Exploiting Structure in Backtracking Algorithms for Propositional and Probabilistic Reasoning

by

Wei Li

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Wei Li

Abstract

Boolean propositional satisfiability (SAT) and probabilistic reasoning represent two core problems in AI. Backtracking based algorithms have been applied in both problems. In this thesis, I investigate structure-based techniques for solving real world SAT and Bayesian networks, such as software testing and medical diagnosis instances.

When solving a SAT instance using backtracking search, a sequence of decisions must be made as to which variable to branch on or instantiate next. Real world problems are often amenable to a divide-and-conquer strategy where the original instance is decomposed into independent sub-problems. Existing decomposition techniques are based on pre-processing the static structure of the original problem. I propose a dynamic decomposition method based on hypergraph separators. Integrating this dynamic separator decomposition into the variable ordering of a modern SAT solver leads to speedups on large real world SAT problems.

Encoding a Bayesian network into a CNF formula and then performing weighted model counting is an effective method for exact probabilistic inference. I present two encodings for improving this approach with noisy-OR and noisy-MAX relations. In our experiments, our new encodings are more space efficient and can speed up the previous best approaches over two orders of magnitude.

The ability to solve similar problems incrementally is critical for many probabilistic reasoning problems. My aim is to exploit the similarity of these instances by forwarding structural knowledge learned during the analysis of one instance to the next instance in the sequence. I propose dynamic model counting and extend the dynamic decomposition and caching technique to multiple runs on a series of problems with similar structure. This allows us to perform Bayesian inference incrementally as the evidence, parameter, and structure of the network change. Experimental results show that my approach yields significant improvements over previous model counting approaches on multiple challenging Bayesian network instances.

Acknowledgments

First, I would like to thank my advisor, Peter van Beek, for continuously supporting me over the years. He gave me inspiration to explore challenging research problems, showed me the need to be persistent to accomplish any goal and kept my thesis accurate about all the details. This thesis would not have been possible without his supervision.

My other committee members have also been very supportive. Pascal Poupart showed me different ways to approach probabilistic reasoning and he brought a much wider perspective to the thesis. Grant Weddell has long been an inspiration to me. His weekly meetings of automatic reasoning have been one of my best learning experiences at UW.

I would like to thank my many friends and colleagues at UW with whom I have had the pleasure of working over the years. These include Huayue Wu, Claude-Guy Quimper, Jiye Li, Wei Zhou, Abid Malik, Meng He, Jinbo Xu, Zhifeng Liu, Jiang Liu, Zhenmei Gu, Lijie Zou, Xinyi Dong, Hao Chen, Yuan Lin, Ning Zhang and Vincent Park.

A special thanks goes to my manager at Autodesk Research, George Fitzmaurice, who is always very supportive. He helped me to schedule my work to complete my defence.

I thank my parents Shaolong Li and Jinfang Liu for giving me life, for educating me, for unconditional support and encouragement to pursue my interests. I greatly thank my parents-in-law for believing in me and supporting me. I also thank my brother Ying Li, for reminding me that my research should always be useful; my sister-in-law, Ying Zou, and her husband, Ahmed Hassan, for sharing their experience of becoming an excellent researcher.

Last, but not least, I am indebted to my wife Bing Zou and my four-year-old daughter Ivy Li for giving me the time and the motivation to finish this thesis.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

In this chapter, I introduce my research area: knowledge representation and reasoning, a subfield of artificial intelligence. I then informally present and motivate the specific problems addressed in this thesis. Finally, I summarize the contributions of the thesis and give an outline of the remainder of the thesis.

1.1 Knowledge Representation and Reasoning

Knowledge representation and reasoning have been central to the field of artificial intelligence since its inception. The central insight is that many interesting problems can be solved by explicitly representing declarative knowledge in some language and answering queries using a general purpose inference engine. The applications of knowledge representation and reasoning systems are numerous and include software and hardware verification, planning, scheduling, autonomous vehicles, medical diagnosis, and computer troubleshooting (see, for example, [83, 85, 102], and the references therein). Many knowledge representation languages and inference mechanisms have been proposed. In this thesis, we are concerned with knowledge representation and reasoning systems based on propositional logic and logical inference and systems based on probabilities and probabilistic inference. For probabilistic inference, my focus is on a convenient graphical representation called Bayesian networks.

The general problem that I address in this thesis is improving the efficiency of inference in these types of knowledge representation and reasoning systems. Unfortunately, inference in propositional logic and Bayesian networks is NP-Hard or worse (see [45, 72]). That is the bad news. More fortunately, much progress has been made over the years and inference engines for propositional logic and Bayesian networks can now solve significantly-sized real-world instances (see [72, 102]). It is my aim here to contribute to this progress.

1.2 Contributions of the Thesis

In general terms, the contributions of the thesis are techniques and algorithms for exploiting the structure of real-world instances within backtracking algorithms for

query answering. Together my results increase the efficiency and improve the scalability of query answering in knowledge representation and reasoning systems, and so increases their applicability in practice. Below I give a more detailed overview of my results. The discussion is divided according to the reasoning task being solved.

For the task of determining whether a propositional CNF formula is satisfiable, the main results are as follows.

Determining whether a propositional formula is satisfiable is most often done with a backtracking algorithm. Previous studies have demonstrated that backtracking search algorithms can be considerably improved if they take advantage of the internal structure of propositional formulas to decompose an instance into independent subproblems. However, most existing decomposition techniques are static and performed prior to search. I propose a dynamic decomposition method based on hypergraph separators. Integrating the separator decomposition into the variable ordering of a modern SAT solver leads to speedups on large real-world satisfiability problems. In comparison to static decomposition based variable orderings, my approach does not need time to construct the full decomposition prior to search, which sometimes needs more time than the solving process itself. Furthermore, my dynamic method can solve hard instances not solvable by previous static approaches within an acceptable amount of time.

For the task of answering a general probabilistic query of the form $P(Q \mid E)$ from a Bayesian network, the main results are as follows.

Previous studies have demonstrated that encoding a Bayesian network into a SAT formula and then performing weighted model counting using a DPLL-based algorithm can be an effective method for exact inference, where DPLL is a backtracking algorithm specialized for SAT that includes unit propagation, conflict recording and backjumping [89]. I present techniques for improving this weighted model counting approach for Bayesian networks with noisy-OR and noisy-MAX relations. In particular, I present space efficient CNF encodings for noisy-OR and noisy-MAX which exploit their structure or semantics. In my encodings, I pay particular attention to reducing the treewidth of the CNF formula and to directly encoding the effect of unit propagation on evidence into the CNF formula, without actually performing unit propagation. I also explore alternative search ordering heuristics for the DPLL-based backtracking algorithm. I experimentally evaluated my techniques on large-scale real and randomly generated Bayesian networks. On these benchmarks, my techniques gave speedups of up to two orders of magnitude over the best previous approaches for Bayesian networks with noisy-OR relations and scaled up to networks with larger numbers of random variables. My techniques extend the model counting approach for exact inference to networks that were previously intractable for the approach.

Further, many real world Bayesian network applications need to update their networks incrementally as new data becomes available. For example, the capability of updating a Bayesian network is crucial for building adaptive systems. I present techniques for improving the efficiency of exact inference in incrementally updated Bayesian networks by exploiting common structure. In particular, I propose and formalize the concept of dynamic weighted model counting and present an algorithm for performing dynamic model counting. The techniques I propose provide a general

approach for reusing partial results generated from answering previous queries based on the same or a similar Bayesian network. My focus is to improve the efficiency of exact inference when the network structure or the parameters or the evidence is updated. I show that my approach can be used to significantly improve inference on multiple challenging Bayesian network instances and other problems encoded as dynamic model counting problems.

A large part of the material in this thesis originates from the following publications.

- Wei Li and Peter van Beek. Guiding Real-World SAT Solving with Dynamic Hypergraph Separator Decomposition. In *Proceedings of the Sixteenth IEEE International Conference on Tools with Artificial Intelligence*, Boca Raton, Florida, 542–548, November, 2004.

- Wei Li, Peter van Beek, and Pascal Poupart. Performing Incremental Bayesian Inference by Dynamic Model Counting. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, Boston, Massachusetts, 1173–1179, July, 2006.

- Wei Li, Pascal Poupart, and Peter van Beek. Exploiting Causal Independence Using Weighted Model Counting. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, Chicago, Illinois, 337–343, July, 2008.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 introduces the concepts and definitions that are fundamental for understanding the thesis. In Chapter 3, I consider a knowledge base expressed in propositional logic and the inference engine is based on model finding. In this chapter, I present my work on dynamically decomposing propositional formulas during the backtracking search. In Chapters 4 & 5, I consider a knowledge base expressed as a Bayesian network and the inference engine is based on weighted model counting. In Chapter 4, I present my work on extending the weighted model counting approach to exact inference to Bayesian networks that contain the widely used noisy-OR and noisy-MAX relations. In Chapter 5, I present my work on improving the efficiency of exact inference in incrementally updated Bayesian networks. Chapter 6 concludes the thesis and suggests some possible future work.

Chapter 2

Background

In this chapter, I review the necessary background in knowledge representation and reasoning (KRR). A KRR system consists of a formal language for expressing declarative knowledge, a semantics which specifies the meaning of sentences in the language, and a reasoning or inference procedure for answering queries (see, e.g., Poole, Mackworth, and Goebel [83]). In the first half of this chapter, I review the case where the formal language of the KRR system is propositional logic and the inference procedure is based on backtracking search for determining satisfiability and model counting. Such systems are suitable for expressing and querying definite knowledge. In the second half of this chapter, I review the case where the formal language of the KRR system is based on probabilities, or more specifically, Bayesian networks. Such systems are suitable for expressing and querying indefinite or uncertain knowledge. For Bayesian networks, various algorithms are available for probabilistic inference. Of particular interest here are inference algorithms that take advantage of a relationship between model counting in propositional satisfiability and inference in Bayesian networks. (For more background on these topics, see van Harmelen, Lifschitz, and Porter [102], Darwiche [25], Koller and Friedman [72], Poole, Mackworth, and Goebel [83], Pearl [82], and Freuder [41].)

2.1 Propositional Logic and Logical Inference

In many real world tasks, such as planning and scheduling, the problem specification and the solution criteria can be expressed succinctly as a set of constraints derived either from expert knowledge or axioms. A problem that allows constraints to be explicitly stated can be naturally formulated and represented as a constraint satisfaction problem (CSP).

Definition 2.1 (CSP)**.** *A* constraint satisfaction problem *consists of a set of variables* $X = \{x_1, \ldots, x_n\}$*; a set of values* $D = \{a_1, ..., a_d\}$*, where each* $x_i \in X$ *has an associated finite domain* $dom(x_i) \subseteq D$ *of possible values; and a collection of constraints.*

Each constraint C is a relation—a set of tuples—over some set of variables, denoted by $vars(C)$. The size of the set $vars(C)$ is called the *arity* of the constraint. A

binary constraint is a constraint of arity two; a *non-binary* constraint is a constraint of arity greater than two. A *solution* to a CSP is an n-tuple $((x_1, a_1), \ldots, (x_n, a_n))$, where each variable is instantiated—i.e., assigned a value—from its domain such that all the constraints are simultaneously satisfied. If there is no solution the CSP is said to be unsatisfiable.

A special case of CSPs is where the variables are Boolean—i.e., each variable has the Boolean domain $\{true, false\}$—and the constraints are specified in propositional logic. I assume that the constraints are in conjunctive normal form (CNF) and are thus written as clauses. A *literal* is a Boolean variable (also called a *proposition*) or its negation and a *clause* is a disjunction of literals. A clause with one literal is called a *unit clause* and the literal in the unit clause is called a *unit literal*. A propositional formula F is in *conjunctive normal form* if it is a conjunction of clauses. Each clause corresponds to a constraint which must be satisfied.

Example 2.1. *For example,* $(x \vee \neg y)$ *is a clause, and the formula,*

$$F = (x \vee \neg y) \wedge (x \vee y \vee z) \wedge (y \vee \neg z \vee w) \wedge (\neg w \vee \neg z \vee v) \wedge (\neg v \vee u),$$

is in CNF, where u*,* v*,* w*,* x*,* y*, and* z *are propositions.*

Without loss of generality, we assume that no clause contains both a positive literal and a negative literal of the same proposition and that the literals in a clause are unique.

Definition 2.2 (SAT)**.** *Given a propositional formula in conjunctive normal form, the problem of determining whether there exists a variable assignment that makes the formula evaluate to* true*, or* $\exists x_1, \ldots, \exists x_n (F(x) = 1)$*?, is called the* Boolean satisfiability problem *or* SAT.

A variable assignment that makes a formula evaluate to *true* is also called a *model*. The problem of counting the number of models of a formula is called *model counting (MC)* or *#SAT*.

2.1.1 Backtracking Search and DPLL

Constraint satisfaction and propositional satisfiability are often solved using backtracking search. A backtracking search for a solution to a CSP or SAT instance can be seen as performing a depth-first traversal of a search tree. The search tree is generated as the search progresses and represents alternative choices that may have to be examined in order to find a solution or prove that no solution exists. Exploring a choice is also called *branching* and the order that choices are explored is also called the *branching strategy* or the *variable ordering heuristic*. Since the first formal statements of backtracking algorithms over 45 years ago [26, 48], many techniques for improving the efficiency of a backtracking algorithm have been suggested and evaluated (see, e.g., [101]). Below I describe the important improvements for backtracking algorithms with respect to SAT solving.

When specialized to SAT solving, backtracking algorithms are often referred to as being DPLL-based (see Algorithm 2.1), in honor of Davis, Putnam, Logemann, and Loveland, the authors of some of the earliest work in the field [27, 26]. DPLL-based SAT solvers can handle large propositional formulas and are widely applied in such problems as planning [71], scheduling [77], and hardware design verifications [11].

Algorithm 2.1: DPLL(F)

input : Propositional formula F in conjunctive normal form
output: Returns *true* if F is satisfiable; *false* otherwise

if F is an empty clause set **then**
  return *true*;
if F contains the empty clause **then**
  return *false*;
if F contains a unit clause **then**
  F = BCP(F);
choose an uninstantiated variable v in F using a variable ordering heuristic;
return (DPLL($F|_{v=false}$) or DPLL($F|_{v=true}$))

Let F denote a propositional formula. I use the value 0 interchangeably with the Boolean value *false* and the value 1 interchangeably with the Boolean value *true*. The notation $F|_{v=false}$ ($F|_{v=true}$) represents a new formula, called the *residual formula* obtained by replacing the variable v with *false* (*true*) in F and simplifying.

Definition 2.3 (Residual formula)**.** *Given a formula F and a variable v in F, the* residual formula $F|_{v=false}$ *($F|_{v=true}$) is obtained from F by,*

- *removing all clauses that contain v and evaluate to true, and*
- *deleting literal v (in case $v = false$) or $\neg v$ (in case $v = true$), from all clauses.*

Let s be a set of instantiated variables in F. The residual formula $F|_s$ is obtained by cumulatively reducing F by each of the variables in s.

Example 2.2. *Consider once again the propositional formula F given in Example 2.1. Suppose x is assigned* false*. The residual formula is given by,*

$$F|_{x=0} = (\neg y) \wedge (y \vee z) \wedge (y \vee \neg z \vee w) \wedge (\neg w \vee \neg z \vee v) \wedge (\neg v \vee u).$$

2.1.2 Boolean Constraint Propagation

As is clear, a CNF formula is satisfied if and only if each of its clauses is satisfied and a clause is satisfied if and only if at least one of its literals is equivalent to 1. In a unit clause, there is no choice and the value of the literal is said to be *forced* or *implied*. The process of *Boolean constraint propagation (BCP)* or *unit propagation* assigns all unit literals to the value 1. As well, the formula is simplified by removing the variables of the unit literals from the remaining clauses and removing clauses that

evaluate to *true* (i.e., the residual formula is obtained). This process keeps looking for new unit clauses and updating the formula until no unit clause remains. BCP and backtracking search are core operations for modern SAT solvers. Most (if not all) current state-of-the-art SAT solvers are based on BCP. In practice, for most SAT problems, a major portion of the run time is spent in the BCP process.

Algorithm 2.2: BCP(F)

input : Propositional formula F in conjunctive normal form
output: Returns simplified formula where no unit clauses remain

```
while there exists a unit clause c in F, where c is of the form (v) or (¬v) do
    remove c from F;
    if clause c is of the form (v) then
        remove ¬v from remaining clauses of F;
        remove every remaining clause that has v;
    else
        remove v from remaining clauses of F;
        remove every remaining clause that has ¬v;
return simplified formula F;
```

The procedure $BCP(F)$ returns the simplified formula, where no more unit clauses remain. In the pseudo-code from Algorithm 2.2, v is a unit literal. BCP performs unit propagation until there are no further implications. A conflict occurs when implications for setting the same variable to both *true* and *false* are produced.

Example 2.3. *Consider again the propositional formula* $F|_{x=0}$ *given in Example 2.2, where* x *has been assigned* false. *The unit clause* $(\neg y)$ *forces* y *to be assigned* false. *The residual formula is given by,*

$$F|_{x=0,y=0} = (z) \wedge (\neg z \vee w) \wedge (\neg w \vee \neg z \vee v) \wedge (\neg v \vee u).$$

In turn, the unit clause (z) *forces* z *to be assigned* true. *Similarly, the assignments* $w = 1$, $v = 1$, *and* $u = 1$ *are forced.*

2.1.3 Conflict Clause Learning

Most modern SAT solvers improve the basic DPLL algorithm through the use of *conflict clause learning* (or nogood recording) [28, 97]. Conflict clause learning is the process of recording information in the form of a new clause when a conflict is found. Consider the following CNF formula,

$$(\neg x \vee \neg y) \wedge (x \vee \neg z) \wedge (y \vee z).$$

If the variables x and y in the above formula have both been set to *false*, the result is the following formula: $\neg z \wedge z$. This is an obvious contradiction and thus it can be determined that the variables x and y cannot both be set to *false*. Therefore we can

modify the original formula to be: $(\neg x \vee \neg y) \wedge (x \vee \neg z) \wedge (y \vee z) \wedge (x \vee y)$. When the solver returns to this formula and sets the variable x to *false* next time, it will then be faced with the unit clause that forces y to be *true*.

