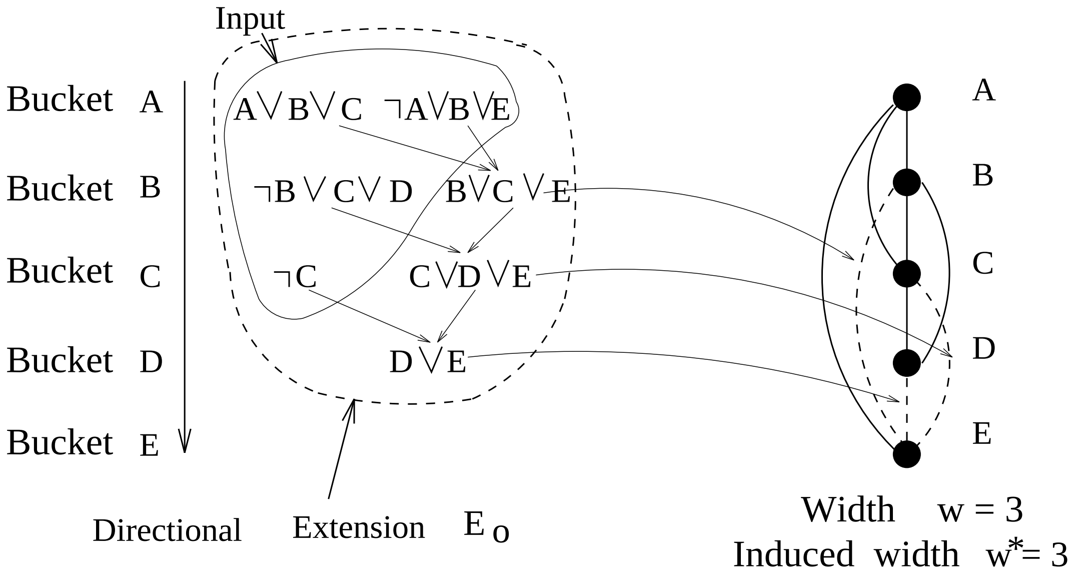


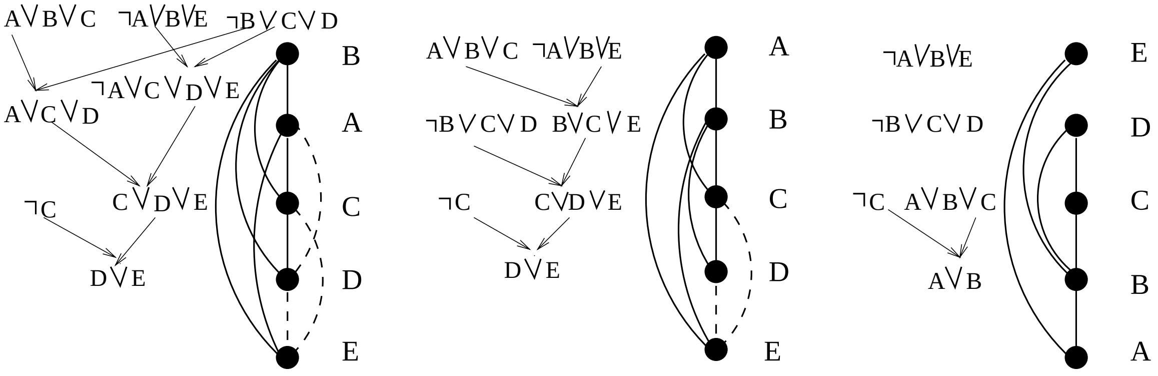
Figure 2.7: The effect of algorithm DR on the interaction graph of theory $\varphi_1 = \{(\neg C), (A \vee B \vee C), (\neg A \vee B \vee E), (\neg B \vee C \vee D)\}$ along the ordering $o = (E, D, C, B, A)$.

type of interaction, each (except E) appearing in two clauses. Nevertheless, D appears positive in both clauses in its bucket, therefore, it will not be resolved upon and can be processed first. Subsequently, B and C appear only negatively in the remaining theory and will not add new clauses. Inspired by these two examples, we will now provide a connection between the algorithm's complexity and two parameters: a topological parameter, called *induced width*, and a syntactic parameter, called *diversity*.

2.4.1 Induced width

In this section we show that the size of the directional extension and therefore the complexity of directional resolution can be estimated using a graph parameter called *induced width*.

As noted before, DR creates new clauses which correspond to new edges in the resulting interaction graph (we say that DR "induces" new edges). Figure 2.7 illustrates again the performance of directional resolution on theory φ_1 along ordering $o = (E, D, C, B, A)$, showing the interaction graph of $E_o(\varphi_1)$ (dashed lines correspond



(a) $w^* = 4$ (b) $w^* = 3$ (c) $w^* = 2$

Figure 2.8: The effect of the ordering on the induced width: interaction graph of theory $\varphi_1 = \{(\neg C), (A \vee B \vee C), (\neg A \vee B \vee E), (\neg B \vee C \vee D)\}$ along the orderings (a) $o_1 = (E, D, C, A, B)$, (b) $o_2 = (E, D, C, B, A)$, and (c) $o_3 = (A, B, C, D, E)$.

to induced edges). Resolving over A creates clause $(B \vee C \vee E)$ which corresponds to a new edge between nodes B and E, while resolving over B creates clause $(C \vee D \vee E)$ which induces a new edge between C and E. In general, processing a bucket of a variable Q produces resolvents that connect all the variables mentioned in that bucket. The concepts of induced graph and induced width are defined to reflect those changes.

Definition 1: Given a graph G, and an ordering of its nodes o, the *parent set* of a node X_i is the set of nodes connected to X_i that precede X_i in o. The size of this parent set is called the *width* of X_i relative to o. The *width of the graph* along o, denoted w_o, is the maximum width over all variables. The *induced graph* of G along o, denoted $I_o(G)$, is obtained as follows: going from $i = n$ to $i = 1$, we connect all the neighbors of X_i preceding it in the ordering. The induced width of G along o, denoted w^*_o, is the width of $I_o(G)$ along o, while the induced width w^* of G is the minimum induced width along any ordering.

For example, in Figure 2.7 the induced graph $I_o(G)$ contains the original (bold) and the induced (dashed) edges. The width of B is 2, while its induced width is 3; the width of C is 1, while its induced width is 2. The maximum width along o is

3 (the width of A), and the maximum induced width is also 3 (the induced width of A and B). Therefore, in this case, the width and the induced width of the graph coincide. In general, however, the induced width of a graph can be significantly larger than its width. Note that in this example the graph of the directional extension, $G(E_o(\varphi))$, coincides with the induced ordered graph of the input theory's graph, $I_o(G(\varphi))$. Generally,

Lemma 1: *Given a theory* φ *and an ordering* o*,* $G(E_o(\varphi))$ *is a subgraph of* $I_o(G(\varphi))$*.* $\Box$

The parents of node X_i in the induced graph correspond to the variables mentioned in $bucket_i$. Therefore, the induced width of a node can be used to estimate the size of its bucket, as follows:

Lemma 2: *Given a theory* φ *and an ordering* $o = (Q_1, ..., Q_n)$*, if* Q_i *has at most* k *parents in the induced graph along* o*, then the bucket of a variable* Q_i *in* $E_o(\varphi)$ *contains no more than* 3^{k+1} *clauses.* $\Box$

We can now derive a bound on the complexity of directional resolution using properties of the problem's interaction graph.

Theorem 4: *(complexity of DR)*
Given a theory φ *and an ordering of its variables* o*, the time complexity of algorithm DR along* o *is* $O(n \cdot 9^{w_o^*})$*, and* $E_o(\varphi)$ *contains at most* $n \cdot 3^{w_o^*+1}$ *clauses, where* w_o^* *is the induced width of* φ*'s interaction graph along* o*.* $\Box$

Corollary 1: *Theories having bounded* w_o^* *for some ordering* o *are tractable.* $\Box$.

Figure 2.8 demonstrates the effect of variable ordering on the induced width, and consequently, on the complexity of DR when applied to theory φ_1. While DR generates 3 new clauses of length 3 along ordering (a), only one binary clause is generated along ordering (c). Although finding an ordering that yields the smallest

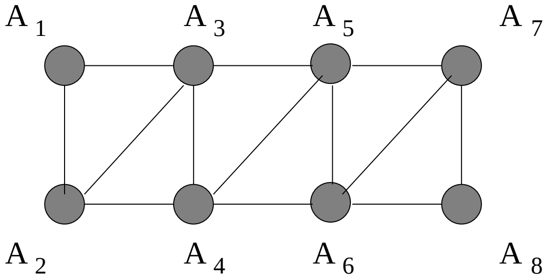


Figure 2.9: The interaction graph of φ_4 in example 4: $\varphi_4 = \{(A_1 \vee A_2 \vee \neg A_3)$, $(\neg A_2 \vee A_4)$, $(\neg A_2 \vee A_3 \vee \neg A_4)$, $(A_3 \vee A_4 \vee \neg A_5)$, $(\neg A_4 \vee A_6)$, $(\neg A_4 \vee A_5 \vee \neg A_6)$, $(A_5 \vee A_6 \vee \neg A_7)$, $(\neg A_6 \vee A_8)$, $(\neg A_6 \vee A_7 \vee \neg A_8)\}$.

induced width is NP-hard [1], good heuristic orderings are currently available [7, 22, 97] and continue to be explored [5]. Furthermore, there is a class of graphs, known as *k-trees*, that have $w^* < k$ and can be recognized in $O(n \cdot exp(k))$ time [1].

Definition 2: (k-trees)

1. A clique of size k (complete graph with k nodes) is a k-tree.
2. Given a k-tree defined on $X_1, ..., X_{i-1}$, a k-tree on $X_1, ..., X_i$ can be generated by selecting a clique of size k and connecting X_i to every node in that clique.

Corollary 2: *If the interaction graph of a theory φ having n variables is a subgraph of a k-tree, then there is an ordering o such that the space complexity of algorithm DR along o (the size of $E_o(\varphi)$) is $O(n \cdot 3^k)$, and its time complexity is $O(n \cdot 9^k)$.* $\square$

Important tractable classes are *trees* ($w^* = 1$) and *series-parallel networks* ($w^* = 2$). These classes can be recognized in polynomial (linear or quadratic) time.

Example 4: Consider a theory φ_n defined on the variables $\{A_1, A_2, ..., A_n\}$. A clause $(A_i \vee A_{i+1} \vee \neg A_{i+2})$ is defined for each odd i, and two clauses $(\neg A_i \vee A_{i+2})$ and $(\neg A_i \vee A_{i+1} \vee \neg A_{i+2})$ are defined for each even i, where $1 \leq i \leq n$. The interaction graph of φ_n for $n = 5$ is shown in Figure 2.9. The reader can verify that the graph is a 3-tree ($w^* = 2$) and that its induced width along the original ordering is 2. Therefore, by theorem 4, the size of the directional extension will not exceed $27n$.

2-SAT

Note that algorithm DR is tractable for 2-cnf theories, because 2-cnfs are closed under resolution (the resolvents are of size 2 or less) and because the overall number of clauses of size 2 is bounded by $O(n^2)$ (in this case, unordered resolution is also tractable), yielding $O(n \cdot n^2) = O(n^3)$ complexity. Therefore,

Theorem 5: *Given a 2-cnf theory φ, its directional extension $E_o(\varphi)$ along any ordering o is of size $O(n^2)$, and can be generated in $O(n^3)$ time.*

Obviously, DR is not the best algorithm for solving 2-SAT, since 2-SAT can be solved in linear time [39]. Note, however, that DR also compiles the theory into one that can produces each model in linear time. As shown in [30], in this case all models can be generated in output linear time.

The graphical effect of unit resolution

Resolution with a unit clause Q or $\neg Q$ deletes the opposite literal over Q from all relevant clauses. It is equivalent to assigning a value to variable Q. Therefore, unit resolution generates clauses on variables that are already connected in the graph, and therefore will not add new edges.

2.4.2 Diversity

The concept of induced width sometimes leads to a loose upper bound on the number of clauses recorded by DR. In Example 4, only six clauses were generated by DR, even without eliminating subsumption and tautologies in each bucket, while the computed bound is $27n = 27 \cdot 8 = 216$. Consider the two clauses $(\neg A \vee B)$ and $(\neg C \vee B)$ and the order $o = A, C, B$. When bucket B is processed, no clause is added because B is positive in both clauses, yet nodes A and C are connected in the induced graph. In this subsection, we introduce a new parameter called *diversity*, that provides a tighter bound on the number of resolution operations in the bucket. Diversity is based on the fact that a proposition can be resolved upon only when it appears both positively and negatively in different clauses.

Definition 3: (diversity)
Given a theory φ and an ordering o, let Q_i^+ (Q_i^-) denote the number of times Q_i appears positively (negatively) in $bucket_i$. The *diversity of* Q_i relative to o, $div(Q_i)$, is defined as $Q_i^+ \times Q_i^-$. The *diversity of an ordering* o, $div(o)$, is the largest diversity of its variables relative to o, and the *diversity of a theory*, div, is the minimal diversity among all orderings.

The concept of diversity yields new tractable classes. For example, if o is an ordering having a zero diversity, algorithm DR adds no clauses to φ, regardless of its induced width.

Example 5: Let $\varphi = \{(G \vee E \vee \neg F), (G \vee \neg E \vee D), (\neg A \vee F), (A \vee \neg E), (\neg B \vee C \vee \neg E), (B \vee C \vee D)\}$. It is easy to see that the ordering $o = (A, B, C, D, E, F, G)$ has diversity 0 and induced width 4.

Theorem 6: *Zero-diversity theories are tractable for DR: given a zero-diversity theory φ having n variables and c clauses, 1. its zero-diversity ordering o can be found in $O(n^2 \cdot c)$ time and 2. DR along o takes linear time.* $\square$

The proof follows immediately from Theorem 8 (see subsection 2.4.3).

Zero-diversity theories generalize the notion of causal theories defined for general constraint networks of multi-valued relations [34]. According to this definition, theories are *causal* if there is an ordering of the propositional variables such that each bucket contains a single clause. Consequently, the ordering has zero diversity. Clearly, when a theory has a non-zero diversity, it is still better to place zero-diversity variables last in the ordering, so that they will be processed first. Indeed, the *pure literal rule* of the original Davis-Putnam resolution algorithm requires processing first all-positive and all-negative (namely, zero-diversity) clauses.

```
**min-diversity** $(\varphi)$
1. **For** $i = n$ to 1 do:
        Choose symbol $Q$ having the smallest diversity
        in $\varphi - \bigcup_{j=i+1}^{n} bucket_j$ and put it in the $i^{th}$ position.
```

Figure 2.10: Algorithm *min-diversity*.

However, the parameter of real interest is the diversity of the directional extension $E_o(\varphi)$, rather than the diversity of φ.

Definition 4: (induced diversity)
The *induced diversity of an ordering* o, $div^*(o)$, is the diversity of $E_o(\varphi)$ along o, and the *induced diversity of a theory*, div^*, is the minimal induced diversity over all its orderings.

Since $div^*(o)$ bounds the number of clauses generated in each bucket, the size of $E_o(\varphi)$ for every o can be bounded by $|\varphi| + n \cdot div^*(o)$. The problem is that computing $div^*(o)$ is generally not polynomial (for a given o), except for some restricted cases. One such case is the class of zero-diversity theories mentioned above, where $div^*(o) = div(o) = 0$. Another case, presented below, is a class of theories having $div^* = 1$. Note that we can easily create examples with high w^* having $div^* \leq 1$.

Theorem 7: *Given a theory* φ *defined on variables* $Q_1,\ldots, Q_n$, *such that each symbol* Q_i *either (a) appears only negatively (only positively), or (b) it appears in exactly two clauses, then* $div^*(\varphi) \leq 1$ *and* φ *is tractable.* $\Box$

```
min-width ($\varphi$)
1. Initialize: $G \leftarrow G(\varphi)$
2. For $i = n$ to 1 do
        1.1. Choose symbol $Q$ having the smallest
             degree in $G$ and put it in the $i^{th}$ position.
        1.2. $G \leftarrow G - \{Q\}$.
```

Figure 2.11: Algorithm *min-width*.

```
min-degree ($\varphi$)
1. Initialize: $G \leftarrow G(\varphi)$
2. For $i = n$ to 1 do
        1.1. Choose symbol $Q$ having the smallest
             degree in $G$ and put it in the $i^{th}$ position.
        1.2. Connect the neighbors of $Q$ in $G$.
        1.3. $G \leftarrow G - \{Q\}$.
```

Figure 2.12: Algorithm *min-degree*.

2.4.3 Ordering heuristics

As previously noted, finding a minimum-induced-width ordering is known to be NP-hard [1]. A similar result can be demonstrated for minimum-induced-diversity orderings. However, the corresponding suboptimal (non-induced) min-width and min-diversity heuristic orderings often provide relatively low induced width and induced diversity. Min-width and min-diversity orderings can be computed in polynomial time by a simple greedy algorithm, as shown in Figures 2.10 and 2.11.

Theorem 8: *Algorithm* min-diversity *generates a minimal diversity ordering of a theory in time* $O(n^2 \cdot c)$*, where* n *is the number of variables and* c *is the number of clauses in the input theory.* □

```
max-cardinality ($\varphi$)
1. For $i = 1$ to n do
       Choose symbol $Q$ connected to maximum number of
       previously ordered nodes in $G$ and put it in the $i^{th}$ position.
```

Figure 2.13: Algorithm *max-cardinality*.

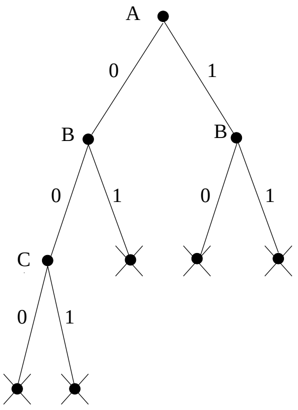
The *min-width* algorithm [22] (Figure 2.11) is similar to the min-diversity, except that at each step we select a variable with the smallest *degree* in the current interaction graph. The selected variable is then placed i-th in the ordering and deleted from the graph.

A modification of min-width ordering, called *min-degree* [41] (Figure 2.12), connects all the neighbors of the selected variable in the current interaction graph before the variable is deleted. Empirical studies demonstrate that the min-degree heuristic usually yields lower-w^* orderings than the induced-width heuristic. In all these heuristics ties are broken randomly.

There are several other commonly used ordering heuristics, such as *max-cardinality* heuristic presented in Figure 2.13. For more details, see [7, 22, 97].

2.5 Backtracking search (DP)

Backtracking search processes the variables in some order, instantiating the next variable if it has a value consistent with previous assignments. If there is no such value (a situation called a *dead-end*), the algorithm *backtracks* to the previous variable and selects an alternative assignment. Should no consistent assignment be found, the algorithm backtracks again. The algorithm explores the *search tree*, in a depth-first manner, until it either finds a solution or concludes that no solution exists. An



```
DP($\varphi$):
Input: A cnf theory $\varphi$.
Output: A decision of whether $\varphi$ is satisfiable.
Unit_propagate($\varphi$);
If the empty clause generated return(false);
else if all variables are assigned return(true);
else
    $Q$ = some unassigned variable;
    return(DP($\varphi \wedge \neg Q$) $\vee$
            DP($\varphi \wedge Q$) )
```

(a) (b)

Figure 2.14: (a) A backtracking search tree along the ordering A, B, C for a cnf theory $\varphi_5 = \{(\neg A \vee B), (\neg C \vee A), \neg B, C\}$ and (b) the Davis-Putnam Procedure.

example of a search tree is shown in Figure 2.14a. This tree is traversed when deciding satisfiability of a propositional theory $\varphi_5 = \{(\neg A \vee B), (\neg C \vee A), \neg B, C\}$. The tree nodes correspond to the variables, while the tree branches correspond to different assignments (0 and 1). Dead-end nodes are crossed out. Theory φ_5 is obviously inconsistent.

There are various advanced backtracking algorithms for solving CSPs that improve the basic scheme using "smart" variable- and value-ordering heuristics ([14], [51]). More efficient backtracking mechanisms, such as *backjumping* [53, 21, 92], constraint propagation (e.g., *arc-consistency*, *forward checking* [59]), or learning (recording constraints) [21, 48, 3] are available. The Davis-Putnam Procedure (DP) [18] shown in Figure 2.14b is a backtracking search algorithm for deciding propositional satisfiability combined with unit propagation. Various branching heuristics augmenting this basic version of DP have been proposed since 1962 [69, 14, 64, 55].

The worst-case time complexity of all backtracking algorithms is exponential in the number of variables while their space complexity is linear. Yet, the average time complexity of DP depends on the distribution of instances [42] and is often much lower then its worst-case bound. Usually, its average performance is affected by rare, but exceptionally hard instances. Exponential-family empirical distributions (e.g., lognormal, Weibull) proposed in recent studies [50, 96] summarize such observations in a concise way. A typical distribution of the number of explored search-tree nodes is shown in Figure 2.15. The distribution is shown for inconsistent problems. As it turns out, consistent and inconsistent CSPs produce different types of distributions (for more details see [50, 51]).

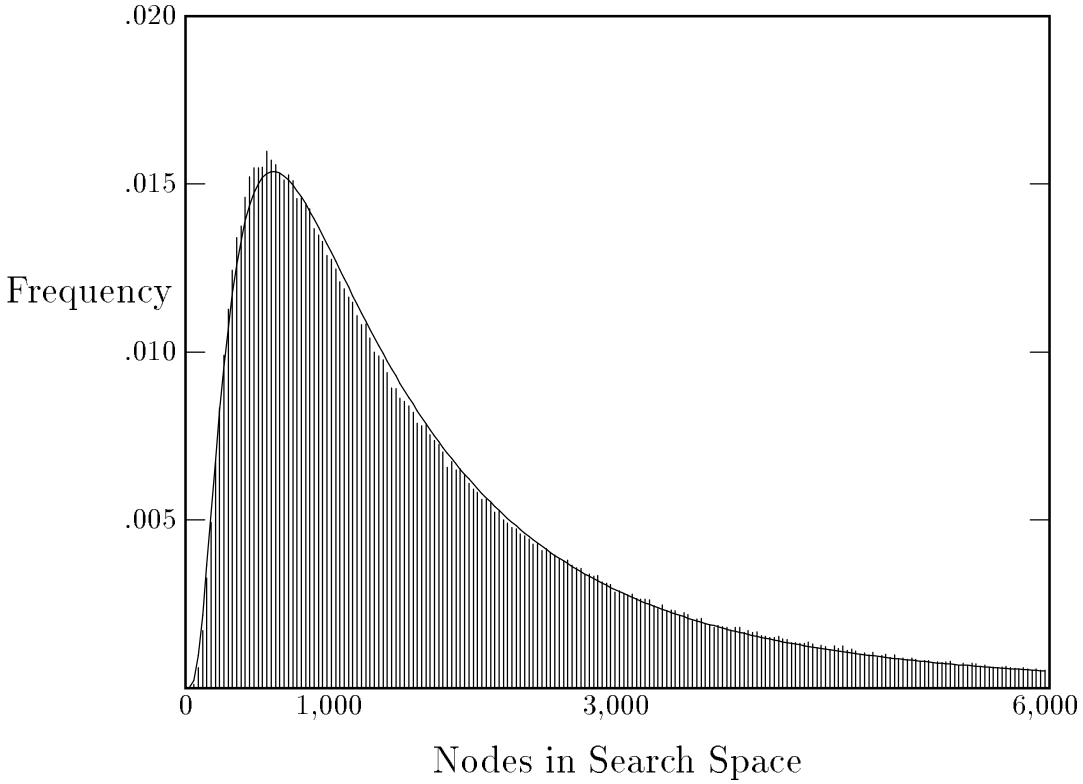


Figure 2.15: An empirical distribution of the number of nodes explored by algorithm BJ-DVO (backjumping+dynamic variable ordering) on 10^6 instances of inconsistent random binary CSPs having N=50 variables, domain size D=6, constraint density C=.1576 (probability of a constraint between two variables), and tightness T=0.333 (the fraction of prohibited value pairs in a constraint).

2.5.1 The proportionate-effect model of backtracking

In this section, we discuss the proportionate-effect model of backtracking [51, 96] that provides additional insights on the characteristics of the runtime distributions and supports the empirical observations. This model is derived under general assumptions which hold for any variable- and value-ordering. The point we are trying to make is that no matter how "smart" ordering heuristics are, it is always possible that backtracking will encounter exceptionally hard problems, although the heuristics can (and should) reduce this possibility.

The performance model is derived for uniform random binary CSP generator that takes four parameters: N, D, T and C. It generates instances with N variables, reach having a domain of size D. The parameter T (tightness) specifies the probability that a value pair in a constraint is disallowed. The parameter C specifies the probability of a binary constraint existing between two variables.

We restrict our attention to the simple backtracking algorithm with a fixed variable ordering $(Y_1, ..., Y_N)$ and to the inconsistent random binary CSPs with the parameters $\langle N, D, T, C\rangle$. We show that the number of nodes on level i of the search tree explored by backtracking is distributed lognormally when i is sufficiently large.

A common method for deriving the lognormal distribution uses the *law of proportionate effect* [16]: if the growth rate of a variable at each step in a process is randomly proportion to its size at that step, then the size of the variable at time n will be approximately lognormally distributed. Formally, if the value of a random variable at time i is X_i, and

$$X_i = X_{i-1} \times b_i, \tag{2.1}$$

where $(b_1, b_2, \ldots, b_n)$ are positive independent random variables, then the distribution of X_i is, for large enough i, lognormally distributed. The law of proportionate effect follows from the central limit theorem, since 2.1 implies

$$log(X_n) = \sum_i^n log(b_i), \tag{2.2}$$

and the sum of independent random variables $log(b_i)$ converges to the normal distribution.

Let X_i be the number of nodes explored at i-th level of the search tree. The branching factor b_i at i-th level of the tree is defined as X_i/X_{i-1}, where for $2 \leq i \leq n$, and $b_1 = D$ (D is the domain size). For $i > 1$, b_i is randomly distributed in $[0, D]$ and specifies how many values of variable Y_{i-1} are consistent with the previous assignment. The probability of a value k for Y_{i-1} being consistent with the assignment to $Y_1, ..., Y_{i-2}$ is

$$p_i = (1 - CT)^{i-2},$$

where C is the probability of a constraint between Y_{i-1} and a previous variable, and T is the probability of a value pair to be prohibited by that constraint. Therefore, the branching factor b_i is distributed binomially with parameter p_i. On each level i, b_i is independent of previous b_j, $j < i$. Note that b_i are non-negative (positive for all levels except the deepest level in the tree reached by backtracking) and can be greater than or less than 1 (since b_i can be zero, the low of proportionate effect is not entirely applicable for some deep levels of the search tree). Then

$$X_i = b_1 \times b_2 \times \ldots \times b_i,$$

is lognormally distributed by the law of proportionate effect.

This derivation applies to the distribution of nodes on each particular level i, where i is large enough. It still remains to be shown how this analysis relates to the distribution of the total number of nodes explored in a tree. In a complete search tree, the total number of nodes $\sum_{i=0}^{N} D^i = (D^{N+1} - 1)/(D - 1) \approx D^N \frac{D}{D-1}$ is proportional to the number of nodes at the deepest level, D^N. A similar relation may be possible to derive for a backtracking search tree.

Satisfiable CSPs do not fit this scheme since the tree traversal is interrupted when a solution is found. Indeed, empirical results reported in [50] point to substantial difference in behavior of satisfiable and unsatisfiable problems. In addition, the model is derived for binary CSPs rather than for SAT, although empirical results suggest

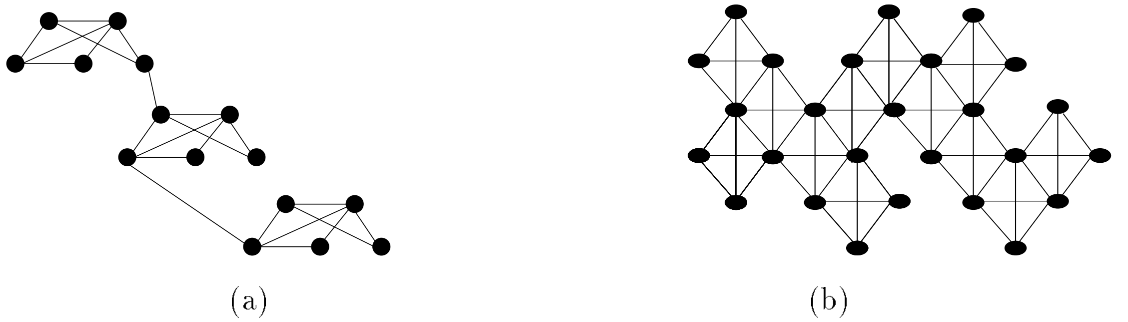

(a) (b)

Figure 2.16: An example of a theory with (a) a chain structure (3 subtheories, 5 variables in each) and (b) a (k,m)-tree structure (k=2, m=2).

the same distribution families in both cases [50, 96]. This model provides a general insight on what type of behavior is expected from backtracking algorithms.

2.6 DP versus DR: empirical evaluation

In this section we present an empirical comparison of DP and DR on different types of cnf theories, including uniform random problems, random chains and (k,m)-trees, and benchmark problems from the Second DIMACS Challenge [3]. The algorithms were implemented in C and tested on SUN Sparc stations. Since we used several machines having different performance (from Sun 4/20 to Sparc Ultra-2), we specify which machine was used for each set of experiments. Reported runtime is measured in seconds.

Algorithm DR is implemented as discussed in Section 2.3. If it is followed by DP using the same fixed variable ordering, no dead-ends will occur (see Theorem 2). Algorithm DP was implemented using the dynamic variable ordering heuristic of *Tableau* [14], a state-of-the-art backtracking algorithm for SAT. This heuristic, called the *2-literal-clause* heuristic, suggests instantiating next a variable that would cause the largest number of unit propagations approximated by the number of 2-literal clauses in which the variable appears. The augmented algorithm significantly

---

[3] Available at *ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/volume/cnf.*

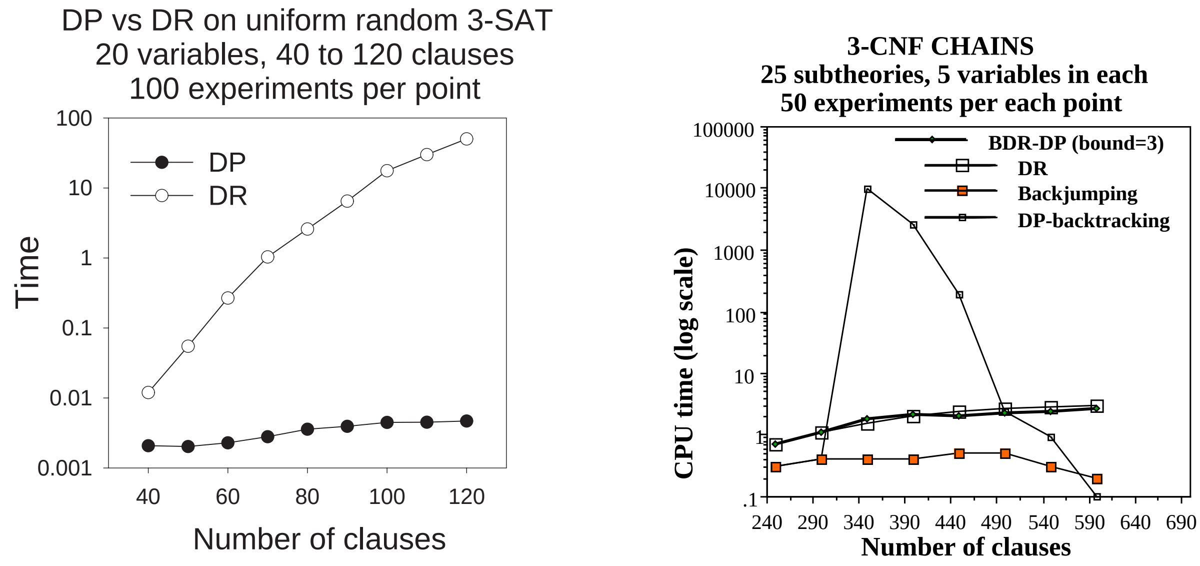
outperforms DP without this heuristic [14].

2.6.1 Random problem generators

To test the algorithms on problems with different structures, several random problem generators were used. The *uniform k-cnfs* generator [82] uses as input the number of variables N, the number of clauses C, and the number of literals per clause k. Each clause is generated by randomly choosing k out of N variables and by determining the sign of each literal (positive or negative) with probability p. In the majority of our experiments $p = 0.5$. Although we did not check for clause uniqueness, for large N it is unlikely that identical clauses will be generated.

Our second generator, *chains*, creates a sequence of independent uniform k-cnf theories (called *subtheories*) and connects each pair of successive cliques by a 2-cnf clause containing variables from two consecutive subtheories in the chain (see Figure 2.16a). The parameters of the generator are the number of cliques, $Ncliq$, the number of variables per clique, N, and the number of clauses per clique, C. A chain of cliques, each having N variables, is a subgraph of a k-tree [1] where $k = 2N-1$ and therefore, has $w^* \leq 2N-1$.

We also used a *(k,m)-tree* generator which generates a tree of cliques each having $(k+m)$ nodes where k is the size of the intersection between two neighboring cliques (see Figure 2.16b, where $k = 2$ and $m = 2$). Given k, m, the number of cliques $Ncliq$, and the number of clauses per clique $Ncls$, the *(k,m)-tree* generator produces a clique of size $k+m$ with $Ncls$ clauses and then generates each of the other $Ncliq-1$ cliques by selecting randomly an existing clique and its k variables, adding m new variables, and generating $Ncls$ clauses on that new clique. Since a k-m-tree can be embedded into a $(k+m-1)$-tree, its induced width is bounded by $k+m-1$ (note that $(k,1)$-trees are conventional k-trees).



(a) uniform random 3-cnfs, $w^* = 10$ to 18

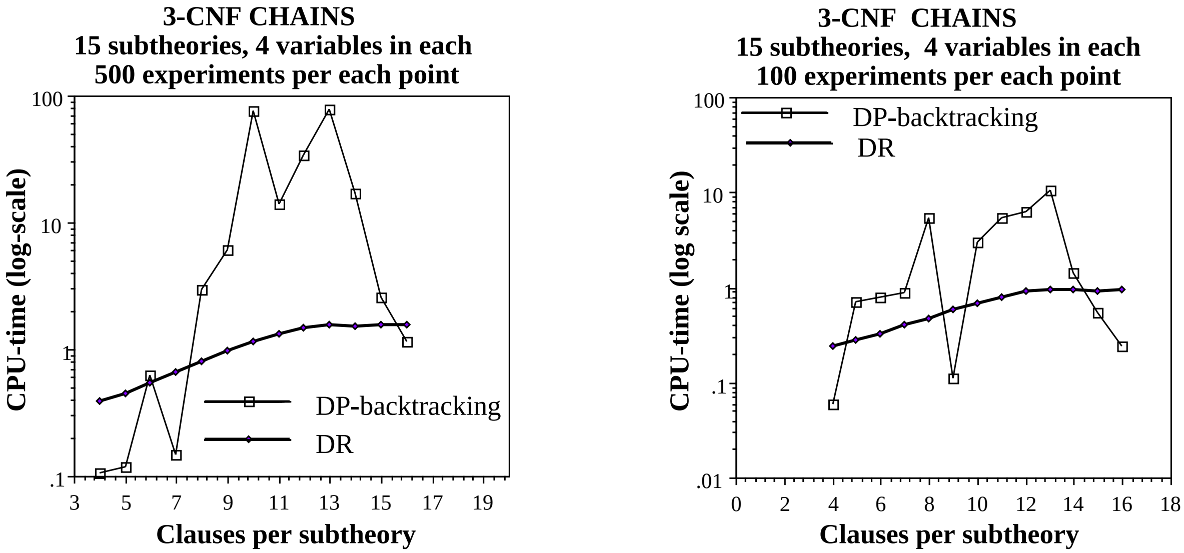
(b) chain 3-cnfs, $w^* = 4$ to 7

Figure 2.17: (a) DP versus DR on uniform random 3-cnfs; (b) DP, DR, BDR-DP(3) and backjumping on 3-cnf chains (Sun 4/20).

2.6.2 Results

As expected, on uniform random 3-cnfs having large w^*, the complexity of DR grew exponentially with the problem density while the performance of DP was much better. Even small problems having 20 variables already demonstrate the exponential behavior of DR (see Figure 2.17a). On larger problems DR often ran out of memory. We did not proceed with more extensive experiments in this case, since the exponential behavior of DR on uniform 3-cnfs is already well-known [52, 57].

However, the behavior of the algorithms on chain problems was completely different. DR was by far more efficient than DP, as can be seen from Table 2.1 and from Figure 2.17b, summarizing the results on 3-cnf chain problems that contain 25 subtheories, each having 5 variables and 9 to 23 clauses (24 additional 2-cnf clauses



(a) input ordering

(b) min-width ordering

Figure 2.18: DR and DP on 3-cnf chains with different orderings (Sun 4/20).

Table 2.1: DR versus DP on 3-cnf chains having 25 subtheories, 5 variables in each, and from 11 to 21 clauses per subtheory (total 125 variables and 299 to 549 clauses). 20 instances per row. The columns show the percentage of satisfiable instances, time and deadends for DP, time and the number of new clauses for DR, the size of largest clause, and the induced width w^*_{md} along the min-diversity ordering. The experiments were performed on Sun 4/20 workstation.

Num of cls	% sat	**DP**		**DR**			
		Time	Dead ends	Time	Number of new clauses	Size of max clause	w^*
299	100	0.4	1	1.4	105	4.1	5.3
349	70	9945.7	908861	2.2	131	4.0	5.3
399	25	2551.1	207896	2.8	131	4.0	5.3
449	15	185.2	13248	3.7	135	4.0	5.5
499	0	2.4	160	3.8	116	3.9	5.4
549	0	0.9	9	4.0	99	3.9	5.2

Table 2.2: DR and DP on hard chains when the number of dead-ends is larger than 5,000. Each chain has 25 subtheories, with 5 variables in each (total of 125 variables). The experiments were performed on Sun 4/20 workstation.

Num of cls	Sat: 0 or 1	**DP**		**DR**
		Time	Dead ends	Time
349	0	41163.8	3779913	1.5
349	0	102615.3	9285160	2.4
349	0	55058.5	5105541	1.9
399	0	74.8	6053	3.6
399	0	87.7	7433	3.1
399	0	149.3	12301	3.1
399	0	37903.3	3079997	3.0
399	0	11877.6	975170	2.2
399	0	841.8	70057	2.9
449	1	655.5	47113	5.2
449	0	2549.2	181504	3.0
449	0	289.7	21246	3.5

Table 2.3: Histograms of the number of deadends (log-scale) for DP on chains having 20, 25 and 30 subtheories, each defined on 5 variables and 12 to 16 clauses. Each column presents results for 200 instances; each row defines a range of deadends; each entry is the frequency of instances (out of total 200) that yield the range of deadends. The experiments were performed on Sun Ultra-2.

Deadends	C=12			C=14			C=16		
	Ncliq			Ncliq			Ncliq		
	20	25	30	20	25	30	20	25	30
$[0, 1)$	103	90	75	75	23	8	7	2	2
$[1, 10)$	81	85	102	102	107	93	73	68	59
$[10, 10^2)$	3	4	7	7	21	24	40	37	43
$[10^2, 10^3)$	2	1	4	4	8	12	20	26	22
$[10^3, 10^4)$	1	3	2	2	10	8	21	10	21
$[10^4, \infty)$	10	17	10	10	31	55	39	57	53

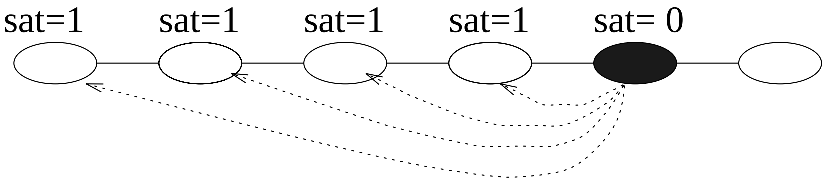


Figure 2.19: An inconsistent chain problem: a naive backtracking is very inefficient when encountering an inconsistent subproblem at the end of the variable ordering.

connect the subtheories in the chain) [4]. A min-diversity ordering was used for each instance. Since the induced width of these problems was small (less than 6, on average), directional resolution solved these problems quite easily. However, DP-backtracking encountered rare but extremely hard problems that contributed to its average complexity. Table 2.2 lists the results on selected hard instances from Table 2.1 (where the number of dead-ends exceeds 5,000).

Similar results were obtained for other chain problems and with different variable orderings. For example, Figure 2.18 graphs the experiments with min-width and input orderings. We observe that min-width ordering may significantly improve the performance of DP relative to the input ordering (compare Figure 2.18a and Figure 2.18b). Still, it did not prevent backtracking from encountering rare, but extremely hard instances.

Table 2.3 presents the histograms demonstrating the performance of DP on chains in more details. The histograms show that in most cases the frequency of easy problems (e.g., less than 10 deadends) decreased and the frequency of hard problems (e.g., more than 10^4 deadends) increased with increasing number of cliques and with increasing number of clauses per clique. Further empirical studies are necessary to investigate phase transition in chains[5].

---

[4] Figure 2.17b also shows the results for algorithms BDR-DP and backjumping discussed later.

[5] The phase transition phenomenon observed in uniform random 3cnf and in CSPs corresponds to a sharp transition from mostly satisfiable to mostly unsatisfiable problems around a particular value of a critical parameter, such as clauses/variables ratio, and is usually associated with a sharp peak in the algorithm's complexity [10, 82, 56, 110, 65].

Table 2.4: DP versus Tableau on 150- and 200-variable uniform random 3-cnfs using the min-degree ordering. 100 instances per row. Experiments ran on Sun Sparc Ultra-2.

Cls	% sat	Tableau time	DP time	DP de
150 variables				
550	1.00	0.3	0.4	81
600	0.93	2.0	3.9	992
650	0.28	4.1	10.1	2439
700	0.04	2.7	7.1	1631
200 variables				
780	0.99	11.6	10.0	1836
820	0.95	48.5	43.7	7742
860	0.40	81.7	125.8	22729
900	0.07	26.6	92.4	17111

In our experiments nearly all of the 3-cnf chain problems that were difficult for DP were unsatisfiable. One plausible explanation is that inconsistent chain theories may have an unsatisfiable subtheory only at the end of the ordering. If all other subtheories are satisfiable then DP will try to re-instantiate variables from the satisfiable subtheories whenever it encounters a dead-end. Figure 2.19 shows an example of a chain of satisfiable theories with an unsatisfiable theory close to the end of the ordering. Min-diversity and min-width orderings do not preclude such a situation. There are enhanced backtracking schemes, such as *backjumping* [53, 54, 21, 91], that are capable of exploiting the structure and preventing useless re-instantiations. Experiments with backjumping confirm that it substantially outperforms DP on the same chain instances (see Figure 2.17b).

The behavior of DP and DR on *(k-m)*-trees is similar to that on chains and will be discussed later in the context of hybrid algorithms.

Table 2.5: Histograms of DP and Tableau runtimes (log-scale) on chains having $Ncliq = 15$, $N = 8$, and C from 21 to 27, 200 instances per column. Each row defines a runtime range, and each entry is the frequency of instances within the range. The experiments were performed on Sun Ultra-2.

Time	C=21	C=23	C=25
Tableau runtime histogram			
$[0, 1)$	195	189	166
$[1, 10)$	0	2	12
$[10, 10^2)$	0	3	14
$[10^2, \infty)$	5	6	8
DP runtime histogram			
$[0, 1)$	193	180	150
$[1, 10)$	2	3	8
$[10, 10^2)$	2	2	11
$[10^2, \infty)$	3	15	31

Comparing different DP implementations

One may raise the question whether our (not highly optimized) DP implementation is efficient enough to be representative of backtracking-based SAT algorithms. We answer this question by comparing our DP with the executable code of Tableau [14].

The results for 150- and 200-variable uniform random 3-cnf problems are presented in Table 2.4. We used min-degree as an initial ordering consulted by both (dynamic-ordering) algorithms Tableau and DP in tie-breaking situations. In most cases, Tableau was 2-4 times faster than DP, while in some DP was faster or comparable to Tableau.

On chains, the behavior pattern of Tableau was similar to that of DP. Table 2.5 compares the runtime histograms for DP and Tableau on chain problems showing that both algorithms were encountering rare hard problems, although Tableau usually encountered hard problems less frequently than DP. Some problem instances that were hard for DP were easy for Tableau, and vice versa.

Therefore, although Tableau is often more efficient than our implementation, this difference does not change the key distinctions made between backtracking- and resolution-based approaches. Most of experiments in this chapter use our implementation of DP [6].

2.7 Combining search and resolution

The complementary properties of DP and DR suggest combining both into a hybrid scheme (note that algorithm DP already includes a limited amount of resolution in the form of unit propagation). We will present two general parameterized schemes integrating bounded resolution with search. The hybrid scheme BDR-DP(i) performs bounded resolution prior to search, while the other scheme called DCDR(b) uses it dynamically during search.

2.7.1 Algorithm BDR-DP(i)

The resolution operation helps detecting inconsistent subproblems and thus can prevent DP from unnecessary backtracking. Yet, resolution can be costly. One way of limiting the complexity of resolution is to bound the size of the recorded resolvents. This yields the incomplete algorithm *bounded directional resolution*, or $BDR(i)$, presented in Figure 2.20, where i bounds the number of variables in a resolvent. The algorithm coincides with DR except that resolvents with more than i variables are not recorded. This bounds the size of the directional extension $E_o^i(\varphi)$ and, therefore, the complexity of the algorithm. The time and space complexity of BDR(i) is $O(n \cdot exp(i))$. The algorithm is sound but incomplete. Algorithm $BDR(i)$ followed by DP is named *BDR-DP(i)* [7]. Clearly, BDR-DP(0) coincides with DP while for $i > w_o^*$ BDR-DP(i) coincides with DR (each resolvent is recorded).

---

[6] Having the source code for DP allowed us more control over the experiments (e.g., bounding the number of deadends) than having only the executable code for Tableau.

[7] Note that DP always uses the 2-literal-clauses dynamic variable ordering heuristic.

```
Bounded Directional Resolution: BDR($i$)
Input: A $cnf$ theory $\varphi$, $o = Q_1, ..., Q_n$, and bound $i$.
Output: The decision of whether $\varphi$ is satisfiable.
If it is, a bounded directional extension $E_o^i(\varphi)$.
1. Initialize: generate a partition of clauses, $bucket_1, ..., bucket_n$,
where $bucket_i$ contains all the clauses whose highest literal is $Q_i$.
2. For $i = n$ to 1 do:
        resolve each pair $\{(\alpha \vee Q_i), (\beta \vee \neg Q_i)\} \subseteq bucket_i$.
        If $\gamma = \alpha \vee \beta$ is empty, return "$\varphi$ is unsatisfiable"
        else if $\gamma$ contains no more than $i$ propositions,
        add $\gamma$ to the bucket of its highest variable.
3.Return $E_o^i(\varphi) = \bigcup_i bucket_i$.
```

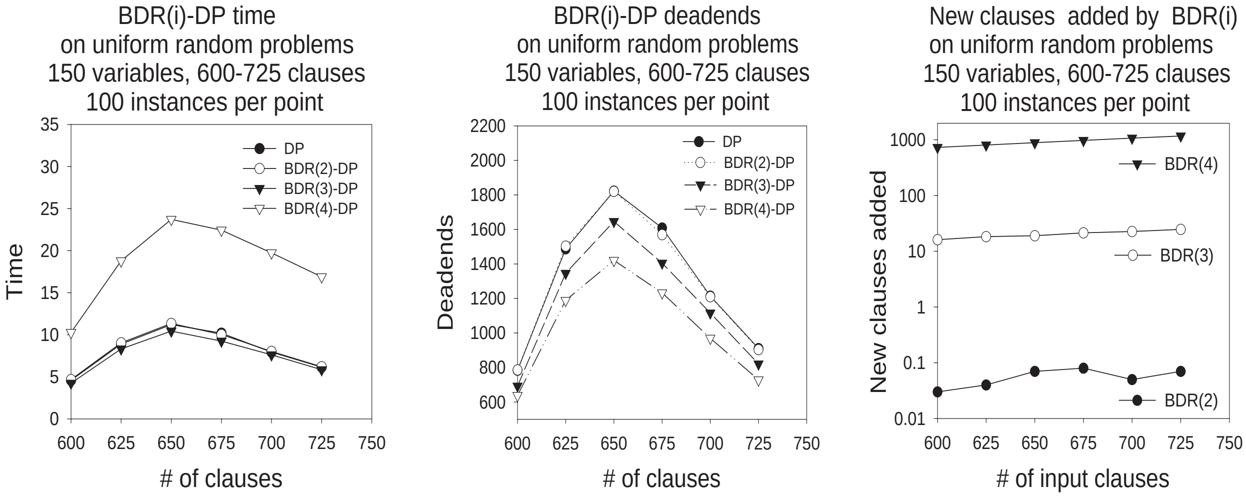
Figure 2.20: Algorithm Bounded Directional Resolution (BDR).

2.7.2 Empirical evaluation of BDR-DP(i)

We tested BDR-DP(i) for different values of i on uniform 3-cnfs, chains, (k,m)-trees, and on DIMACS benchmarks. In most cases, BDR-DP(i) achieved its optimal performance for intermediate values of i.

Table 2.6: DP versus BDR-DP(i) for $2 \leq i \leq 4$ on uniform random 3-cnfs with 150 variables, 600 to 725 clauses, and positive literal probability $p = 0.5$. The induced width w_o^* along the min-width ordering varies from 107 to 122. Each row presents average values on 100 instances (Sun Sparc 4).

Num of cls	**DP**		**BDR-DP(2)**				**BDR-DP(3)**				**BDR-DP(4)**				w_o^*
	Time	Dead ends	BDR time	DP time	Dead ends	New cls	BDR time	DP time	Dead ends	New cls	BDR time	DP time	Dead ends	New cls	
600	4.6	784	0	4.6	786	0	0.1	4.1	692	16	1.7	8.5	638	731	113
625	8.9	1487	0	8.9	1503	0	0.1	8.2	1346	18	1.9	16.8	1188	805	114
650	11.2	1822	0.1	11.2	1821	0	0.1	10.3	1646	19	2.3	21.4	1421	889	115
675	10.2	1609	0.1	9.9	1570	0	0.1	9.1	1405	21	2.6	19.7	1232	975	116
700	7.9	1214	0.1	7.9	1210	0	0.1	7.5	1116	23	3	16.6	969	1071	117
725	6.1	910	0.1	6.1	904	0	0.1	5.7	820	25	3.5	13.3	728	1169	118



(a) time (b) deadends (c) new clauses

Figure 2.21: BDR-DP(i) on a class of uniform random 3-cnf problems. (150 variables, 600 to 725 clauses). The induced width along the min-width ordering varies from 107 to 122. Each data point corresponds to 100 instances. Note that the plots for DP and BDR(2)-DP in (a) and (b) almost coincide (the white-circle plot for BDR(2)-DP overlaps with the black-circle plot for DP).

Performance on uniform 3-cnfs

These results for BDR-DP(i) ($0 \leq i \leq 4$) on a class of uniform random 3-cnfs are presented in Table 2.6. It shows the average time and number of deadends for DP, the average BDR(i) time, DP time and the number of deadends after preprocessing, as well as the average number of new clauses added by BDR(i). An alternative summary of the same data is given in Figure 2.21, comparing DP and BDR-DP(i) time. It also demonstrates the increase in the number of clauses and the corresponding reduction in the number of deadends. For $i = 2$, almost no new clauses are generated (Figure 2.21c). Indeed, the graphs for DP and BDR-DP(2) practically coincide. Incrementing i by 1 results in a two orders of magnitude increase in the number of generated clauses, while the number of deadends decreases by 100-200, as shown in Figure 2.21c.

The results suggest that BDR-DP(3) is the most cost-effective on these problem

Table 2.7: DP versus BDR-DP(i) for $i = 3$ and $i = 4$ on uniform 3-cnfs with 200 variables, 900 to 1400 clauses, and with positive literal probability $p = 0.7$. Each row presents mean values on 20 experiments.

Num of cls	**DP**		**BDR-DP(3)**				**BDR-DP(4)**			
	Time	Dead ends	BDR time	DP time	Dead ends	New cls	BDR time	DP time	Dead ends	New cls
900	1.1	0	0.3	1.1	0	11	8.4	1.7	1	657
1000	2.7	48	0.4	1.6	14	12	13.1	2.7	21	888
1100	8.8	199	0.6	27.7	685	18	20.0	50.4	729	1184
1200	160.2	3688	0.8	141.5	3271	23	28.6	225.7	2711	1512
1300	235.3	5027	1.0	219.1	4682	28	39.7	374.4	4000	1895
1400	155.0	3040	1.2	142.9	2783	34	54.4	259.0	2330	2332

Table 2.8: DP versus BDR-DP(3) on uniform random 3-cnfs with $p = 0.5$ at the phase-transition point (C/N=4.3): 150 variables and 645 clauses, 200 variables and 860 clauses, 250 variables and 1075 clauses. The induced width w_o^* was computed for the min-width ordering. The results in the first two rows summarize 100 experiments, while the last row represents 40 experiments.

$< vars, cls >$	**DP**		**BDR-DP(3)**				w_o^*
	Time	Dead ends	BDR time	DP time	Dead ends	New cls	
$< 150, 650 >$	11.2	1822	0.1	10.3	1646	19	115
$< 200, 860 >$	81.3	15784	0.1	72.9	14225	18	190
$< 250, 1075 >$	750	115181	0.1	668.8	102445	19	1094

classes (see Figure 2.21a). It is slightly faster than DP and BDR-DP(2) (BDR-DP(2) coincides with DP on this problem set) and significantly faster than BDR-DP(4). Table 2.6 shows that BDR(3) takes only 0.1 second on average, while BDR(4) takes up to 3.5 seconds and indeed generates many more clauses. Observe also that DP is slightly faster when applied after BDR(3). Interestingly, for $i = 4$ the time of DP almost doubles although fewer deadends are encountered. For example, in Table 2.6, for the problem set with 650 clauses, DP takes on average 11.2 seconds but after preprocessing by BDR(4) it takes 21.4 seconds. This can be explained by the significant increase in the number of clauses that need to be consulted by DP. Thus, as i increases beyond 3, DP's performance is likely to worsen while at the same time the complexity of preprocessing grows exponentially in i. Table 2.7 presents additional results for problems having 200 variables where $p = 0.7$ [8].

Finally, we observe that the effect of BDR(3) is proportional to the theory size. In Table 2.8 we compare the results for three classes of uniform 3-cnf problems in the phase transition region. While this improvement was marginal for 150-variable problems (from 11.2 seconds for DP to 10.3 seconds for BDR-DP(3)), it was more pronounced on 200-variable problems (from 81.3 to 72.9 seconds), and on 250-variable problems (from 929.9 to 830.5 seconds). In all those cases the average speed-up is about 10%.

Our tentative empirical conclusion is that $i = 3$ is the optimal parameter for BDR-DP(i) on uniform random 3-cnfs.

Performance on chains and (k,m)-trees

The experiments with chains showed that BDR-DP(3) easily solved almost all instances that were hard for DP. In fact, the performance of BDR-DP(3) on chains was comparable to that of DR and backjumping (see Figure 2.17b).

---

[8] Note that the average decrease in the number of deadends is not always monotonic: for problems having 1000 clauses, DP has an average of 48 deadends, BDR-DP(3) yields 14 deadends, but BDR-DP(4) yields 21 deadends. This may occur because DP uses dynamic variable ordering.

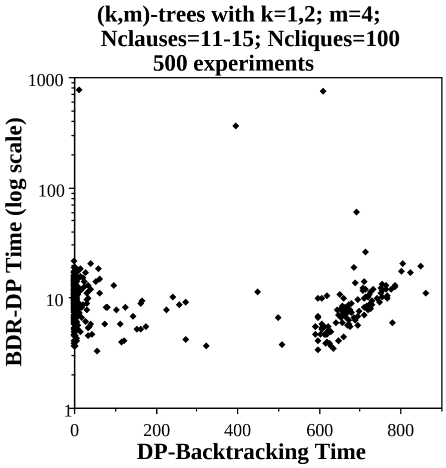
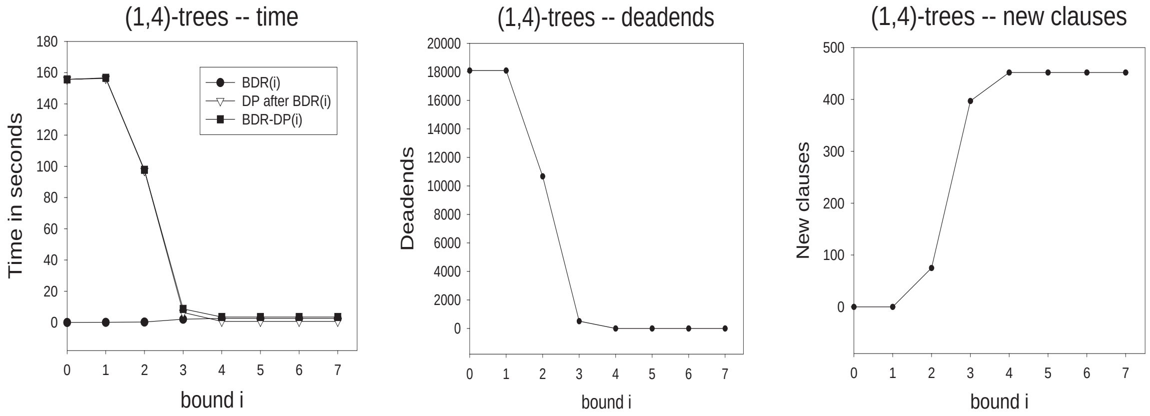


Figure 2.22: DP and BDR-DP(3) on (k-m)-trees, k=1,2, m=4, Ncliq=100, and Ncls=11 to 15. 50 instances per each set of parameters (total of 500 instances), an instance per point.