Adding conflict clauses can make an instance easier to solve because it prunes parts of the search tree. However, additional clauses can also slow down the search process. First, the additional clauses can introduce an additional overhead in the variable ordering heuristic. This overhead can be significant when a large number of clauses are added. Second, the addition of non-effective clauses can prevent the formation of more effective clause by pruning the sub-tree in which the other clause would have been created.

2.1.4 Variable Ordering Heuristics for DPLL

Variable ordering heuristics for solving SAT can be either static or dynamic, depending on when the ordering is created. A static ordering is created before the search begins and used without modification during DPLL. A dynamic ordering is created during the search and takes into account dynamic and local characteristics of the search space. Empirical evidence shows that dynamic variable ordering heuristics consistently outperform static variable ordering heuristics, even though they introduce additional overhead into the search. Both static and dynamic heuristics are based on various guiding principles.

1. *Fail-first principle.* The idea of the fail-first principle is to pick the most constrained variable so as to detect failure as early as possible. The Maximum Occurrence of Clauses of Minimal Size (MOM) family of heuristics [38] are examples of heuristics that follow this principle.

2. *Propagation-first principle.* Heuristics guided by this principle give preference to a variable that can create a simple sub-problem through facilitation of unit propagation. An example is the the two sided J-W heuristic [60].

3. *Conflict analysis based principle.* This strategy can be viewed as attempting to satisfy the most recent conflict clauses. It is dynamic, since it gives preference to information received recently and therefore adjusts itself quickly to changes in the formula. It also has low overhead, since the statistics can be updated during conflict analysis. The conflict analysis based principle has been successfully applied in many SAT solvers, including the heuristics used in zChaff (Variable State Independent Decaying Sum ordering, or VSIDS) [104] and BerkMin [47].

2.1.5 Model Counting

Given a Boolean propositional formula F in CNF, the model counting problem (#SAT) is to determine the number of distinct assignments to variables for which the formula evaluates to *true*. For example, the number of models of F in Example 2.2 is 18.

The model counting problem can be solved with two different classes of algorithms: DPLL-based backtracking search and local search (see, e.g., [49] and references therein). Both methods have been applied in SAT. However, DPLL-based approaches determine the exact number of models, while local search approaches estimate the number of models with or without a correctness guarantee. Here, we focus on exact methods using DPLL-based exhaustive search. Algorithm 2.3 extends DPLL for SAT by adding the number of satisfiable instantiations together.

Algorithm 2.3: DPLL_MC(F)

input : Propositional formula F in conjunctive normal form
output: Returns the number of models of F

if F is an empty clause set **then**
  return 1;
else if F contains the empty clause **then**
  return 0;
else
  choose an uninstantiated variable v in F using a variable ordering heuristic;
  return (DPLL_MC($F|_{v=false}$) + DPLL_MC($F|_{v=true}$))

However, the simple extended DPLL algorithm shown above would not scale up to real world tasks. To avoid counting solutions of the same subproblems, *component caching* is applied in most modern model counting engines [9, 88, 100].

The definition of "component" is tightly related with the graph representation and tree decomposition of CNFs. Every CNF formula F is associated with an underlying hypergraph $H = (V, E)$, where V is the set of variables appearing in F and each clause generates a hyperedge in E containing the variables of the clause.

The *primal graph* of a hypergraph H is a graph G where V is the same set of variables in H and there exists an edge between the pair (v, w), $v \neq w$, such that v and w are in the same hyperedge.

Definition 2.4 (Tree decomposition of a primal graph [84, 13])**.** *A tree decomposition of a graph, $G = (V, E)$, is a triple (N, F, M), where*

- *N is a set of nodes and $F \subset N \times N$ is a set of arcs such that (N, F) forms a tree,*
- *$M : N \rightarrow 2^V$ forms a subset of vertices with each tree node,*
- *$\bigcup_{n \in N} M(n) = V$,*
- *for every edge $e \in E$, there is a node $n \in N$, such that $e \subseteq M(n)$,*
- *for every $n_1, n_2, n_3 \in N$, if n_2 is on the (unique) path between n_1 and n_3, then $M(n_1) \cap M(n_3) \subseteq M(n_2)$.*

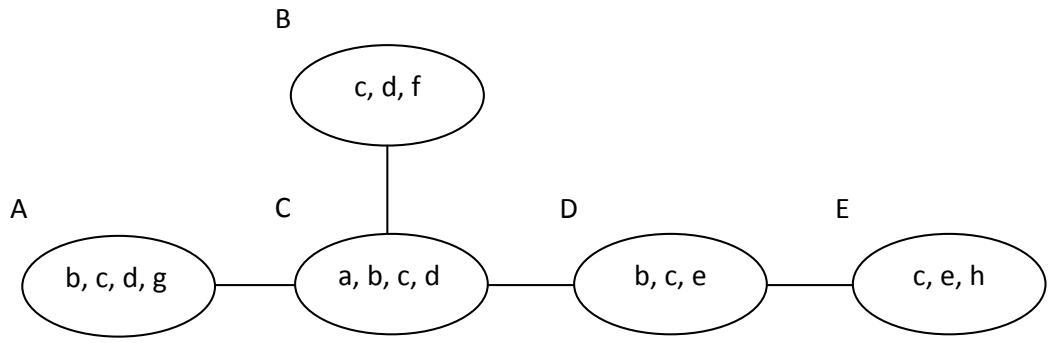


Figure 2.1: Tree decomposition of CNF formula F of Example 2.4.

The *treewidth* of a tree decomposition (N, F, M) is $\max_{n \in N} |M(n)| - 1$. The treewidth of a graph G is the minimum treewidth over all possible tree decompositions of G. Tree decompositions and treewidth can be generalized to hypergraphs in a straightforward manner; and the treewidth of a hypergraph is equal to the treewidth of its primal graph [1].

Example 2.4. *For example, consider the formula,*

$$F = (b \vee \neg c \vee e) \wedge (b \vee c \vee g) \wedge (c \vee d \vee \neg g) \wedge (\neg a \vee b \vee c) \wedge (a \vee b \vee \neg d) \wedge (\neg c \vee e \vee h) \wedge (c \vee \neg d \vee f).$$

A tree decomposition of the hypergraph associated with formula F is shown in Figure 2.1. The treewidth of the decomposition is 3.

The best decomposition leads to a time complexity in $O(n \cdot d^{w+1})$, where w is the treewidth of the network [84]. Because of the inefficiency of the approach, tree decomposition is often used when searching for all the solutions, such as model counting. Sang et al. [88] proposed to used component caching to reduce the required space in decomposition.

A *rooted* tree decomposition of a graph G is a tree decomposition (N, F, M) of G where some node $n \in N$ has been designated to be the root. An important property of a rooted tree decomposition of a graph G that has a small treewidth is that the graph G has small separators. The *separator* of a node $n \in N$ in a rooted tree decomposition is the set $S = M(n) \cap M(parent(n))$, where $parent(n)$ denotes the parent of node n in the rooted tree. Instantiating the Boolean variables in the separator S decomposes the initial formula into separate subproblems or components which can then be solved independently. The first subproblem is given by the node n and all of its descendants. The second subproblem is given by the remaining nodes. In component caching, disjoint components of the formula, generated dynamically during a DPLL search, are cached so that they only have to be solved once.

Definition 2.5 (Component)**.** *A* component *of a CNF formula F is a set of clauses Φ, the variables of which are disjoint from the variables in the remaining clauses $F - \Phi$.*

Example 2.5. *Consider once again the tree decomposition introduced in Example 2.4 and let the node labeled C in Figure 2.1 be designated the root of the tree. The*

separator of node D is given by,

$$\begin{aligned} S &= M(D) \cap M(parent(D)) \\ &= \{b, c, e\} \cap \{a, b, c, d\} \\ &= \{b, c\}. \end{aligned}$$

Instantiating the variables in {b, c} gives two components defined by the sets of variables {e, h} and {a, d, f, g}.

2.1.6 Stochastic Local Search

Constraint satisfaction and propositional satisfiability are also often solved using stochastic local search. A stochastic local search for a solution to a CSP or SAT instance can be seen as performing a traversal of a search graph. The search graph is generated as the search progresses and each node most often represents a complete assignment to each of the variables. A cost function, which applies to nodes, is then used to guide the search to a low cost or satisficing solution. A common cost function measures how many of the constraints or clauses are not satisfied by the current assignments to the variables. Many techniques for improving the efficiency of a stochastic local search algorithm have been suggested and evaluated (see, e.g., [63] and references therein).

Stochastic local search algorithms have several advantages including finding high quality solutions to optimization problems and sometimes quickly finding satisficing solutions, if they exist. Unfortunately, stochastic local search is not suitable for showing unsatisfiability and not as suitable for counting solutions. Our interest is in backtracking search algorithms and exact methods. Nevertheless, stochastic local search is an interesting and useful approach in CSP and SAT.

2.2 Bayesian Networks and Probabilistic Inference

In many real world tasks, such as diagnosis and classification, one must deal with significant uncertainty about the state of the world and our own observations of that world. For example, in medical diagnosis the patient's true disease may be only indirectly inferred through limited observations of symptoms and those symptoms may have unclear causal relationships with the possible diseases. In such application domains, a knowledge representation and reasoning system based on probabilistic models and probabilistic inference can often be used to advantage. Here our focus is on Bayesian networks [82], which are a fundamental building block of many AI applications.

For our purposes, a *probabilistic model* of an application domain consists of a set of random variables $\boldsymbol{X} = \{X_1, \ldots, X_n\}$; a set of values $D = \{a_1, ..., a_d\}$, where each $X_i \in \boldsymbol{X}$ has an associated finite domain $dom(X_i) \subseteq D$ of possible values; and a *joint probability distribution* $P(X_1, \ldots, X_n)$ over the possible assignments to the variables in $\boldsymbol{X}$. One can view an assignment of a value to each random variable in the model as

a complete specification of a state of the domain and the joint probability distribution as giving the probability of that state of the domain.

Example 2.6. *Consider a probabilistic model with four random variables, A, B, C, and D, where* $dom(A) = \{a_1, a_2\}$, ..., $dom(D) = \{d_1, d_2\}$. *One possible joint probability distribution is shown in Table 2.1.*

Table 2.1: Joint probability distribution over the possible assignments to the random variables A, B, C, and D, where $dom(A) = \{a_1, a_2\}$, ..., $dom(D) = \{d_1, d_2\}$.

A	B	C	D	$P(A, B, C, D)$
a_1	b_1	c_1	d_1	0.00135
a_1	b_1	c_1	d_2	0.00015
a_1	b_1	c_2	d_1	0.00280
a_1	b_1	c_2	d_2	0.00070
a_1	b_2	c_1	d_1	0.00855
a_1	b_2	c_1	d_2	0.01995
a_1	b_2	c_2	d_1	0.00665
a_1	b_2	c_2	d_2	0.05985
a_2	b_1	c_1	d_1	0.09113
a_2	b_1	c_1	d_2	0.01013
a_2	b_1	c_2	d_1	0.02700
a_2	b_1	c_2	d_2	0.00675
a_2	b_2	c_1	d_1	0.17213
a_2	b_2	c_1	d_2	0.40163
a_2	b_2	c_2	d_1	0.01913
a_2	b_2	c_2	d_2	0.17213

Give a joint probability distribution, one can answer probabilistic queries of interest, such as conditional probability queries of the form $P(\boldsymbol{Q} \mid \boldsymbol{E})$, where $\boldsymbol{E}$ is a subset of the random variables called the evidence variables and $\boldsymbol{Q}$ is a subset of the random variables called the query variables. Unfortunately, as shown the joint probability grows exponentially in the sizes of the domains of the random variables. This has led to methods for compactly representing a joint probability distribution, such as Bayesian networks.

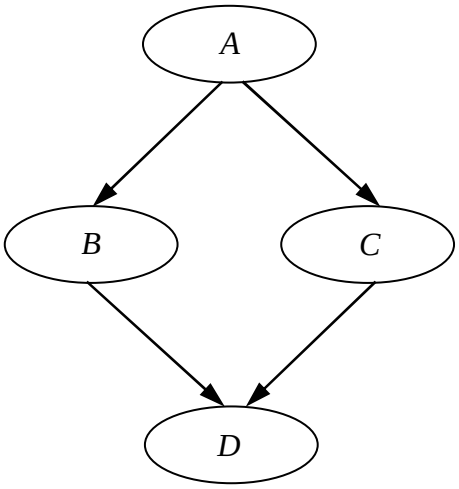
Definition 2.6 (Bayesian network)**.** *A* Bayesian network *representation of a probabilistic model is a directed acyclic graph where the nodes are the random variables and each node is labeled with a conditional probability table (CPT) which specifies the strengths of the influences of the parent nodes on the child node.*

Example 2.7. *Consider once again the probabilistic model with four random variables, A, B, C, and D, where* $dom(A) = \{a_1, a_2\}$, ..., $dom(D) = \{d_1, d_2\}$. *Figure 2.2 shows a Bayesian network representation of the joint probability distribution given in Table 2.1.*

One can view a Bayesian network as a factorized representation of the joint probability distribution as any entry in the joint probability distribution can be expressed as the product,

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i \mid parents(X_i)), \tag{2.1}$$

where n is the size of the Bayesian network and $parents(X_i)$ is the set of parents of X_i in the directed graph. One can also view a Bayesian network as an encoding of a collection of conditional independence assumptions or postulates. It is the conditional independence assumptions that allow the factorization and lead to a compact encoding of the joint probability distribution.



$P(A = a_1)$	$P(A = a_2)$
0.1	0.9

A	$P(B = b_1 \mid A)$
a_1	0.05
a_2	0.15

A	$P(B = b_2 \mid A)$
a_1	0.95
a_2	0.85

A	$P(C = c_1 \mid A)$
a_1	0.30
a_2	0.75

A	$P(C = c_2 \mid A)$
a_1	0.70
a_2	0.25

B	C	$P(D = d_1 \mid B, C)$	$P(D = d_2 \mid B, C)$
b_1	c_1	0.9	0.1
b_1	c_2	0.8	0.2
b_2	c_1	0.3	0.7
b_2	c_2	0.1	0.9

Figure 2.2: A Bayesian network over random variables A, B, C, and D, where $dom(A) = \{a_1, a_2\}$, ..., $dom(D) = \{d_1, d_2\}$.

2.2.1 Variable Elimination

The two standard exact algorithms for Bayesian networks are variable elimination and junction tree. Junction tree algorithms are often preferred as they pre-compute results and so can answer queries faster. However, there are large real-world networks that junction tree cannot deal with due to time and space complexities. In such networks, variable elimination can sometimes still answer queries because it permits pruning of irrelevant variables.

The brute-force method approach of computing the joint probability of a Bayesian network is to multiply all the conditional probabilities of the network. Then other marginal or conditional probability can be obtained from it. However, the complexity of this method grows exponentially with the size of the network. One of the ways of

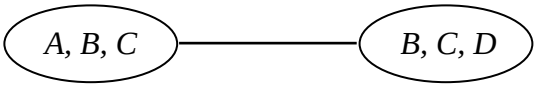


Figure 2.3: Junction tree of the Bayesian network shown in Figure 2.2.

avoiding this problem is to sum out some variables before multiplying all the factors. This is the intuition behind variable elimination algorithms.

Example 2.8. *Consider once again the Bayesian network shown in Figure 2.2. Given the evidence $D = d_1$, the posteriori probability of A can be computed using variable elimination [5] as follows:*

$$\begin{aligned} P(a, d_1) &= \sum_b \sum_c P(a, b, c, d_1) \\ &= \sum_b \sum_c P(a) \cdot P(b \mid a) \cdot P(c \mid a) \cdot P(d_1 \mid b, c) \\ &= P(a) \cdot \sum_b P(b \mid a) \cdot \sum_c P(c \mid a) \cdot P(d_1 \mid b, c) \\ P(a \mid d_1) &= \frac{P(a, d_1)}{P(d_1)} = \frac{P(a, d_1)}{\sum_a P(a, d_1)} \end{aligned}$$

Variable elimination requires an elimination ordering—an ordering in which to eliminate or process the random variables. The elimination ordering can have a great impact on the efficiency of the algorithm. Good heuristics exist for selecting an elimination ordering (see, e.g., [72] and references therein).

2.2.2 Junction Trees

Variable elimination algorithms are inefficient if the undirected graph underlying the Bayesian network contains cycles. We can avoid cycles if we turn highly-interconnected subsets of the nodes into independent nodes. A junction tree is an undirected tree in which a set of random variables have a Markov property: a tree decomposition maps a Bayesian network into a tree whose nodes contains all the nodes of the graph; for each node pair u, v of tree nodes with intersection S, all tree nodes on the path between u and v contain S. For example, Figure 2.3 shows the junction tree of Bayesian network in Figure 2.2. A node in the junction tree corresponds to the other variables required to remove a variable during variable elimination. An arc in the junction tree shows the flow of data in the elimination computation. The width of a junction tree is the maximum size of the set $V(n)$ over all the nodes of the tree minus 1. For example, the width of the junction tree shown in Figure 2.3 is 2. The treewidth of a Bayesian network is the minimum width of any junction tree for that network. It is NP hard to determine treewidth. However, good heuristics exist for constructing junction trees of small width (see, e.g., [72] and references therein).