Table 2.9: BDR-DP(3) and DP (termination at 20,000 dead ends) on (k, m)-trees, k=1,2, m=4, Ncliq=100, and Ncls=11 to 14. 50 experiments per each row.

		DP		**BDR-DP(3)**			
Number of	% sat	Time	Dead ends	BDR(3) time	DP after BDR(3) time	DP after BDR(3) dead ends	Number of new clauses
(1,4)-tree, *Ncls* = 11 to 14, *Ncliq* = 100 (total: 401 vars, 1100-1400 cls)							
1100	60	233.2	7475	5.4	17.7	2	298
1200	18	352.5	10547	7.5	1.2	7	316
1300	2	328.8	9182	9.8	0.25	3	339
1400	0	174.2	4551	11.9	0.0	0	329
(2,4)-tree, *Ncls* = 11 to 14, *Ncliq* = 100 (total: 402 vars, 1100-1400 cls)							
1100	36	193.7	6111	4.1	23.8	568	290
1200	12	160.0	4633	6.0	1.6	25	341
1300	2	95.1	2589	8.4	0.1	0	390
1400	0	20.1	505	10.3	0.0	0	403



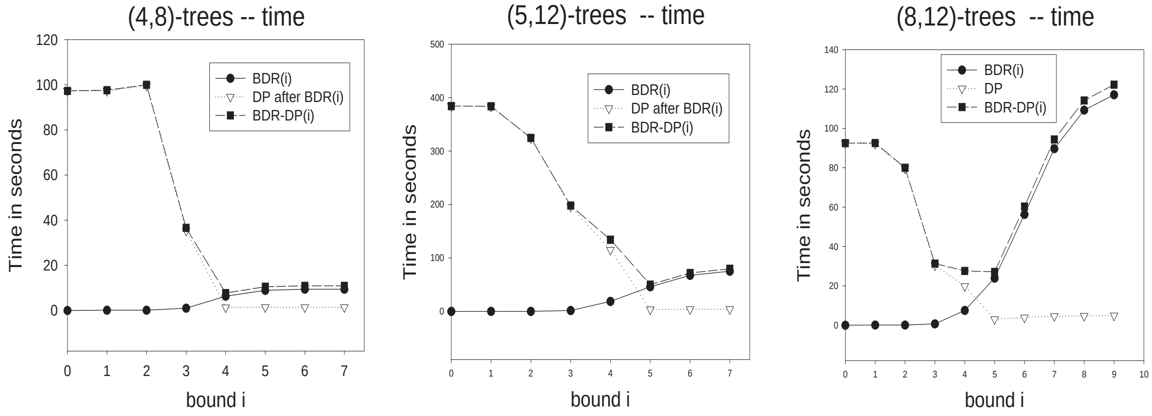
(a) time (b) deadends (c) new clauses

Figure 2.23: BDR-DP(i) on 100 instances of (1,4)-trees, $Ncliq = 100$, $Ncls = 11$, $w^*_{md} = 4$ (termination at 50,000 deadends). (a) Average time, (b) the number of deadends, and (c) the number of new clauses are plotted as functions of the parameter i. Note that the plot for BDR-DP(i) practically coincides with the plot for DP when $i \leq 3$, and with DP when $i > 3$.

Experimenting with (k, m)-trees, while varying the number of clauses per clique, we discovered again exceptionally hard problems for DP. The results on (1,4)-trees and on (2,4)-trees are presented in Table 2.9. In these experiments we terminated DP once it exceeded 20,000 dead-ends (around 700 seconds). This happened in 40% of (1,4)-trees with $Ncls = 13$, and in 20% of (2,4)-trees with $Ncls = 12$. density). Figure 2.22 shows a scatter diagram comparing DP and BDR-DP(3) time on the same data set together with an additional 100 experiments on (k,m)-trees having 15 cliques (total of 500 instances).

As in the case of 3-cnf chains we observed that the majority of the exceptionally hard problems were unsatisfiable. For fixed m, when k is small and the number of cliques is large, hard instances for DP appeared more frequently.

The behavior of BDR-DP(i) as a function of i on structured bounded-w^* theories is demonstrated in Figures 2.23 and 2.24. In these experiments we used min-degree



(a) (4,8)-trees, $w^*_{md} = 9$ (b) (5,12)-trees, $w^*_{md} = 12$ (c) (8,12)-trees, $w^*_{md} = 14$

Figure 2.24: BDR-DP(i) on 3 classes of (k,m)-tree problems: (a) (4,8)-trees, $Ncliq = 60$, $Ncls = 23$, $w^*_{md} = 9$, (b) (5,12)-trees, $Ncliq = 60$, $Ncls = 36$, $w^* = 12$, and (c) (8,12)-trees, $Ncliq = 50$, $Ncls = 34$, $w^* = 14$ (termination at 50,000 deadends). 100 instances per each problem class. Average time, the number of dead-ends, and the number of new clauses are plotted as functions of the parameter i.

ordering that yielded smaller average w^* (denoted w^*_{md}) than input ordering, min-width ordering, and min-cardinality ordering. Appendix A compares the results for all four orderings in Tables 1-3.

Figure 2.23 shows results for (1,4)-trees, while Figure 2.24a presents the results for (4,8)-trees, (5,12)-trees, and (8,12)-trees. Each point represents an average over 100 instances. We observed that for relatively low-w^* (1,4)-trees preprocessing time is not increasing when $i > 3$ since BDR(4) coincides with DR (Figure 2.23a), while for high-w^* (8,12)-trees the preprocessing time grows quickly with increasing i (Figure 2.23c). Since DP time after BDR(i) usually decreases monotonically with i, the total time of BDR-DP(i) is optimal for some intermediate values of i. We observe that for (1,4)-trees, BDR-DP(3) is most efficient, while for (4,8)-trees and for (5,12)-trees the optimal parameters are $i = 4$ and $i = 5$, respectively. For (8,12)-trees, the values

Table 2.10: Tableau, DP, DR, and BDR-DP(i) for i=3 and 4 on the Second DIMACS Challenge benchmarks. The experiments were performed on Sun Sparc 5 workstation.

Problem	**Tableau** time	**DP** time	Dead ends	**DR** time	**BDR-DP(3)**			**BDR-DP(4)**			w^*
					time	Dead ends	New cls	time	Dead ends	New cls	
aim-100-2_0-no-1	2148	>8988	$> 10^8$	*	0.9	5	26	0.60	0	721	54
dubois20	270	3589	3145727	0.2	349	262143	30	0.2	0	360	4
dubois21	559	7531	6291455	0.2	1379	1048575	20	0.2	0	390	4
ssa0432-003	12	45	4787	4	132	8749	950	40	1902	1551	19
bf0432-007	489	8688	454365	*	46370	677083	10084	*	*	*	131

$i = 3, 4$, and 5 provide the best performance.

BDR-DP(i), DP, DR, and Tableau on DIMACS benchmarks

We tested DP, Tableau, DR and BDR-DP(i) for i=3 and i=4 on the benchmark problems from the Second DIMACS Challenge. The results presented in Table 2.10 are quite interesting: while all benchmark problems were relatively hard for both DP and Tableau, some of them had very low w^* and were solved by DR in less than a second (e.g., *dubois20* and *dubois21*). On the other hand, problems having high induced width, such as *aim-100-2_0-no-1* ($w^* = 54$) and *bf0432-007* ($w^* = 131$) were intractable for DR, as expected. Algorithm BDR-DP(i) was often better than both "pure" DP and DR. For example, solving the benchmark *aim-100-2_0-no-1* took more than 2000 seconds for Tableau, more than 8000 seconds for DP, and DR ran out of memory, while BDR-DP(3) took only 0.9 seconds and reduced the number of DP deadends from more than 10^8 to 5. Moreover, preprocessing by BDR(4), which took only 0.6 seconds, made the problem backtrack-free. Note that the induced width of this problem is relatively high ($w^* = 54$). Interestingly, for some DIMACS problems (e.g., *ssa0432-003* and *bf0432-007*) preprocessing by BDR(3) actually worsened the performance of DP. Similar phenomenon was observed in some rare cases for (k,m)-trees (Figure 2.22).

In summary, *BDR-DP(i)* with intermediate values of i is overall more cost-effective than both DP and DR. On unstructured random uniform 3-cnfs BDR-DP(3) is comparable to DP, on low-w^* chains it is comparable to DR, and on intermediate-w^* (k,m)-trees, BDR-DP(i) for $i = 3, 4, 5$ outperforms both DR and DP. We believe that the transition from i=3 to i=4 on uniform problems is too sharp, and that intermediate levels of preprocessing may provide a more refined trade-off.

2.7.3 Algorithm DCDR(b)

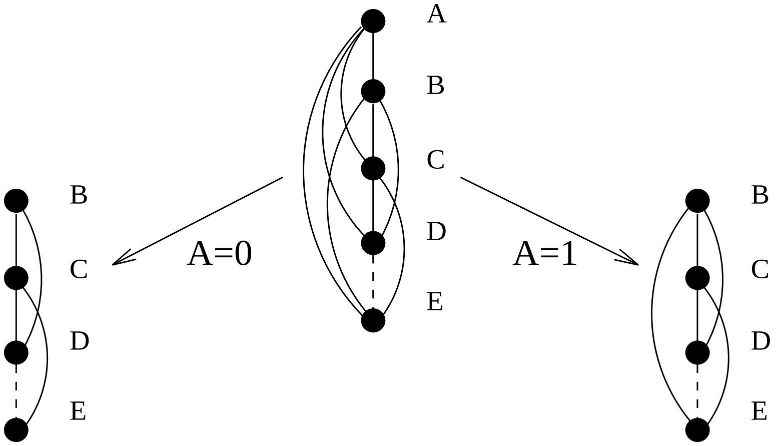


Figure 2.25: The effect of conditioning on A on the interaction graph of theory $\varphi = \{(\neg C \vee E), (A \vee B \vee C), (\neg A \vee B \vee E), (\neg B \vee C \vee D)\}$.

The second method of combining DP and DR that we consider uses resolution dynamically during search. We propose a class of hybrid algorithms that select a set of *conditioning* variables (also called a *cutset*), such that instantiating those variables results in a low-width theory tractable for DR [9]. The hybrids run DP on the cutset variables and DR on the remaining ones, thus combining the virtues of both approaches. Like DR, they exploit low-w^* structure and produce an output theory that facilitates model generation, while using less space and allowing less average time, like DP.

---

[9] This is a generalization of the cycle-cutset algorithm proposed in [33] which transforms the interaction graph of a theory into a tree.

The description of the hybrid algorithms uses a new notation introduced below. An instantiation of a set of variables $C \subseteq X$ is denoted $I(C)$. The theory φ conditioned on the assignment $I(C)$ is called a *conditional theory* of φ relative to $I(C)$, and is denoted as $\varphi_{I(C)}$. The effect of conditioning on C is deletion of variables in C from the interaction graph. Therefore the *conditional interaction graph* of φ with respect to $I(C)$, denoted $G(\varphi_{I(C)})$, is obtained from the interaction graph of φ by deleting the nodes in C (and all their incident edges). The *conditional width* and *conditional induced width* of a theory φ relative to $I(C)$, denoted $w_{I(C)}$ and $w^*_{I(C)}$, respectively, are the width and induced width of the interaction graph $G(\varphi_{I(C)})$.

For example, Figure 2.25 shows the interaction graph of theory $\varphi = \{(\neg C \vee E)$, $(A \vee B \vee C)$, $(\neg A \vee B \vee E)$, $(\neg B \vee C \vee D)\}$ along the ordering $o = (E, D, C, B, A)$ having width and induced width 4. Conditioning on A yields two conditional theories: $\varphi_{A=0} = \{(\neg C \vee E)$, $(B \vee C)$, $(\neg B \vee C \vee D)\}$, and $\varphi_{A=1} = \{(\neg C \vee E)$, $(B \vee E)$, $(\neg B \vee C \vee D)\}$. The ordered interaction graphs of $\varphi_{A=0}$ and $\varphi_{A=1}$ are also shown in Figure 2.25. Clearly, $w_o(B) = w^*_o(B) = 2$ for theory $\varphi_{A=0}$, and $w_o(B) = w^*_o(B) = 3$ for theory $\varphi_{A=1}$. Note that, besides deleting A and its incident edges from the interaction graph, an assignment may also delete some other edges (e.g., $A = 0$ removes the edge between B and E because the clause $(\neg A \vee B \vee E)$ becomes satisfied).

The conditioning variables can be selected in advance ("statically"), or during the algorithm's execution ("dynamically"). In our experiments, we focused on the dynamic version *Dynamic Conditioning + DR (DCDR)* that was superior to the static one.

Algorithm DCDR(b) guarantees that the induced width of variables that are resolved upon is bounded by b. Given a consistent partial assignment $I(C)$ to a set of variables C, the algorithm performs resolution over the remaining variables having $w^*_{I(C)} < b$. If there are no such variables, the algorithm selects a variable and attempts to assign it a value consistent with $I(C)$. The idea of DCDR(b) is demonstrated in Figure 2.26 for the theory $\varphi = \{(\neg C \vee E), (A \vee B \vee C \vee D)$, $(\neg A \vee B \vee E \vee D)$,

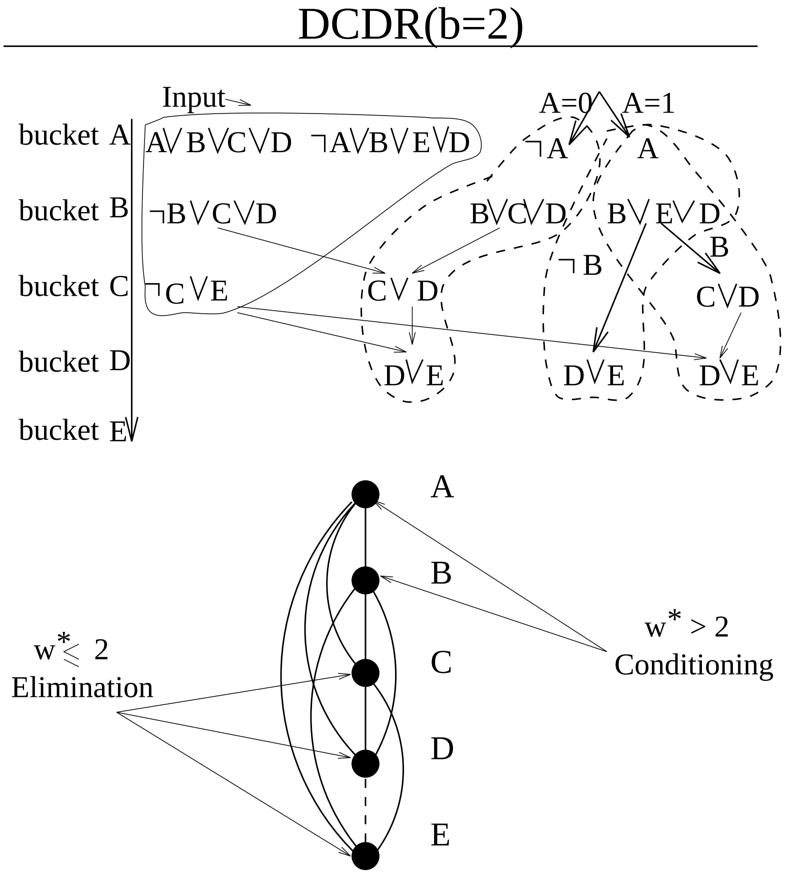


Figure 2.26: A trace of DCDR(2) on the theory $\varphi = \{(\neg C \vee E), (A \vee B \vee C), (\neg A \vee B \vee E), (\neg B \vee C \vee D)\}$.

$(\neg B \vee C \vee D)\}$. Assume that we run DCDR(2) on φ. Every variable is initially connected to at least 3 other variables in $G(\varphi)$. As a result, no resolution can be done and a conditioning variable is selected. Assume that A is selected. Assignment $A = 0$ adds the unit clause $\neg A$ which causes unit resolution in $bucket_A$, and produces a new clause $(B \vee C \vee D)$ from $(A \vee B \vee C \vee D)$. The assignment $A = 1$ produces clause $(B \vee E \vee D)$. In Figure 2.26, the original clauses are shown on the left as a partitioning into buckets. The new clauses are shown on the right, within the corresponding search-tree branches.

Following the branch for $A = 0$ we get a conditional theory $\{(\neg B \vee C \vee D)$, $(B \vee C \vee D)$, $(\neg C \vee E)\}$. Since the degrees of all the variables in the corresponding (conditional) interaction graph are now 2 or less, we can proceed with resolution. We select B, perform resolution in its bucket, and record the resolvent $(C \vee D)$ in $bucket_C$. The resolution in $bucket_C$ creates clause $(D \vee E)$. At this point, the algorithm terminates, returning the assignment $A = 0$, and the conditional directional extension $\varphi \wedge$ $(B \vee C \vee D) \wedge$ $(C \vee D) \wedge$ $(D \vee E)$.

The alternative branch of $A = 1$ results in the conditional theory $\{(B \vee E \vee D)$, $(\neg B \vee C \vee D)$, $(\neg C \vee E)\}$. Since each variable is connected to three other variables, no resolution is possible. Conditioning on B yields the conditional theory $\{(E \vee D)$, $(\neg C \vee E)\}$ when $B = 0$, and the conditional theory $\{(C \vee D), (\neg C \vee E)\}$ when $B = 1$. In both cases, the algorithm terminates, returning $A = 1$, the assignment to B, and the corresponding conditional directional extension.

Algorithm DCDR(b) (Figure 2.27) takes as an input a propositional theory φ and a parameter b bounding the size of resolvents. Unit propagation is performed first (lines 1-2). If no inconsistency is discovered, DCDR proceeds to its primary activity: choosing between resolution and conditioning. While there is a variable Q connected to at most b other variables in the current interaction graph conditioned on the current assignment, DCDR resolves upon Q (steps 4-9). Otherwise, it selects an unassigned variable (step 10), adds it to the cutset (step 11), and continues recursively with the conditional theory $\varphi \wedge \neg Q$. An unassigned variable is selected using the same dynamic variable ordering heuristic that is used by DP. Should the theory prove inconsistent the algorithm switches to the conditional theory $\varphi \wedge Q$. If both positive and negative assignments to Q are inconsistent the algorithm backtracks to the previously assigned variable. It returns to the previous level of recursion and the corresponding state of φ, discarding all resolvents added to φ after the previous assignment was made. If the algorithm does not find any consistent partial assignment it decides that the theory is inconsistent and returns an empty cutset and an empty directional extension. Otherwise, it returns an assignment $I(C)$ to the cutset C, and the conditional directional extension $E_o(\varphi_{I(C)})$ where o is the variable ordering dynamically constructed by the algorithm. Clearly, the conditional induced width

DCDR(φ, X, b)

Input: A cnf theory φ over variables X; a bound b.

Output: A decision of whether φ is satisfiable. If it is, an assignment $I(C)$ to its conditioning variables, and the conditional directional extension $E_o(\varphi_{I(C)})$.

```
if unit_propagate($\varphi$) = false, return(false);
else $X \leftarrow X - \{$ variables in unit clauses $\}$
if no more variables to process, return true;
else while $\exists Q \in X$ s.t. $degree(Q) \leq b$ in the current graph
        resolve over Q
        if no empty clause is generated,
            add all resolvents to the theory
        else return false
        $X \leftarrow X - \{Q\}$
Select a variable $Q \in X$; $X \leftarrow X - \{Q\}$
$C \leftarrow C \cup \{Q\}$;
return( DCDR($\varphi \wedge \neg Q$, $X$, $b$) $\vee$
        DCDR($\varphi \wedge Q$, $X$, $b$) ).
```

Figure 2.27: Algorithm DCDR(b).

$w^*_{I(C)}$ of φ's interaction graph with respect to o and to the assignment $I(C)$ is bounded by b.

Theorem 9: (DCDR(b) soundness and completeness) *Algorithm DCDR(b) is sound and complete for satisfiability. If a theory φ is satisfiable, any model of φ consistent with the output assignment $I(C)$ can be generated backtrack-free in $O(|E_o(\varphi_{I(C)})|)$ time where o is the ordering computed dynamically by DCDR(b).* $\Box$

Theorem 10: (DCDR(b) complexity) *The time complexity of algorithm DCDR(b) is $O(n2^{\alpha \cdot b + |C|})$, where C is the largest cutset ever conditioned upon by the algorithm, and $\alpha \leq log_2 9$. The space complexity is $O(n \cdot 2^{\alpha \cdot b})$.* $\Box$

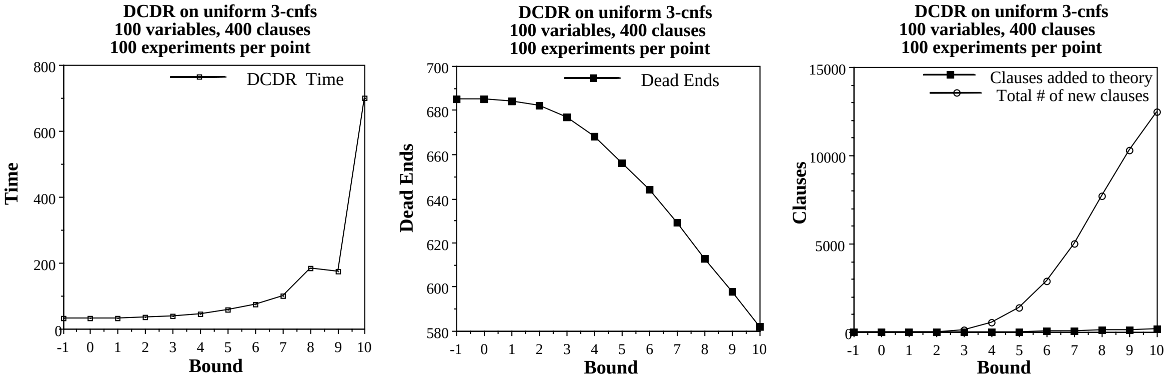
The parameter b can be used to control the trade-off between search and resolution. If $b \geq w^*_o(\varphi)$, where o is the ordering used by DCDR(b), the algorithm coincides with

DR having time and space complexity exponential in $w^*(\varphi)$. It is easy to show that the ordering generated by DCDR(b) in case of no conditioning yields a min-degree ordering. Thus, given b and a min-degree ordering o, we are guaranteed that DCDR(b) coincides with DR if $w^*_o \leq b$. If $b < 0$, the algorithm coincides with DP. Intermediate values of b allow trading space for time. As b increases, the algorithm requires more space and less time (see also [25]). However, there is no guaranteed worst-case time improvement over DR. It was shown [7] that the size of the smallest *cycle-cutset* C (a set of nodes that breaks all cycles in the interaction graph, leaving a tree, or a forest), and the smallest induced width, w^*, obey the relation $|C| \geq w^* - 1$. Therefore, for $b = 1$, and for a corresponding cutset C_b, $\alpha \cdot b + |C_b| \geq w^* + \alpha - 1 \geq w^*$, where the left side of this inequality is the exponent that determines complexity of DCDR(b) (Theorem 10). In practice, however, backtracking search rarely demonstrates its worst-case performance and thus the average complexity of DCDR(b) is superior to its worst-case bound as will be confirmed by our experiments.

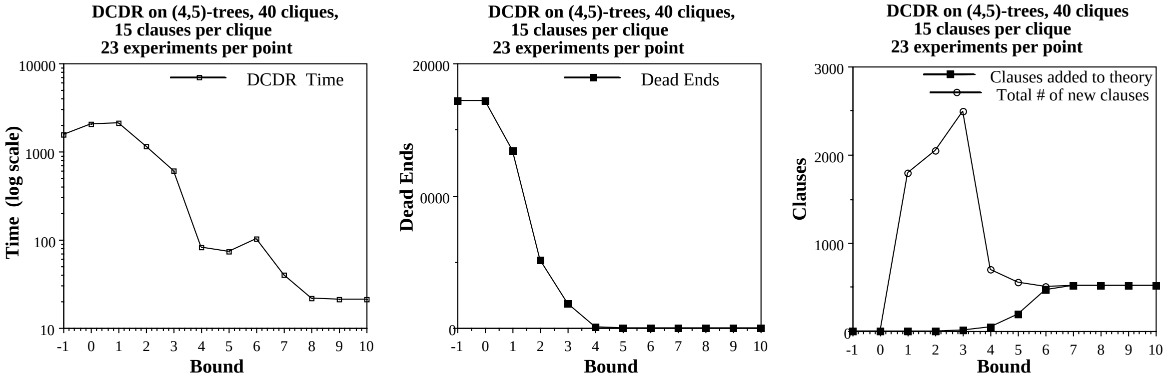
Algorithm DCDR(b) uses the 2-literal-clause ordering heuristic for selecting conditioning variables as used by DP. Random tie-breaking is used for selecting the resolution variables.

2.7.4 Empirical evaluation of DCDR(b)

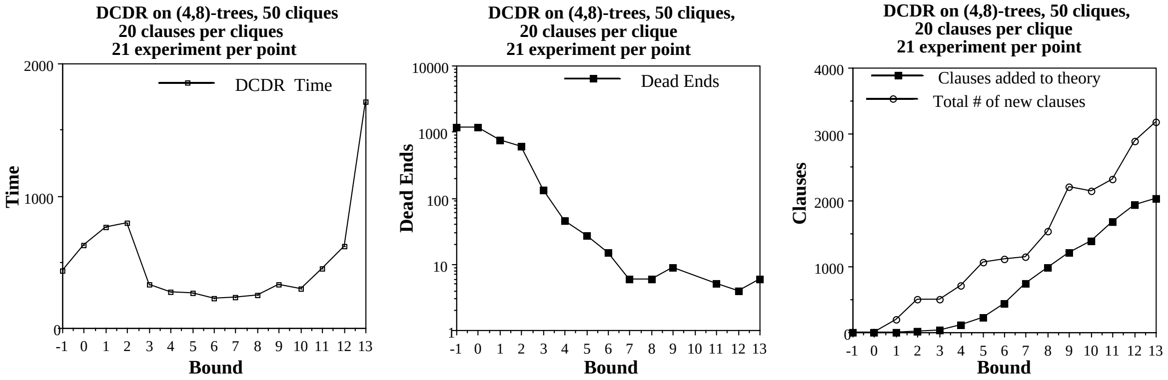
We evaluated the performance of DCDR(b) as a function of b. We tested problem instances in the 50%-satisfiable region (the phase transition region). The results for different b and three different problem structures are summarized in Figures 2.28-2.30. Figure 2.28(a) presents the results for uniform 3-cnfs having 100 variables and 400 clauses. Figures 2.28(b) and 2.28(c) focus on $(4,5)$-trees and on $(4,8)$-trees, respectively. We plotted the average time, the number of dead-ends, and the number of new clauses generated as functions of the bound b (we plot both the total number of generated clauses and the number of clauses actually added to the output theory excluding tautologies and subsumed clauses).



(a) uniform 3-cnfs



(b) (4,5)-trees



(c) (4,8)-trees

Figure 2.28: DCDR(b) on three different classes of 3-cnf problems. Average time, the number of dead-ends, and the number of new clauses are plotted as functions of the parameter b.

As expected, the performance of DCDR(b) depends on the induced width of the theories. We observed three different patterns:

- On problems having large w^*, such as uniform 3-cnfs in the phase-transition region (see Figure 2.28), the time complexity of DCDR(b) is similar to DP when b is small. However, when b increases, the CPU time grows exponentially. Apparently, the decline in the number of dead ends is too slow relative to the exponential (in b) growth in the total number of generated clauses. However, the number of new clauses actually *added* to the theory grows slowly. Consequently, the final conditional directional extensions have manageable sizes. We obtained similar results when experimenting with uniform theories having 150 variables and 640 clauses.

- Since DR is equivalent to DCDR(b) whenever b is equal or greater then w^*, for theories having small induced width, DCDR(b) indeed coincides with DR even for small values of b. Figure 2.28(b) demonstrates this behavior on (4,5)-trees with 40 cliques, 15 clauses per clique, and induced width 6. For $b \geq 8$, the time, the total number of clauses generated, as well as the number of new clauses added to the theory, do not change. With small values of b ($b = 0, 1, 2, 3$), the efficiency of DCDR(b) was sometimes worse than that of DCDR(-1), which is equivalent to DP, due to the overhead incurred by extra clause generation (a more accurate explanation is still required).

- On (k, m)-trees having larger size of cliques (Figure 2.28(c)), intermediate values of b yielded a better performance than both extremes. DCDR(-1) is still inefficient on structured problems while large induced width made pure *DR* too costly time- and space-wise. For (4,8)-trees, the optimal values of b appear between 5 and 8.

Figure 2.29 summarizes the results for DCDR(-1), DCDR(5), and DCDR(13) on the three classes of problems. The intermediate bound b=5 seems to be overall more cost-effective than both extremes, b= -1 and b=13.

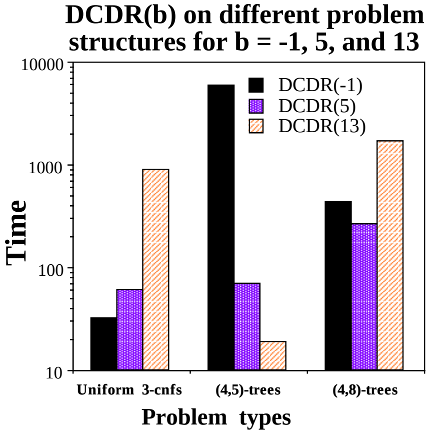


Figure 2.29: Relative performance of DCDR(b) for $b = -1, 5, 13$ on different types of problems.

Figure 2.30 describes the average number of resolved variables which indicates the algorithm's potential for knowledge compilation. When many variables are resolved upon, the resulting conditional directional extension encodes a larger portion of the models, all sharing the assignment to the cutset variables.

2.8 Related work

Directional resolution belongs to a family of elimination algorithms first analyzed for optimization tasks in dynamic programming [7] and later used in constraint satisfaction [100, 33] and in belief networks [76]. In fact, DR can be viewed as an adaptation of the constraint-satisfaction algorithm *adaptive consistency* to propositional satisfiability where the project-join operation over relational constraints is replaced by resolution over clauses [33, 37]. Using the same analogy, bounded resolution can be related to bounded consistency-enforcing algorithms, such as arc- path- and i-consistency [78, 43, 22], while bounded directional resolution, BDR(i), parallels directional i-consistency [33, 37]. Indeed, one of this chapter's contributions is transferring constraint satisfaction techniques to the propositional framework.

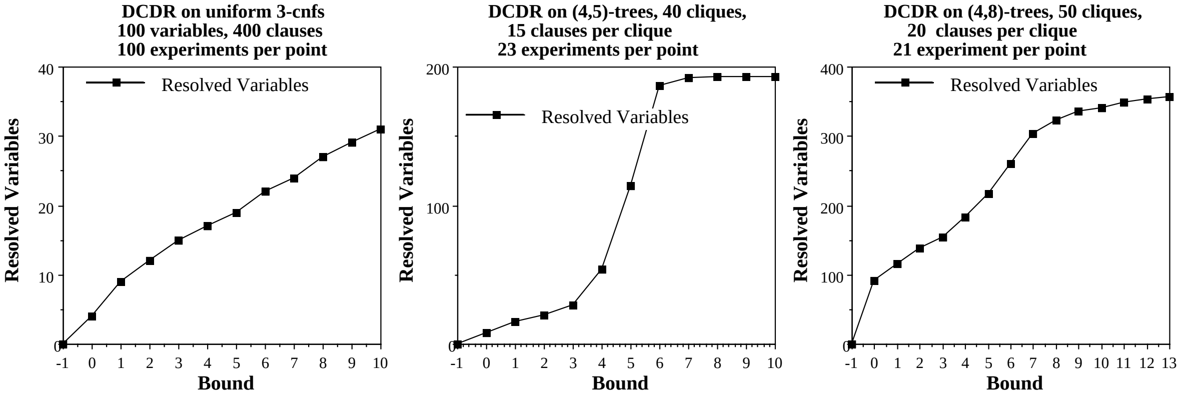


Figure 2.30: DCDR: the number of resolved variables on different problems.

The recent success of constraint processing which can be attributed to techniques combining search with limited forms of constraint propagation (e.g., forward-checking, MAC, constraint logic programming [59, 53, 99, 67]) that motivated our hybrid algorithms. In the SAT community, a popular form of combining constraint propagation with search is unit-propagation in DP. Our work extends this idea.

The hybrid algorithm BDR-DP(i), initially proposed in [35], corresponds to applying directional i-consistency prior to backtracking search for constraint processing. This approach was empirically evaluated for some constraint problems in [32]. However, those experiments were restricted to small and relatively easy problems, for which only a very limited amount of preprocessing was cost-effective. The presented experiments with BDR-DP(i) suggest that the results in [32] were too preliminary and that the idea of preprocessing before search is viable and should be further investigated.

Our second hybrid algorithm, DCDR(b), proposed first in [94], generalizes the cycle-cutset approach that was presented for constraint satisfaction [21] using static variable ordering. This idea of alternating search with bounded resolution was also suggested and evaluated independently by van Gelder in [55], where a generalization of unit resolution known as *k-limited resolution* was proposed. This operation requires that the operands and the resolvent have at most k literals each. The hybrid algorithm proposed in [55] computes *k-closure* (namely, it applies k-limited resolution iteratively and eliminates subsumed clauses) between branching steps in DP-backtracking. This algorithm, augmented with several branching heuristics, was tested for k=2 (the combination called $2cl$ algorithm), and demonstrated its superiority to DP, especially on larger problems. Algorithm DCDR(b) computes a *subset* of b-closure between its branching steps [10]. In this chapter, we study the impact of b on the effectiveness of hybrid algorithms over different problem structures, rather than focus on a fixed b.

2.9 Summary and conclusions

The chapter compares two popular approaches to solving propositional satisfiability, backtracking search and resolution, and proposes two parameterized hybrid algorithms. We analyze the complexity of the original resolution-based Davis-Putnam algorithm, called here directional resolution (DR)), as a function of the induced width of the theory's interaction graph. Another parameter called diversity provides an additional refinement for tractable classes. Our empirical studies confirm previous results showing that on uniform random problems DR is indeed very inefficient. However, on structured problems such as k-tree embeddings, having bounded induced width, directional resolution outperforms the popular backtracking-based Davis-Putnam-Logemann-Loveland Procedure (DP).

The two parameterized hybrid schemes, BDR-DP(i) and DCDR(b), allow a flexible combination of backtracking search with directional resolution. Both schemes use a parameter that bounds the size of the resolvents recorded. The first scheme, BDR-DP(i), uses bounded directional resolution BDR(i) as a preprocessing step, recording only new clauses of size i or less. The effect of the bound was studied empirically over

---

[10] DCDR(b) performs resolution on variables that are connected to at most b other variables; therefore, the size of resolvents is bounded b. It does not, however, resolve over the variables having degree higher than b in the conditional interaction graph, although such resolutions can sometimes produce clauses of size not larger than b.

both uniform and structured problems, observing that BDR-DP(i) frequently achieves its optimal performance for intermediate levels of i, outperforming both DR and DP. We also believe that the transition from i=3 to i=4 is too sharp and that intermediate levels of preprocessing are likely to provide even better trade-off. Encouraging results are obtained for BDR-DP(i) on DIMACS benchmark, where the hybrid algorithm easily solves some of the problems that were hard both for DR and DP.

The second hybrid scheme uses bounded resolution *during* search. Given a bound b, algorithm DCDR(b) instantiates a dynamically selected subset of conditioning variables such that the induced width of the resulting (conditional) theory and therefore the size of the resolvents recorded does not exceed b. When $b \leq 0$, $DCDR(b)$ coincides with DP, while for $b \geq w^*_o$ (on the resulting ordering o) it coincides with directional resolution. For intermediate b, DCDR(b) was shown to outperform both extremes on intermediate-w^* problem classes.

For both schemes selecting the bound on the resolvent size allows a flexible scheme that can be adapted to the problem structure and to computational resources. Our current "rule of thumb" for DCDR(b) is to use small b when w^* is large, relying on search, large b when w^* is small, exploiting resolution, and some intermediate bound for intermediate w^*. Additional experiments are necessary to further demonstrate the spectrum of optimal hybrids relative to problem structures.

Chapter 3

Exploiting Causal Independence

3.1 Introduction

Belief networks [1] are a powerful and convenient tool for probabilistic reasoning, successfully used in many practical applications, including medical diagnosis, hardware troubleshooting, and noisy-channel communication. However, (exact) reasoning in belief networks is known to be NP-hard [13]. Commonly used structure-exploiting algorithms such as *join-tree propagation* [75, 68, 105] and *variable elimination* [114, 23] are time and space exponential in the network parameter known as *induced width* (the size of largest clique induced by the above inference algorithms). The induced width is often large, especially in networks with large *families* (a family is a group of nodes that participate in the same conditional probability table, or CPT). Even CPT specification, which is exponential in the family size, may be intractable in such networks. One way to cope with this problem is to make structural assumptions that simplify the CPT specification. In this chapter, we focus on an assumption known as *causal independence* [63, 111, 114], where multiple causes contribute independently to a common effect. The causal-independence assumption is commonly used in large practical networks, such as CPCS and QMR-DT networks for medical diagnosis [90].

---

[1] Also known as *Bayesian networks*, *causal networks*, or *graphical models*.

The causal-independence assumption simplifies CPT specification from exponential to linear in family size. However, the question is to what extent it is possible to maintain such concise representation during inference. Some computational benefits of causal independence have already been demonstrated by previously proposed approaches that include *network transformations* [63, 84], the variable-elimination algorithm *VE1* [114], and the algorithm *Quickscore* for a special class of *two-layer noisy-OR (BN2O)* networks. Our work extends the existing approaches in several ways.

- We provide the connection between the network transformations and algorithm VE1, which can be viewed as variable-elimination inference applied to a transformed network subject to some variable ordering restrictions. We show that the ordering restrictions imposed by VE1 may sometimes lead to a unnecessary complexity increase, and describe a general variable-elimination scheme, called *ci-elim-bel*, that improves VE1 by accommodating any variable ordering.

- We investigate the impact of causal independence on finding a most probable explanation (MPE), finding a maximum a posteriori hypothesis (MAP), and finding a maximum expected utility (MEU) decision. We show that, while causal independence can significantly reduce the complexity of belief updating and finding MAP and MEU, it is generally not effective for MPE. Finally, we outline the algorithms for finding MAP and MEU in causally-independent networks.

- The complexity of the above algorithms is analyzed using the notion of the induced width of the transformed network (called the *effective induced width*). We demonstrate that, when a proper variable ordering is used, the effective induced width is never larger than the induced width of the *moral graph* of the original network (the graph where the parents in each family are connected), and can be as small as the induced width of the original (unmoralized) directed acyclic graph. For example, exploiting causal independence in polytrees with families of size m reduces the induced width from m to 2. An advantage of the graph-based complexity analysis is that the anticipated computational benefits of exploiting causal independence can be evaluated in advance and compared with those of general-purpose algorithms.

- Finally, we show that causal independence allows a more efficient propagation of evidence. A causally-independent network can be transformed into one that combines probabilistic and deterministic relations. Constraint-propagation techniques, such as *relational arc-consistency*, can propagate evidence and simplify subsequent probabilistic inference. We present an evidence-propagation scheme for causally-independent networks generalizing the property of noisy-OR: observing a particular value of one variable, such as $z = 0$ in $z = x \vee y$, allows to deduce value assignments to some other variables. Subsequently, we present an algorithm for arbitrary noisy-OR networks that uses evidence propagation and generalizes the *Quickscore* algorithm [60].

This chapter is organized as follows. Sections 3.2 and 3.3 provide background on probabilistic inference and on causal independence. Section 3.4 introduces the notion of transformed networks, while Section 3.5 focuses on belief updating using transformed networks. Subsection 3.5.1 analyzes the computational benefits of exploiting causal independence, while subsection 3.5.2 shows the relation between network transformations and algorithm VE1. Section 3.6 extends the analysis to the tasks of finding MPE, MAP and MEU. Evidence propagation in causally-independent networks is discussed in Section 3.7. Section 3.8 concludes this chapter. Proofs of some theorems can be found in Appendix B.

3.2 Inference in belief networks: an overview

This section provides a background on belief networks, probabilistic tasks and inference algorithms, focusing on the *bucket-elimination* approach [23].

A belief network is a *directed acyclic graph*, where the nodes represent random variables and the edges denote probabilistic dependencies among those variables, quantified by conditional probabilities. A formal definition is given after introducing some basic notation and terminology.

A *directed graph* is a pair $G = \{V, E\}$, where $V = \{X_1, ..., X_n\}$ is a set of nodes and $E = \{(X_i, X_j)|X_i, X_j \in V, i \neq j\}$ is a set of edges. Two nodes X_i and X_j are called *neighbors* if there is an edge between them (either (X_i, X_j) or (X_j, X_i)). We say that X_i *points* to X_j if $(X_i, X_j) \in E$; X_i is called a *parent* of X_j, while X_j is called a *child* of X_i. The set of parent nodes of X_i is denoted $pa(X_i)$, or pa_i, while the set of child nodes of X_i is denoted $ch(X_i)$, or ch_i. We call a node and its parents a *family.* A directed graph is *acyclic* if it has no directed cycles. In an *undirected graph*, the directions of the edges are ignored: (X_i, X_j) and (X_j, X_i) are identical. A directed graph is *singly-connected* (also known as a *polytree*), if its underlying undirected graph (called *skeleton graph*) has no (undirected) cycles. Otherwise, it is called *multiply-connected.* A directed *ordered* graph is a pair (G, o), where G is a directed graph and o is an ordering of its variables, $o = (X_1, ..., X_n)$. Given a subgraph $G' = \{V', E'\}$ of G, where $V' \subseteq V$, the ordering o' of G' obtained by deleting $X_i \notin V'$ from o is called the *restriction* of o to G' (we also say that o' *agrees* with o on the set of nodes V').

Let $X = \{X_1, ..., X_n\}$ be a set of random variables having *domains* $D_1, ..., D_n$, respectively. A *belief network* is a pair (G, P), where $G = (X, E)$ is a directed acyclic graph representing the variables as nodes and $P = \{P(x_i|pa_i)|i = 1, ..., n\}$ is the set of conditional probabilities defined for each variable X_i and its parents pa_i in G. A belief network represents a joint probability distribution over X having the product form

$$P(x_1, ..., x_n) = \Pi_{i=1}^{n} P(x_i|pa_i). \tag{3.1}$$

where x stnads for value of X (when there is no confusion, the lower-case letters will sometimes denote variables as well).

The *moral graph* G^M of a belief network (G, P) is obtained by connecting all the parents of each node and dropping the directionality of edges. The original

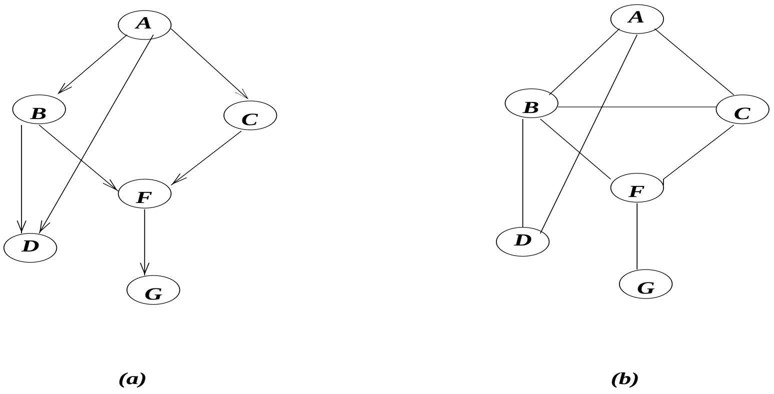


Figure 3.1: (a) a belief network representing the joint probability distribution $P(g, f, d, c, b, a) = P(g|f)P(f|c,b)P(d|b,a)P(b|a)P(c|a)P(a)$, and (b) its moral graph.

directed acyclic graph G will be also called *unmoral graph.* We will use the terms "moral" ("unmoral") graph and "moral" ("unmoral") network interchangeably. An *evidence* $e = \bigcup_j (X_j = d_j)$, where $d_j \in D_j$, is an instantiated subset of variables. We denote variables by upper-case letters, and use lower-case letters for the corresponding domain values.

Example 6: Consider the belief network that represents

$$P(g,f,d,c,b,a) = P(g|f)P(f|c,b)P(d|b,a)P(b|a)P(c|a)P(a).$$

Its acyclic directed graph is shown in Figure 3.1a, and the corresponding moral graph is shown in Figure 3.1b. In this case, for example, $pa(F) = \{B, C\}$, $pa(B) = \{A\}$, $pa(A) = \emptyset$, $ch(A) = \{B, D, C\}$.

The following reasoning tasks are defined over belief networks:

1. *belief updating,* i.e. finding the posterior probability $P(Y|e)$ of *query* nodes $Y \subset X$ given evidence e (this task is often referred to as *probabilistic inference*);

2. finding a *most probable explanation (MPE)*, i.e. finding a maximum probability assignment to unobserved variables given evidence;

3. finding *maximum a posteriori hypothesis (MAP)*, i.e. finding a maximum probability assignment to a subset of *hypothesis* variables given evidence;

4. given a *utility* function, finding the *maximum expected utility (MEU)* decision, namely, finding an assignment to a set of *decision* nodes that maximizes the expected utility.

All these tasks are known to be NP-hard [13]. However, there exists a polynomial propagation algorithm for singly-connected networks [88]. The two common approaches to extending this algorithm to *exact* inference in multiply-connected networks are the *cycle-cutset* approach, also called *conditioning*, and *join-tree clustering* [88, 76, 103]. Join-tree clustering is closely related to the *variable-elimination* approach [17, 114, 23]. We focus here on a general variable-elimination scheme called *bucket-elimination* [23] that allows a unifying approach to various reasoning tasks.

3.2.1 The bucket-elimination scheme

A bucket-elimination algorithm accepts as an input an ordered set of variables and a set of functions, such as propositional clauses, constraints, or conditional probability functions. Each variable is associated with a *bucket*. All functions defined on variable X_i and on lower-index variables are placed in the bucket of X_i. Bucket-elimination processes each bucket, from last to first, applying an *elimination operator* which eliminates the bucket's variable and computes a new function that summarizes the effect of this variable on the rest of the problem. The new function is placed in an appropriate lower bucket. Some elimination operators are defined below:

Definition 5: Given a function h defined over subset of variables S, where $X \in S$, the functions $(\min_X h)$, $(\max_X h)$, $(mean_X h)$, and $(\sum_X h)$ are defined over $U = S - \{X\}$ as follows. For every $U = u$, $(\min_X h)(u) = \min_x h(u,x)$, $(\max_X h)(u) = \max_x h(u,x)$, $(\sum_X h)(u) = \sum_x h(u,x)$, and $(mean_X h)(u) = \sum_x \frac{h(u,x)}{|X|}$, where $|X|$ is the cardinality of X's domain. 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 $U = \cup_j S_j$. For every $u \in U$, $(\Pi_j h_j)(u) = \Pi_j h_j(u_{S_j})$, and $(\sum_j h_j)(u) = \sum_j h_j(u_{S_j})$.

Next, we review the bucket-elimination algorithms for belief updating and for finding the MPE [23]. Without loss of generality, consider the task of updating the belief in X_1. Given a belief network (G, P), and a variable ordering $o = (X_1, ..., X_n)$, the belief $P(x_1|e)$ is defined as:

$$P(x_1|e) = \frac{P(x_1, e)}{P(e)} = \alpha P(x_1, e) = \alpha \sum_{X/\{X_1\}} \prod_i P(x_i|pa_i) = \alpha \sum_{x_2} \ldots \sum_{x_n} \prod_i P(x_i|pa_i), \tag{3.2}$$

where α is a normalizing constant. By the distributivity law,

$$\sum_{x_2} \ldots \sum_{x_n} \prod_i P(x_i|pa_i) = F_1 \sum_{x_2} F_2 \ldots \sum_{x_n} F_n, \tag{3.3}$$

where each $F_i = \prod_x P(x|pa(x))$ is the product of all probabilistic components defined on X_i and *not* defined on any variable X_j for $j > i$. The set of all such components is initially placed in the *bucket* of X_i (denoted *bucket*$_i$). Algorithm *elim-bel* [23] shown in Figure 3.2 computes the sums in the equation (3.3) sequentially from right to left, eliminating variables from X_n to X_1. For each X_i, the algorithm multiplies the components of *bucket*$_i$, then sums over X_i, and puts the resulting function in the bucket of its highest-index variable. If X_i is observed ($X_i = a$), then X_i is replaced by a independently in each of the bucket's components, and each result is placed in its highest-variable buckets. The following example illustrates *elim-bel* on the network in Figure 3.3a.

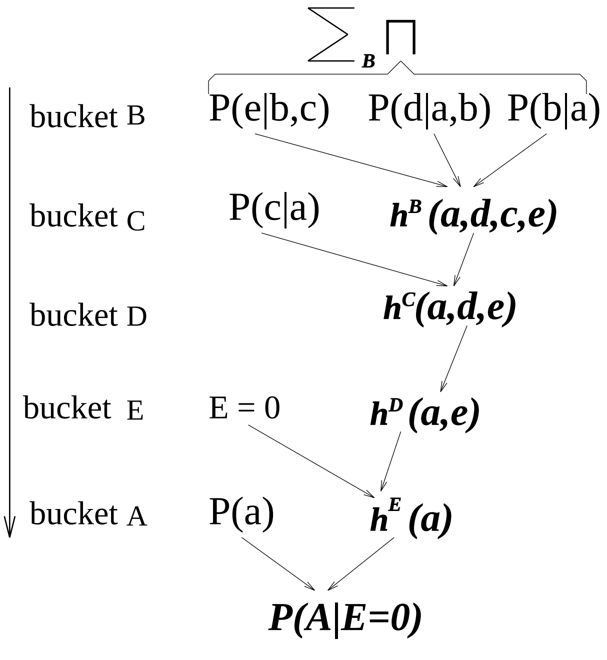
Example 7: Given the belief network in Figure 3.3, the ordering $o = (A, E, D, C, B)$, and evidence $E = 0$, $Bel(a) = P(a|E = 0) = \alpha P(a, E = 0)$ is computed as follows:

$$P(a, E = 0) = \sum_{E=0,d,c,b} P(a, b, d, c, e) = \sum_{E=0,d,c,b} P(a)P(c|a)P(e|b, c)P(d|a, b)P(b|a) =$$

$$P(a) \sum_{E=0} \sum_{d} \sum_{c} P(c|a) \sum_{b} P(e|b, c)P(d|a, b)P(b|a).$$

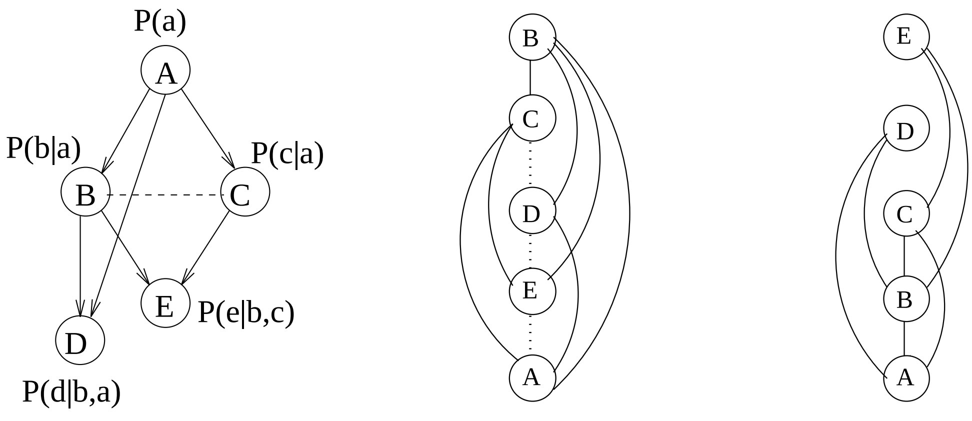
```
Algorithm elim-bel(BN,o,e)
Input: A belief network $BN = (G, P)$, ordering $o$, evidence $e$.
Output: $P(x_1|e)$, the belief in $X_1$ given evidence $e$.

  1. Initialize: Partition $P = \{P_1, ..., P_n\}$ into buckets $bucket_1$, ..., $bucket_n$, where
     $bucket_p$ contains all matrices $h_1, h_2, ..., h_t$ whose highest-index variable is $X_i$.

  2. Backward: for $p = n$ to 1 do

       • If $X_p$ is observed ($X_p = a$), replace $X_p$ by $a$ in each $h_i$
         and put the result in its highest-variable bucket.

       • Else compute $h^p = \sum_{X_p} \Pi_{i=1}^{t} h_i$
         and place $h^p$ in its highest-variable bucket.

  3. Return $Bel(x_1) = \alpha P(x_1) \cdot \Pi_i h_i(x_1)$,
     where each $h_i$ is in $bucket_1$ and $\alpha$ is a normalizing constant.
```

(a) Algorithm *elim-bel.*



(b) A trace of *elim-bel.*

Figure 3.2: (a) Algorithm *elim-bel* and (b) an example of the algorithm's execution.



(a) (b) (c)

Figure 3.3: (a) A belief network, (b) its induced graph along $o = (A, E, D, C, B)$, and (c) its induced graph along $o = (A, B, C, D, E)$.

Using bucket elimination, we get:
1. bucket B: $h^B(a, d, c, e) = \sum_b P(e|b, c)P(d|a, b)P(b|a)$
2. bucket C: $h^C(a, d, e) = \sum_c P(c|a)h^B(a, d, c, e)$
3. bucket D: $h^D(a, e) = \sum_d h^C(a, d, e)$
4. bucket E: $h^E(a) = h^d(a, E = 0)$
5. bucket A: $Bel(a) = P(a|E = 0) = \alpha P(a)h^E(a)$,
where α is a normalizing constant. Figure 3.2(b) shows a schematic trace of the algorithm.

Note that the elimination procedure can be simplified by processing evidence nodes first, thus placing them last in the ordering. In example 7, placing E last in the ordering yields:
1. bucket E: $h^E(b, c) = P(e = 0|b, c)$
2. bucket B: $h^B(a, d, c) = \sum_b h^E(b, c)P(d|a, b)P(b|a)$
3. bucket C: $h^C(a, d) = \sum_c P(c|a)h^B(a, d, c)$
4. bucket D: $h^D(a) = \sum_d h^C(a, d)$
5. bucket A: $Bel(a) = P(a|E = 0) = \alpha P(a)h^E(a)$,

```
Algorithm elim-mpe(BN,o,e)
Input: A belief network $BN = (G, P)$, ordering $o$, evidence $e$.
Output: An MPE assignment.

  1. Initialize: Partition $P = \{P_1, ..., P_n\}$ into buckets $bucket_1$, ..., $bucket_n$, where
     $bucket_p$ contains all matrices $h_1, h_2, ..., h_t$ whose highest-index variable is $X_i$.

  2. Backward: for $p = n$ to 1 do

       • If $X_p$ is observed ($X_p = a$), replace $X_p$ by $a$ in each $h_i$
         and put the result in its highest-variable bucket.

       • Else compute $h^p = \max_{x_p} \Pi_{i=1}^t h_i$,
         $x_p^{opt} = \arg\max_{x_p} \Pi_{i=1}^t h_i$,
         and place $h^p$ in its highest-variable bucket.

  3. Forward: for $p = 1$ to $n$,
     given $X_1 = x_1^{opt}, ..., X_{p-1} = x_{p-1}^{opt}$, assign $x_p^{opt}$ to $X_p$.

  4. Return the assignment $x^{opt} = (x_1^{opt}, ..., x_n^{opt})$.
```

Figure 3.4: Algorithm *elim-mpe*.

Eliminating an evidence variable is equivalent to replacing this variable by its observed value in all relevant probabilistic components. This simplifies computation, and corresponds to removing the evidence node from the moral network. From now on, *we always assume that evidence nodes are eliminated first, and refer to the variable ordering of the remaining network.*

Similar bucket-elimination algorithms were derived for the tasks of finding MPE, MAP, and MEU [23]. Given a belief network (G, P), and given a variable ordering $o = (X_1, ..., X_n)$, the MPE task is to find

$$MPE = \max_{x_1} \ldots \max_{x_n} \prod_i P(x_i | pa_i). \tag{3.4}$$

Using the product form of the joint probability distribution, we move maximization over each variable as far to the right as possible:

$$MPE = \max_{x_1} F_1 \ldots \max_{x_{n-1}} F_{n-1} \max_{x_n} F_n. \tag{3.5}$$

This leads to the bucket-elimination algorithm *elim-mpe* [23] (see Figure 3.4), which is quite similar to *elim-bel*, except that maximization is replaced by summation. Bucket-elimination algorithms *elim-map* and *elim-meu* for finding MAP and MEU, respectively, are presented in [23].

An important property of bucket-elimination algorithms is that their performance can be predicted using a graph parameter called *induced width* [33] (also known as *tree-width* [2]), which describes the largest clique created in the problem's graph and corresponds to the largest function recorded by the algorithm. Formally, the induced width is defined as follows. Given a (directed or undirected) graph G, the *width* of X_i along ordering o is the number of X_i's neighbors preceding X_i in o. The *width of the graph* along o, denoted w_o, is the maximum width along o. The *induced graph* of G along o is obtained by connecting the preceding neighbors of each X_i, going from $i = n$ to $i = 1$. The induced width along o, denoted w_o^*, is the width of the induced graph along o, while the induced width w^* is the minimum induced width along any ordering. For example, Figures 3.3b and 3.3c depict the induced graphs (induced edges are shown as dashed lines) of the moral graph in Figure 3.3a along the orderings $o = (A, E, D, C, B)$ and $o' = (A, B, C, D, E)$, respectively. We get $w_o^* = 4$ and $w_{o'}^* = 2$. It can be shown that the induced width of node X_i bounds the number of arguments of any function computed in $bucket_i$. Therefore, the complexity of bucket-elimination algorithms is time and space exponential in w^*. Note, that we assume that evidence nodes are eliminated from the moral graph before computing its induced width. Assuming finite domains of size not greater than d,

Theorem 11: *[23] The complexity of bucket-elimination algorithms is $O(nd^{w_o^*+1})$, where n is the number of variables in a belief network, and w_o^* is the induced width of the moral graph along ordering o after all evidence nodes are removed.*

The induced width will vary depending on the variable ordering. Although finding a minimum-w^* ordering is NP-hard [2], good heuristic algorithms are available [7, 22, 97]. For more details on bucket-elimination and induced width see [23, 27].