2.2.3 Recursive Conditioning

The basic idea of recursive conditioning is to recursively decompose a Bayesian network reasoning problem by branching on variables [22]. Recursive conditioning determines a static variable ordering before branching starts. Once a subproblem is created, recursive conditioning attempts to allocate enough space to cache the results of all evaluated subproblems. Unique features of the algorithm are that it can use as much space as is available and it offers a smooth tradeoff between time and space.

2.2.4 Weighted Model Counting

There are natural polynomial-time reductions between the Bayesian inference problem and model counting problems [4]. Darwiche has described a method for compiling Bayesian networks as a set of multi-linear functions [24]. As well, recent work on inference in Bayesian networks reduces the problem to weighted model counting of CNFs [23, 76, 89]. Weighted model counting is a generalization of model counting.

Definition 2.7. *A weighted model counting problem consists of a CNF formula F and for each variable v in F, a weight for each literal: $weight(v)$ and $weight(\neg v)$. Let s be an assignment of a value to every variable in the formula F that satisfies the formula; i.e., s is a model of the formula. The weight of s is the product of the weights of the literals in s. The solution of a weighted model counting problem is the sum of the weights of all satisfying assignments; i.e.,*

$$weight(F) = \sum_{s} \prod_{l \in s} weight(l),$$

where the sum is over all possible models and the product is over the literals in that model.

Any backtracking algorithm for SAT can be easily extended to count the number of models by simply forcing it to backtrack whenever a solution is found [4]. Algorithm 2.4, the basic weighted model counting algorithm, shows that a DPLL-based backtracking search algorithm can also easily be extended to determine the weighted model count of a CNF formula F [89].

Algorithm 2.4: BWMC(F)

input : Propositional formula F in conjunctive normal form
output: Returns sum of the weights of all models of F

if F is an empty clause set **then**
  return 1;
if F contains the empty clause **then**
  return 0;
choose an uninstantiated variable v in F using a variable ordering heuristic;
return BWMC($F|_{v=0}$) $\times$ $weight(\neg v)$ + BWMC($F|_{v=1}$) $\times$ $weight(v)$;

Darwiche [23, 19] proposed an encoding of a Bayesian network into weighted model counting of a propositional formula in conjunctive normal form. Darwiche's encoding proceeds as follows. At each step, I illustrate the encoding using the Bayesian network shown in Figure 2.2. To improve clarity, I refer to the random variables in the Bayesian network as "nodes" and reserve the word "variables" for the Boolean variables in the resulting propositional formula.

- For each value of each node in the Bayesian network, an *indicator variable* is created,

$$\begin{array}{rcl@{\qquad}rcl} A & : & I_{a_1},\, I_{a_2}, & C & : & I_{c_1},\, I_{c_2}, \\ B & : & I_{b_1},\, I_{b_2}, & D & : & I_{d_1},\, I_{d_2}. \end{array}$$

- For each node, *indicator clauses* are generated which ensure that in each model exactly one of the corresponding indicator variables for each node is true,

$$\begin{array}{rcl@{\qquad}rcl} A & : & (I_{a_1} \vee I_{a_2}) \wedge (\neg I_{a_1} \vee \neg I_{a_2}), & C & : & (I_{c_1} \vee I_{c_2}) \wedge (\neg I_{c_1} \vee \neg I_{c_2}), \\ B & : & (I_{b_1} \vee I_{b_2}) \wedge (\neg I_{b_1} \vee \neg I_{b_2}), & D & : & (I_{d_1} \vee I_{d_2}) \wedge (\neg I_{d_1} \vee \neg I_{d_2}). \end{array}$$

- For each conditional probability table (CPT) and for each non-zero parameter (probability) value in the CPT, a *parameter variable* is created,

$$\begin{array}{rcl@{\qquad}rcl} A & : & P_{a_1}, \quad P_{a_2}, & C & : & P_{c_1|a_1}, \quad P_{c_1|a_2}, \\ & & & & & P_{c_2|a_1}, \quad P_{c_2|a_2}, \\ B & : & P_{b_1|a_1}, \quad P_{b_1|a_2}, & D & : & P_{d_1|b_1,c_1}, \quad P_{d_1|b_1,c_2}, \\ & & P_{b_2|a_1}, \quad P_{b_2|a_2}, & & & P_{d_1|b_2,c_1}, \quad P_{d_1|b_2,c_2}, \\ & & & & & P_{d_2|b_1,c_1}, \quad P_{d_2|b_1,c_2}, \\ & & & & & P_{d_2|b_2,c_1}, \quad P_{d_2|b_2,c_2}. \end{array}$$

- For each parameter variable, a *parameter clause* is generated. A parameter clause asserts that the conjunction of the corresponding indicator variables implies the parameter variable and vice-versa,

$$\begin{array}{rcl@{\qquad}l} A & : & I_{a_1} \Leftrightarrow P_{a_1} & I_{a_2} \Leftrightarrow P_{a_2} \\ B & : & I_{a_1} \wedge I_{b_1} \Leftrightarrow P_{b_1|a_1} & I_{a_2} \wedge I_{b_1} \Leftrightarrow P_{b_1|a_2} \\ & & I_{a_1} \wedge I_{b_2} \Leftrightarrow P_{b_2|a_1} & I_{a_2} \wedge I_{b_2} \Leftrightarrow P_{b_2|a_2} \\ C & : & I_{a_1} \wedge I_{c_1} \Leftrightarrow P_{c_1|a_1} & I_{a_2} \wedge I_{c_1} \Leftrightarrow P_{c_1|a_2} \\ & & I_{a_1} \wedge I_{c_2} \Leftrightarrow P_{c_2|a_1} & I_{a_2} \wedge I_{c_2} \Leftrightarrow P_{c_2|a_2} \\ D & : & I_{b_1} \wedge I_{c_1} \wedge I_{d_1} \Leftrightarrow P_{d_1|b_1c_1} & I_{b_2} \wedge I_{c_1} \wedge I_{d_1} \Leftrightarrow P_{d_1|b_2c_1} \\ & & I_{b_1} \wedge I_{c_2} \wedge I_{d_1} \Leftrightarrow P_{d_1|b_1c_2} & I_{b_2} \wedge I_{c_2} \wedge I_{d_1} \Leftrightarrow P_{d_1|b_2c_2} \\ & & I_{b_1} \wedge I_{c_1} \wedge I_{d_2} \Leftrightarrow P_{d_2|b_1c_1} & I_{b_2} \wedge I_{c_1} \wedge I_{d_2} \Leftrightarrow P_{d_2|b_2c_1} \\ & & I_{b_1} \wedge I_{c_2} \wedge I_{d_2} \Leftrightarrow P_{d_2|b_1c_2} & I_{b_2} \wedge I_{c_2} \wedge I_{d_2} \Leftrightarrow P_{d_2|b_2c_2}. \end{array}$$

- A weight is assigned to each literal in the propositional formula. Each positive literal of a parameter variable is assigned a weight equal to the corresponding probability entry in the CPT table,

$$
\begin{array}{lcll}
A & : & weight(P_{a_1}) = P(a_1) & weight(P_{a_2}) = P(a_2) \\
B & : & weight(P_{b_1|a_1}) = P(b_1 \mid a_1) & weight(P_{b_1|a_2}) = P(b_1 \mid a_2) \\
 & & weight(P_{b_2|a_1}) = P(b_2 \mid a_1) & weight(P_{b_2|a_2}) = P(b_2 \mid a_2) \\
C & : & weight(P_{c_1|a_1}) = P(c_1 \mid a_1) & weight(P_{c_1|a_2}) = P(c_1 \mid a_2) \\
 & & weight(P_{c_2|a_1}) = P(c_2 \mid a_1) & weight(P_{c_2|a_2}) = P(c_2 \mid a_2) \\
D & : & weight(P_{d_1|b_1,c_1}) = P(d_1 \mid b_1, c_1) & weight(P_{d_1|b_1,c_2}) = P(d_1 \mid b_1, c_2) \\
 & & weight(P_{d_1|b_2,c_1}) = P(d_1 \mid b_2, c_1) & weight(P_{d_1|b_2,c_2}) = P(d_1 \mid b_2, c_2) \\
 & & weight(P_{d_2|b_1,c_1}) = P(d_2 \mid b_1, c_1) & weight(P_{d_2|b_1,c_2}) = P(d_2 \mid b_1, c_2) \\
 & & weight(P_{d_2|b_2,c_1}) = P(d_2 \mid b_2, c_1) & weight(P_{d_2|b_2,c_2}) = P(d_2 \mid b_2, c_2)
\end{array}
$$

  All other literals (both positive and negative) are assigned a weight of 1. I.e., $weight(I_{a_1}) = weight(\neg I_{a_1}) = \cdots = weight(I_{d_2}) = weight(\neg I_{d_2}) = 1$ and $weight(\neg P_{a_1}) = \cdots = weight(\neg P_{d_2|b_2,c_2}) = 1$.

Sang, Beame, and Kautz [89] (hereafter, just Sang) introduced an alternative encoding of a Bayesian network into weighted model counting of a CNF formula. Sang's encoding creates fewer variables and clauses, but the size of generated clauses of multi-valued variables can be larger. As with Darwiche's encoding presented above, I illustrate Sang's encoding using the Bayesian network shown in Figure 2.2.

- As in Darwiche's encoding, for each node, *indicator variables* are created and *indicator clauses* are generated which ensure that in each model exactly one of the corresponding indicator variables for each node is true.

- Assume that the values of the nodes are linearly ordered. For each CPT entry $P(Y = y \mid \boldsymbol{X})$ such that y is not the last value in the domain of Y, a parameter variable $P_{y|\boldsymbol{X}}$ is created; e.g.,

$$
\begin{array}{lcll}
A & : & P_{a_1}, & \\
B & : & P_{b_1|a_1}, & P_{b_1|a_2}, \\
C & : & P_{c_1|a_1}, & P_{c_1|a_2}, \\
D & : & P_{d_1|b_1,c_1}, & P_{d_1|b_1,c_2}, \\
 & & P_{d_1|b_2,c_1}, & P_{d_1|b_2,c_2}.
\end{array}
$$

- For each CPT entry $P(Y = y_i \mid \boldsymbol{X})$, a parameter clause is generated. Let the ordered domain of Y be $\{y_1, \ldots, y_k\}$ and let $\boldsymbol{X} = x_1, \ldots, x_l$. If y_i is not the last value in the domain of Y, the clause is given by,

$$I_{x_1} \wedge \cdots \wedge I_{x_l} \wedge \neg P_{y_1|\boldsymbol{X}} \wedge \cdots \wedge \neg P_{y_{i-1}|\boldsymbol{X}} \wedge P_{y_i|\boldsymbol{X}} \Rightarrow I_{y_i}.$$

If y_i is the last value in the domain of Y, the clause is given by,

$$I_{x_1} \wedge \cdots \wedge I_{x_l} \wedge \neg P_{y_1|\boldsymbol{X}} \wedge \cdots \wedge \neg P_{y_{k-1}|\boldsymbol{X}} \Rightarrow I_{y_k}.$$

For my running example, the following parameter clauses would be generated,

$$\begin{array}{lll}
A & : & P_{a_1} \Rightarrow I_{a_1} \qquad\qquad \neg P_{a_1} \Rightarrow I_{a_2} \\
B & : & I_{a_1} \wedge P_{b_1|a_1} \Rightarrow I_{b_1} \qquad I_{a_1} \wedge \neg P_{b_1|a_1} \Rightarrow I_{b_2} \\
 & & I_{a_2} \wedge P_{b_1|a_2} \Rightarrow I_{b_1} \qquad I_{a_2} \wedge \neg P_{b_1|a_2} \Rightarrow I_{b_2} \\
C & : & I_{a_1} \wedge P_{c_1|a_1} \Rightarrow I_{c_1} \qquad I_{a_1} \wedge \neg P_{c_1|a_1} \Rightarrow I_{c_2} \\
 & & I_{a_2} \wedge P_{c_1|a_2} \Rightarrow I_{c_1} \qquad I_{a_2} \wedge \neg P_{c_1|a_2} \Rightarrow I_{c_2} \\
D & : & I_{b_1} \wedge I_{c_1} \wedge P_{d_1|b_1,c_1} \Rightarrow I_{d_1} \qquad I_{b_1} \wedge I_{c_1} \wedge \neg P_{d_1|b_1,c_1} \Rightarrow I_{d_2} \\
 & & I_{b_1} \wedge I_{c_2} \wedge P_{d_1|b_1,c_2} \Rightarrow I_{d_1} \qquad I_{b_1} \wedge I_{c_2} \wedge \neg P_{d_1|b_1,c_2} \Rightarrow I_{d_2} \\
 & & I_{b_2} \wedge I_{c_1} \wedge P_{d_1|b_2,c_1} \Rightarrow I_{d_1} \qquad I_{b_2} \wedge I_{c_1} \wedge \neg P_{d_1|b_2,c_1} \Rightarrow I_{d_2} \\
 & & I_{b_2} \wedge I_{c_2} \wedge P_{d_1|b_2,c_2} \Rightarrow I_{d_1} \qquad I_{b_2} \wedge I_{c_2} \wedge \neg P_{d_1|b_2,c_2} \Rightarrow I_{d_2}.
\end{array}$$

- A weight is assigned to each literal in the propositional formula. As in Darwiche's encoding, the weight of literals for indicator variables is always 1. The weight of literals for each parameter variable $P_{y|\boldsymbol{X}}$ is given by,

$$\begin{aligned}
weight(P_{y|\boldsymbol{X}}) &= P(y \mid \boldsymbol{X}), \\
weight(\neg P_{y|\boldsymbol{X}}) &= 1 - P(y \mid \boldsymbol{X}).
\end{aligned}$$

Let F be the CNF encoding of a Bayesian network (either Darwiche's encoding or Sang's encoding). A general query $P(\boldsymbol{Q} \mid \boldsymbol{E})$ on the network can be answered by,

$$\frac{weight(F \wedge Q \wedge E)}{weight(F \wedge E)}, \tag{2.2}$$

where Q and E are propositional formulas which enforce the appropriate values for the indicator variables that correspond to the known values of the random variables. For example, given the query $P(A = a_2 \mid C = c_2, D = d_1)$, we would calculate, $weight(F \wedge I_{a_2} \wedge I_{c_2} \wedge I_{d_1})/weight(F \wedge I_{c_2} \wedge I_{d_1})$.

In this section, I compared variable elimination, junction tree, recursive conditioning and weighted model counting. In general, "good" variable orderings are the fundamental of all the approaches. For example, the process of "ordering" the factors and the sums often results in a more efficient computation for the variable elimination algorithm (see Example 2.8). Naturally, in networks with loops, the main difficulty of variable elimination consists is finding the optimal elimination order. The ability to manage the cache dynamically is also of crucial importance for space efficiency during the computation. Table 2.2 shows that weighted model counting is one of the only reasoning approaches that can use a dynamic strategy for both variable ordering and cache management.

Table 2.2: A comparison of algorithms for answering probabilistic queries.

algorithm	branch ordering	cache space	cache 'good'	cache 'nogood'
variable elimination	dynamic	dynamic	yes	yes
junction tree	static	none	no	no
recursive conditioning	static	static	yes	no
weighted model counting	dynamic	dynamic	yes	yes

Encoding inference in a Bayesian network as weighted model counting on a CNF formula has several advantages over traditional reasoning approaches: it allows advances in SAT solving techniques—such as non-chronological backtracking, clause learning and efficient variable selection heuristics—to be used; it provides a standard framework for evaluating different approaches and ideas, and it allows for dynamic decompositions and unit propagation.

2.3 Summary

In this chapter, I described the necessary background in knowledge representation and reasoning (KRR) systems based on propositional logic and KRR systems based on Bayesian networks and I reviewed algorithms for answering queries in both types of systems.

In the next chapter, I present my work on dynamically decomposing propositional formulas in CNF, for improving the efficiency of SAT solvers and SAT model counters.

Chapter 3

Exploiting Structure in Propositional Reasoning: Dynamic Decomposition

The general solution of satisfiability problems is NP-Complete. Although state-of-the-art SAT solvers can efficiently obtain the solutions of many real-world instances, there are still a large number of real-world SAT families which cannot be solved in a reasonable amount of time. Much effort has been devoted to devising techniques for taking advantage of the internal structure of SAT instances. However, most existing decomposition techniques are based on preprocessing the *static* structure of the original problem. In this chapter, I present a *dynamic* decomposition method based on hypergraph separators. Integrating the separator decomposition into the variable ordering of a modern SAT solver leads to speedups on large real-world satisfiability problems. In comparison to static decomposition based variable orderings (such as, for example, Huang and Darwiche's [64] dTree method), my approach does not need time to construct the full tree decomposition, which sometimes takes more time than the solving process itself.

My primary focus is to achieve speedups on large real-world satisfiability problems. I combined the state-of-the-art SAT solver zChaff [104] with dynamic hypergraph separator decomposition and empirically evaluated it on SAT 2002 competition benchmarks. My results show that the new solver can significantly outperform both regular zChaff and zChaff integrated with dTree decomposition in solving real-world problems. Furthermore, the new solver solved more hard instances than the dTree decomposition within a given cutoff time limit.

3.1 Structure in CSP and SAT instances

In this section, I review the definitions and concepts for capturing the structure of real-world CSP and SAT instances (for more background on this topic, see Dechter [29]). The term "structure" means the instance's structural properties, which can be represented as properties of the constraint graph or constraint hypergraph. Many

real world problems can be broken up and separated into disconnected components or subproblems by various decomposition approaches based on their following nature.

- Real-world applications are built and designed in a modular way. Modularization with minimal inter-connectivity is encouraged in industrial design and development.
- Each module uses distinct name spaces and variables.
- Small sets of variables are designed to control an application.

The underlying structure of a propositional CNF formula can be captured using what are called the primal graph and primal hypergraph representations.

Definition 3.1 (Primal graph)**.** *Given a propositional formula F in CNF, its* primal graph *representation $G = (V, E)$ is a graph whose vertex set V consists of the variables in F, and an edge connects each pair of nodes whose variables appear in the same clause.*

Let $H = (V, E)$ be a hypergraph with vertex set V and edge set E. A hypergraph is a generalization of a graph where a hyperedge can be any non-empty subset of V.

Definition 3.2 (Primal hypergraph)**.** *Given a propositional formula F in CNF, its* primal hypergraph *representation $H = (V, E)$ is a hypergraph whose vertex set V consists of the variables in F, and there is a hyperedge for each clause in F which consists of the set of all the variables that appear in that clause.*

Example 3.1. *Let F be the following propositional formula,*

$$F = (x \vee y \vee z) \wedge (y \vee \neg z \vee w) \wedge (\neg w \vee \neg z \vee v) \wedge (\neg v \vee u),$$

Figure 3.1 shows both the primal graph and the primal hypergraph of formula F, where the hyperedges are $\{x, y, z\}$, $\{y, z, w\}$, $\{w, z, v\}$, and $\{v, u\}$.

The underlying structure of a propositional CNF formula can also be captured using what are called the dual graph and dual hypergraph representations.

Definition 3.3 (Dual graph)**.** *Given a propositional formula F in CNF, its* dual graph *representation $G = (V, E)$ is a graph whose vertex set V consists of the clauses in F, and an edge connects each pair of nodes whose clauses share a variable.*

Definition 3.4 (Dual hypergraph)**.** *Given a propositional formula F in CNF, its* dual hypergraph *representation $H = (V, E)$ is a hypergraph whose vertex set V consists of the clauses in F, and there is a hyperedge for each Boolean variable in F which consists of the set of all the clauses (vertices) that contain that variable.*

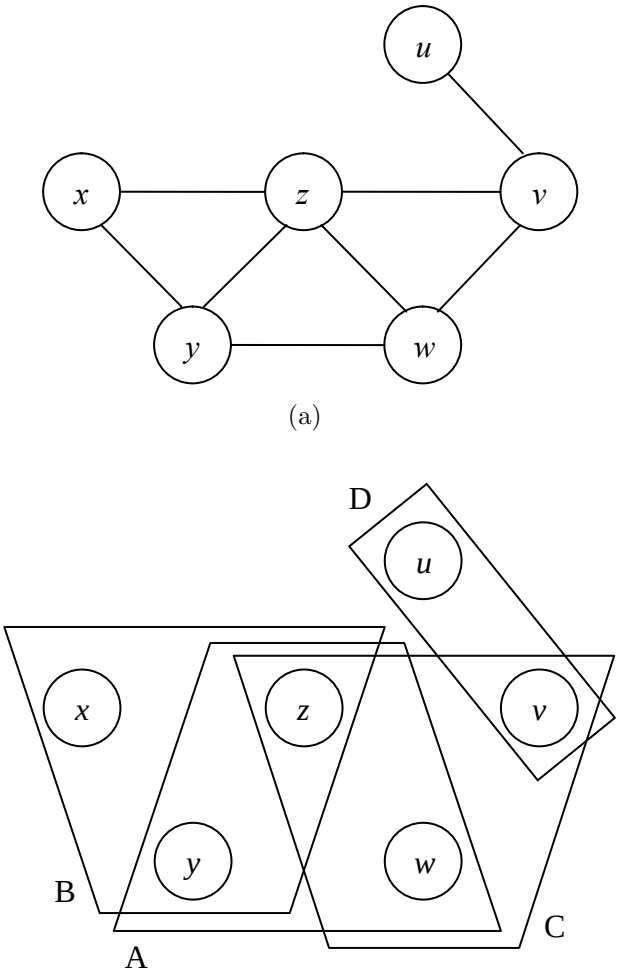


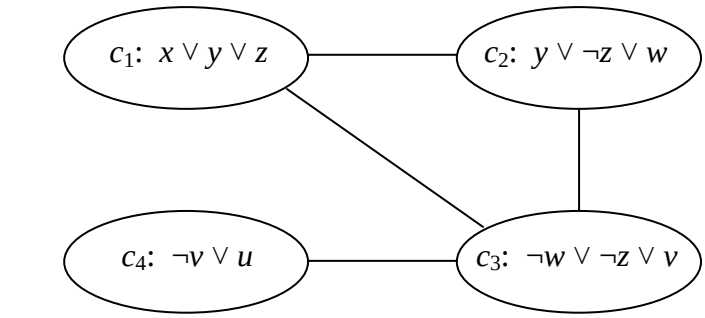
Figure 3.1: (a) Primal graph and (b) primal hypergraph of the propositional formula $F = (x \vee y \vee z) \wedge (y \vee \neg z \vee w) \wedge (\neg w \vee \neg z \vee v) \wedge (\neg v \vee u)$ in Example 3.1. The edges in the primal hypergraph are $\{x, y, z\}$, $\{y, z, w\}$, $\{w, z, v\}$, and $\{v, u\}$.

Example 3.2. *Once again, let F be the following propositional formula,*

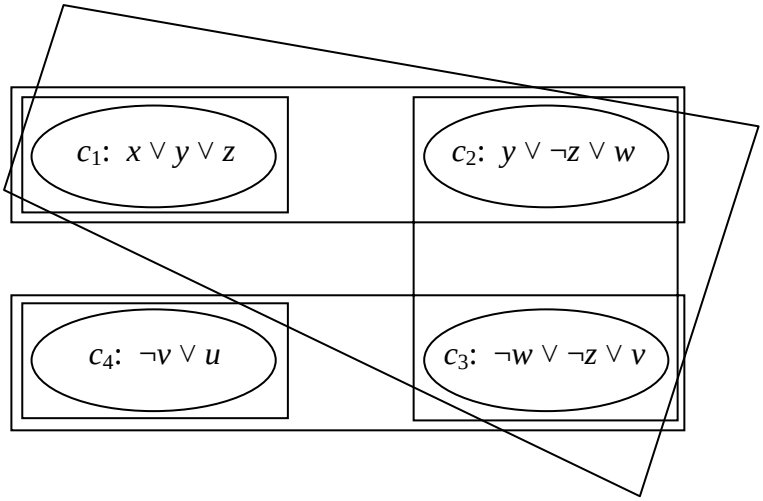
$$F = (x \vee y \vee z) \wedge (y \vee \neg z \vee w) \wedge (\neg w \vee \neg z \vee v) \wedge (\neg v \vee u),$$

Figure 3.2 shows both the dual graph and the dual hypergraph of formula F. Let c_1, c_2, c_3, and c_4 be the four clauses in the formula. The dual hypergraph $H = (V, E)$ has $V = \{c_1, c_2, c_3, c_4\}$ and $E = \{\{c_1\}, \{c_1, c_2\}, \{c_1, c_2, c_3\}, \{c_2, c_3\}, \{c_3, c_4\}, \{c_4\}\}$, corresponding to the clauses in which x, y, z, w, v, and u appear, respectively.

In an undirected graph G, two vertices u and v are called *connected* if G contains a path from u to v. Otherwise, they are called disconnected. A graph is called *connected* if every pair of distinct vertices in the graph can be connected through some path. A *cut* or *vertex cut* of a connected graph G is a set of vertices whose



(a)



(b)

Figure 3.2: (a) Dual graph and (b) dual hypergraph of the propositional formula $F = (x \vee y \vee z) \wedge (y \vee \neg z \vee w) \wedge (\neg w \vee \neg z \vee v) \wedge (\neg v \vee u)$ in Example 3.2. The edges in the dual hypergraph are $\{c_1\}$, $\{c_1, c_2\}$, $\{c_1, c_2, c_3\}$, $\{c_2, c_3\}$, $\{c_3, c_4\}$, and $\{c_4\}$

removal renders G disconnected. The *connectivity* of G is the size of a smallest vertex cut. A graph is called *k-connected* if its connectivity is k or greater. 2-connectivity is also called *biconnectivity* and 3-connectivity is also called *tri-connectivity*. A vertex whose removal disconnects a connected graph is called an *articulation point*. A graph without any articulation point is biconnected. The *biconnected components* of a graph are the maximal biconnected subgraphs.

A *path* in a hypergraph connecting vertex x and y is a sequence $x = x_1, E_1, x_2, \ldots, x_i, E_i, x_{i+1}, \ldots, E_p, x_{p+1} = y$ with $x_i, x_{i+1} \in E_i$. In a hypergraph H, two vertices x and y are called *connected* if H contains a path from x to y. H is called *connected* if every pair of distinct vertices in the hypergraph can be connected through some path. *A separator of a hypergraph* G is a set of hyperedges whose removal chops G into disjoint sub-hypergraphs whose sizes stand in some sought relation.

My focus is on how to decompose a SAT instance into independent subproblems by analyzing its hypergraph. Existing decomposition techniques are based on preprocess-

ing the static structure of the original problem. I propose a dynamic decomposition method based on hypergraph separators. Integrating this dynamic separator decomposition into the variable ordering of a modern SAT solver leads to speedups on large real world SAT instances. I also observe that previously proposed conflict-guided variable ordering heuristics do not work well when combined with static decomposition strategies.

3.2 Related Work

In this section, I relate my work to previously proposed methods for using structure to guide backtracking search in CSPs and SAT.

As explained above, a CSP/SAT instance can be represented as a graph. Such graphical representations form the basis of structure-guided variable ordering heuristics. Real problems often do contain much structure and on these problems the advantages of structure-guided heuristics include that structural parameters can be used to bound the worst-case of a backtracking algorithm and structural goods and nogoods can be recorded and used to prune large parts of the search space. Unfortunately, a current limitation of these heuristics is that they can break down in the presence of constraints or clauses that encompass all or nearly all of the variables, which can be common in practice. A further disadvantage is that structure-guided heuristics are usually static or nearly static.

Freuder [39] may have been the first to propose a structure-guided variable ordering heuristic. Consider the constraint graph where there is a vertex for each variable in the CSP and there is an edge between two vertices x and y if there exists a constraint C such that both $x \in vars(C)$ and $y \in vars(C)$.

Definition 3.5 (Width)**.** *Let the vertices in a constraint graph be ordered. The* width *of an ordering is the maximum number of edges from any vertex v to vertices prior to v in the ordering. The width of a constraint graph is the minimum width over all orderings of that graph.*

Consider the static variable ordering corresponding to an ordering of the vertices in the graph. Freuder [39] shows that the static variable ordering is backtrack-free if the level of strong k-consistency (a form of constraint propagation) is greater than the width of the ordering. Freuder [39] also shows that there exists a backtrack-free static variable ordering if the level of strong consistency is greater than the width of the constraint graph. Freuder [40] generalizes these results to static variable orderings which guarantee that the number of nodes visited in the search can be bounded *a priori*.

Dechter and Pearl [31] propose a static variable ordering which first instantiates variables which cut cycles in the constraint graph. Once all cycles have been cut, the constraint graph is a tree and can be solved quickly using a form of constraint propagation called arc consistency [39]. Sabin and Freuder [87] refine and test this proposal within an algorithm that maintains arc consistency. They show that, on

random binary problems, a variable ordering that cuts cycles can out perform state-of-the-art heuristics.

As a further example of a structure-guided variable ordering heuristic, Moskewicz et al. [79], in their Chaff solver for SAT, propose that the choice of variable should be biased towards variables that occur in recently recorded nogoods.

A well-known technique in algorithm design on graphs is divide-and-conquer using graph separators.

Definition 3.6 (Separator)**.** *A separator of a graph is a subset of the vertices or the edges which, when removed, separates the graph into disjoint subgraphs.*

A graph can be recursively decomposed by successively finding separators of the resulting disjoint subgraphs leading to a tree of separators or a decomposition tree. The technique has been widely applied in constraint graphs and graphical models such as Bayesian networks (see, e.g., [32, 12, 65]). Since the solution of both CSP and SAT problems are NP-Complete, tree decomposition plays an important role to bound their runtime. The worst-case complexity of solving SAT problems using the original DPLL is $O(m2^n)$, where m is the number of clauses and n is the number of variables. So an efficient general algorithm is not expected to be found. In order to improve this worst-case complexity, a variety of structural decomposition methods have been investigated. The best known tree decomposition leads to a time complexity in $O(n2^{w+1})$, where w is the width of the hypergraph representation of the SAT problem.

The idea of decomposing a CSP into smaller pieces with limited interconnections was first explored by Freuder [40]. Freuder showed that a binary CSP (a CSP with only constraints over pairs of variables) can be decomposed into smaller "biconnected components" and solved separately. Gyssens [51] extended this idea to hypergraphs by introducing hinge decomposition. Freuder's approach [40] is to obtain a static variable ordering by finding the articulation nodes and constructs a tree whose nodes are biconnected components of the original constraint graph. An articulation node is a separator of size one; i.e., it consists of a single vertex. In this process, the original constraint graph is assumed to be connected; if it is not, each piece can simply be considered separately. After finding all the articulation nodes and biconnected components, one can construct a decomposition tree whose nodes are the biconnected components (see Algorithm 3.1). Then we can generate a static variable ordering—an ordering for the backtracking algorithm—by traversing the decomposition tree in pre-order [40].

Freuder and Quinn [42] propose a static variable ordering heuristic based on a recursive decomposition of the constraint graph. The idea is that the separators (called cutsets in [42]) give groups of variables which, once instantiated, decompose the CSP and the disconnected components can then be solved independently. Freuder and Quinn also propose a special-purpose backtracking algorithm to correctly use the variable ordering to get additive behavior rather than multiplicative behavior when solving the independent problems.

Once all of the variables in a separator have been instantiated, the backtracking algorithm has a choice of which subproblem to work on next. Amir and McIlraith [3]

Algorithm 3.1: Construct a Tree With Biconnected Components

input : A constraint satisfaction problem (CSP)
output: A decomposition tree

```
choose one of the components as the root and put it in a queue Q;
while Q is not empty do
    remove a component c from Q;
    for each component d which shares an articulation point with c and is not
    already in the tree do
        make d a child of c in the tree;
        put d in Q;
```

present a heuristic that solves the most constrained subproblem first. However, no experimental evaluation is performed. Various methods have also been proposed for recursively decomposing the constraint graph. Aloul, Markov and Sakallah [2] present a static variable ordering based on a recursive min-cut bisection of the hypergraph for a SAT instance and show that this can improve the performance of SAT solvers. However, most modern SAT solvers use a dynamic variable ordering because they are overwhelmingly more effective.

Huang and Darwiche [64] present a recursive decomposition technique called dTree. Further, Huang and Darwiche [64] show that the special-purpose backtracking algorithm proposed by Freuder and Quinn [42] is not needed; a regular backtracking algorithm with a technique called backjumping incorporated is sufficient. This is an interesting result as backjumping is now standard in all SAT solvers. Because the separators in Freuder and Quinn's approach are found prior to search, the pre-established variable groupings never change during the execution of the backtracking search. However, Huang and Darwiche note that within these groupings the variable ordering can be dynamic and any one of the existing variable ordering heuristics can used. Huang and Darwiche [64] show that their divide-and-conquer approach can be effective on hard SAT problems. Independently Bjesse et al. [12] present a proposal similar in substance to Huang and Darwiche. However, no experimental evaluation is performed.

My work on exploiting structure in propositional reasoning takes as its starting point Freuder and Quinn's [42] idea of constructing a variable ordering heuristic by recursively decomposing the hypergraph of a SAT instance. However, all previous work [42, 3, 2, 64, 12] has only considered static decomposition methods; i.e., the decomposition tree is fully constructed prior to the backtracking search. In my work I consider dynamic decomposition where the decomposition can happen during the backtracking search. The consequences are twofold. First, with static decomposition methods, the pre-established variable groupings never change during the backtracking search. The result is that the variable ordering heuristic is limited to be quite static. My experimental work shows that the result is that these heuristics sometimes considerably reduce the amount of Boolean constraint propagation and in such cases do not improve the efficiency of solving real world instances. Using a dynamic de-

composition can lead to considerably more propagation earlier in the tree. Second, a static decomposition technique constructs the whole decomposition tree prior to backtracking search, and this preprocessing time can be considerable. Using a dynamic decomposition, the separator based decomposition is done on-the-fly and can stop at any time during the decomposition process, thus avoiding unnecessary work.

3.3 Dynamic Tree Decomposition for Variable Ordering Heuristics

In this section, I present techniques for taking advantage of structure for improving solvers for propositional satisfiability. In particular, I present several techniques for dynamically recognizing and incorporating structural information into the search algorithms and the variable orderings which guide the search algorithms.

I begin by providing the intuitions behind why my proposals based on dynamic structural information can have an advantage over previous proposals based on static decompositions.