Note that the induced width cannot be lower than $|F|-1$ where $|F|$ is the size of largest family in the network. The next section introduces a simplifying assumption about CPTs called *causal independence* that allows decomposing large families into smaller ones.

3.3 The causal independence assumption

The specification of the conditional probability tables (CPTs) is exponential in the family size which may be sometimes prohibitively large. For example, dozens of different diseases may cause the same symptom, such as fever. In such cases, simplifying assumptions about the nature of the probabilistic dependencies should be made. This chapter will focus on the *causal independence* assumption often used in real-life applications, such as medical-diagnosis CPCS and QMR-DT networks [90].

Causal independence assumes that several causes contribute independently to a common effect. Consider the following example inspired by [89]. A burglary alarm can be turned on by one of possible causes, such as a burglary, an earthquake, or something else. A belief network describing this situation is shown in Figure 3.5a, where the "effect" node e stands for the state of the alarm (on or off), while the nodes c_1, c_2,...,c_n represent possible causes, such as burglary or earthquake. Specification of the full CPT, namely, the probability of alarm given every combination of its parents' values, would be intractable for large n. Moreover, it is generally difficult to assess such joint probabilities from the causal mechanisms that are unrelated to each other (belong to different "frames of knowledge" [89]). For example, the probability of the alarm not turning on in the case of a burglary depends on the burglar's skills, which are unrelated to earthquakes. It is more natural to specify separately the probability of alarm given burglary, and the probability of alarm given earthquake. How should we combine those two? Assume there were no uncertainty, and the alarm would always turn on if either a burglary, or an earthquake, or any of the other specified

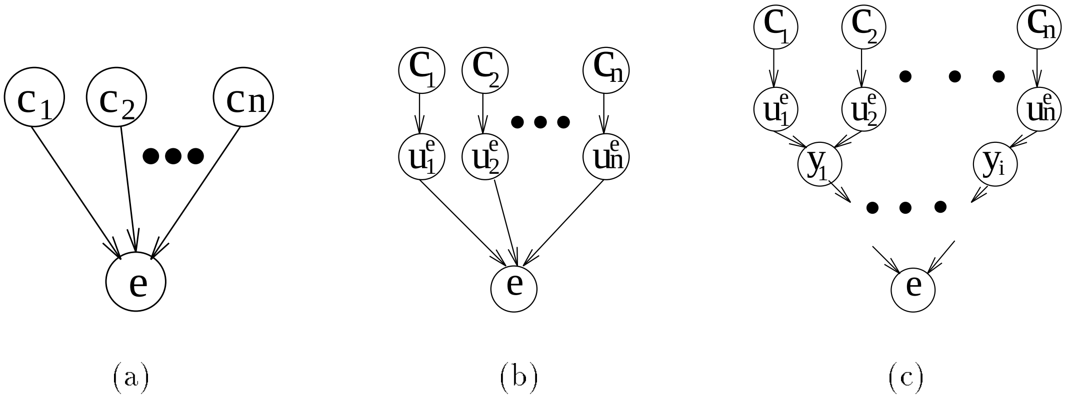


Figure 3.5: (a) a causally-independent family, (b) its dependence graph, and (c) its binary-tree transformation.

causes occurred. Then we could use a simple disjunctive model

$$c_1 \vee c_2 \vee ... \vee c_n \rightarrow e.$$

However, in real-life, the alarm may not always turn on in the presence of one or more causes, since it may be broken, or a burglar may be smart, or some other hidden *inhibiting mechanism* [89] may be present. Those hidden mechanisms can be modeled by adding hidden variables to the network, as described below, thus transforming the *logical-OR* into the *noisy-OR* [72].

Several generalizations of noisy-OR model were proposed in [111, 61, 62] and summarized in [63] under the collective name *causal independence (CI)*. Most generally, causal independence assumes that a probabilistic relation between a set of causes $c_1, ..., c_n$ and an effect e can be decomposed into a "noisy transformation" of each cause c_i into a *hidden* variable u_i, and a deterministic function $e = g(u_1, ..., u_n)$. Such structure can be captured by a *dependence graph* depicting the hidden variables explicitly (see Figure 3.5b).

A special kind of causal independence, called *decomposable causal independence* [63], implies that the function $g(u_1, ..., u_n)$ can be decomposed into a series of binary functions. Following [114], we assume that $g(u_1, ..., u_n) = u_1 * ... * u_n$, where $*$ is a commutative and associative binary operator (e.g., logical OR, logical AND, addition). This class of causally-independent relations includes several commonly used models, such as noisy-OR, noisy-MAX, noisy-AND, and noisy-adder (e.g., linear-Gaussian model). From now on, we assume decomposable causal independence calling it simply "causal independence". Formally,

Definition 6: [114] Let $c_1, ..., c_m$ be the parents of e in a belief network. The variables $c_1, ..., c_m$ are said to be *causally-independent* w.r.t. e (called *convergent* variable) if there exists a set of random variables $u_1^e, ..., u_m^e$ (called *hidden* variables), a set of conditional probability functions $f(u_i^e, c_i) = P(u_i^e|c_i)$, and a binary commutative and associative operator $*$ such that

1. for each i, u_i^e is independent of any of c_j and u_j^e, $i \neq j$, given c_i, i.e.
$$P(u_i^e|c_1, ..., c_n, u_1^e, ..., u_m^e) = P(u_i^e|c_i), \text{ and}$$
2. $e = u_1^e * \ldots * u_m^e$.

The family $e, c_1, ..., c_m$ will be called a *causally-independent family.* If all the families in a belief network are causally-independent, then the network will be called a *causally-independent belief network (CI-network).* A causally-independent network $BN = (G, P)$ over the set of random variables X is defined by a directed acyclic graph G (where the nodes correspond to variables in X) a set of functions $P = P_{CI} \cup P_{priors}$ over $X \cup U$, where U is the set of hidden variables, $P_{CI} = \{P(u_i^e|c_i)|e \in X, \; pa(e) \neq \emptyset, \; c_i \in pa(e)\}$ is the set of conditional probabilities introduced for each causally-independent family, and P_{priors} is the set of prior probabilities on the *root* nodes (nodes without parents).

The next section shows how causal independence can be used to decompose large families into smaller ones which allows computing CPT entries in linear time and can be further exploited by probabilistic inference algorithms.

3.4 Binary-tree network transformations

Given the input specification of a causally-independent family (i.e., the set of functions $\{P(u_i^e|c_i)|i = 1,...,m\}$), the CPT of a causally-independent family can be computed as

$$P(e|c_1,...c_m) = \sum_{\{u_1^e...u_m^e|e=u_1^e*...*u_m^e\}} \prod_{i=1}^{m} P(u_i^e|c_i). \tag{3.6}$$

Since the operation $*$ is commutative and associative, $e = u_1^e * ... * u_m^e$ can be decomposed into a sequence of pairwise computations, such as

$$e = u_1^e * y_1,\;\; y_1 = u_2^e * y_2,\;\; ...,\;\; y_{m-2} = u_{m-1}^e * u_m^e,$$

Then each CPT entry can be computed in $O(m)$ time as follows [114]:

$$P(e|c_1,...c_m) = \sum_{\{u_1^e...u_m^e|e=u_1^e*...*u_m^e\}} \prod_{i=1}^{m} P(u_i^e|c_i) =$$

$$\sum_{\{u_1^e,y_1|e=u_1^e*y_1\}} P(u_1^e|c_1)\dots \sum_{\{u_{m-1}^e,u_m^e|y_{m-2}=u_{m-1}^e*u_m^e\}} P(u_{m-1}^e|c_{m-1})P(u_m^e|c_m) = \tag{3.7}$$

$$\sum_{u_1^e,y_1} P'(e|u_1^e,y_1)P(u_1^e|c_1)\dots \sum_{u_{m-1}^e,u_m^e} P'(y_{m-2}|u_{m-1}^e,u_m^e)P(u_{m-1}^e|c_{m-1})P(u_m^e|c_m), \tag{3.8}$$

where y_j denote *hidden* variables introduced for keeping the intermediate results, and $P'(x|y,z)$ are new (deterministic) CPTs defined as follows: $P'(x|y,z) = 1$ if $x = y*z$, and 0 otherwise.

Clearly, there are many different ways of decomposing $e = u_1^e * ... * u_m^e$ into a sequence of binary operations, each corresponding to a traversal of some *binary computation tree.*

Definition 7: [binary computation tree]

Given an expression $e = u_1^e*...*u_m^e$ where $*$ is commutative and associative, its *binary computation tree* is a directed rooted binary tree having the root e and leaves u_i^e, where each non-leaf node y is pointed to by its two children[2] y_l and y_k and associated with the operation $y = y_l * y_k$.

---

[2] Note that the children of y in such directed binary tree will be called parents of y in the corresponding belief network.

This leads to the notion of a *binary-tree transformation* that decomposes a causally-independent family of size $m+1$ (Figure 3.5a) into an equivalent network having families of size at most 3, as we will show.

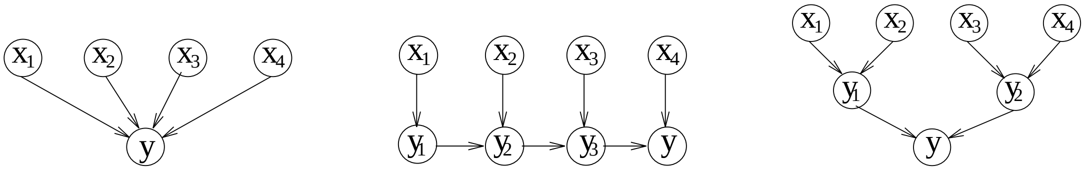
Definition 8: [binary-tree transformation]
Given a causally-independent family F having the parents c_1,...,c_m and the child e such that $e = u_1^e * ... * u_m^e$, its *dependence graph* includes the variables u_i^e explicitly as shown in Figure 3.5b. Given a causally-independent network $BN = (G, P)$, a *transformed graph* G_T of the BN is obtained by first replacing each family in G by its dependence graph, and then replacing the resulting family $e = u_1^e * ... * u_m^e$ with a binary computation tree (Figure 3.5c). The nodes of the BN are called *input* nodes, while both the nodes u_i^e introduced by causal independence and the nodes y_j introduced by binary computation trees will be called *hidden* nodes. A *binary-tree transformation* (a *transformed network*) of the $BN = (G, P)$ is a belief network $T_{BN} = (G_T, P, P')$ where G_T is a transformed graph, each u_i^e is associated with the function $P(u_i^e|c_i) \in P$, and each node x in a binary computation tree of some family, having parents y and z, is associated with $P'(x|y,z) \in P'$ defined as $P'(x|y,z) = 1$ if $x = y * z$, and 0 otherwise.

Equations 3.6-3.8 show that the causally-independent CPTs can be derived by belief updating in a transformed network (see equation 3.8). Moreover,

Theorem 12: [equivalence of belief updating in BN and T_{BN}]
Given a causally-independent belief network BN, a transformed network T_{BN}, and an evidence e, computing $P(x_1|e)$ over the BN is equivalent to computing $P(x_1|e)$ over the T_{BN}.

Proof: Given a belief network $BN = (G, P)$ over the set of variables $X = \{X_1, ..., X_n\}$, the task of updating the belief in X_1 given evidence e is to find

$$P(x_1|e) = \alpha \sum_{x_2,\ldots,x_n} \prod_i P(x_i|pa_i). \tag{3.9}$$



(a) (b) (c)

Figure 3.6: (a) A belief network, (b) a temporal transformation, and (c) a parent-divorcing transformation.

Assuming causal independence, we can generate a transformed network T_{BN} as discussed above. Let c_j^i denote the parents of x_i in G, and let u_j^i and y_k^i denote the hidden variables associated with the family of x_i in the T_{BN}. Then each $P(x_i|pa_i)$ in (3.9) can be replaced by the corresponding expression (3.8) yielding

$$\sum_{x_2,...,x_n} \prod_i \sum_{u_1^i, y_1^i} P'(e|u_1^i, y_1^i) P(u_1^i|c_1^i) \ldots \sum_{u_{m-1}^i, u_m^i} P'(y_{m-2}^i|u_{m-1}^i, u_m^i) P(u_{m-1}^i|c_{m-1}^i) P(u_m^i|c_m^i), \tag{3.10}$$

where $P'(x|y,z)$ are defined as $P'(x|y,z) = 1$ if $x = y * z$, and 0 otherwise. Namely,

$$P(x_1|e) = \alpha \sum_{X-\{X_1\}} \prod_i P(x_i|pa_i) = \alpha \sum_{x_1,...,x_n} \sum_H \prod_{z \in Z} P(z|pa(z)), \tag{3.11}$$

where $H = U \cup Y$ is the set of all the hidden variables $U = \{u_j^i | i = 1, ..., n\}$ and $Y = \{y_l^i | i = 1, ..., n\}$, contributed by causal independence and by binary computation trees, respectively, and where $Z = U \cup Y \cup X - \{X_1\}$ is the set of variables in the T_{BN}. Thus, computing $P(x_1|e)$ over BN (equation 3.9) is equivalent to computing $P(x_1|e)$ over T_{BN} (equation 3.11), which concludes the proof. □

The idea of network transformations was already proposed in [63] (*"temporal transformation"*) and [84] (*"parent divorcing"*). Our definition extends these approaches by including of the u_i^e variables associated with causal independence explicitly and by avoiding any commitment to a particular binary computation tree. Temporal transformations, for example, assume that causes are activated one after another at subsequent time points (thus the name "temporal"). Variables u_i^e are not specified explicitly but rather "hidden" within the new CPTs. For example, given a causally-independent network in Figure 3.6a, a possible temporal-transformation order is $y = ((u_1^y * u_2^y) * u_3^y) * u_4^y$ which yields

$$P(y|x_1, x_2, x_3, x_4) = \sum_{\{u_1^y, u_2^y, u_3^y, u_4^y | y = u_1^y * u_2^y * u_3^y * u_4^y\}} P(u_1^y|x_1)P(u_2^y|x_2)P(u_3^y|x_3)P(u_4^y|x_4) =$$

$$= \sum_{\{y_3, u_4^y | y = y_3 * u_4^y\}} \sum_{\{y_2, u_3^y | y_3 = y_2 * u_3^y\}} \sum_{\{y_1, u_2^y | y_2 = y_1 * u_2^y\}} P(y_1|x_1)P(u_2^y|x_2)P(u_3^y|x_3)P(u_4^y|x_4) =$$

$$= \sum_{y_3} \sum_{\{u_4^y | y = y_3 * u_4^y\}} P(u_4^y|x_4) \sum_{y_2} \sum_{\{u_3^y | y = y_2 * u_3^y\}} P(u_3^y|x_3) \sum_{y_1} \sum_{\{u_2^y | y_2 = y_1 * u_2^y\}} P(u_2^y|x_2)P(y_1|x_1).$$

If we define the new conditional probabilities for y_1, y_2, y_3:
$P(y_1|x_1) = P(u_1^y|x_1)$,
$P(y_2|y_1, x_2) = \sum_{\{u_2^y | y_2 = y_1 * u_2^y\}} P(u_2^y|x_2)$,
$P(y_3|y_2, x_3) = \sum_{\{u_3^y | y_3 = y_2 * u_3^y\}} P(u_3^y|x_3)$,
$P(y|y_3, x_4) = \sum_{\{u_4^y | y = y_3 * u_4^y\}} P(u_4^y|x_4)$,
then we get

$$P(y|x_1, x_2, x_3, x_4) = \sum_{y_3} P(y|y_3, x_4) \sum_{y_2} P(y_3|y_2, x_3) \sum_{y_1} P(y_2|y_1, x_2)P(y_1|x_1),$$

yielding the temporal transformation network in Figure 3.6b. The parent-divorcing method [84], on the other hand, would use the order $y = (u_1^y * u_2^y) * (u_3^y * u_4^y)$ which yields the transformed network in Figure 3.6c.

Clearly, network transformations are not deterministic: there are many feasible binary-tree transformation of each causally-independent family. As noted in [63], some transformations yield better performance than others. Finding a transformation that allows the best performance is generally hard. However, generating any particular transformation network is easy:

Theorem 13: [transformation complexity]

The complexity of transforming a causally-independent belief network BN into a transformed network T_{BN} and its size are $O(nmd^3)$, where n is the number of variables, m is the largest family size, and d is the domain size.

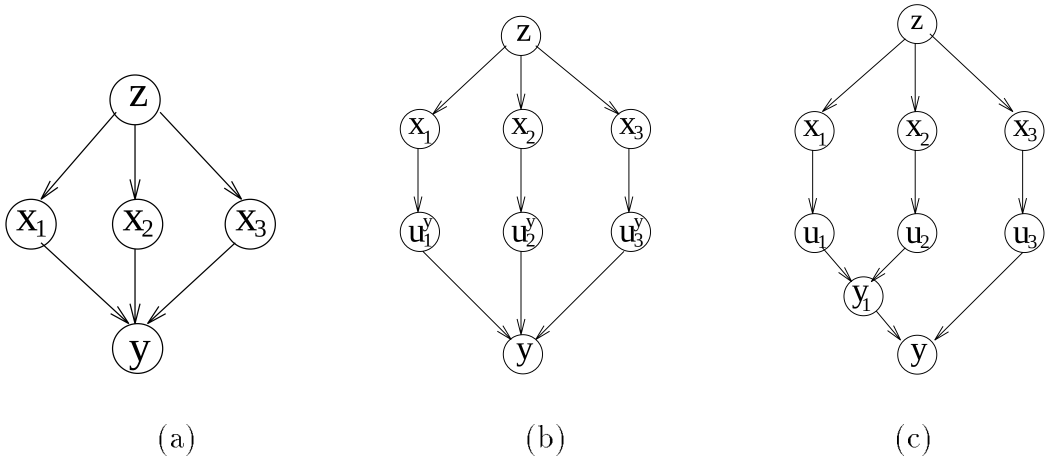


Figure 3.7: (a) A belief network, (b) its dependence graph, and (c) its transformed network.

Proof: For each causally-independent family having m parents, any of its binary-tree transformations introduces no more than $2m$ hidden variables (the number of nodes in a binary tree having m leaves). Consequently, there are no more than $2m$ new CPTs, each defined on at most 3 variables, yielding $O(md^3)$ time and space complexity. Since a T_{BN} has $O(nm)$ nodes and families of size at most 3, its specification requires $O(nmd^3)$ space. □

The next section shows how belief updating on transformed networks yields computational savings.

3.5 Belief updating in CI-networks

Once a transformed network is available, a standard inference technique such as join-tree clustering or a variable-elimination algorithm can be applied. Algorithms that exploit causal independence are called "causally-informed", while standard algorithms ignoring this assumption are called "causally-blind". The next few paragraphs demonstrate the application of *elim-bel* to the transformed network.

Example 8: Consider the *noisy-OR* network in Figure 3.7a. Causal independence is explicitly depicted by the dependence graph in Figure 3.7b. The query $P(x_3|x_1 = 0)$ can be computed as follows:

$$P(x_3|x_1 = 0) = \alpha \sum_{z,y,x_2,x_1=0} P(z)P(x_1|z)P(x_2|z)P(x_3|z)P(y|x_1, x_2, x_3) =$$

$$\alpha \sum_{z,y,x_2,x_1=0} P(z)P(x_1|z)P(x_2|z)P(x_3|z) \sum_{\{u_1^y,u_2^y,u_3^y|y=u_1^y\vee u_2^y\vee u_3^y\}} P(u_1^y|x_1)P(u_2^y|x_2)P(u_3^y|x_3).$$

Assuming now the transformed network in Figure 3.7c, we get

$$P(x_3|x_1 = 0) = \alpha \sum_z \sum_y \sum_{x_2} \sum_{x_1=0} P(z)P(x_1|z)P(x_2|z)P(x_3|z)\times$$

$$\times \sum_{\{y_1,u_3^y|y=y_1\vee u_3^y\}} \sum_{\{u_1^y,u_2^y|y_1=u_1^y\vee u_2^y\}} P(u_1^y|x_1)P(u_2^y|x_2)P(u_3^y|x_3).$$

Pushing the summation over x_1 and x_2 as far to the right as possible yields

$$P(x_3|x_1 = 0) = \alpha \sum_z P(z)P(x_3|z) \sum_y \sum_{\{y_1,u_3^y|y=y_1\vee u_3^y\}} P(u_3^y|x_3)\times$$

$$\times \sum_{\{u_1^y,u_2^y|y_1=u_1^y\vee u_2^y\}} \sum_{x_2} P(x_2|z)P(u_2^y|x_2) \sum_{x_1=0} P(x_1|z)P(u_1^y|x_1) =$$

$$= \alpha \sum_z P(z)P(x_3|z) \sum_y \sum_{y_1} \sum_{u_3^y} P'(y|y_1, u_3^y)P(u_3^y|x_3)\times$$

$$\times \sum_{u_1^y} \sum_{u_2^y} P'(y_1|u_1^y, u_2^y) \sum_{x_2} P(x_2|z)P(u_2^y|x_2) \sum_{x_1=0} P(x_1|z)P(u_1^y|x_1).$$

The last expression can be computed by summation from right to left along the ordering $o = (x_3,\ z,\ y,\ y_1,\ u_3^y,\ u_1^y,\ u_2^y,\ x_2,\ x_1)$ which indeed corresponds to applying algorithm *elim-bel* to the transformed network in Figure 3.7c along ordering o as follows:

1. **Bucket** x_1 : $\{x_1 = 0, P(x_1|z), P(u_1^y|x_1)\} \rightarrow$
   $h^{x_1}(u_1^y, z) = \sum_{x_1=0} P(x_1|z)P(u_1^y|x_1) \rightarrow$ put into bucket of u_1^y.

2. **Bucket** x_2: $\{P(x_2|z), P(u_2^y|x_2)\} \rightarrow$
   $h^{x_2}(u_2^y, z) = \sum_{x_2} P(x_2|z)P(u_2^y|x_2) \rightarrow$ put into bucket of u_2^y.

3. **Bucket** u_2^y: $\{h^{x_2}(u_2^y, z), P'(y_1|u_1^y, u_2^y)\} \rightarrow$
   $h^{u_2^y}(u_1^y, y_1, z) = \sum_{u_2^y} P'(y_1|u_1^y, u_2^y) h^{x_2}(u_2^y, z) \rightarrow$ put into bucket of u_1^y.

4. **Bucket** u_1^y: $\{h^{x_1}(u_1^y, z), h^{u_2^y}(u_1^y, y_1, z)\} \rightarrow$
   $h^{u_1^y}(y_1, z) = \sum_{u_1^y} h^{x_1}(u_1^y, z) h^{u_2^y}(u_1^y, y_1, z) \rightarrow$ put into bucket of y_1.

5. **Bucket** u_3^y: $\{P(u_3^y|x_3), P'(y|y_1, u_3^y)\} \rightarrow$
   $h^{u_3^y}(y, y_1, x_3) = \sum_{u_3^y} P(u_3^y|x_3) P'(y|y_1, u_3^y) \rightarrow$ put into bucket of y_1.

6. **Bucket** y_1: $\{h^{u_1^y}(y_1, z), h^{u_3^y}(y, y_1, x_3)\} \rightarrow$
   $h^{y_1}(y, z, x_3) = \sum_{y_1} h^{u_1^y}(y_1, z) h^{u_3^y}(y, y_1, x_3) \rightarrow$ put into bucket of y.

7. **Bucket** y: $\{h^{y_1}(y, z, x_3)\} \rightarrow$
   $h^y(x_3, z) = \sum_y h^{y_1}(y, z, x_3) \rightarrow$ put into bucket of z.

8. **Bucket** z: $\{P(z), P(x_3|z), h^y(x_3, z)\} \rightarrow$
   $h^z(x_3) = \sum_z P(z) P(x_3|z) h^y(x_3, z) \rightarrow$ put into bucket of x_3.

9. **Bucket** x_3: $\{h^z(x_3)\} \rightarrow$
   $P(x_3|x_1 = 0) = \alpha h^z(x_3)$, where α is a normalizing constant.

The complexity of the computation in each bucket is $O(d^3)$ since the arity of each function in a bucket is not larger than 3. In contrast, the complexity of *elim-bel* applied to the input belief network (Figure 3.7a) is $O(nd^4)$, where n is the number of variables.

```
Algorithm ci-elim-bel(BN,e)

Input: A belief network $BN = (G, P)$, and evidence $e$.
Output: $P(x_1|e)$.
1. Generate a transformed network $T_{BN}$.
2. Return elim-bel($T_{BN}$,o,e), where $o$ is some ordering of the $T_{BN}$.
```

Figure 3.8: Algorithm ci-elim-bel

The causally-informed algorithm *ci-elim-bel* that first generates a transformed network and then applies algorithm *elim-bel* is presented in Figure 3.8.

3.5.1 Complexity analysis

This section makes a few observations regarding the performance gain when exploiting causal independence for belief updating. Recall that G^M denotes the moral graph of a belief network $BN = (G, P)$, G_T denotes the graph of a transformed network, and G_T^M denotes its moral graph. We will use "the induced width of the network" as a shorthand for "the induced width of the network's graph". As an immediate implication of theorems 11 and 13,

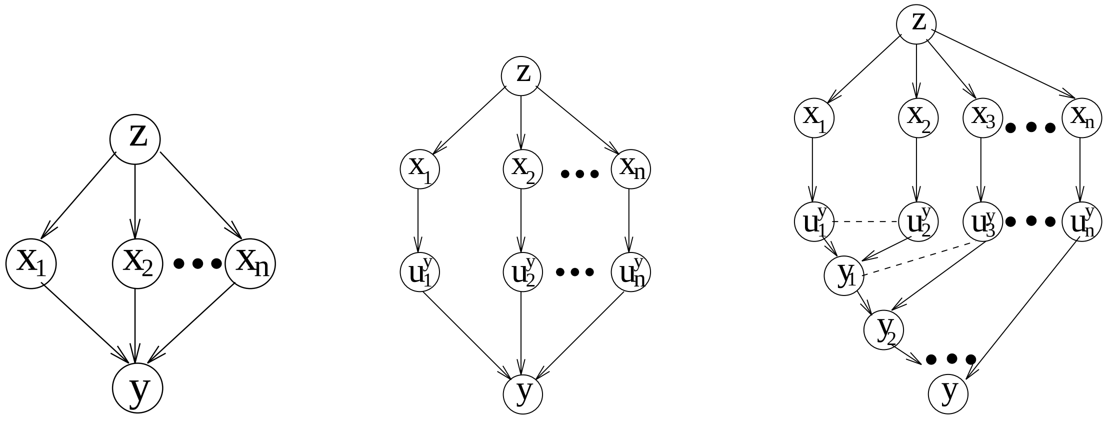
Corollary 3: [complexity of *ci-elim-bel* on a T_{BN}]
Given a causally-independent belief network BN and a transformed network T_{BN}, the complexity of elim-bel *on T_{BN} and therefore the complexity of* ci-elim-bel *is $O(nmd^{w_o^*+1})$, where w_o^* is the induced width of G_T^M along o.* □

The question is how $w^*(G_T^M)$ which characterizes the performance of *elim-bel* on a transformed network compares to $w^*(G^M)$ that determines the complexity of *elim-bel* on the original network. We define the *effective induced width* w_e^* as $w_e^* = min_{G_T} w^*(G_T^M)$. We next present several classes of causally-independent networks that allow significant reduction in the effective induced width.

Performance gains due to causal indpendence

Theorem 14: [polytrees]
Given a causally-independent poly-tree BN having n nodes, domains of size d, and no more than m parents in each family, the complexity of ci-elim-bel *on the BN is $O(nmd^3)$.*

Proof: Any transformed network of a polytree is a polytree having families of size 3 or smaller. It therefore allows an ordering having $w^* = 2$ (starting with query



(a) (b) (c)

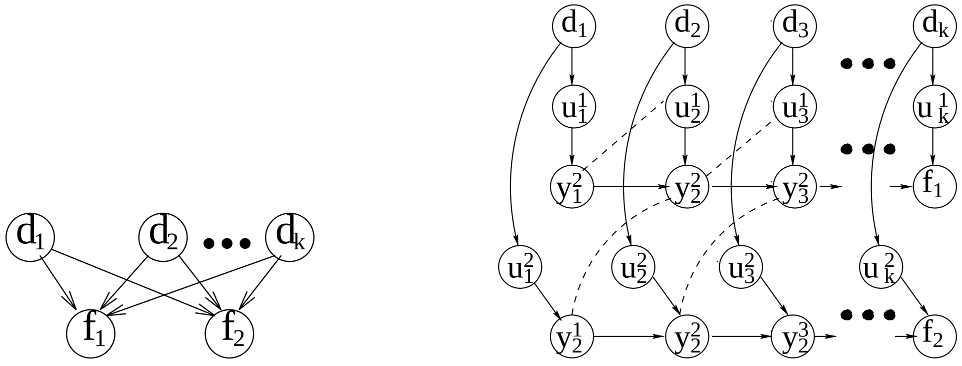
Figure 3.9: (a) A belief network, (b) its dependence graph, and (c) its moral transformed network.

nodes). Therefore, the complexity of *elim-bel* on any T_{BN} (and thus the complexity of *ci-elim-bel*) is $O(nmd^3)$. □

Theorem 14 generalizes Pearl's result for noisy-OR polytrees [89]. Remember that the standard causally-blind polytree algorithm is exponential (namely, $O(nd^m)$) in size of a largest parent set. The following example introduces another class of multiply-connected causally-independent networks that allow linear complexity.