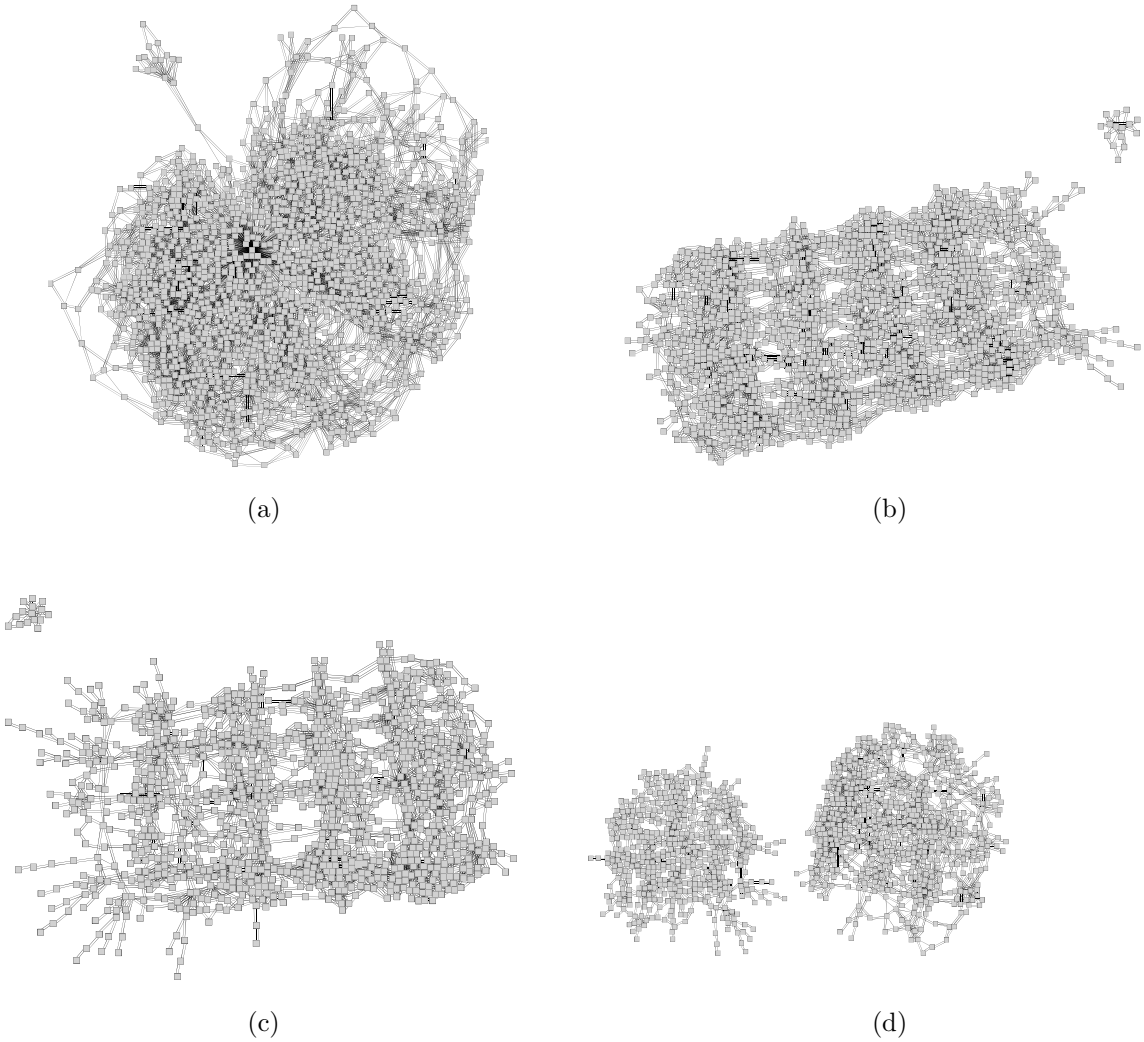
3.3.1 Static Decomposition versus Dynamic Decomposition

The structure of a SAT encoded real world problem changes dramatically during the running time of DPLL. A disadvantage of static decomposition is that one can always expect a better result if one analyzes the structure dynamically; i.e., during the backtracking search.

The visualization approach proposed in [95] provides an empirical tool to observe and analyze the structure of real-world SAT problems dynamically. It shows that long implication chains often exist in these instances. Unit propagation is a look-ahead strategy for all of the cutting-edge SAT solvers. Since most of the variables on the implication chains are instantiated after making a relatively small number of decisions, the internal structure of real-world instances often change dramatically in different parts of the search tree.

Example 3.3. *Figure 3.3(a) shows the primal graph of the bounded model checking instance dp05s05, which is from the dining philosophers problem. Figure 3.3(b) shows the result after setting proposition 1283 to false and performing unit propagation. In Figures 3.3(b)–3.3(d), independent subproblems are naturally occurring. Figures 3.6(a)–3.6(d) show the corresponding dual graph of dp05s05. The snapshots are from applying zChaff to the instance, where zChaff is using the Variable State Independent Decaying Sum branching heuristic. It can be seen that the problem can be decomposed into quite large independent subproblems after instantiating just 10 variables. Thus, small separators that would not be discovered statically before search can be discovered dynamically during the search.*

The second disadvantage of statically decomposing a SAT instance is that a static structure-guided heuristic can limit Boolean constraint propagation (BCP) and break



(a) (b)

(c) (d)

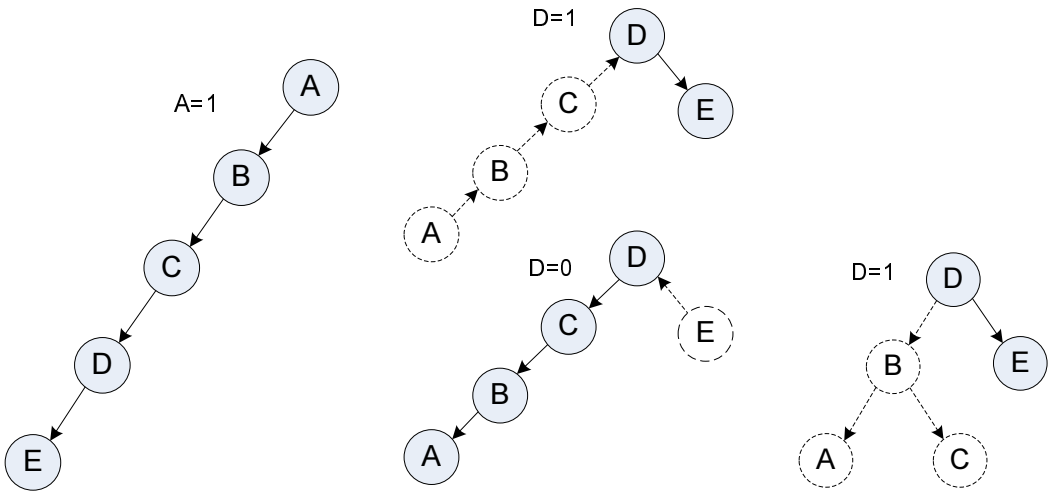
Figure 3.3: (a) Primal graph of original dp05s05 problem; (b) primal graph at decision level 1 after one variable has been instantiated; (c) primal graph at decision level 10; and (d) decomposed primal graph at decision level 10.

the implication chains. An implication chain is an assignment of a set of variables $v_0, v_1, \ldots, v_k$, where v_0 is a decision variable, v_i is given an implied value before v_j whenever $0 < i < j$, and for each $i < k$, v_i appears in the antecedent clause c_{i+1} of v_{i+1}. It is very common to observe noticeably many implication chains of considerable length in real-world instances. However, when an instance is decomposed into several subproblems during search, BCP cannot pass from one subproblem to another.

Example 3.4. *Consider the following CNF formula,*

$$(\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg C \vee D) \wedge (\neg D \vee E),$$

where A, B, C, D and E are in the implication chain with variable A being assigned the value 1 *(see Figure 3.4(a)). Compare 3.4(a), which is the implication chain of the*



(a) $\{A, B, C, D, E\}$ (b) $\{A, B, C\}$ $\{D, E\}$ (c) $\{A, B\}$ $\{C\}$ $\{D, E\}$

Figure 3.4: Comparison of the maximum length of the implication chains with different decomposition methods for the CNF formula, $(\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg C \vee D) \wedge (\neg D \vee E)$, in Example 3.4.

above CNF formula without decomposition, to Figure 3.4(b) and Figure 3.4(c). The latter two figures show that a variable ordering heuristic that focuses on separators and decomposing the instance can result in shorter implication chains and so cause more decisions to be made in the search for a solution.

Thus, there is a trade-off between decomposition and BCP. By continually applying unit propagation and updating the formula until no unit clauses remain, BCP can instantiate an ordered variable sequence or implication chain. Every implication chain has a start clause. When a problem is decomposed into several subproblems, it is hoped that the subproblem with which we begin contains all the start clauses and that the implication chains are not cut by any subproblem. Otherwise, BCP cannot "activate" all the unit clauses along the chain.

Example 3.5. *Consider once again the formula* $F = (x \vee y \vee z) \wedge (y \vee \neg z \vee w) \wedge (\neg w \vee \neg z \vee v) \wedge (\neg v \vee u)$*, of Example 3.1. Figure 3.1 shows the corresponding primal graph and primal hypergraph and Figure 3.5 shows one possible tree decomposition for this hypergraph. The decomposed problem can be solved using many possible subproblem orderings. The following table shows four of the possible orderings and the resulting maximum number of implications and minimum number of decisions needed to find a solution when using DPLL to solve each subproblem in the given order.*

subproblem ordering	*max #implications*	*min #decisions*
$A \rightarrow B \rightarrow C \rightarrow D$	*3*	*3*
$B \rightarrow A \rightarrow C \rightarrow D$	*5*	*1*
$C \rightarrow A \rightarrow B \rightarrow D$	*2*	*4*
$D \rightarrow C \rightarrow A \rightarrow B$	*1*	*5*

As the example demonstrates, the ordering in which the subproblems are solved can have a significant influence on the amount of work needed to find a solution.

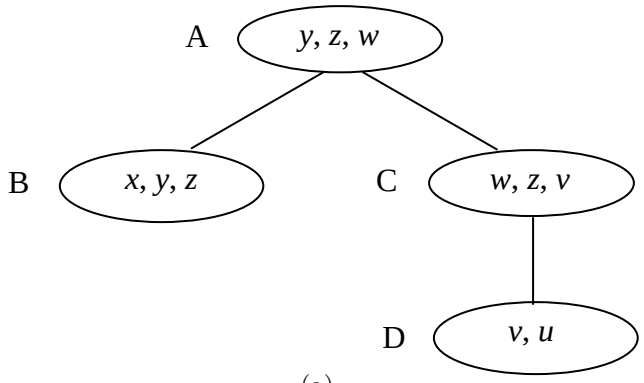


Figure 3.5: (a) A rooted tree decomposition of the hypergraph shown in Figure 3.1. (b) As discussed in Example 3.5, the subproblems can be solved in different possible orders.

Unfortunately, none of the existing static decomposition approaches can avoid shortening implication chains. Here, I propose several dynamic decomposition methods that can help keep implication chains safe.

3.3.2 A Dynamic Structure-guided Variable Ordering Heuristic for SAT

Since the structure of a SAT problem changes dramatically during the running time of DPLL, here I propose to use a dynamic decomposition method based on finding vertex separators of the residual primal graph. Finding vertex separators naturally leads to a divide-and-conquer strategy. The separator becomes the root of the corresponding tree structure, while the subtrees become the subproblems induced by the separator.

Definition 3.7 (Residual primal graph)**.** *The* residual primal graph $G(\phi) = (V(\phi), E(\phi))$ *is based on the residual formula $\phi = F|_s$, where F is the original formula and s is the current partial assignment. Each node in $G(\phi)$ represents a variable which has not been instantiated in $\phi = F|_s$. If $\phi_2 = F|_{s_2}$ can be* extended *from $\phi_1 = F|_{s_1}$, then $G(\phi_2)$ is a subgraph of $G(\phi_1)$, where $V(\phi_2) \subseteq V(\phi_1)$ and $E(\phi_2) \subseteq E(\phi_1)$.*

To find a vertex separator of the residual primal graph, I find a hyperedge separator of the dual hypergraph representation of the residual formula $\phi = F|_s$. A separator of a hypergraph G is a set of hyperedges whose removal chops G into disjoint subhypergraphs whose sizes stand in some sought relation. The formal basis for this is given by the following lemma.

Lemma 3.1. *A hyperedge separator of a dual hypergraph is also a vertex separator of the primal graph.*

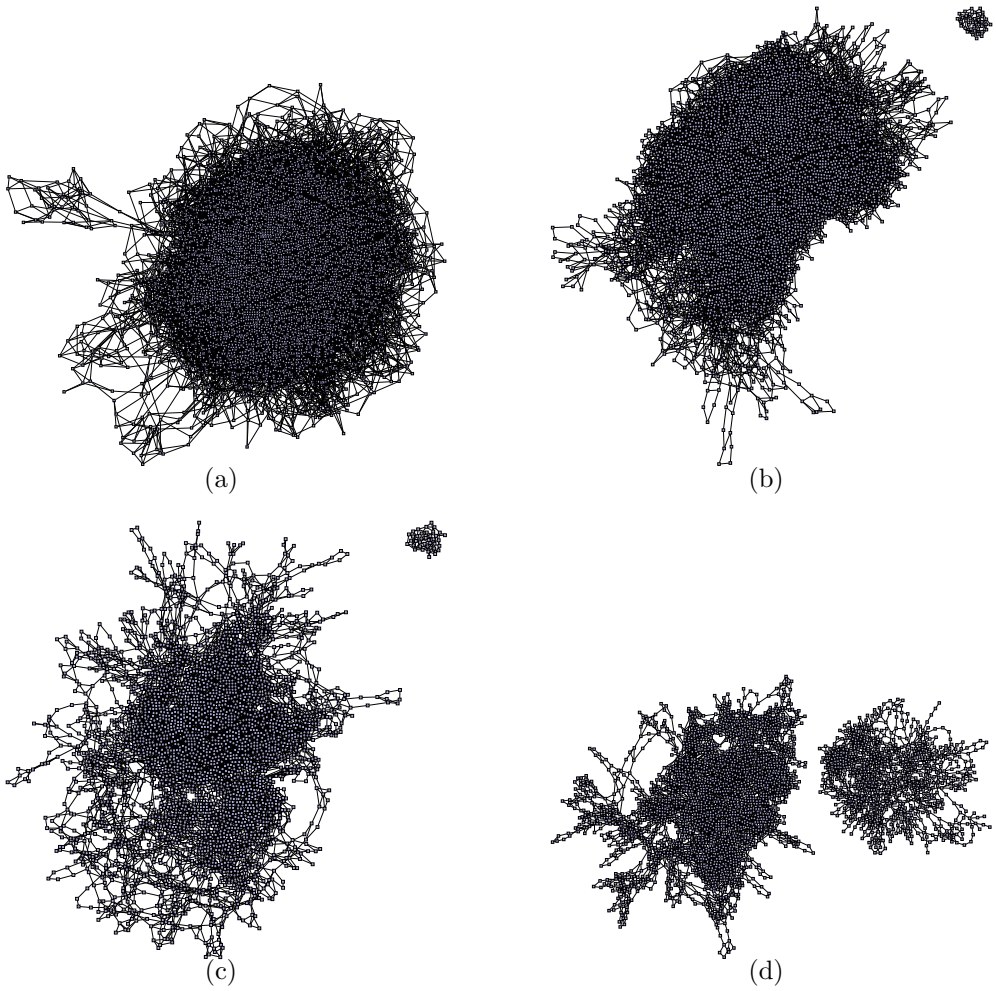


Figure 3.6: (a) Dual graph of original dp05s05 problem; (b) dual graph at decision level 1 after one variable has been instantiated; (c) dual graph at decision level 10; and (d) decomposed dual graph at decision level 10.

I next discuss the motivation for finding a vertex separator of the primal graph by finding a hyperedge separator of the dual hypergraph. The primary reason is that the dual hypergraph is a *compact* representation of a propositional formula. To explain, let n be the number of variables in the formula and m be the number of clauses. In most formulas, m is much larger than n. Table 3.1 shows example statistics for the benchmarks I used in my experiments. As a result, the dual hypergraph, which has a hyperedge for each variable, has a relatively small number of hyperedges compared to the number of dual variables. As well, the dual hypergraph works well with current hypergraph partitioning technology such as the multi-level refinement method (see, e.g., [69, 70]). In contrast, directly finding a vertex separator on the primal graph works less well for two reasons. First, the primal graph is missing some structural information because some of the clause information is lost when represented as a regular graph. Second, the primal graph is quite dense and has many more edges

Table 3.1: Mean of the clause/variable ratio for each of the selected benchmarks.

benchmark	ratio
bmc1	2.4
bmc2	3.2
bart	4.8
homer	5.1
lisa	5.5
cmpadd	2.8
matrix	3.3
fpga_routing	16.0
graph_coloring	7.7
onestep_rand_net	2.9
multistep_rand_net	2.9
ezfact	6.4
qg	29.0

than the dual hypergraph. Together, but perhaps primarily for the first reason, the multi-level refinement methods work much less well when applied to the primal graph for a CNF formula.

Most of the current hypergraph partition tools focus on optimizing the following quality measures [69, 70]:

- *Hyperedge Cut*: A hyperedge cut is the number of the hyperedges that span multiple partitions.
- *Sum of External Degrees*: The external degree of a partition is defined as the number of hyperedges, that are incident but not fully inside this partition.
- *Scaled Cost*: The scaled cost is defined as the sum of the vertex weights of a partition.

Definition 3.8 (Dynamic separator)**.** *Given a connected* residual primal graph $G(\phi)$*, where* $\phi = F|_{s_0}$*, a* dynamic separator *is a set* $S_1 \subset V(\phi)$ *such that the residual primal graph* $G(\phi')$ *induced by* $\phi' = F|_{s_0 \cup s_1}$ *is no longer a connected graph.*

To explore the problem structure dynamically, I propose a dynamic variable ordering heuristic based on dynamic separator decomposition. Once we have the hypergraph representation of a formula F, the entire formula can be decomposed into smaller subgraphs (subproblems) giving a divide-and-conquer strategy (see Figure 3.7).

My algorithm is presented as Algorithm 3.2. DSD_DPLL() takes three inputs: the propositional formula F, corresponding primal graph G, and S, the separator of G, whose initial value is empty. After unit propagation, we create and maintain a separator of G. The next variable is chosen from S until all the variables are

instantiated and F is decomposed into several subproblems. The separator created after unit propagations is based on the residual graph of the current partial solution. F is *true* when all its subproblems are *true*, otherwise it is *false*.

Algorithm 3.2: DSD_DPLL(F, S, G)

input : Propositional formula F, corresponding primal graph G, and the current separator S
output: Returns *true* if F is satisfiable; *false* otherwise

```
if F is an empty clause set then
    return true;
if F contains an inconsistent clause then
    return false;
if F contains a unit clause C then
    F = Unit_Propagation(F,C);
if S is empty then
    S = Separator(G);
if there is no uninstantiated variable in S then
    for each constraint graph partition G_i do
        if DSD_DPLL(F, φ, G_i) = false then
            return false;
    return true;
else
    choose an uninstantiated variable v from S;
    return DSD_DPLL(F|_{v=0}, S, G) or DSD_DPLL(F|_{v=1}, S, G);
```

3.3.3 Multiple Principle Guided Variable Ordering Heuristics

The dynamic way of decomposition is a form of reasoning by cases or assumptions. To solve a complicated SAT or MC problem, we try to simplify it by considering a number of residual formulas and the residual primal graphs, which correspond to a set of mutually exclusive and exhaustive assumptions. We then solve each of the cases under its corresponding assumption, and combine the results to obtain a solution to the original problem. The main problem of complete dynamic decomposition is that a large separator will lead to a blow up in the number of cases that has to be considered. In Figure 3.7, for example, if the top separator contains k variables, the decomposition would have to consider 2^k cases.

In the process of complete dynamic decomposition process, the complexity depends on the number of decisions. At the same decision level, the more decisions we make, the more times we need to update old separators. I use the decision distribution diagram to show that the separator size of real-world instances has an influence on the complexity of dynamic decomposition. The diagram is generated by recording

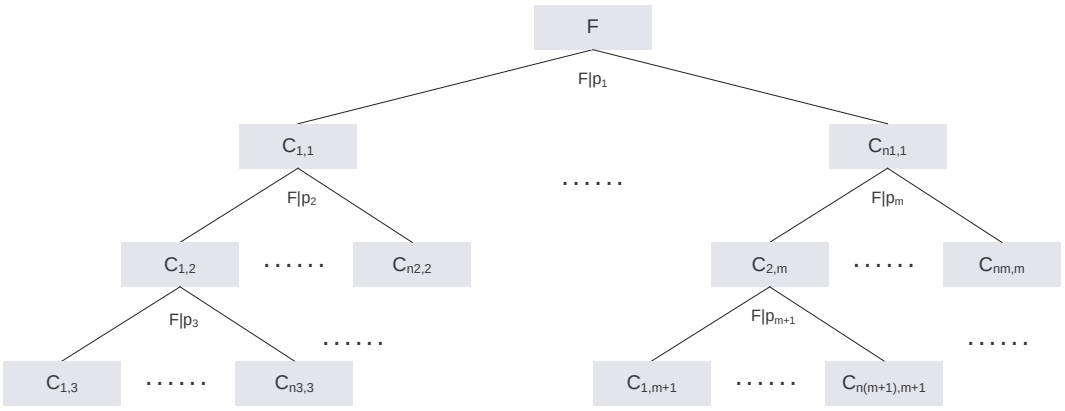


Figure 3.7: Components are created dynamically based on the residual formula, $\Phi = F|p$, where F is the original formula, p is the partial assignment at a node in the search tree, $p_1 \subset p_2 \subset \ldots \subset p_m \subset p_{m+1}$, and $p_1 \subset p_2$ represents that p_1 is a partial solution of p_2.

the number of decisions made at every decision level of the dynamic decomposition. Figure 3.8 is the decision distribution diagram of a random circuit checking instance. This diagram shows that the separator needs to be updated frequently at the root of the search tree. In contrast, Figure 3.9 shows that bart11, an instance of circuit model checking, has an easy decision making process at the beginning. Generally, the second case is more welcome since the easily solvable variables simplify the problem right before the dynamic decomposition.

A technique in SAT heuristic design is to combine multiple variable ordering heuristics that are based on different strategies. There are three basic approaches of combining different heuristics:

1. Weighted Sum Functions: One example of this approach is *Variable State Aware Decaying Sum* (VSADS), which is a dynamic heuristic designed for DPLL-based model counting engines [90]. It combines both Variable State Independent Decaying Sum Ordering (VSIDS) and the Dynamic Largest Combined Sum (DLCS). DLCS makes its branching decisions based only on the number of occurrences of a variable in the residual formula, while VSIDS is based on the most recently learned conflict clauses. The VSADS score of a variable is the combined weighted sum of its VSIDS score and its DLCS score,

$$\begin{aligned} score(VSADS) &= p \times score(VSIDS) + \\ & \quad q \times score(DLCS), \end{aligned}$$

   where p and q are some given constants. VSADS is the default dynamic variable ordering heuristic of Cachet.

2. Dominated Heuristic: The dTree group ordering heuristic [64] discussed above uses a binary decomposition tree to generate ordered variable groups before

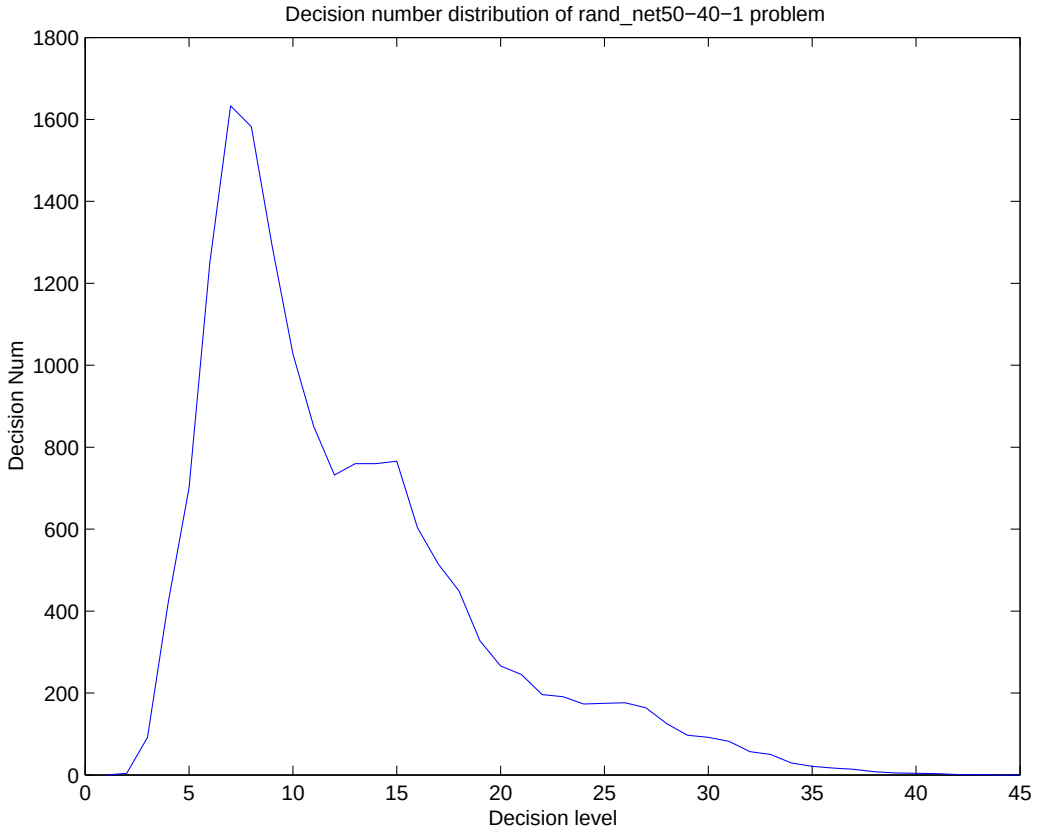


Figure 3.8: The decision distribution diagram of randnet50401 (a random circuit checking instance) showing the number of decisions made at each level in the DPLL search tree.

search. The order of variables within a group is then decided dynamically during search by another dynamic heuristic H. In this case, the structure-guided heuristic dominates H.

3. Dynamic Selection of Heuristics: Herbstritt [57] proposed the idea that not to apply one branching heuristic during the whole search process, but to give each heuristic the possibility to make a decision assignment from time to time. This method has not been implemented in modern SAT solvers.

It is widely accepted that the application of a "good" variable ordering heuristic is essential for solving a SAT problem. Here, I focus on the impact of two variable ordering heuristics based on different guiding principles: "conflict analysis based guiding principle" and "structure guiding principle". I propose several approaches which can, (i) apply several dynamic variable ordering heuristics, and (ii) use structural information to dynamically select a heuristic after each decision during the whole search process. I show that my approach can result in faster and more robust behavior for SAT algorithms.

I propose a structural variable ordering heuristic, *enhanced static decomposition (ESD)*, which is based on the static decomposition tree. My approach follows the

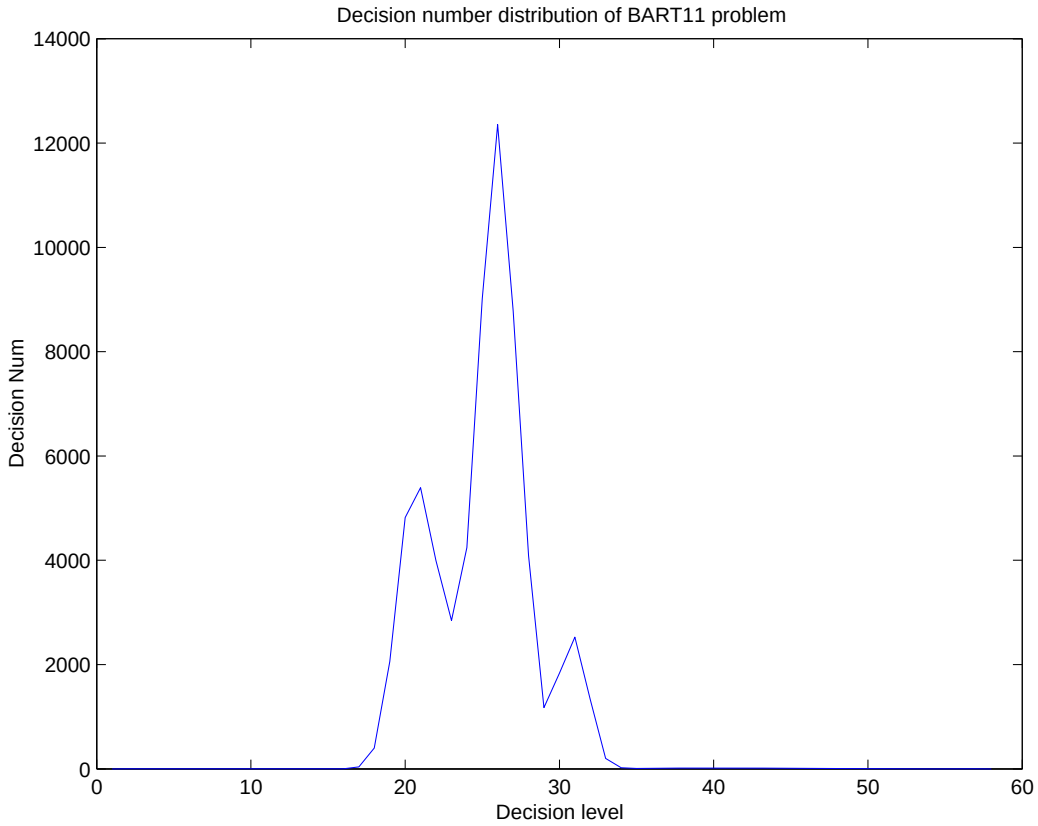


Figure 3.9: The decision distribution diagram of BART11 (an instance of circuit model checking) showing the number of decisions made at each level in the DPLL search tree.

ideas behind the weight sum method of combining various heuristics into one heuristic. ESD first decomposes the CNF formula before backtracking search using the same approach as in [64]: given the hypergraph separator decomposition, ESD generates a decomposition tree. Each node in the decomposition tree represents a variable group. During DPLL, ESD dynamically adds top variables selected by the dynamic variable ordering heuristics to the current variable group. Whenever the next decision variable is needed, an uninstantiated variable is chosen from the updated group. Compared with the completely recursive dynamic decomposition, ESD is more practical and easy to implement. After adding high ranking variables into the separator and instantiating them, the long implication chains are "started" at the beginning of the decomposition tree.

Huang and Darwiche's [64] heuristic decides the order of variables within a group dynamically, but the order of groups is determined statically. Cachet [88] uses one dynamic variable ordering heuristic, while disconnected components are detected passively. A structure based heuristic that recursively decomposes the SAT problem cannot avoid breaking the unit propagation chains.

Heuristics guided by conflict analysis have been successfully applied in major

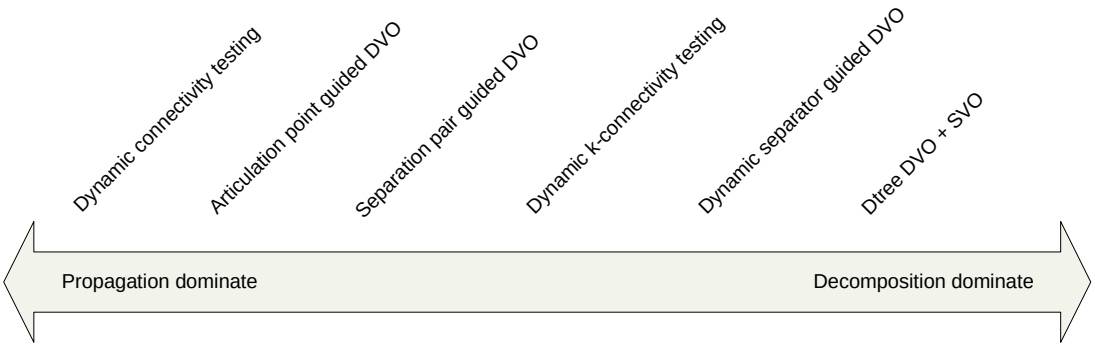


Figure 3.10: Approaches to integrating structure-guided variable ordering heuristics and propagation-first variable ordering heuristics.

SAT solvers. Here, my goal is to find a more practical and efficient way to integrate such heuristics into structure-guided heuristics. Figure 3.10 gives several possible approaches for combining structure-guided variable ordering heuristics and conflict analysis based variable ordering heuristics.

I can improve my method using the idea of dominated heuristics by limiting the search tree level to reduce its overhead. By this approach, we only decompose the residual primal graph when the depth in the search tree is less than some constant. In my experiments, I use a value of one or two as the cutoff.

3.4 Experimental Evaluation

In this section, I empirically evaluate the effectiveness of my proposals for structure-guided variable orderings.

In my implementation, a CNF is represented by a dual weighted hypergraph. To find separators, I use the hMETIS algorithm package [58, 69]. The problem of computing an optimal partition of a hypergraph is NP-complete and hence potentially time-consuming. The hMETIS package uses a heuristic method, multi-level hypergraph partition, to quickly find separators. The basic idea of multi-level algorithms is to construct a sequence of successively smaller hypergraphs by collapsing appropriate vertices, then find a partition of the small coarsened hypergraph, and finally obtain the approximated separator of the original hypergraph from the coarsest hypergraph step by step. An option to hMETIS is a parameter k, the number of partitions in which to decompose the hypergraph. In my experiments, I always used $k = 2$, referred to here as 2-way partitioning. As well, following Huang and Darwiche's [64] work on the dTree method, I chose the balance factor as 15:85, which means that the percentage of vertices in the two partitions should be in a ratio of 15 to 85 or as close to this ratio as possible.

Generally, there are two different ways to merge vertices together to form single vertices in the next level coarse hypergraph: edge coarsening and hyperedge coarsening. In hyperedge coarsening, vertices are grouped together that correspond to

Table 3.2: For the selected benchmarks, total CPU time (sec.) and number of improved instances (impr.) for zChaff compared to the proposed method (Separator) which uses 2-way ESD + MASF + dynamically adding variables.

benchmark	zChaff	Separator	#solved/#inst.	sat/unsat	impr.
bmc1	0	0	4/4	unsat	3
bmc2	770	**387**	5/6	unsat	4
bart	**35**	44	3/21	sat	1
homer	3164	**586**	9/15	unsat	9
lisa	1782	**996**	9/14	sat	6
cmpadd	**4**	5	8/8	unsat	5
matrix	31	**23**	2/5	unsat	2
fpga_routing	**27**	38	27/32	mixed	10
graph_coloring	**11517**	11690	150/300	mixed	107
onestep_rand_net	443	**125**	15/16	unsat	9
multistep_rand_net	**180**	506	2/16	unsat	0
ezfact	1367	**1270**	31/41	mixed	18
qg	191	**162**	10/19	mixed	7

hyperedges. In my experiments, I found that the coarsening scheme was an important factor for producing high quality hypergraph separator decomposition and preference was given to hyperedge coarsening.

I enhanced the separator decomposition with dynamic variable adding. Before the variables in the separator are instantiated (zChaff uses the VSIDS heuristic), a group of variables with highest scores are added to the current separator. After all the variables in the separator of a hypergraph have been instantiated, the sub-hypergraphs of the current separator are updated to eliminate variables implied by the instantiations of the variables in the separator.

I also consider dynamic subproblem ordering heuristics. When there are several subproblems, the algorithm must decide which subproblem to solve first. Most modern SAT solvers have a dynamic variable ordering heuristic. For example, zChaff uses Variable State Independent Decaying Sum (VSIDS) heuristic. To combine VSIDS with subproblem ordering, each hyperedge is given a weight, which dynamically changes with VSIDS. Four dynamic subproblem ordering heuristics were tested in preliminary experiments,

MVSF — subproblem with maximum VSIDS value is solved first,
MASF — subproblem with maximum average VSIDS value is solved first,
SSF — subproblem with the shortest clause is solved first, and
MCSF — subproblem that is most constrained is solved first [3].

The MASF subproblem ordering heuristic was found to work the best in preliminary experiments and it was used in all of the experiments that I report here.