Example 9: ["Source-sink" networks]

The causally-independent network BN in Figure 3.9a has a "source" node z and "sink" node y, and a middle layer of nodes x_1,...,x_n, such that there is an edge from z to each x_i, and from each x_i to y. Given the transformed network in Figure 3.9c, and the ordering $o = (z, y, x_n, ..., x_2, x_1)$ over the BN, the induced width $w^*_o(G)$ of the original network's graph is 2. The reader can check that the induced width $w^*_{o'}(G^M_T)$ of the moral transformed network along the ordering $o' = (z, y, y_{n-1}, ..., y_2, y_1, u^y_1, u^y_2, ..., u^y_n, x_1, x_2, ..., x_n)$ is also 2. Note that the induced width $w^*(G^M)$ of the moral BN is n. Thus, while the causally-blind representation and the corresponding inference algorithms require $O(d^n)$ time and space, exploiting causal independence reduces the



(a) (b)

Figure 3.10: (a) A k-2-network BN and (b) a moral graph of a transformed network T_{BN}.

complexity to $O(nd^3)$.

The examples above demonstrated that the effective induced width of a causally-independent network can be as small as the induced width $w^*(G)$ of its (unmoral) directed acyclic graph. The following result identifies a general class of such networks.

Theorem 15: [best-case complexity]

If convergent variables (i.e. children in causally-independent families) in a network $BN = (G, P)$ never have more than one common parent (e.g., Figure 3.9a, but not 3.10a), then for every transformed network T_{BN}, $w^*(G_T^M)$ equals the induced width of the (unmoral) graph G.

Proof: See Appendix B. □

On the other hand, $w^*(G)$ is also a lower bound on the effective induced width:

Theorem 16: *For every belief network $BN = (G, P)$, $w_e^* \geq w^*(G)$.*

Proof: Assume the opposite and let o be an ordering of some transformed network yielding an induced width $w = w_o^*(G_T^M)$, where $w < w^*(G)$. Then the restriction o' of o to the input nodes must have an induced width $w' = w_{o'}^*(G)$ that is not larger than w, thus leading to the contradiction: $w_{o'}^*(G) < w^*(G)$, but $w^*(G) = min_o w_o^*(G)$. □

Thus, *the best we can expect is that the effective induced width would equal the induced width of the (unmoral) directed acyclic graph.* Another example of network structures that can greatly benefit from exploiting causal independence are *k-n-networks.*

Example 10: [k-n-networks]
Figure 3.10 depicts a *k-n-network*, which is a two-layer network with k nodes in the top layer, each connected to n nodes in the bottom layer (here $n = 2$). An example is a *binary-node 2-layer noisy-OR (BN2O)* network used in medical-diagnosis QMR-DT system [90], where the top layer nodes, d_i, represent diseases that cause findings f_j in the bottom layer. Clearly, the induced width of the moral *k-2-network* in Figure 3.10a is at least k (the number of parents). Consider now the moral graph of the transformed network in Figure 3.10b, along orderings where parents appear after their children, such as $o = (f_1, f_2,\ y^1_{k-1},\ y^2_{k-1},\ \dots\ ,\ y^1_1,\ y^2_1,\ d_k,\ \dots,\ d_1,\ u^1_1,\ \dots,\ u^1_k,\ u^2_1,$ $\dots,\ u^2_k)$. Each u^1_j for $j > 1$ has three preceding neighbors in o, d_j, y^1_{j-1}, and y^1_j, thus $w^*_o(u^1_j) = 3$; eliminating u^1_j connects d_j, y^1_{j-1}, and y^1_j. Similarly, for $k > 1$, $w^*_o(u^2_k) = 3$ and eliminating each u^2_k connects d_k, y^2_{k-1}, and y^2_k. Since d_1 is now connected to y^1_1 and y^2_1, its induced width is 2, and eliminating d_1 connects y^1_1 and y^1_2. Every d_i for $i \geq 2$ is now connected to y^{i-1}_1, y^{i-1}_2, y^i_1, and y^i_2 in the induced graph of G^M_T in Figure 3.10b, so that $w^*_o(d_i) = 4$. Eliminating all d_i creates a chain of cliques of size 4 (each defined on y^{i-1}_1, y^{i-1}_2, y^i_1, and y^i_2). Subsequently, $w^*_o(y^2_j) = 3$ and $w^*_o(y^1_j) = 2$ for $j = 1, \dots, k-1$. It is easy to see that the induced width of the transformed network along o is constant ($w^* = 4$) rather than linear in k, yielding an exponential speed-up. This example generalizes to an arbitrary *k-n-network* yielding $w^*_o(G^M_T) = 2n$ along any ordering o where parents appear after their children. It can be therefore shown that

Theorem 17: [k-n-networks]
The complexity of algorithm ci-elim-bel *applied to a causally-independent k-n-network is* $O(d^{min\{k,2n\}})$.

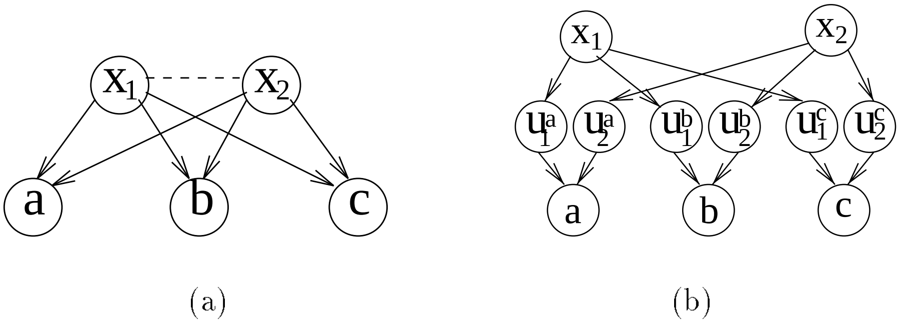


Figure 3.11: (a) A 2-3 network and (b) its transformed network.

Proof: See Appendix B. □

Therefore, when $k > 2n$, exploiting causal independence can exponentially decrease the complexty of belief updating, while for $k \leq 2n$ causally-blind algorithms (yielding $w^* = k$) are best.

Some pitfalls of network transformations

All the previous examples exploited causal independence by processing parents in a family before their hidden variables. However, this ordering is not always good. Sometimes it is best to ignore causal independence and reconstruct some of the original CPTs by eliminating the hidden variables first.

Example 11: Consider the 2-3-network in Figure 3.11a, the transformed network T_{BN} in Figure 3.11b, and the ordering $o = (a, b, c,\ u_1^c,\ u_2^c,\ u_1^b,\ u_2^b,\ u_1^a,\ u_2^a,\ x_1,\ x_2)$. Eliminating each parent x_i interconnects u_i^a, u_i^b, and u_i^c; eliminating u_2^a , and then u_1^a, connects a to the rest of the hidden variables. As a result, $w_o^*(T_{BN}) = 5$. However, the induced width of the original moral graph (Figure 3.11a) along a restricted ordering $o' = (a,\ b,\ c,\ x_1,\ x_2)$, is just 3. Indeed, a different ordering of the transformed network where the hidden variables are eliminated first has also induced width of 3. This ordering, however, restores the original CPTs. In other words, for an arbitrary *k-n-network*, restoring the original CPTs yielding the effective induced width k is preferrable when $k < 2n$ (see theorem 17). As we can see, careless use of causal independence (namely, selecting a bad ordering of the transformed network) may actually increase the complexity of inference.

It can be shown that, although a transformed network includes new (hidden) nodes, no new edges are induced between the input nodes:

Theorem 18: [w^* **on the input variables]**
Given a causally-independent belief network $BN = (G, P)$ *having induced width* $w^*_o(G)$ *along* o*, and given a transformed network* T_{BN}*, there exists an extension of* o *to* o' *such that the induced width of the input variables in* T_{BN} *along* o'*, computed only with respect to the edges induced between the input variables, is not larger than* $w^*_o(G)$.

Proof: See the Appendix B. □

Therefore, the effective induced width may increase only because of new induced edges connected to at least one hidden node; thus we focus on a proper ordering of hidden nodes. Figure 3.12 presents an "order-correction" procedure for constructing an appropriate ordering of the hidden variables which guarantees that *ci-elim-bel* is never worse than the causally-blind algorithm. The procedure starts with some initial ordering of the transformed network and then, if necessary, restores some CPTs, so that the w^*_e is never larger than the $w^*(G^M)$. In summary,

Corollary 4: *Given a belief network* $BN = (G, P)$*, then for every transformed network* $T_{BN} = (G_T, P, P')$ *the induced width* $w^*(G^M_T)$ *satisfies*

$$w^*(G) \leq w^*(G^M_T) \leq w^*(G^M).$$

3.5.2 The connection between previous approaches

We next identify ordering restrictions that reduce *ci-elim-bel* to the previously proposed network transformation approaches of [63, 84] and to the variable-elimination algorithm VE1 [114].

```
Procedure: Order(BN, $T_{BN}$)
Input:  a network BN, and a transformed network $T_{BN}$.
Output: an ordering $o_T$ of $T_{BN}$ s.t. $w^*_{o_T} \leq w^*_{o_I}$,
        where $w^*_{o_I}$ is the restriction of $o_T$ to the variables of BN.
1. find a good heuristic ordering $o_T$ of the transformed network.
   Let $o_I$ be the restriction of $o_T$ to BN.
2. For each variable $x$ in $o_T$, going from last to first,
       If $w^*_{o_T}(x) > w^*_{o_I}$:
           If $x$ is an input variable, move all hidden variables $y_i^x$
           related to $x$ to the top of $o_T$.
           Else if $x = y_i^z$ is a hidden variable of some convergent variable $z$,
           put all $z$'s hidden variables at the top of $o_T$.
3. Return $o_T$.
```

Figure 3.12: Procedure *order*.

Temporal transformations and parent-divorcing

As noted in section 3.3, the only difference between our notion of binary-tree network transformations and the previously proposed transformations (temporal [63] and parent divorcing [84]) is that we explicitly specify the variables u_i^e contributed by causal independence. Let us denote by TT_{BN} and PD_{BN} an arbitrary temporal transformation and a parent-divorcing transformation of a belief network BN, respectively, while G^M_{TT} and G^M_{PD} denote their moral graphs. Clearly, there is a one-to-one correspondence between TT_{BN} or PD_{BN} and the binary-tree transformation T_{BN} obtained by inserting the additional layer of hidden nodes u_i^e into each family of TT_{BN} or PD_{BN}.

Proposition 1: [ordering restrictions of network transformations]
Given a belief network BN, applying elim-bel *to TT_{BN} or to PD_{BN} along o is equivalent to applying* elim-bel *to the corresponding transformed network T_{BN} along ordering $o' = (o, u)$, where u is some ordering of the hidden variable u_i^x.* $\Box$

Theorem 19: [w^* of TT_{BN} and PD_{BN}]
Given a TT_{BN} (respectively, a PD_{BN}), and its corresponding T_{BN}, then for any

ordering o $w_o^*(G_{TT}^M) = w_{o'}^*(G_T^M)$ *(respectively,* $w_o^*(G_{PD}^M) = w_{o'}^*(G_T^M)$*) where* $o' = (o, u)$ *extends o to* T_{BN} *and u is some ordering of the hidden variable* u_i^x*.*

Proof: Since variables u_i^e in T_{BN} are not connected to each other, and are connected to exactly two other variables, and since they appear last in the ordering (eliminated first), the width and the induced width of each u_i^e is 2 (see Figure 3.5c), which does not exceed $w_o^*(TT_{BN})$. Same holds for PD_{BN}. □

Algorithm VE1

A variable-elimination belief-updating algorithm VE1 for causally-independent networks [114] was derived directly from the factorization of the joint probability distribution (expression 3.6). The algorithm takes as an input a causally-independent belief network and an ordering of the input variables. The hidden variables are eliminated implicitly using an operator $\otimes$ [114].

Remember that the child in a causally-independent family is called a *convergent* variable. The operator $\otimes$ is defined on two functions f and g as follows [114]:

$$f \otimes g(e_1 = \alpha_1, ..., e_k = \alpha_k, Y) =$$

$$\sum_{\{\alpha_1^1, \alpha_1^2 | \alpha_1^1 * \alpha_1^2 = \alpha_1\}} \cdots \sum_{\{\alpha_k^1, \alpha_k^2 | \alpha_k^1 * \alpha_k^2 = \alpha_k\}} f(e_1 = \alpha_1^1, ..., e_k = \alpha_k^1, Y_1) \cdot g(e_1 = \alpha_1^2, ..., e_k = \alpha_k^2, Y_2),$$

where $e_1, ..., e_k$ are convergent variables that appear both in f and g, and $Y = Y_1 \cup Y_2$ is the set of all the other variables appearing as arguments of f and g.

We now rederive VE1 for a network in Figure 3.11a. Assume that the task is to find the belief in c in the absence of evidence (i.e., to find the marginal $P(c)$). Using notation $f^i(e, c_i) = P(u_i^e, c_i)$ and the operator $\otimes$, the expression for $P(c)$ is

$$\sum_b \sum_a \sum_{\{u_1^a, u_2^a | a = u_1^a \vee u_2^a\}} \sum_{\{u_1^b, u_2^b | b = u_1^b \vee u_2^b\}} \sum_{\{u_1^c, u_2^c | c = u_1^c \vee u_2^c\}} \sum_{x_2} P(u_2^a | x_2) P(u_{x_2}^b | x_2) P(u_{x_2}^c | x_2) \times$$

$$\times \sum_{x_1} P(u_1^a | x_1) P(u_2^b | x_2) P(u_1^c | x_1) = \tag{3.12}$$

$$\sum_b \sum_a \sum_{\{u_1^a, u_2^a | a = u_1^a \vee u_2^a\}} \sum_{\{u_1^b, u_2^b | b = u_1^b \vee u_2^b\}} \sum_{\{u_1^c, u_2^c | c = u_1^c \vee u_2^c\}} f(u_2^a, u_{x_2}^b, u_{x_2}^c) g(u_1^a, u_2^b, u_1^c) \tag{3.13}$$

can be written as follows:

$$P(c) = \sum_b \sum_a (\sum_{x_2} f_a^2(a, x_2) f_b^2(b, x_2) f_c^2(c, x_2)) \otimes (\sum_{x_1} f_a^1(a, x_1) f_b^1(b, x_1) f_c^1(c, x_1)). \tag{3.14}$$

VE1 computes this expression from right to left along a given ordering of the input variables, and can be written as a bucket-elimination algorithm, where hidden variables are eliminated implicitly inside the buckets of input variables. In each bucket, VE1 applies the operator $\otimes$ to the appropriate functions first, and then sums over the bucket's variable:

1. **Bucket** x_1 : $\{f_a^1(a, x_1), f_b^1(b, x_1), f_c^1(c, x_1)\} \rightarrow$
   $h^{x_1}(a, b, c) = \sum_{x_1=0} f_a^1(a, x_1) f_b^1(b, x_2) f_c^1(c, x_1) \rightarrow$ put in bucket of a.

2. **Bucket** x_2 : $\{f_a^1(a, x_2), f_b^2(b, x_2), f_c^2(c, x_2)\} \rightarrow$
   $h^{x_2}(a, b, c) = \sum_{x_1=0} f_a^2(a, x_2) f_b^2(b, x_2) f_c^2(c, x_2) \rightarrow$ put in bucket of a.

3. **Bucket** a : $\{h^{x_1}(a, b, c), h^{x_2}(a, b, c)\} \rightarrow$
   3.1. $h(a, b, c) = h^{x_1}(a, b, c) \otimes h^{x_2}(a, b, c)$
   3.2. $h^a(b, c) = \sum_a h(a, b, c) \rightarrow$ put in bucket of b.

4. **Bucket** b : $\{h(a, b, c)\} \rightarrow h^b(c) = \sum_b h^a(b, c) \rightarrow$ put in bucket of c.

5. **Bucket** c : $\{h^b(c)\} \rightarrow P(c) = \alpha h^b(c)$, where α is a normalizing constant.

The computations above can be viewed as applying *elim-bel* to the transformed network in Figure 3.11b along o' that agrees with $o = (c, b, a, x_2, x_1)$ on the input nodes and satisfies in addition certain constraints imposed by operator $\otimes$ (to be discussed shortly). In our example the appropriate ordering $o' = (c, b, a, u^c_{x_1}, u^c_{x_2}, u^b_{x_1}, u^b_{x_2}, u^a_{x_1}, u^a_{x_2}, x_1, x_2)$ is used (note that the operation $h^{x_1}(a, b, c) \otimes h^{x_2}(a, b, c)$ in step 3.1 is equivalent to summing over $\{u^c_1, u^c_2\}$, $\{u^b_1, u^b_2\}$, and $\{u^a_1, u^a_2\}$).

The implicit ordering restrictions of VE1 may sometimes worsen the algorithm's performance. In the example above, the complexity of VE1 along $o = (c, b, a, x_1, x_2)$

is $O(d^6)$ since in expression 3.13 the operator $\otimes$ computes a product of two functions f and g defined on 6 variables (the resulting function corresponds to a clique on 6 hidden variables in the induced graph of the transformed network). On the other hand, *elim-bel* applied along another ordering $(c, b, a, x_1, x_2,\ u^c_{x_1},\ u^c_{x_2},\ u^b_{x_1},\ u^b_{x_2},\ u^a_{x_1},\ u^a_{x_2})$ first restores the original CPTs $P(a|x_1, x_2)$, $P(b|x_1, x_2)$, and $P(c|x_1, x_2)$ yielding $O(d^5)$ complexity. Since VE1 is unable to eliminate hidden variables independently of input variables (e.g., u^a_1 and u^a_2 cannot be eliminated before x_1 and x_2), it is inferior to *elim-bel*. Formally,

Theorem 20: [ordering restrictions of VE1]

Given a belief network $BN = (G, P)$ and its ordering o, there exists a $T_{BN} = (G_T, P, P')$ such that VE1 along o is equivalent to elim-bel applied to T_{BN} along o' satisfying the following conditions: 1. o' agrees with o on input nodes; 2. a hidden node u in the T_{BN} appear in o' before (is eliminated after) some input node x connected to u in the induced graph of G_T along o'; 3. o' must agree with a depth-first traversal order of the binary-tree of each family F.

Proof: The sequence of VE1's computations corresponds to variable-elimination in some transformed network T_{BN}. VE1 operates over the input variables; hidden variables are eliminated implicitly using operator $\otimes$ within the buckets of the input variables (in each bucket, $\otimes$ is applied first, then the bucket's variable is summed out). Therefore, the ordering of the input variables is preserved, i.e. condition 1 is satisfied. Condition 2 is also obvious: to be eliminated by $\otimes$, a hidden variable u must appear in the bucket of some input node x; this can happen only if u and x appear as arguments of same function, either originally defined in the transformed network T_{BN}, or recorded by VE1. In both cases, u and x must be connected in the induced graph of T_{BN}. Finally, condition 3 follows from the definition of the operator $\otimes$. It is easy to see that using $\otimes$ dynamically constructs a binary computation tree and its traversal ordering, since each summation $\sum_{\{y_l, y_k | y = y_l * y_k\}}$ eliminates a pair (y_l, y_k) of hidden variables and "creates" a new hidden variable y. This implies a *depth-first* traversal order of the emerging binary computation tree. □

3.5.3 Summary

The impact of causal independence on belief updating can be summarized as follows:

1. The two belief-updating schemes exploiting causal independence, such as network transformations [63, 84], and algorithm VE1 [114], can be viewed as applying algorithm *ci-elim-bel* along a specific variable ordering of a transformed network.

2. On polytrees, causally-informed algorithms can decrease the complexity of inference from $O(Nd^m)$ to $O(Nmd^3)$, where N is the number of nodes in the polytree, d is the domain size, and m bounds the parent set size.

3. Exploiting causal independence is most effective when $w^*(G_T^M) = w^*(G)$.

4. Exploiting causal independence in *k-n-networks* yields $O(d^{min\{k,2n\}})$ complexity resulting in exponential savings when $k > 2n$.

5. Algorithm *ci-elim-bel* may be sometimes worse than *elim-bel* unless the ordering is carefully chosen. An ordering-correction procedure that avoids "bad" orderings was presented.

6. Due to its ordering restrictions, VE1 cannot use the ordering-correction procedure and may sometimes be exponentially worse than *ci-elim-bel.*

3.6 Optimization tasks: MPE, MAP, and MEU

In optimization problems, it is not always possible to take advantage of causal independence since maximization and summation cannot be permuted, namely:

$$\max_x \sum_y f(x,y) \neq \sum_y \max_x f(x,y).$$

This restriction may not allow orderings that exploits CPT decomposition.

Consider the task of finding a *most probable explanation (MPE)*:

$$MPE = \max_{x_1,\ldots,x_N} \prod_i P(x_i|pa_i) = \max_{x_1} F_1 \ldots \max_{x_N} F_N,$$

where $F_i = \prod_x P(x|pa(x))$ is the product of all probabilistic components such that either $x = x_i$, or $x_i \in pa(x)$. The bucket elimination algorithm *elim-mpe* sequentially eliminates x_i from right to left. For example, given the causally-independent family in Figure 3.5a, having 3 parents c_1, c_2 and c_3,

$$MPE = \max_{c_1,c_2,c_3,e} P(c_1)P(c_2)P(c_3)P(e|c_1,c_2,c_3) =$$

$$= \max_{c_1} P(c_1) \max_{c_2} P(c_2) \max_{c_3} P(c_3) \max_{e} \sum_{\{y_1,u_1^e|e=y_1*u_1^e\}} P(u_1^e|c_1)\times$$

$$\times \sum_{\{u_2^e,u_3^e|y_1=u_2^e*u_3^e\}} P(u_2^e|c_2)P(u_3^e|c_3).$$

While belief updating task uses only the commutative and associative summation operation, the MPE task involves both maximization and summation that cannot be permuted. Therefore, the hidden variables must be summed out before maximizing over c_1, c_2, and c_3. This reconstructs the CPT on the whole family, so that causal independence has no effect.

Nevertheless, the two other optimization tasks, MAP and MEU, can still benefit from causal independence, because they involve summation over a subset of the input variables which can be permuted with summations over the hidden variables. By definition,

$$MAP = \max_{x_1,\ldots,x_m} \sum_{x_{m+1},\ldots,x_N} \prod_i P(x_i|pa_i),$$

where $X_1, ..., X_k$ are the *hypothesis* variables. For example, given the network in Figure 3.5a and hypothesis e,

$$MAP = \max_{e} \sum_{c_1,c_2,c_3} P(c_1)P(c_2)P(c_3)P(e|c_1,c_2,c_3).$$

Decomposing the causally-independent $P(e|c_1, c_2, c_3)$ and rearranging the summation order yields:

$$\max_e \sum_{c_1} P(c_1) \sum_{c_2} P(c_2) \sum_{c_3} P(c_3) \sum_{\{y_1, u_1^e | e = y_1 * u_1^e\}} P(u_1^e|c_1) \sum_{\{u_2^e, u_3^e | y_1 = u_2^e * u_3^e\}} P(u_2^e|c_2) P(u_3^e|c_3) =$$

$$= \max_e \sum_{\{y_1, u_1^e | e = y_1 * u_1^e\}} \sum_{\{u_2^e, u_3^e | y_1 = u_2^e * u_3^e\}} \sum_{c_1} P(c_1) P(u_1^e|c_1) \sum_{c_2} P(c_2) P(u_2^e|c_2) \sum_{c_3} P(c_3) P(u_3^e|c_3).$$

Clearly, the summations over the parents c_i and the corresponding hidden variables can be permuted. In general, a decomposition of the CPTs due to causal independence can be exploited when (at least some) parents in the causally-independent family are not included in the hypothesis. Clearly, the MAP task defined on a causally-independent network can be formulated as the same task on a transformed network, where the set of hypothesis variables remains the same.

Algorithm *ci-elim-map* is shown in Figure 3.13. Given a belief network BN, the algorithm first computes a transformed network T_{BN} where the families having a hypothesis variable as a child are not transformed. Then variable-elimination algorithm *elim-map*[23] for finding MAP is applied to the T_{BN} along an ordering o that starts with the hypothesis variables.

Algorithm ci-elim-map(BN,e,H)

Input: A belief network $BN = (G, P)$, and evidence e,
and a subset of hypothesis variables $H \subseteq X$.
Output: An assignment $h = \arg\max_a P(H = a|e)$.
1. Compute a T_{BN} s.t. the families where the child is
a hypothesis variable are not transformed.
2. Return *elim-map*(T_{BN},o,e,H), where o is an ordering
of T_{BN} starting with the variables in H.

Figure 3.13: Algorithm ci-elim-map

Similarly to MAP, computing MEU requires summation over a subset of the variables and maximization over the rest of them. By definition,

$$MEU = \max_{x_1,...,x_m} \sum_{x_{m+1},...,x_N} \prod_i P(x_i|pa_i) U(x_1, ..., x_N),$$

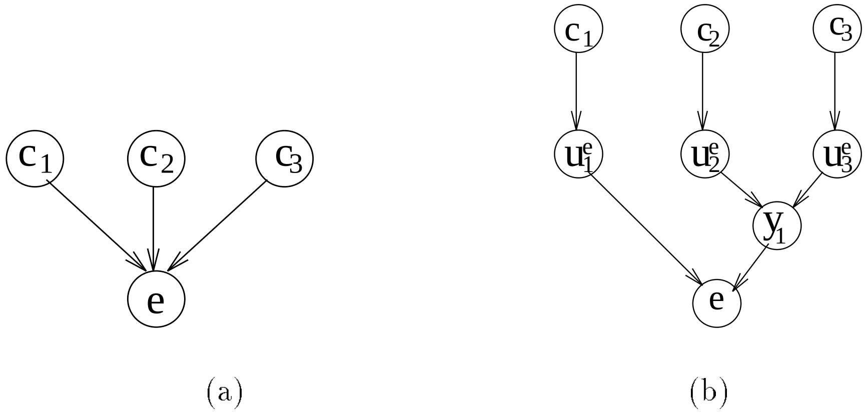


Figure 3.14: (a) a causally-independent belief network and (b) its transformed graph.

where $x_1, ..., x_m$ are *decision* variables (usually denoted $d_1, ..., d_m$), $x_{m+1}, ..., x_N$ are *chance* variables, and $U(x_1, ..., x_N)$ is a *utility function*. The utility is often assumed to be *decomposable*: $U(x_1, ..., x_N) = \sum r(x_i)$, where $r(x_i)$ are individual utilities, or *rewards*. A belief network with decision nodes and a utility function is also called an *influence diagram*.

A bucket elimination algorithm *elim-meu* for computing the MEU was presented in [23]. It makes a simplifying assumption that *the decision variables have no parents*, and they can be placed at the beginning of the ordering. We will now show how causal independence can be exploited using the following example.

Assume that c_1 in Figure 3.14a is a decision variable (denote $d = c_1$), and that the utility function is decomposable, $U(d, e, c_2, c_3) = r(d) + r(e) + r(c_2) + r(c_3)$. Then

$$MEU = \max_d \sum_{e,c_2,c_3} P(c_2)P(c_3)P(e|d, c_2, c_3)U(d, e, c_2, c_3).$$

Decomposing $P(e|d, c_2, c_3)$ into pairwise sums according to some binary-tree transformation yields:

$$\max_d \sum_{e,c_2,c_3} P(c_2)P(c_3) \sum_{\{y_1, u_1^e | e = y_1 * u_1^e\}} P(u_1^e|d) \sum_{\{u_2^e, u_3^e | y_1 = u_2^e * u_3^e\}} P(u_2^e|c_2)P(u_3^e|c_3)U(d, e, c_2, c_3).$$

Pushing the summations over c_2 and c_3 to the right and decomposing the sums over the pairs of hidden variables using new CPTs $P'(x|y,z)$ yields

$$\max_d \sum_e \sum_{y_1} \sum_{u_1^e} P'(e|y_1,u_1^e) P(u_1^e|d) \sum_{u_2^e} \sum_{u_3^e} P'(y_1|u_2^e,u_3^e) \sum_{c_2} P(c_2)P(u_2^e|c_2) \times$$

$$\times \sum_{c_3} P(c_3)P(u_3^e|c_3)U(d,e,c_2,c_3),$$

where $P'(e|y_1,u_1^e) = 1$ if $e = y_1 * u_1^e$ and 0 otherwise, and $P'(y_1|u_2^e,u_3^e) = 1$ if $y_1 = u_2^e * u_3^e$ and 0 otherwise. Using decomposability of the utility function, the summation over c_3 can be written as

$$\sum_{c_3} P(c_3)P(u_3^e|c_3)[r(d)+r(e)+r(c_2)+r(c_3)] =$$

$$= [\sum_{c_3} P(c_3)P(u_3^e|c_3)](r(d)+r(e)+r(c_2)) + \sum_{c_3} P(c_3)P(u_3^e|c_3)r(c_3) =$$

$$= [\sum_{c_3} P(c_3)P(u_3^e|c_3)][r(d)+r(e)+r(c_2)+\frac{\sum_{c_3} P(c_3)P(u_3^e|c_3)r(c_3)}{\sum_{c_3} P(c_3)P(u_3^e|c_3)}] =$$

$$= \lambda^{c_3}(u_3^e)[r(d)+r(e)+r(c_2)+\mu^{c_3}(u_3^e)],$$

where

$$\lambda^{c_3}(u_3^e) = \sum_{c_3} P(c_3)P(u_3^e|c_3), \mu^{c_3}(u_3^e) = \frac{\theta_{c_3}(u_3^e)}{\lambda^{c_3}(u_3^e)}, \text{ and } \theta^{c_3}(u_3^e) = \sum_{c_3} P(c_3)P(u_3^e|c_3)r(c_3).$$

The next summation (over c_2) can be computed similarly, and so is the next one over c_1.

Algorithm *ci-elim-meu* for computing MEU in causally-independent networks is shown in Figure 3.15. Since causal independence is not defined for decision nodes, the transformed network transforms only the families of chance nodes. As with *ci-elim-bel* and *ci-elim-map*, *ci-elim-meu* first selects a transformed network T_{BN}. Then *elim-meu* [23] is applied along some ordering o that starts with the decision nodes. This algorithm partitions into buckets all the network components (including the utility components). Then it processes chance nodes, from last to first, computing the new λ- and μ-functions. Finally, the buckets of the decision nodes are processed by maximization and an optimal assignment to those nodes is generated.

```
Algorithm ci-elim-meu(BN,U, e)

Input: A belief network $BN = (G, P)$, a decomposed utility function $U = \{r_1(x_1), ..., r_n(x_n)\}$, and evidence $e$.
Output: An assignment $d$ to the set of decision variables $D$ that maximizes the expected utility.
1. Compute a $T_{BN}$ (transform only families of chance nodes).
2. Return elim-meu($T_{BN}$,U,o,e), where $o$ is an ordering of $T_{BN}$ starting with the decision nodes.
```

Figure 3.15: Algorithm ci-elim-meu

A trace of *elim-meu* [23] applied to the transformed network in Figure 3.14b along $o = (d,\, e,\, y_1,\, u_1^e,\, u_2^e,\, u_3^e,\, c_2,\, c_3)$ is shown below. In each bucket we will have probability components λ_i and utility components μ_j. We compute a new pair of λ and μ in each bucket, as demonstrated below, and place them in the appropriate lower buckets:

1. **Bucket** $c_3 : \{P(c_3), P(u_3^e|c_3), r(c_3)\} \rightarrow$
   $\lambda^{c_3}(u_3^e) = \sum_{c_3} P(c_3)P(u_3^e|c_3)$,
   $\mu^{c_3}(u_3^e) = \frac{\theta_{c_3}(u_3^e)}{\lambda^{c_3}(u_3^e)}$,
   where $\theta^{c_3}(u_3^e) = \sum_{c_3} P(c_3)P(u_3^e|c_3)r(c_3)$. Put $\lambda^{c_3}(u_3^e)$ and $\mu^{c_3}(u_3^e)$ into bucket of u_3^e.

2. **Bucket** $c_2 : \{P(c_2), P(u_2^e|c_2), r(c_2)\} \rightarrow$
   $\lambda^{c_2}(u_2^e) = \sum_{c_2} P(c_2)P(u_2^e|c_2)$,
   $\mu^{c_2}(u_2^e) = \frac{\theta_{c_2}(u_2^e)}{\lambda^{c_2}(u_2^e)}$,
   where $\theta^{c_2}(u_2^e) = \sum_{c_2} P(c_2)P(u_2^e|c_2)r(c_2)$. Put $\lambda^{c_2}(u_2^e)$ and $\mu^{c_2}(u_2^e)$ into bucket of u_2^e.

3. **Bucket** $u_3^e : \{P'(y_1|u_2^e, u_3^e), \lambda^{c_3}(u_3^e), \mu^{c_3}(u_3^e)\} \rightarrow$
   $\lambda^{u_3^e}(y_1, u_2^e) = \sum_{u_3^e} P'(y_1|u_2^e, u_3^e)\lambda^{c_3}(u_3^e)$,
   $\mu^{u_3^e}(y_1, u_2^e) = \frac{\theta_{u_3^e}(y_1, u_2^e)}{\lambda^{u_3^e}(y_1, u_2^e)}$,
   where $\theta^{u_3^e}(y_1, u_2^e) = \sum_{u_3^e} P'(y_1|u_2^e, u_3^e)\lambda^{c_3}(u_3^e)\mu^{c_3}(u_3^e)$. Put $\lambda^{u_3^e}(y_1, u_2^e)$ and $\mu^{u_3^e}(y_1, u_2^e)$ into bucket of u_2^e.

4. **Bucket** u_2^e : $\{\lambda^{u_3^e}(y_1, u_2^e), \mu^{u_3^e}(y_1, u_2^e)\} \rightarrow$
   $\lambda^{u_2^e}(y_1) = \sum_{u_2^e} \lambda^{u_3^e}(u_2^e)$,
   $\mu^{u_2^e}(y_1) = \frac{\theta_{u_2^e}(y_1)}{\lambda^{u_2^e}(y_1)}$, where $\theta^{u_2^e}(y_1) = \sum_{u_2^e} \lambda^{u_3^e}(y_1, u_2^e)\mu^{u_3^e}(y_1, u_2^e)$. Put $\lambda^{u_2^e}(y_1)$ and $\mu^{u_2^e}(y_1)$ into bucket of y_1.

5. **Bucket** u_1^e : $\{P(u_1^e|d), P'(e|y_1, u_1^e)\} \rightarrow$
   $\lambda^{u_1^e}(y_1, d, e) = \sum_{u_1^e} P(u_1^e|d)P'(e|y_1, u_1^e)$. Put $\lambda^{u_1^e}(y_1, d, e)$ into bucket of y_1.

6. **Bucket** y_1 : $\{\lambda^{u_1^e}(y_1, d, e), \lambda^{u_2^e}(y_1), \mu^{u_2^e}(y_1)\} \rightarrow$
   $\lambda^{y_1}(d, e) = \sum_{y_1} \lambda^{u_1^e}(y_1, d, e)\lambda^{u_2^e}(y_1)$,
   $\mu^{y_1}(d, e) = \frac{\theta_{y_1}(d,e)}{\lambda^{y_1}(d,e)}$, where $\theta^{y_1}(d, e) = \sum_{y_1} \lambda^{u_1^e}(y_1, d, e)\lambda^{u_2^e}(y_1)\mu^{u_2^e}(y_1)$. Put $\lambda^{y_1}(d, e)$ and $\mu^{y_1}(d, e)$ in the bucket of e.

7. **Bucket** e : $\{\lambda^{y_1}(d, e), \mu^{y_1}(d, e)\} \rightarrow$
   $\lambda^e(d) = \sum_e \lambda^{y_1}(d, e)$,
   $\mu^e(d) = \frac{\theta_e(d)}{\lambda^e(d)}$, where $\theta^e(d) = \sum_e \lambda^{y_1}(d, e)\mu^{y_1}(d, e)$. Put $\lambda^e(d)$ and $\mu^e(d)$ in the bucket of d.

8. **Bucket** d : $\{r(d), \lambda^e(d), \mu^e(d)\} \rightarrow$
   $MEU = \max_d \lambda^e(d)[r(d) + \mu^e(d)]$,
   $d^{max} = \arg\max_d \lambda^e(d)[r(d) + \mu^e(d)]$.
   Return MEU and d^{max}.

Theorem 21: *The complexity of ci-elim-map and ci-elim-meu is* $O(Nmd^{w_o^*+1})$, *where* w_o^* *is the effective induced width along ordering* o. $\Box$

For each algorithm, the transformed graph G_T can be inspected a priori to determine the benefits of causal independence.

3.7 Exploiting evidence in CI-networks

Causal independence offers additional benefits in the presence of evidence. In some cases (e.g., for noisy-OR and noisy-AND CPTs) this may lead to exponential performance improvement.

Recall that the specification of a causally-independent CPT includes deterministic conditional probabilities, or *functional constraints*, of the form $y = y_1 * ... * y_n$. An observation of y imposes additional constraints on the hidden variables y_i. For example, if the operation $*$ is logical AND, then $y = y_1 \wedge \ldots \wedge y_n$ and evidence $y = 1$ imply $y_i = 1$, for $i = 1, ..., n$. Similarly, in the case of logical OR, observation $y = 0$ implies $y_i = 0$, for $i = 1, ..., n$. Another example: given that all variables are integers from 0 to 10, and given that $y = y_1 + ... + y_n$, an observation $y = 2$ imposes a constraint $y_1 + ... + y_n = 2$ which restricts the domain of each y_i to $\{0, 1, 2\}$, and rules out some of the remaining combinations of y_i. The examples above demonstrate the feasibility of constraint propagation known as enforcing *relational arc-consistency* [37] (see Appendix B for more details). We will focus on a special case when an observation of y implies a single value of each y_i (as is the case for noisy-OR and noisy-AND), namely on *evidence propagation*.

The procedure *Propagate_evidence* is presented in Figure 3.16. Given a belief network that includes deterministic (constraints), and a set of observations e, the procedure processes constraints that includes an observed variable, deducing as many new observations as possible, until no new observations can be deduced. This procedure parallels *arc-consistency* [78, 43, 22] in binary constraint networks, or *unit propagation* in propositional theories [19].

3.7.1 Noisy-OR networks

In this section we focus on evidence propagation in *noisy-OR* networks, a particular class of causally-independent networks. The algorithms discussed below exploit the fact that, given $z = x \vee y$, an observation $z = 0$ implies $x = 0$ and $y = 0$. In

```
Propagate_evidence($D, e$)

Input: A belief network $(G, P)$, where $P$ includes
deterministic CPTs (constraints), and evidence $e = \{(x_i = a_i) | i = 1, ..., k\}$.
Output: Extended set of observations $e'$.
1. Initialization: $e' \leftarrow e$
2. While $e'$ is not empty, let $(x = a) \in e'$,
          $e' \leftarrow e' - \{(x = a)\}$
          for each constraint $C$ that includes $x$
              if $C$ and $x = a$ imply $y = b$ /* $y$ is a variable in $C$ */
                  $e' \leftarrow e' \cup (y = b)$
3. Return $e'$.
```

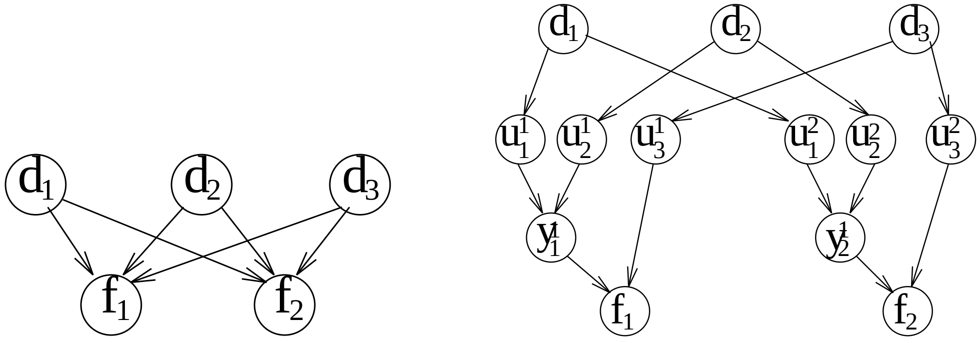
Figure 3.16: Procedure **Propagate_evidence**

some cases, this leads to an exponential speed-up in inference. The algorithms can be generalized for any operator $*$ having the property that given $z = x * y$, an observation $z = a$ implies singleton assignments $x = b$ and $y = c$. As mentioned before, logical AND has this property. Another examples are the operations MAX and $+$, when a is the minimal value in the domain of the variables.

Consider as an example the class of *Binary-Node 2-layer Noisy-OR (BN2O)* networks used in the QMR-DT medical database [90]. A fully-connected BN2O network with k diseases and n findings is a k-n-network, where the top-layer nodes d_i, $i = 1, ..., k$, represent diseases, and the bottom-layer nodes f_j, $j = 1, ..., n$ represent findings (see Figure 3.17a). The assignment $f_i = 0$ is a *negative* finding, while the assignment $f_i = 1$ is a *positive* finding.

The complexity of inference in a fully-connected BN2O network using algorithm *ci-elim-bel* is $O(exp(min\{k, 2n\}))$ (theorem 17). This complexity may be reduced by evidence propagation, especially in the presence of negative observations.

Example 12: Consider a BN2O network in Figure 3.17a and its transformed network in Figure 3.17b. Assume the query $P(d_1 | f_1 = 0, f_2 = 0)$. Evidence propagation in the transformed network assigns the value 0 to the nodes y_1, u_3^1, y_2, and u_3^2, and, consequently, to the nodes u_1^1, u_2^1, u_1^2, and u_2^2. Since all instantiated variables are



(a) (b)

Figure 3.17: (a) A BN2O network; (b) its transformed network.

processed first, the rest of the variables can be processed by *elim-bel* as follows:

1. **Bucket** d_3:$\{P(u_1^3=0|d_3), P(u_2^3=0|d_3)\} \rightarrow$
   $h^{d_3} = \sum_{d_3} P(u_1^3=0|d_3)P(u_2^3=0|d_3)$ is a constant.
2. **Bucket** d_2:$\{P(u_1^2=0|d_2), P(u_2^2=0|d_2)\} \rightarrow$
   $h^{d_2} = \sum_{d_2} P(u_1^2=0|d_2)P(u_2^2=0|d_3)$ is a constant.
3. **Bucket** d_1: $\{P(u_1^1=0|d_1), P(u_2^1=0|d_1)\} \rightarrow$
   $P(d_1|e) = \alpha P(u_1^1=0|d_1)P(u_2^1=0|d_1)$, where α is a normalization constant.

Note that normalization in bucket d_1 makes computing the constants in the buckets of d_3 and d_2 unnecessary. Thus, the complexity is linear in the number of hidden variables u_i^1, which equals the number of findings. On the other hand, the complexity of *elim-bel* applied to the original network is (at least) exponential in the number of diseases, since they form a clique in the moral graph.

The effect of negative evidence propagation on the complexity of *ci-elim-bel* is summarized in the following theorem:

Theorem 22: *Given a BN2O network BN having k diseases, n findings, and evidence e that consists of p positive and $(n-p)$ negative findings, applying algorithm ci-elim-bel augmented with Propagate_evidence on the transformed network T_{BN} takes*

$O(2^{min\{k,2p\}})$ time and space.

Proof: Evidence propagation assigns 0 to all hidden variables associated with negative findings in a transformed network. Assume that those hidden variables are last in the ordering. Namely, for each $f_i = 0$, the algorithm replaces u_j^i by 0 in each component $P(d_j|u_j^i)$. As mentioned before, instantiating a variable in a (moral) belief network is equivalent to removing it from the network. Therefore, all negative findings and their corresponding hidden variables can be now removed from the transformed network, leaving only positive findings in the lower layer. The result is a k-p-network, which yields the complexity of $O(2^{min\{k,2p\}})$. $\square$

3.7.2 Quickscore algorithm: an overview

In this section, we review algorithm *Quickscore* [60] for BN2O networks, which has better complexity bound for BN2O than *ci-elim-bel* achieves even when augmented by evidence propagation. Subsequently, we extend Quickscore to general noisy-OR networks.

As shown in [60], Quickscore takes $O(2^p)$ time, where p is the number of positive findings in a BN2O network. Note that for $k < p$ *ci-elim-bel* is superior and takes $O(2^k)$ time assuming that f_i nodes are processed first. Therefore, an algorithm for inference in BN2O networks should incorporate both approaches:

1. if $k < p$, run *ci-elim-bel*;
2. otherwise, run Quickscore.

The complexity of this modified scheme is $O(2^{min\{k,p\}})$, which is better than the complexity $O(2^{min\{k,2p\}})$ of *ci-elim-bel* augmented by *Propagate_evidence*.

Generally, a variable elimination algorithm decomposes the sum in the equation (3.2) into a product of linear number of functions, where computing each function is exponential in the number of its arguments. Alternatively, *Quickscore* algorithm decomposes the sum into exponential number of summands, each computed in linear time. Following [60], we next rederive algorithm *Quickscore*.

Consider a BN2O network for k diseases and n findings, where p findings are positive and the rest of them are negative. Let F^+ denote the set of positive findings, and let F^- denote the set of negative findings. Then

$$P(d_1|F^+, F^-) = \alpha \sum_{d_2,\ldots,d_k} \prod_{i=1}^{k} P(d_i) \prod_{f_j \in F^-} P(f_j = 0|d_1, \ldots, d_k) \prod_{f_l \in F^+} P(f_l = 1|d_1, \ldots, d_k) =$$

$$= \alpha \sum_{d_2,\ldots,d_k} \prod_{i=1}^{k} P(d_i) P^- P^+, \tag{3.15}$$

where α is a normalization constant, and where

$$P^- = \prod_{f_j \in F^-} P(f_j = 0|d_1, \ldots, d_k) = \prod_{f_j \in F^-} \prod_{i=1}^{k} P(u_i^j = 0|d_i), \tag{3.16}$$

and

$$P^+ = \prod_{f_l \in F^+} P(f_l = 1|d_1, \ldots, d_k) = \prod_{f_l \in F^+} (1 - \prod_{i=1}^{k} P(u_i^l = 0|d_i)) =$$

$$= \sum_{F' \in 2^{F^+}} (-1)^{|F'|} \prod_{f_l \in F'} \prod_{i=1}^{k} P(u_i^l = 0|d_i). \tag{3.17}$$

The notation 2^S is a shorthand for the power set of S, i.e. the set of all possible subsets of S. Then

$$P^- \cdot P^+ = \sum_{F' \in 2^{F^+}} (-1)^{|F'|} \prod_{i=1}^{k} \prod_{f_j \in F' \cup F^-} P(u_i^j = 0|d_i), \tag{3.18}$$

and, consequently,

$$P(d_1|F^+, F^-) = \alpha \sum_{d_2,\ldots,d_k} \prod_{i=1}^{k} P(d_i) P^- P^+ =$$

$$\alpha \sum_{F' \in 2^{F^+}} (-1)^{|F'|} \sum_{d_2,\ldots,d_k} \prod_{i=1}^{k} [P(d_i) \prod_{f_j \in F' \cup F^-} P(u_i^j = 0|d_i)] =$$

$$= \alpha \sum_{F' \in 2^{F^+}} (-1)^{|F'|} \prod_{i=1}^{k} \sum_{d_i} [P(d_i) \prod_{f_j \in F' \cup F^-} P(u_i^j = 0|d_i)]. \tag{3.19}$$

Computing each component

$$P(d_i)\prod_{f_j\in F'\cup F^-}P(u_i^j=0|d_i)$$

is linear in $|F'\cup F^-|\leq n$, so that computing each

$$\prod_{i=1}^{k}\sum_{d_i}[P(d_i)\prod_{f_j\in F'\cup F^-}P(u_i^j=0|d_i)] \tag{3.20}$$

is $O(kn)$ and the total complexity of computing $P(d_1|F^+,F^-)$ is $O(kn2^p)$, since $\sum_{d_i}$ introduces only a constant factor equal to the maximum domain size of d_i, and since the outer sum $\sum_{F'\in 2^{F^+}}$ has $2^{|F^+|}=2^p$ summands.

Now recall that $O(2^{2p})$ complexity of *ci-elim-bel* (theorem 22 for $k>2p$) is the result of performing $O(p)$ quadratic operations over the pairs of hidden variables that correspond to p positive evidence nodes (the details are given in the proof of the theorem 17). On the other hand, Quickscore requires only $O(2^p)$ time since it does not have hidden variables, but performs 2^p summations. In the next section, we generalize the formula used by Quickscore to the case of arbitrary noisy-OR networks and incorporate it into algorithm *NOR-elim-bel.*

3.7.3 Algorithm NOR-elim-bel

Assume a noisy-OR network defined on n nodes x_1,...,x_n, where the first k nodes are not observed. The next p nodes are positive findings $F^+=\{x_{k+1},...,x_{k+p}\}$, and the rest of the nodes are negative findings $F^-=\{x_{k+p+1},...,x_n\}$. As usual, α is a normalization constant, u_j^i denotes a hidden variable associated with node x_i and its j-th parent, and 2^S denotes the power set of S. Then $P(x_1|e)$, where $e=F^-\cup F^+$, can be computed as

$$P(x_1|e)=\alpha\sum_{x_2...x_k}PP^-P^+,$$

where

$$P=\prod_{i=1}^{k}P(x_i|pa(x_i)),$$

$$P^- = \prod_{x_j \in F^-} P(x_j = 0|pa(x_j)) = \prod_{x_j \in F^-} \prod_{x_m \in pa(x_j)} P(u^j_m = 0|x_m),$$

and

$$P^+ = \prod_{x_l \in F^+} P(x_l = 1|pa(x_l)) = \prod_{x_l \in F^+} (1 - \prod_{x_q \in pa(x_l)} P(u^l_q = 0|x_q)) =$$

$$= \sum_{F' \in 2^{F^+}} (-1)^{|F'|} \prod_{x_l \in F'} \prod_{x_q \in pa(x_l)} P(u^l_q = 0|x_q) = \sum_{F' \in 2^{F^+}} (-1)^{|F'|} P',$$

and where

$$P' = \prod_{x_l \in F'} \prod_{x_q \in pa(x_l)} P(u^l_q = 0|x_q).$$

The first two terms, P and P^-, can be moved inside the summation $\sum_{F' \in 2^{F^+}}$ in the last term, P^+. Also, the summation $\sum_{x_2 \ldots x_k}$ can be performed after $\sum_{F' \in 2^{F^+}}$, namely:

$$P(x_1|e) = \alpha \sum_{x_2 \ldots x_k} PP^- \sum_{F' \in 2^{F^+}} (-1)^{|F'|} P' = \alpha \sum_{F' \in 2^{F^+}} (-1)^{|F'|} \sum_{x_2 \ldots x_k} PP^- P' =$$

$$= \alpha \sum_{F' \in 2^{F^+}} (-1)^{|F'|} P_{F'},$$

where

$$P_{F'} = \sum_{x_2 \ldots x_k} PP^- P' =$$

$$= \sum_{x_2 \ldots x_k} \prod_{i=1}^{k} P(x_i|pa(x_i)) [\prod_{x_j \in F^-} \prod_{x_m \in pa(x_j)} P(u^j_m = 0|x_m)] [\prod_{x_l \in F'} \prod_{x_q \in pa(x_l)} P(u^l_q = 0|x_q)] =$$

$$= \sum_{x_2 \ldots x_k} \prod_{i=1}^{k} P(x_i|pa(x_i)) \prod_{x_j \in F^- \cup F'} \prod_{x_m \in pa(x_j)} P(u^j_m = 0|x_m). \qquad (3.21)$$

Note that the last expression is equivalent to joint probability distribution of x_1 together with the negative observations for the nodes in $F^- \cup F'$, and positive observations for the nodes in $F^+ - F'$, namely the joint

$$P(x_1, \{x_i = 0|x_i \in F^- \cup F'\}, \{x_j = 1|x_j \in F^+ - F'\}), \qquad (3.22)$$

which can be computed using a bucket-elimination algorithm. In particular, we can use *elim-bel*, preprocessed by evidence propagation. The algorithm *elim-bel* will be

```
Algorithm NOR-elim-bel

Input: A noisy-OR network $BN$, and evidence $e = F^- \cup F^+$,
       where $F^-$ is the set of 0-valued nodes and $F^+$ is the set of 1-valued nodes.
Output: $Bel(x_1) = P(x_1|e)$.
Initialization:
1. $Bel(x_1) \leftarrow 0$
2. $T_{BN} \leftarrow$ a transformed network of BN
3. For each $F' \in 2^{F^+}$
4.     $e' \leftarrow F^- \cup \{x_i = 0 | x_i \in F'\} \cup \{x_i = 1 | x_i \in F^+ - F'\}$
5.     $e'' \leftarrow e' \cup \{u_j^i = 0 | x_i \in F' \cup F^-\}$ /* evidence propagation */
6.     $o' \leftarrow$ an ordering of nodes in $T_{BN}$ s.t. observed nodes $e''$ are last
7.     $P_{F'} \leftarrow$ elim-bel$(T_{BN}, o', e'')$
8.     $Bel(x_1) \leftarrow Bel(x_1) + (-1)^{|F'|} P_{F'}$
Return $Bel(x_1)$.
```

Figure 3.18: Algorithm NOR-elim-bel

invoked 2^p times, once for each $F' \in 2^{F^+}$ yielding algorithm *NOR-elim-bel* in Figure 3.18. Note, that step 5 of the algorithm is equivalent to the evidence propagation.

It is easy to see that for BN2O networks, *NOR-elim-bel* coincides with *Quickscore*: the product of $P(x_i|pa(x_i))$ is simply $P(d_i)$ where d_i are the upper-layer nodes excluding the query node d_1.

Theorem 23: *Given a transformed network T_{BN} of a noisy-OR belief network BN, an evidence $e = F^+ \cup F^-$, where F^+ and F^- are positive and negative observations, respectively, and an ordering o of T_{BN}, the complexity of the algorithm* NOR-elim-bel *is $O(nm \cdot 2^{|F^+| + w_o^*})$, where n is the number of nodes in BN, m is the largest family size in BN, and w_o^* is the induced width of T_{BN} along o.*

Proof: Network transformation in step 2 takes $O(nm)$ time (see theorem 13). The for-loop (steps 3-8) is executed once for each $F' \in 2^{F^+}$, i.e. $2^{|F^+|}$ times. Within the loop, steps 4-6 can be performed in linear time. Since evidence propagation of F^- results in a smaller set of observations then the evidence propagation of $F^- \cup F'$, the set of nodes in o is a superset of the nodes in o' (step 6). Assume that o and o' agree on their common nodes, then $w_o^* \geq w_{o'}^*$. Therefore, step 7 that runs *elim-bel*(T_{BN},o',e") takes no more than $O(nm \cdot 2^{w_o^*})$ time, thus yielding the total complexity of $O(nm\ 2^{w_o^*}\ 2^{|F^+|}) = O(nm\ 2^{|F^+|+w_o^*})$. □

Given a particular evidence, we can decide upfront whether to use *NOR-elim-bel* or the regular *ci-elim-bel* by comparing the exponents in the corresponding complexity bounds, namely, comparing $|F^+| + w_o^*$ to w_e^*.

3.8 Conclusions

This chapter investigated the impact of causal independence on probabilistic inference. Our contributions are:

- We showed the connection between the previously existing approaches to belief updating in causally-independent networks, such as network transformations [63, 84] and variable-elimination algorithm VE1 [114]. Each of those methods is a special cases of inference over *binary-tree transformed networks*. The ordering restrictions of VE1 may sometimes lead to a unnecessary complexity increase. We describe a general variable-elimination scheme, called *ci-elim-bel*, that improves VE1 by accommodating any variable ordering over the transformed networks.

- We extended the causally-informed algorithms to other probabilistic tasks, such as finding a most probable explanation (MPE), finding a maximum a posteriori hypothesis (MAP), and finding the maximum expected utility (MEU). Surprisingly, while causal independence can significantly reduce the complexity of belief updating and finding MAP and MEU, it has, generally, no effect on MPE.

- We showed that the complexity of causally-informed algorithms for tasks such as belief updating, finding MAP and MEU is exponential in the induced width of a transformed network, called the effective induced width. The effective induced width does not exceed the induced width of the original network's moral graph and may be as small as the induced width of the unmoral graph. Consequently, exploiting causal independence reduces complexity, often by an exponential factor.

- We augmented algorithm *ci-elim-bel* with *evidence propagation* using relational arc-consistency, and also incorporated this approach into a new algorithm *NOR-elim-bel* for noisy-OR networks, which generalizes algorithm Quickscore for BN2O networks [60].

We identified several network topologies, such as poly-trees, "source-sink" multiply-connected networks, and k-n-networks, where exploiting causal independence leads to an exponential complexity reduction relative to standard causally-blind algorithms. Empirical evaluation is necessary to assess the ultimate virtues of the methods discussed.

Chapter 4

Approximate inference

4.1 Introduction

Automated reasoning tasks such as constraint satisfaction and optimization, probabilistic inference, decision-making, and planning are generally hard (NP-hard). One way to cope with this computational complexity is to identify tractable problem classes. Another way is to design algorithms that compute *approximate* rather than exact solutions.

Although approximation within given error bounds is also known to be NP-hard [86, 98], there are approximation strategies that work well in practice. One approach advocates *anytime algorithms*. These algorithms can be interrupted at any time producing the best solution found thus far [20, 8]. Another approach is to identify problem classes that can be solved approximately within given error bounds, thus applying the idea of tractability to approximation.