Table 3.3: For the selected benchmarks, total CPU time (sec.) and number of improved instances (impr.) for zChaff+dTree compared to the proposed method (Separator) which uses 2-way ESD + MASF + dynamically adding variables.

benchmark	zChaff+ dTree	dTree Time	Separator	#solved/ #inst.	sat/ unsat	impr.
bmc1	0.01	3	0.01	4/4	unsat	4
bmc2	0.07	2.5	**0.04**	1/6	unsat	1
bart	150	2	**44**	3/21	sat	3
homer	**216**	13	586	9/15	unsat	0
lisa	**700**	87	1451	11/14	sat	6
cmpadd	**0.76**	31	4.57	8/8	unsat	8
matrix	90	6	**23**	2/5	unsat	2
fpga_routing	11	2076	**8**	17/32	mixed	17
graph_coloring	40767	230	**19761**	160/300	mixed	129
one_step_randnet	240	380	**125**	15/16	unsat	11
ezfact	759	179	**543**	31/41	mixed	22
qg	182	503	**162**	10/19	mixed	7

multistep_rand_net was omitted as dTree had a stack overflow error on these instances.

The benchmark instances used in my experiments are selected from the industrial and handmade categories of the SAT Competition 2002 [94, 62]. Experiments were performed on all of the industrial instances from the competition but I omit from my experimental results all instances that both dTree and my method could not solve. The experiments were performed on a PC with a 2.67GHz Pentium 4 processor and 1Gb of RAM. Each runtime is the average of 10 runs with a 15 minute CPU cut-off limit. All runs that did not complete in the time limit did not contribute to the average. The time limit is longer than the SAT 2002 competition (see [94]).

In Table 3.2, we compare the runtime of zChaff and enhanced static separator decomposition (ESD). The times shown represent the total time for the instances which were solved within the time limit. If a method did not solve an instance, I charged the maximum time limit. The column which lists the number of solved instances shows the total number of instances which both methods could solve. For example, for the benchmark "bmc2", the benchmark contains six instances, five of which could be solved by both methods (the column with heading #solved/#inst) and for four of the instances, our method (Separator) improved over zChaff (the column with heading impr.) and, conversely, for two of the instances, zChaff performed either as well or better.

Table 3.3 reports the comparative performance of zChaff+dTree and zChaff+ESD. The dTree Time reports the time of constructing a dTree and zChaff+dTree reports the runtime of zChaff with the variable group ordering from the dTree. In contrast, the runtime of finding the graph separator decomposition is included in the runtime of solving the instances. My experimental results also show that the separator de-

Table 3.4: Max decision level (max), number of decisions (dec.), and number of implications (impl.) for zChaff compared to the proposed method (Separator) which uses 2-way DSD_DPLL + MASF.

	zChaff			Dynamic Separator		
benchmark	max	dec.	impl.	max	dec.	impl.
lisa19_3_a	65	181,568	36,789,189	**43**	**61,296**	**9,359,648**
lisa19_99_a	75	262,798	55,335,471	**39**	**31,149**	**4,925,937**
lisa20_99_a	62	99,361	19,340,591	**43**	**85,672**	**14,808,291**
homer06	124	56,754	1,071,432	**103**	**17,713**	**212,052**
homer07	124	110,607	2,126,296	**107**	**29,398**	**420,020**
homer08	141	134,769,269	**8,053**	**123**	**73,096**	642,087
homer09	178	237,750	5,176,136	**150**	**139,264**	**1,666,138**
homer10	**197**	**283,545**	7,590,761	203	387,330	**7,285,721**
homer11	156	121,239	2,412,764	**133**	**46,633**	**672,700**
homer12	162	242,994	4,871,283	**140**	**123,244**	**1,871,968**
homer13	173	300,403	6,093,351	**154**	**137,859**	**246,163**
homer14	194	588,502	13,377,128	**170**	**315,076**	**5,790,068**
homer15	258	1,127,302	31,367,313	**239**	**629,895**	**16,626,020**
homer16	198	369,264	7,156,381	**156**	**193,118**	**2,441,261**
homer17	203	394,490	7,980,266	**170**	**291,448**	**5,691,210**
Hanoi4	39	4,696	309,408	**30**	**1,508**	**153,196**

composition can solve much harder instances than dTree decomposition. Among the 11 industrial problems, there is only one case—the multi-step Rand-net problem—in which zChaff is much faster. However, most instances of this problem cannot be solved by any solver I tested.

In contrast to ESD, a completely dynamic separator decomposition method constructs a new separator each time a new decision is made. Because of the overhead of propagation synchronization, the runtime of the dynamic separator decomposition is very slow. 70% of instances cannot be solved in 15 minutes. However, the solver using dynamic separator decomposition often makes many fewer decisions and implications than zChaff and the static separator decomposition (see Table 3.4). Thus the completely dynamic method shows some promise, although the implementation still needs much work. Currently, I generate the dual hypergraph from scratch each time. A realistic implementation would incrementally maintain the dual hypergraph during the backtracking search.

3.5 Summary

I presented dynamic decomposition methods based on hypergraph separators. Integrating the hypergraph separator based decomposition into the variable ordering

of a modern SAT solver led to speedups on large real-world satisfiability problems. Compared with dTree, my approach does not need time to construct the full tree decomposition, which sometimes needs more time than the solving process. The dynamic separator decomposition shows promise in that it significantly decreases the number of decisions for some real-world problems.

Chapter 4

Exploiting Structure in Probabilistic Reasoning: Efficient Encodings

Previous studies have demonstrated that encoding a Bayesian network into a CNF formula and then performing weighted model counting using a backtracking search algorithm can be an effective method for exact inference in Bayesian networks. In this chapter, I present techniques for improving this approach for Bayesian networks with noisy-OR and noisy-MAX relations—two relations which are widely used in practice as they can dramatically reduce the number of probabilities one needs to specify. My techniques extend the weighted model counting approach for exact inference in Bayesian networks to networks that were previously intractable for the approach. I begin by reviewing the noisy-OR and noisy-MAX relations and previous approaches for exact inference in Bayesian networks with noisy-OR/MAX relations. I then present my space efficient CNF encodings for noisy-OR/MAX, which exploit the structure or semantics of the relations, and prove the correctness of the encodings. I also explore alternative search ordering heuristics for the DPLL-based backtracking algorithm. Finally, I present the results of an extensive empirical evaluation of my proposed encodings on large-scale real and randomly generated Bayesian networks. In the experiments, my techniques gave speedups of up to two orders of magnitude over the best previous approaches for Bayesian networks with noisy-OR/MAX relations and scaled up to networks with larger numbers of random variables.

4.1 Patterns for CPTs: Noisy-OR and Noisy-MAX

In this section, I review noisy-OR and noisy-MAX relations, two common patterns for conditional probability tables (CPTs).

A Bayesian network consists of a directed acyclic graph where the nodes represent the random variables and each node is labeled with a conditional probability table (CPT) which represents the strengths of the influences of the parent nodes on the child node (see Chapter 2). Assuming Boolean random variables, the CPT of a child

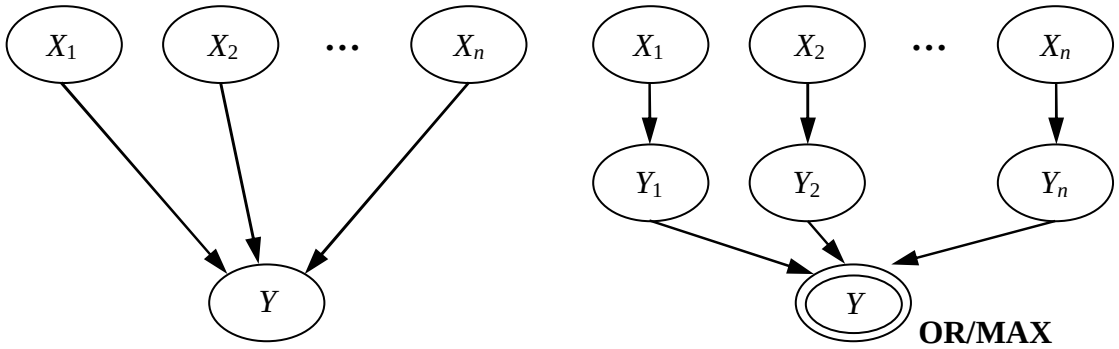


Figure 4.1: Bayesian networks with noisy-OR/MAX relations. (a) General causal structure, where causes $X_1, \ldots, X_n$ lead to effect Y; and (b) decomposed form for the noisy-OR and noisy-MAX relations. The node with a double border is a deterministic node with the designated logical relationship (OR) or arithmetic relationship (MAX).

node with n parents requires one to specify 2^{n+1} probabilities. More generally, if all of the random variables have domain size d, one must specify d^{n+1} probabilities. This presents a practical difficulty and has led to the introduction of patterns for CPTs which require one to specify many fewer parameters (e.g., [50, 82, 34]).

The two most widely used patterns in practice are the noisy-OR relation and its generalization, the noisy-MAX relation [50, 82]. These relations assume a form of causal independence and allow one to specify a CPT with just n parameters in the case of the noisy-OR and $(d-1)^2 n$ parameters in the case of the noisy-MAX, where n is the number of parents of the node and d is the size of the domains of the random variables. The noisy-OR/MAX relations have been successfully applied in the knowledge engineering of large real-world Bayesian networks, such as the Quick Medical Reference-Decision Theoretic (QMR-DT) project [78] and the Computer-based Patient Case Simulation system [81]. As well, Zagorecki and Druzdzel [103] show that in three real-world Bayesian networks, noisy-OR/MAX relations were a good fit for up to 50% of the CPTs in these networks and that converting some CPTs to noisy-OR/MAX relations gave good approximations when answering probabilistic queries. This is surprising, as the CPTs in these networks were not specified using the noisy-OR/MAX assumptions and were specified as full CPTs. Their results provide additional evidence for the usefulness of noisy-OR/MAX relations.

With the noisy-OR relation one assumes that there are different causes $X_1, \ldots, X_n$ leading to an effect Y (see Figure 4.1(a)), where all random variables are assumed to have Boolean-valued domains. Each cause X_i is either present or absent, and each X_i in isolation is likely to cause Y and the likelihood is not diminished if more than one cause is present. Further, one assumes that all possible causes are given and when all causes are absent, the effect is absent. Finally, one assumes that the mechanism or reason that inhibits a X_i from causing Y is independent of the mechanism or reason that inhibits a X_j, $j \neq i$, from causing Y.

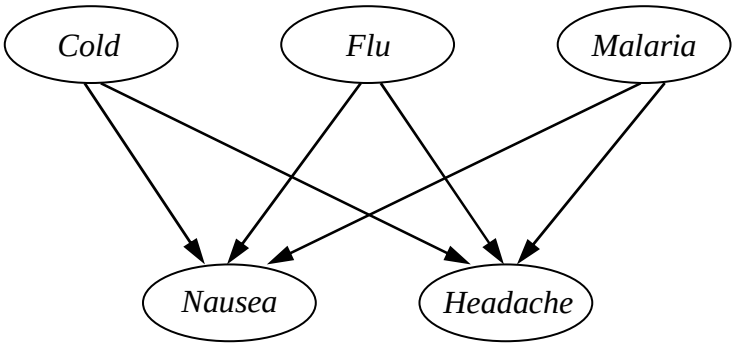


Figure 4.2: Example of a causal Bayesian network with causes (diseases) *Cold*, *Flu*, and *Malaria* and effects (symptoms) *Nausea* and *Headache*.

Example 4.1. *Consider the Bayesian network shown in Figure 4.2. The diseases cold, flu, and malaria are all likely to cause the symptom of nausea and the likelihood of having nausea does not diminish if more than one of these diseases is present. As well, one does not always feel nauseous when one has a cold or the flu and (it may be assumed that) the physiological mechanism that inhibits a cold from causing nausea is independent of the physiological mechanism that inhibits the flu from causing nausea.*

A noisy-OR relation specifies a CPT using n parameters, $q_1, \ldots, q_n$, one for each parent, where q_i is the probability that Y is false given that X_i is true and all of the other parents are false,

$$P(Y = 0 \mid X_i = 1, X_j = 0_{[\forall j, j \neq i]}) = q_i. \tag{4.1}$$

From these parameters, the full CPT representation of size 2^{n+1} can be generated using,

$$P(Y = 0 \mid X_1, \ldots, X_n) = \prod_{i \in T_x} q_i \tag{4.2}$$

and

$$P(Y = 1 \mid X_1, \ldots, X_n) = 1 - \prod_{i \in T_x} q_i \tag{4.3}$$

where $T_x = \{i \mid X_i = 1\}$ and $P(Y = 0 \mid X_1, \ldots, X_n) = 1$ if T_x is empty. The last condition (when T_x is empty) corresponds to the assumptions that all possible causes are given and that when all causes are absent, the effect is absent; i.e.,

$$P(Y = 0 \mid X_1 = 0, \ldots, X_n = 0) = 1.$$

These assumptions are not as restrictive as may first appear. One can always introduce an additional random variable X_0 that is a parent of Y but itself has no parents. The variable X_0 represents all of the other reasons that could cause Y to occur. The node X_0 and the prior probability $P(X_0)$ are referred to as a *leak node* and the *leak probability*, respectively. In what follows, I continue to refer to all possible causes as $X_1, \ldots, X_n$ where it is understood that one of these causes could be a leak node.

Table 4.1: Parameters for the noisy-ORs at node *Nausea* and at node *Headache* for the Bayesian network shown in Figure 4.2, assuming all of the random variables are Boolean.

$$
\begin{aligned}
P(Nausea = 0 \mid Cold = 1, Flu = 0, Malaria = 0) &= 0.6 \\
P(Nausea = 0 \mid Cold = 0, Flu = 1, Malaria = 0) &= 0.5 \\
P(Nausea = 0 \mid Cold = 0, Flu = 0, Malaria = 1) &= 0.4 \\
P(Headache = 0 \mid Cold = 1, Flu = 0, Malaria = 0) &= 0.3 \\
P(Headache = 0 \mid Cold = 0, Flu = 1, Malaria = 0) &= 0.2 \\
P(Headache = 0 \mid Cold = 0, Flu = 0, Malaria = 1) &= 0.1
\end{aligned}
$$

Example 4.2. *Consider once again the Bayesian network shown in Figure 4.2. Suppose that the random variables are Boolean representing the presence or the absence of the disease or the symptom, that there is a noisy-OR at node Nausea and at node Headache, and that the parameters for the noisy-ORs are as given in Table 4.1. The full CPT for the node Nausea is given by,*

C	F	M	$P(Nausea = 0 \mid C, F, M)$	$P(Nausea = 1 \mid C, F, M)$
0	0	0	1.00	0.00
0	0	1	0.40	0.60
0	1	0	0.50	0.50
0	1	1	$0.20 = 0.5 \times 0.4$	0.80
1	0	0	0.60	0.40
1	0	1	$0.24 = 0.6 \times 0.4$	0.76
1	1	0	$0.30 = 0.6 \times 0.5$	0.70
1	1	1	$0.12 = 0.6 \times 0.5 \times 0.4$	0.88

where C, F, and M are short for Cold, Flu, and Malaria, respectively.

An alternative way to view a noisy-OR relation is as a decomposed probabilistic model. In the decomposed model shown in Figure 4.1(b), one only has to specify a small conditional probability table at each node Y_i given by $P(Y_i \mid X_i)$, instead of an exponentially large CPT given by $P(Y \mid X_1, \ldots, X_n)$. In the decomposed model, $P(Y_i = 0 \mid X_i = 0) = 1$, $P(Y_i = 0 \mid X_i = 1) = q_i$, and the CPT at node Y is now deterministic and is given by the OR logical relation. The OR operator can be converted into a full CPT as follows,

$$
P(Y \mid Y_1, \ldots, Y_n) = \begin{cases} 1, & \text{if } Y = Y_1 \vee \cdots \vee Y_n, \\ 0, & \text{otherwise.} \end{cases}
$$

The probability distribution of an effect variable Y is given by,

$$P(Y \mid X_1, \ldots, X_n) = \sum_{Y = Y_1 \vee \cdots \vee Y_n} \left(\prod_{i=1}^{n} P(Y_i \mid X_i) \right),$$

where the sum is over all configurations or possible values for $Y_1, \ldots, Y_n$ such that the OR of these Boolean values is equal to the value for Y.

Similarly, in Pearl's [82] decomposed model, one only has to specify n probabilities to fully specify the model (see Figure 4.3); i.e., one specifies the prior probabilities $P(I_i)$, $1 \leq i \leq n$. In this model, causes always lead to effects unless they are prevented or inhibited from doing so. The random variables I_i model this prevention or inhibition.

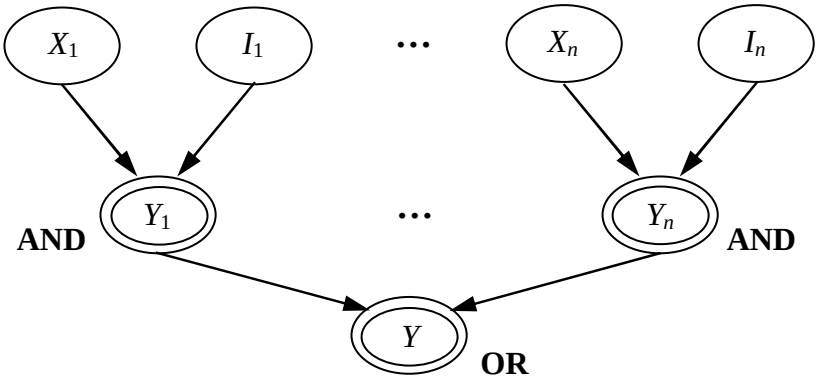


Figure 4.3: Pearl's [82] decomposed form of the noisy-OR relation. Nodes with double borders are deterministic nodes with the designated logical relationship.

These two decomposed probabilistic models (Figure 4.1(b), and Figure 4.3) can be shown to be equivalent in the sense that the conditional probability distribution $P(Y \mid X_1, \ldots, X_n)$ induced by both of these networks is just the original distribution for the network shown in Figure 4.1(a). It is important to note that both of these models would still have an exponentially large CPT associated with the effect node Y if the deterministic OR node were to be replaced by its full CPT representation. In other words, these decomposed models address ease of modeling or representation issues, but they do not address efficiency of reasoning issues.