In this chapter we present a family of parameterized algorithms that allow a flexible trade-off between accuracy and efficiency and that can be combined in an anytime algorithm. We provide conditions under which the approximation algorithms find an exact solution and identify regions of good performance.

The class of *mini-bucket* approximation algorithms we propose imports the idea of *local inference* from constraint networks to probabilistic reasoning and combinatorial optimization using the *bucket-elimination* framework. Bucket-elimination is a unifying algorithmic scheme that generalizes non-serial dynamic programming to enable complex problem-solving and reasoning activities. Among the algorithms that can be expressed as bucket-elimination are *directional-resolution* for propositional satisfiability [36], *adaptive-consistency* for constraint satisfaction [33], *Fourier* and *Gaussian elimination* for linear inequalities [74], *dynamic-programming* for combinatorial optimization [7], as well as many algorithms for probabilistic inference [26].

In all these areas problems are represented by a set of variables and by a set of dependencies (e.g., constraints, cost functions, and probabilities) that can be captured by a graph. The algorithms infer and record new dependencies which amounts to adding new edges to the graph. Generally, representing a dependence among k variables (k is called *arity* of a dependence) requires enumerating $O(exp(k))$ tuples. As a result, the complexity of inference is time and space exponential in the arity of the largest dependence recorded which corresponds to the size of largest clique created in the graph and is known as *induced-width.*

Local inference approximation algorithms like *i-consistency* [44, 22] bound the computational complexity by restricting to i the arity of recorded dependencies. Known special cases are *arc-consistency* ($i = 2$) and *path-consistency* ($i = 3$) [78, 43, 22]. Indeed, the recent success of constraint-processing algorithms can be attributed primarily to this class of algorithms, either used as stand-alone, incomplete algorithms, or incorporated within backtracking search [28, 29]. The idea and benefit of local consistency algorithms are demonstrated in Figure 4.1. The figure shows that while exact algorithms may record arbitrarily large constraints, i-consistency algorithms decide consistency of smaller subproblems, recording constraints of size i or less.

In this chapter we present and analyze a local inference approximation scheme for probabilistic tasks of belief updating, finding the most probable explanation, finding the maximum a posteriori hypothesis, and for optimization tasks in general. We

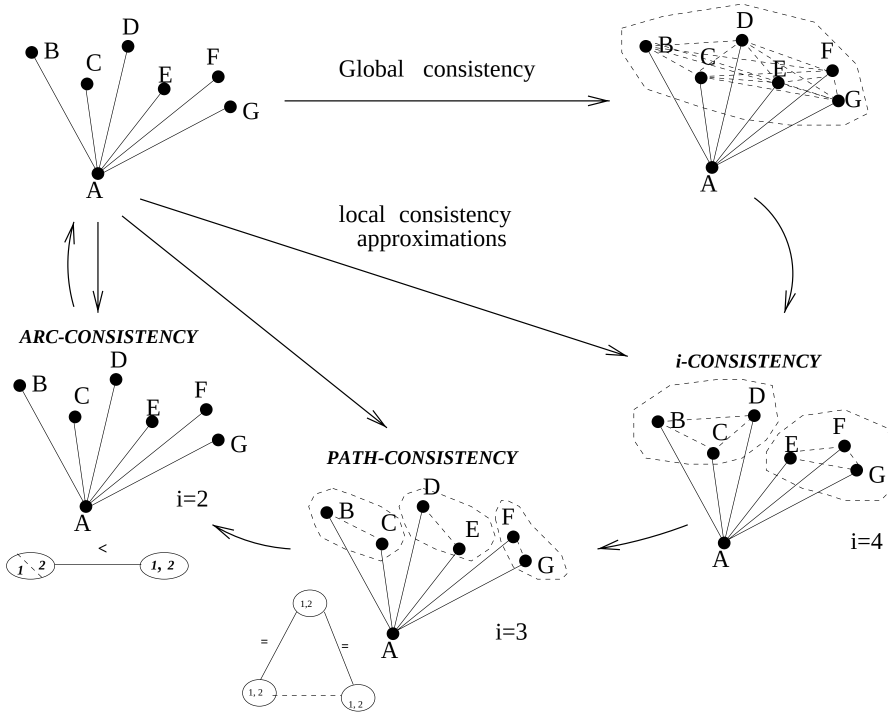


Figure 4.1: From global to local consistency: algorithm i-consistency and its particular cases path-consistency (i=3) and arc-consistency (i=2).

identify regions of completeness and demonstrate promising empirical results obtained both on randomly generated networks and on realistic domains such as medical diagnosis and probabilistic decoding.

For the necessary definitions and preliminaries, the reader is referred to the section 3.2 of the previous chapter. This chapter is organized as follows. In the next three sections we present and analyze the mini-bucket approximation for the probabilistic inference tasks of finding a *most probable explanation (MPE)*, *belief-updating (BEL)* and finding a *most probable a posteriori hypothesis (MAP)*. Section 4.5 presents the mini-bucket algorithm for optimization problems. Section 4.6 identifies cases of completeness, Section 4.7 discusses extensions to anytime algorithms and heuristic search and Section 4.8 discusses related work. In Section 4.9 empirical evaluation is carried out for the MPE task. Encouraging results are obtained on randomly generated noisy-OR networks, on the CPCS networks for medical diagnosis [90], and on classes of probabilistic decoding problems. Section 4.10 provides concluding remarks and discusses the future work.

4.2 Approximating the MPE

As we saw in Section 2, the complexity of bucket-elimination is determined by the complexity of processing each bucket and is time and space exponential in the number of variables in the bucket, $w^* + 1$, where w^* is the induced-width of the network's moral graph along the elimination ordering. Consequently, the algorithm is infeasible when w^* is large, primarily due to memory requirements.

Since the complexity of processing a bucket is tied to the arity of the functions being recorded, we propose to approximate these functions by a collection of smaller-arity functions. Let $h_1, ..., h_t$ be the functions in the bucket of X_p, and let $S_1, ..., S_t$ be the sets of variables on which those functions are defined. When *elim-mpe* processes the bucket of X_p, the function $h^p = max_{X_p} \Pi_{i=1}^{t} h_i$ is computed. Since, for two non-negative functions $Z(x)$ and $Y(x)$, $\max_x Z(x) \cdot Y(x) \leq \max_x Z(x) \cdot \max_x Y(x)$, a simple approximation idea is to compute an upper bound on h^p by "migrating" the maximization inside the multiplication. Namely, $g^p = \Pi_{i=1}^{t} \max_{X_p} h_i$ is an upper bound on h^p, since it replaces each h_i in $\Pi_{i=1}^{t} h_i$ by $\max_{X_p} h_i$. Maximization is now applied separately to smaller-arity functions h_i, yielding a lower complexity. This idea can be generalized to any partitioning of a set of functions $h_1, ..., h_t$ into subsets that we will call *mini-buckets*. Namely, let $Q = \{Q_1, ..., Q_r\}$ be a partitioning into mini-buckets of the functions $h_1, ..., h_t$ in X_p's bucket. The mini-bucket Q_l contains the functions $h_{l_1}, ..., h_{l_r}$. The complete algorithm *elim-mpe* computes $h^p = \max_{X_p} \Pi_{i=1}^{t} h_i$, which can be rewritten as $h^p = \max_{X_p} \Pi_{l=1}^{r} \Pi_{l_i} h_{l_i}$. By migrating the maximization operator into each mini-bucket we compute: $g_Q^p = \Pi_{l=1}^{r} \max_{X_p} \Pi_{l_i} h_{l_i}$. The functions $\max_{X_p} \Pi_{l_i} h_{l_i}$ are placed separately into their highest-variable buckets and the algorithm proceeds with the next variable. Note that functions without arguments (i.e., constants) are placed in the lowest bucket. The product of constants collected in the

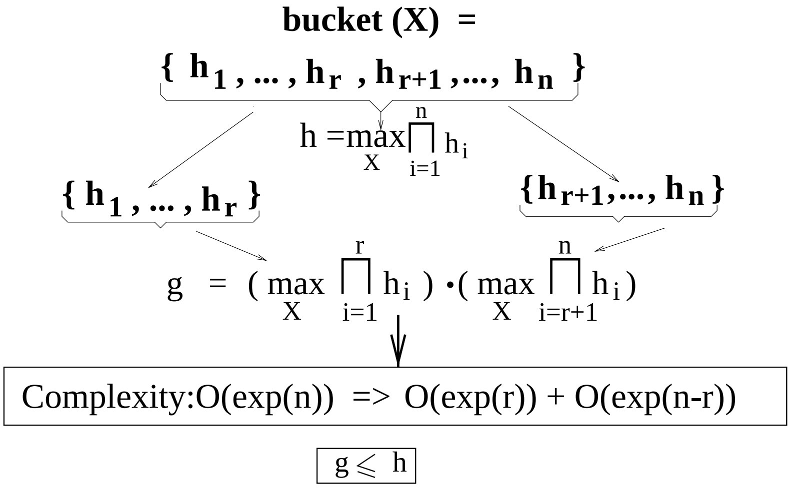


Figure 4.2: The idea of mini-bucket approximation.

first bucket is an upper bound on the MPE.

The idea of mini-bucket partitioning is demonstrated in Figure 4.2. The bucket of variable X having n functions is split into two mini-buckets of size r and $(n-r)$, $r \leq n$. This decreases the complexity exponentially, from $O(exp(n))$ to $O(exp(r)) + O(exp(n-r))$. Clearly, as the mini-buckets get smaller both the complexity and the accuracy decrease. This yields a family of approximation algorithms having adjustable trade-off between accuracy and efficiency.

Definition 9: Given two partitionings Q' and Q'' over the same set of elements, Q' is a refinement of Q'' if and only if for every set $A \in Q'$ there exists a set $B \in Q''$ such that $A \subseteq B$.

Proposition 2: *If Q'' is a refinement of Q' in the bucket of X_p, then $h^p \leq g^p_{Q'} \leq g^p_{Q''}$.*

Proof: Given a partitioning of $Q = \{h_1, ..., h_t\}$ into $Q_1 = \{h_1, ..., h_r\}$ and $Q_2 = \{h_{r+1}, ..., h_t\}$.

$$h^p = \max_{X_p} (\Pi_{i \in Q_1} h_i)(\Pi_{j \in Q_2} h_j). \tag{4.1}$$

The approximating function, g^p, for this partitioning is:

$$g^p_Q = (\max_{X_p} \Pi_{i \in Q_1} h_i) \cdot (\max_{X_p} \Pi_{j \in Q_2} h_j). \tag{4.2}$$

Thus, g^p_Q is derived from h^p by replacing the second product in h^p, $\Pi_{j \in Q_2} h_j$, by $\max_{X_p} \Pi_{j \in Q_2} h_j$. Consequently, $h^p \leq g^p_Q$. In general, given $Q' = \{Q'_1, ..., Q'_m\}$, by induction on the number of mini-buckets m,

$$g^p_{Q'} = \prod_{j=1}^{m} (\max_{X_p} \Pi_{l \in Q'_j} h_l), \tag{4.3}$$

and, consequently, $h^P \leq g^p_{Q'}$.

By definition, given a refinement $Q'' = \{Q''_1, ..., Q''_k\}$ of a partitioning $Q' = \{Q'_1, ..., Q'_m\}$, each mini-bucket $i \in \{1, ..., k\}$ of Q'' belongs to some mini-bucket $j \in \{1, ..., m\}$ of Q'. In other words, each mini-bucket j of Q' is further partitioned into the corresponding mini-buckets of Q'' as follows: $Q'_j = \{Q''_{j1}, ..., Q''_{jl}\}$. Then

$$g^p_{Q''} = \prod_{i=1}^{k} (\max_{X_p} \Pi_{l \in Q''_i} h_l) = \prod_{j=1}^{m} \prod_{Q''_i \subseteq Q'_j} (\max_{X_p} \Pi_{l \in Q''_i} h_l) \geq \prod_{j=1}^{m} (\max_{X_p} \Pi_{l \in Q'_j} h_l) = g^p_{Q'}, \tag{4.4}$$

which concludes the proof. □

The mini-bucket approximation algorithm *approx-mpe(i,m)* is described in Figure 4.3a. It has two parameters that control the partitioning.

Definition 10: Let H be a collection of functions $h_1, ..., h_t$ defined on subsets of variables $S_1, ..., S_t$, respectively. We will say that a function f is *subsumed* by a function h if any argument of f is also an argument of h. A partitioning of $h_1, ..., h_t$ is *canonical* if any function f subsumed by other functions is placed into the bucket of one of those subsuming functions. A partitioning Q into mini-buckets is an (i, m)-partitioning if, and only if, (1) it is canonical, (2) at most m non-subsumed functions participate in each mini-bucket, (3) the total number of variables in a mini-bucket does not exceed i, and (4) the partitioning is *refinement-maximal*, namely, there is no other (i, m)-partitioning that it refines.

Not every combination of i and m yields a feasible partitioning. However, it is easy to see that:

Proposition 3: *If the bound i on the number of variables in a mini-bucket is not smaller than the maximum family size, then, for any value of $m > 0$, there exists an (i, m)-partitioning of each bucket.*

Proof: For $m = 1$, each mini-bucket contains one family. The arity of the recorded functions will only decrease and thus in each bucket an $(i, 1)$-partitioning always exists. Any (i, m)-partitioning that satisfies conditions 1-3 (but not necessarily condition 4), always includes all $(i, 1)$-partitionings satisfying conditions 1-3. Therefore, the set of (i, m)-partitionings satisfying conditions 1-3 is never empty, and there exists an (i, m)-partitioning satisfying conditions 1-4. $\square$

Although the two parameters i and m are not independent they do allow a flexible control of the mini-bucket scheme. The properties of the mini-bucket algorithms are summarized in the following theorem.

Theorem 24: *Algorithm* approx-mpe*(i, m) computes an upper bound to the MPE. Its time complexity is $O(m \cdot exp(2i))$ and its space complexity is $O(m \cdot exp(i))$, where $i \leq n$ and $m \leq 2^i$. For $m = 1$, the algorithm is time and space $O(m \cdot exp(|F|))$, where $|F|$ is the maximum family size.*

Proof: Since *approx-mpe(i,m)* computes an upper bound in each bucket it yields an overall upper bound on the resulting MPE. The complexity of *approx-mpe(i,m)* can be derived as follows. Processing a bucket is linear in the number of its mini-buckets. Since in each mini-bucket there are at most m functions having arity of at most i (i.e., of size at most $exp(i)$), multiplication takes at most $O(m \cdot exp(2i))$ time-wise and $O(exp(i))$ space-wise. The number of mini-buckets is bounded by 2^i, the number of subsets of size i. For $m = 1$, each mini-bucket contains only one family and perhaps some subsumed functions. The arity of the recorded functions will only decrease, thus yielding time and space complexity of $O(m \cdot exp(|F|))$, where $|F|$ is maximum family size. $\square$

In general, as m and i increase we get more accurate approximations. However, a clear hierarchy can only be obtained relative to the partial order of refinement.

Example 13: Figure 4.3b illustrates how algorithms *elim-mpe* and *approx-mpe(i,m)* for $i = 3$ and $m = 2$ process the network in Figure 4.3a along the ordering (A, E, D, C, B). First, all functions are partitioned into buckets that are the same for both algorithms. The exact algorithm *elim-bel* sequentially processes the variables B, C, D, and E, recording the new functions (shown in boldface) $h^B(a,d,c,e)$, $h^C(a,d,e)$, $h^D(a,e)$, and $h^E(a)$. Then, in the bucket of A, it computes MPE $= \max_a P(a)h^E(a)$. Subsequently, an MPE assignment $(A = a', B = b', C = c', D = d', E = 0)$ where $E = 0$ is an evidence is computed for each variable from A to B by selecting a value that maximizes the product of functions in the corresponding buckets conditioned on the previously assigned values. Namely, $a' = \arg\max_a P(a)h^E(a)$, $e' = 0$, $d' = \arg\max_d h^C(a', d, e = 0)$, and so on.

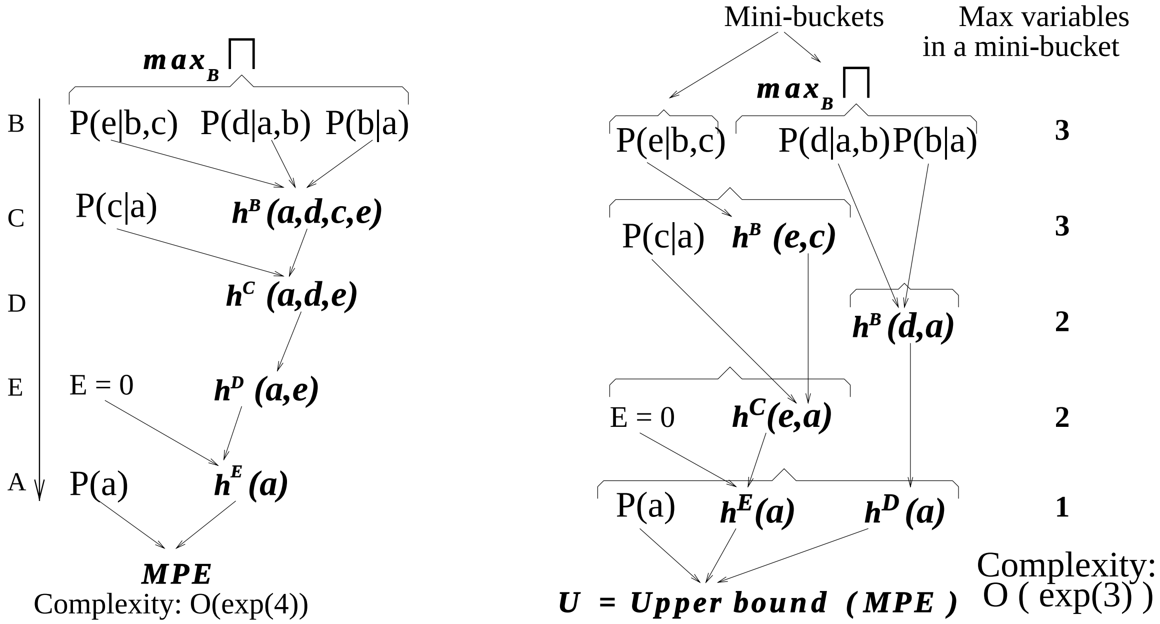
On the other hand, the approximation algorithm *approx-mpe(3,2)* runs as follows. Since the bucket of B includes five variables it is split into two mini-buckets $\{P(e|b,c)\}$ and $\{P(d|a,b), P(b|a)\}$, each containing no more than 3 variables, as shown in Figure 4.3b (tie-breaking is arbitrary when selecting mini-buckets). The new functions $h^B(e,c)$ and $h^B(d,a)$ are computed separately in each mini-bucket and are placed in their highest-variable buckets. In each of the remaining buckets the number of variables is not larger than 3 and therefore no mini-bucket partitioning occurs. An upper bound on the MPE value is computed by maximizing over A the product of functions in A's bucket. A suboptimal MPE tuple is computed similarly to MPE tuple by assigning a value to each variable that maximizes the product of functions in the corresponding bucket, given the assignments to the previous variables.

Note, that *approx-mpe(3,2)* does not produce new functions on more than $i - 1$ (i.e., 2) variables, while the exact algorithm *elim-mpe* records a function on 4 variables.

The probability of the tuple generated by *approx-mpe* provides a lower bound on the MPE and can be computed using the joint-probability's product form. Thus,

```
Algorithm approx-mpe(i,m)
Input: A belief network $BN = (G, P)$, an ordering $o$, evidence $e$.
Output: An upper and a lower bounds on the MPE given evidence $e$.
1. Initialize: Partition $P = \{P_1, ..., P_n\}$ into buckets $bucket_1, \ldots, bucket_n$, where
   $bucket_p$ contains all matrices $h_1, h_2, ..., h_t$ whose highest-index variable is $X_i$.
2. Backward: for $p = n$ to 1 do
   • If $X_p$ is observed ($X_p = a$), replace $X_p$ by $a$ in each $h_i$ and put the result
     in its highest-variable bucket (put constants in $bucket_1$).
   • Else  for $h_1, h_2, ..., h_t$ in $bucket_p$ do
     Generate an $(i, m)$-mini-bucket-partitioning, $Q' = \{Q_1, ..., Q_r\}$.
     for each $Q_l \in Q'$ containing $h_{l_1}, ...h_{l_t}$, compute $h^l = max_{X_p} \Pi_{i=1}^{t} h_{l_i}$ and add it
         to the bucket of the highest-index variable in $U_l \leftarrow \bigcup_{i=1}^{t} S_{l_i} - \{X_p\}$, where
         $S_{l_i}$ is the set of arguments of $h_{l_i}$ (put constants in $bucket_1$).
3. Forward: for $p = 1$ to $n$, given $X_1 = x_1^{opt}, ..., X_{p-1} = x_{p-1}^{opt}$,
   assign a value $x_p^{opt}$ to $X_p$ that maximizes the product of all functions in $bucket_p$.
4. Return the assignment $x^{opt} = (x_1^{opt}, ..., x_n^{opt})$, a lower bound $L = P(x^{opt})$,
and an upper bound $U = \prod_{i=1}^{t} h^i$ on the MPE, where $h^i$ are constants in $bucket_1$.
```

(a) Algorithm *approx-mpe(i,m)*.



(b) A trace of *elim-mpe* (c) A trace of *approx-mpe(3,2)*.

Figure 4.3: (a) Algorithm *approx-mpe(i,m)* and the performance comparison of (b) *elim-mpe* and (c) *approx-mpe(3,2)*.

algorithm *approx-mpe(i,m)* computes an interval $[L, U]$ containing the MPE.

4.3 Approximating belief update

As shown in Section 3.2 the bucket-elimination algorithm *elim-bel* for belief assessment is identical to *elim-mpe* except that maximization is replaced by summation. Let e be a set of observations. Algorithm *elim-bel* finds $P(x_1, e)$ and then computes $P(x_1|e) = \alpha P(x_1, e)$ where α is the normalization constant. This procedure has only a backward phase (see Figure 3.2a). When processing the bucket of X_p, we multiply all its matrices, $h_1, ..., h_t$, defined over subsets $S_1, ..., S_t$, and sum over X_p. The computed function is $h^p : U_p \rightarrow \Re$, where $h^p = \sum_{X_p} \Pi_{i=1}^t h_i$, and $U_p = \cup_i S_i - X_p$. Once all the buckets are processed, the updated probability $P(x, e)$ is available in the bucket of X_1.

The mini-bucket idea can be applied to belief updating as follows. Let $Q\prime = \{Q_1, ..., Q_r\}$ be a partitioning into mini-buckets of the functions $h_1, ...h_t$ in X_p's bucket. The exact algorithm *elim-bel* computes $h^p = \sum_{X_p} \Pi_{i=1}^t h_i$, which can be rewritten as $h^p = \sum_{X_p} \Pi_{l=1}^r \Pi_{l_i} h_{l_i}$. If we follow the MPE approximation precisely and migrate the summation operator into each mini-bucket, we will compute $f_{Q\prime}^p = \Pi_{l=1}^r \sum_{X_p} \Pi_{l_i} h_{l_i}$. This, however, is an unnecessarily large upper bound of h^P since each $\Pi_{l_i} h_{l_i}$ is replaced by $\sum_{X_p} \Pi_{l_i} h_{l_i}$. Instead, we process the first mini-bucket separately and get $h^p = \sum_{X_p} (\Pi_{l_1} h_{l_1}) \cdot (\Pi_{l=2}^r \Pi_{l_i} h_{l_i})$. Subsequently, instead of bounding a function of X by its sum over X, we can bound $i > 1$, by its maximum over X, which yields $g_{Q\prime}^p = (\sum_{X_p} \Pi_{l_1} h_{l_1}) \cdot (\Pi_{l=2}^r \max_{X_p} \Pi_{l_i} h_{l_i})$. Clearly,

Proposition 4: *For every partitioning* Q, $h^p \leq g_Q^p \leq f_Q^p$. *Also, if* Q'' *is a refinement partitioning of* Q', *then* $h^p \leq g_{Q'}^p \leq g_{Q''}^p$. $\square$

The proof is similar to the one for MPE.

In summary, an upper bound g^p of h^p can be obtained by processing one of X_p's mini-buckets by summation and the rest by maximization. We can also compute a

```
Algorithm approx-bel-max(i,m)
Input: A belief network $BN = (G, P)$, an ordering $o$, and evidence $e$.
Output: an upper bound on $P(x_1, e)$.
1. Initialize: Partition $P = \{P_1, ..., P_n\}$ into buckets
   $bucket_1$, . . ., $bucket_n$, where $bucket_p$ contains all
   matrices $h_1, h_2, ..., h_t$ whose highest-index variable is $X_i$.
2. Backward: for $p = n$ to 1 do
   • If $X_p$ is observed ($X_p = a$), replace $X_p$ by $a$ in each $h_i$ and put the result
     in its highest-variable bucket (put constants in $bucket_1$).
   • Else for $h_1, h_2, ..., h_t$ in $bucket_p$ do
     Generate an $(i, m)$-mini-bucket-partitioning, $Q' = \{Q_1, ..., Q_r\}$.
     for each $Q_l \in Q'$, containing $h_{l_1}, ...h_{l_t}$, do
         If $l = 1$ compute $h^l = \sum_{X_p} \Pi_{i=1}^{t} h_{1_i}$
         Else compute $h^l = max_{X_p} \Pi_{i=1}^{t} h_{l_i}$
         Add $h^l$ to the bucket of the highest-index variable in $U_l \leftarrow \bigcup_{i=1}^{t} S_{l_i} - \{X_p\}$,
         where $S_{l_i}$ is the set of arguments of $h_{l_i}$ (put constants in $bucket_1$).
3. Return the product of functions in the bucket of $X_1$,
   which is an upper bound on $P(x_1, e)$ (denoted $g(x_1)$).
```

Figure 4.4: algorithm approx-bel-max(i,m)

lower bound, or the mean value by applying to each mini-bucket ($i > 1$) the *min* or the *mean* operator, respectively. Algorithm *approx-bel-max(i,m)* that uses the maximizing elimination operator is described in Figure 4.4. Algorithms *approx-bel-min* and *approx-bel-mean* can be obtained by replacing the operator *max* by *min* and by *mean*, respectively.

4.3.1 Normalization

Note that *aprox-bel-max* computes an upper bound on $P(x_1, e)$ but not on $P(x_1|e)$. If an exact value of $P(e)$ is not available, deriving a bound on $P(x_1|e)$ from a bound on $P(x_1, e)$ may not be easy. For example, $\frac{g(x_1)}{sum_{x_1} g(x_1)}$, where g is the upper bound on $P(x_1, e)$, is not necessarily an upper bound on $P(x_1|e)$. However, we can derive a lower bound f on $P(e)$ using *approx-bel-min* (in this case the observed variables initiate the ordering), and then compute $\frac{g(x_1)}{f}$ as an upper bound on $P(x_1|e)$.

4.4 Approximating the MAP

Algorithm *elim-map* for computing the MAP presented in [24] is a combination of *elim-mpe* and *elim-bel*; some of the variables are eliminated by summation, the rest by maximization. Consequently, its mini-bucket approximation is a mix of *approx-mpe* and *approx-bel-max*.

Given a belief network, a subset of hypothesis variables $A = \{A_1, ..., A_k\}$ and some evidence, the problem is to find an assignment to the hypothesized variables that maximizes their probability given evidence e. Formally, we wish to compute

$$\max_{\bar{a}_k} P(\bar{a}_k|e) = (\max_{\bar{a}_k} \sum_{\bar{x}_{k+1}^n} \Pi_{i=1}^n P(x_i, e|x_{pa_i}))/P(e) \tag{4.5}$$

when $x = (a_1, ..., a_k, x_{k+1}, ..., x_n)$. Since $P(e)$ is a normalization constant, this is equivalent to computing $P(\bar{a}_k|e)$. The bucket-elimination algorithm for MAP, *elim-map* [24], assumes only orderings in which the hypothesized variables appear first and thus are processed last by the algorithm. The algorithm has a backward phase as usual but its forward phase is relative to the hypothesis variables only. The application of the mini-bucket scheme to *elim-map* is a straightforward extension to *approx-mpe* and *approx-bel-max*. We partition each bucket to mini-buckets as before. If the bucket's variable is a summation variable we apply the rule we have in *approx-bel-max* in which one mini-bucket is approximated by summation and the rest by maximization. When the algorithm reaches the hypothesis buckets their processing is identical to that of *approx-mpe*. Algorithm *approx-map(i,m)* is described in Figure 4.5.

Example 14: We will next demonstrate the mini-bucket approximation for MAP on an example inspired by *probabilistic decoding* [77, 45] [1]. Consider a belief network

---

[1] Probabilistic decoding is discussed in more details in Section 4.9.5.

```
Algorithm approx-map(i,m)
Input: A belief network $BN = (G, P)$, a subset of variables $A = \{A_1, ..., A_k\}$,
an ordering of the variables, $o$, in which the $A$'s appear first, and evidence $e$.
Output: An upper bound on the MAP and an assignment $A = a$.
1. Initialize: Partition $P = \{P_1, ..., P_n\}$ into buckets $buck$
```