The noisy-MAX relation (see [82, 50, 56, 33]) is a generalization of the noisy-OR to non-Boolean domains. With the noisy-MAX relation, one again assumes that there are different causes $X_1, \ldots, X_n$ leading to an effect Y (see Figure 4.1(a)), but now the random variables may have multi-valued (non-Boolean) domains. The domains of the variables are assumed to be ordered and the values are referred to as the degree or the severity of the variable. Each domain has a distinguished lowest degree 0 representing the fact that a cause or effect is absent. As with the noisy-OR relation, one assumes that all possible causes are given and when all causes are absent, the effect is absent. Again, these assumptions are not as restrictive as first appears, as one can incorporate a leak node. As well, one assumes that the mechanism or reason

that inhibits a X_i from causing Y is independent of the mechanism or reason that inhibits a X_j, $j \neq i$, from causing Y.

Example 4.3. *Consider once again the Bayesian network shown in Figure 4.2. Multi-valued domains allow one to model the severity of a symptom. Here, the nausea that sometimes accompanies a cold, the flu, or malaria is likely to be much more severe in the presence of the flu or malaria and perhaps more mild in the presence of a cold.*

Let d_X be the number of values in the domain of some random variable X. For simplicity of notation and without loss of generality, I assume that the domain of a variable X is given by the set of integers $\{0, 1, \ldots, d_X - 1\}$. A noisy-MAX relation with causes $X_1, \ldots, X_n$ and effect Y specifies a CPT using the parameters,

$$P(Y = y \mid X_i = x_i, X_j = 0_{[\forall j, j \neq i]}) = q_{i,y}^{x_i} \qquad \begin{array}{l} i = 1, \ldots, n, \\ y = 0, \ldots, d_Y - 1, \\ x_i = 1, \ldots, d_{X_i} - 1. \end{array} \tag{4.4}$$

If all of the domain sizes are equal to d, a total of $(d-1)^2 n$ non-redundant probabilities must be specified. From these parameters, the full CPT representation of size d^{n+1} can be generated using,

$$P(Y \leq y \mid \boldsymbol{X}) = \prod_{\substack{i=1 \\ x_i \neq 0}}^{n} \sum_{y'=0}^{y} q_{i,y'}^{x_i} \tag{4.5}$$

and

$$P(Y = y \mid \boldsymbol{X}) = \begin{cases} P(Y \leq 0 \mid \boldsymbol{X}) & \text{if } y = 0, \\ P(Y \leq y \mid \boldsymbol{X}) - P(Y \leq y - 1 \mid \boldsymbol{X}) & \text{if } y > 0. \end{cases} \tag{4.6}$$

where $\boldsymbol{X}$ represents a certain configuration of the parents of Y, $\boldsymbol{X} = x_1, \ldots, x_n$, and $P(Y = 0 \mid X_1 = 0, \ldots, X_n = 0) = 1$; i.e., if all causes are absent, the effect is absent.

Example 4.4. *Consider once again the Bayesian network shown in Figure 4.2. Suppose that the diseases are Boolean random variables and the symptoms Nausea and Headache have domains {absent = 0, mild = 1, severe = 2}, there is a noisy-MAX at node Nausea and at node Headache, and the parameters for the noisy-MAX at node Nausea are as given in Table 4.2. The full CPT for the node Nausea is given by,*

C	F	M	$P(N = a \mid C, F, M)$	$P(N = m \mid C, F, M)$	$P(N = s \mid C, F, M)$
0	0	0	1.000	0.000	0.000
0	0	1	0.100	0.400	0.500
0	1	0	0.500	0.200	0.300
0	1	1	$0.050 = 0.5 \times 0.1$	0.300	0.650
1	0	0	0.700	0.200	0.100
1	0	1	$0.070 = 0.7 \times 0.1$	0.380	0.550
1	1	0	$0.350 = 0.7 \times 0.5$	0.280	0.370
1	1	1	$0.035 = 0.7 \times 0.5 \times 0.1$	0.280	0.685

Table 4.2: Parameters for the noisy-MAX at node *Nausea* for the Bayesian network shown in Figure 4.2, assuming the diseases are Boolean random variables and the symptom *Nausea* has domain {absent = 0, mild = 1, severe = 2}.

$$
\begin{aligned}
P(\mathit{Nausea} = \text{absent} \mid \mathit{Cold} = 1, \mathit{Flu} = 0, \mathit{Malaria} = 0) &= 0.7 \\
P(\mathit{Nausea} = \text{mild} \mid \mathit{Cold} = 1, \mathit{Flu} = 0, \mathit{Malaria} = 0) &= 0.2 \\
P(\mathit{Nausea} = \text{severe} \mid \mathit{Cold} = 1, \mathit{Flu} = 0, \mathit{Malaria} = 0) &= 0.1 \\
P(\mathit{Nausea} = \text{absent} \mid \mathit{Cold} = 0, \mathit{Flu} = 1, \mathit{Malaria} = 0) &= 0.5 \\
P(\mathit{Nausea} = \text{mild} \mid \mathit{Cold} = 0, \mathit{Flu} = 1, \mathit{Malaria} = 0) &= 0.2 \\
P(\mathit{Nausea} = \text{severe} \mid \mathit{Cold} = 0, \mathit{Flu} = 1, \mathit{Malaria} = 0) &= 0.3 \\
P(\mathit{Nausea} = \text{absent} \mid \mathit{Cold} = 0, \mathit{Flu} = 0, \mathit{Malaria} = 1) &= 0.1 \\
P(\mathit{Nausea} = \text{mild} \mid \mathit{Cold} = 0, \mathit{Flu} = 0, \mathit{Malaria} = 1) &= 0.4 \\
P(\mathit{Nausea} = \text{severe} \mid \mathit{Cold} = 0, \mathit{Flu} = 0, \mathit{Malaria} = 1) &= 0.5
\end{aligned}
$$

where C, F, M, and N are short for the variables Cold, Flu, Malaria, and Nausea, respectively, and a, m, and s are short for the values absent, mild, and severe, respectively. As an example calculation, $P(\mathit{Nausea} = \mathit{mild} \mid \mathit{Cold} = 0, \mathit{Flu} = 1, \mathit{Malaria} = 1) = ((0.5+0.2) \times (0.1+0.4)) - (0.05) = 0.3$ *As a second example,* $P(\mathit{Nausea} = \mathit{mild} \mid \mathit{Cold} = 1, \mathit{Flu} = 1, \mathit{Malaria} = 1) = ((0.7+0.2) \times (0.5+0.2) \times (0.1+0.4)) - (0.035) = 0.28$

As with the noisy-OR relation, an alternative view of a noisy-MAX relation is as a decomposed probabilistic model (see Figure 4.1(b)). In the decomposed model, one only has to specify a small conditional probability table at each node Y_i given by $P(Y_i \mid X_i)$, where $P(Y_i = 0 \mid X_i = 0) = 1$ and $P(Y_i = y \mid X_i = x) = q^x_{i,y}$. Each Y_i models the effect of the cause X_i on the effect Y in isolation; i.e., the degree or the severity of the effect in the case where only the cause X_i is not absent and all other causes are absent. The CPT at node Y is now deterministic and is given by the MAX arithmetic relation. This corresponds to the assumption that the severity or the degree reached by the effect Y is the maximum of the degrees produced by each cause if they were acting independently; i.e., the maximum of the Y_i's. This assumption is only valid if the effects do not accumulate. The MAX operator can be converted into a full CPT as follows,

$$
P(Y \mid Y_1, \ldots, Y_n) = \begin{cases} 1, & \text{if } Y = \max\{Y_1, \ldots, Y_n\}, \\ 0, & \text{otherwise.} \end{cases}
$$

The probability distribution of an effect variable Y is given by,

$$P(Y \mid X_1, \ldots, X_n) = \sum_{Y=\max\{Y_1,\ldots,Y_n\}} \left(\prod_{i=1}^{n} P(Y_i \mid X_i) \right),$$

where the sum is over all configurations or possible values for $Y_1, \ldots, Y_n$ such that the maximum of these values is equal to the value for Y. In both cases, however, making the CPTs explicit is often not possible in practice, as their size is exponential in the number of causes and the number of values in the domains of the random variables.

4.2 Related Work

In this section, I relate my work to previously proposed methods for exact inference in Bayesian networks with noisy-OR/MAX relations.

I am considering here the problem of exact inference in Bayesian networks which contain noisy-OR/MAX relations. One method for solving such networks is to replace each noisy-OR/MAX by its full CPT representation and then use any of the well-known algorithms for answering probabilistic queries such as variable elimination or tree clustering/jointree. However, in general—and in particular, for the networks that I use in my experimental evaluation—this method is impractical. A more fruitful approach for solving such networks is to take advantage of the structure or the semantics of the noisy-OR/MAX relations to improve both time and space efficiency (e.g., [52, 80, 20, 55, 105, 98, 35, 17]).

Quickscore [52] was the first efficient exact inference algorithm for Boolean-valued *two-layer* noisy-OR networks. Chavira, Allen and Darwiche [17] present a method for *multi-layer* noisy-OR networks and show that their approach is significantly faster than Quickscore on randomly generated two-layer networks. Their approach proceeds as follows: (i) transform the noisy-OR network into a Bayesian network with full CPTs using Pearl's decomposition (see Figure 4.3), (ii) translate the network with full CPTs into CNF using a *general* encoding (see Chapter 2), and (iii) compile the CNF into an arithmetic circuit. In my experiments, I show that my *special-purpose* encodings of noisy-OR can be more space and time efficient and scale to much harder problems.

Many alternative methods have been proposed to decompose a noisy-OR/MAX by adding hidden or auxiliary nodes and then solving using adaptations of variable elimination or tree clustering (e.g., [80, 20, 55, 98, 35]).

Olesen et al. [80] proposed to reduce the size of the distribution for the OR/MAX operator by decomposing a deterministic OR/MAX node with n parents into a set of binary OR/MAX operators. The method, called parent divorcing, constructs a binary tree by adding auxiliary nodes Z_i such that Y and each of the auxiliary nodes has exactly two parents. Figure 4.4 shows the decomposition tree constructed by parent divorcing for four causes $X_1, \ldots, X_4$ and effect Y. Heckerman [53] presented a sequential decomposition method again based on adding auxiliary nodes Z_i and decomposing into binary MAX operators. Here one constructs a linear decomposition tree (see Figure 4.5). Both methods require similar numbers of auxiliary nodes

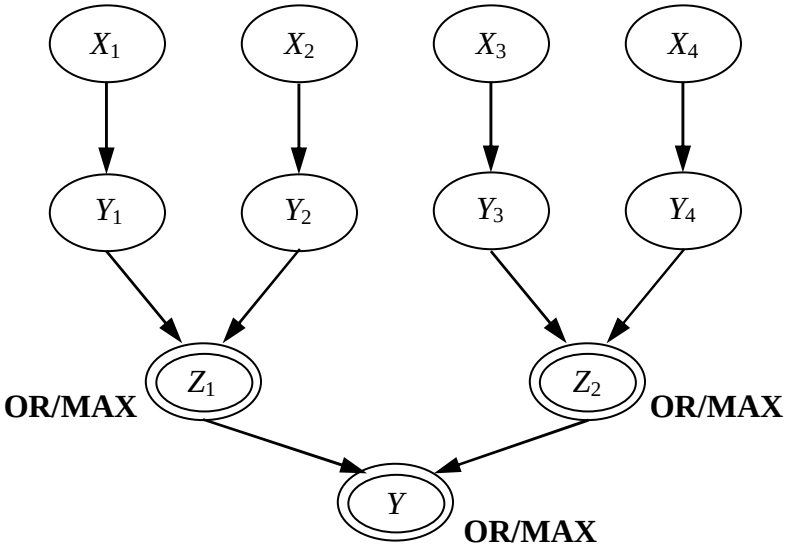


Figure 4.4: Example of parent divorcing for a noisy-OR/MAX with four causes $X_1, \ldots, X_4$ and effect Y (Olesen et al. [80]).

and similarly sized CPTs. However, as Takikawa and D'Ambrosio [98] note, using either parent divorcing or sequential decomposition, many decomposition trees can be constructed from the same original network—depending on how the causes are ordered—and the efficiency of query answering can vary exponentially when using variable elimination or tree clustering, depending on the particular query and the choice of ordering.

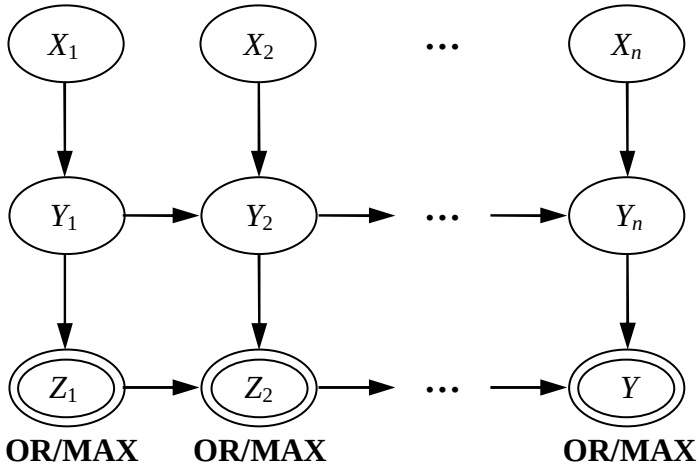


Figure 4.5: Example of the sequential decomposition for a Bayesian network with a noisy-OR/MAX with causes $X_1, \ldots, X_n$ and effect Y (Heckerman [53]).

To take advantage of causal independence models, Díez [33] proposed an algorithm for the noisy-MAX/OR. By introducing one auxiliary variable Y', Díez's method leads to a complexity of $O(nd^2)$ for singly connected networks, where n is the number of causes and d is the size of the domains of the random variables. However, for net-

works with loops it needs to be integrated with local conditioning. Takikawa and D'Ambrosio [98] proposed a similar multiplicative factorization approach. The complexity of their approach is $O(\max(2^d, nd^2))$. However, Takikawa and D'Ambrosio's approach allows more efficient elimination orderings in the variable elimination algorithm, while Díez's method enforces more restrictions on the orderings. More recently, Díez and Galán [35] proposed a multiplicative factorization which improves on this previous work, as it has the advantages of both methods. I use their auxiliary graph as the starting point for one of my CNF encodings. In my experiments, I perform a detailed empirical comparison of their approach using variable elimination against my proposals on large Bayesian networks.

In my work, I build upon the DPLL-based weighted model counting approach of Sang, Beame, and Kautz [89] (see Chapter 2). Their general encoding assumes full CPTs and yields a parameter clause for each CPT parameter. However, this approach is impractical for large-scale noisy-OR/MAX networks. Our special-purpose encodings extend the weighted model counting approach for exact inference to networks that were previously intractable for the approach.

4.3 Efficient Encodings of Noisy-OR into CNF

In this section, I present techniques for improving the weighted model counting approach for Bayesian networks with noisy-OR relations. In particular, I present two space efficient CNF encodings for noisy-OR which exploit their semantics. In my encodings, I pay particular attention to reducing the treewidth of the CNF formula and to directly encoding the effect of unit propagation on evidence into the CNF formula, without actually performing unit propagation. I also take advantage of the Boolean domains to simplify the encodings. I use as a running example the Bayesian network shown in Figure 4.2. In the subsequent section, I generalize to noisy-MAX relations.

4.3.1 Weighted CNF Encoding 1: An Additive Encoding

Let there be causes $X_1, \ldots, X_n$ leading to an effect Y and let there be a noisy-OR relation at node Y (see Figure 4.1(a)), where all random variables are assumed to have Boolean-valued domains.

In my first weighted model encoding method (WMC1), I introduce an indicator variable I_Y for Y and an indicator variable I_{X_i} for each parent of Y. I also introduce a parameter variable P_{q_i} for each parameter q_i, $1 \leq i \leq n$ in the noisy-OR (see Equation 4.1). The weights of these variables are as follows,

$$\begin{aligned} weight(I_{X_i}) = weight(I_Y) &= 1 \\ weight(P_{q_i}) &= 1 - q_i \\ weight(\neg P_{q_i}) &= q_i \end{aligned}$$

The noisy-OR relation can then be encoded as the formula,

$$(I_{X_1} \wedge P_{q_1}) \vee (I_{X_2} \wedge P_{q_2}) \vee \cdots \vee (I_{X_n} \wedge P_{q_n}) \Leftrightarrow I_Y. \tag{4.7}$$

The formula can be seen to be an encoding of Pearl's well-known decomposition for noisy-OR (see Figure 4.3).

Example 4.5. *Consider once again the Bayesian network shown in Figure 4.2 and the parameters for the noisy-ORs shown in Table 4.1. The WMC1 encoding introduces the five Boolean indicator variables I_C, I_F, I_M, I_N, and I_H, each with weight 1; and the six parameter variables $P_{0.6}$, $P_{0.5}$, $P_{0.4}$, $P_{0.3}$, $P_{0.2}$, and $P_{0.1}$, each with $weight(P_{q_i}) = 1 - q_i$ and $weight(\neg P_{q_i}) = q_i$. Using Equation 4.7, the noisy-OR at node Nausea can be encoded as,*

$$(I_C \wedge P_{0.6}) \vee (I_F \wedge P_{0.5}) \vee (I_M \wedge P_{0.4}) \Leftrightarrow I_N.$$

To illustrate the weighted model counting of the formula, suppose that nausea and malaria are absent and cold and flu are present (i.e., Nausea = 0, Malaria = 0, Cold = 1, and Flu = 1; and for the corresponding indicator variables I_N and I_M are false and I_C and I_F are true). The formula can be simplified to,

$$(P_{0.6}) \vee (P_{0.5}) \Leftrightarrow 0.$$

There is just one model for this formula, the model that sets $P_{0.6}$ to false and $P_{0.5}$ to false. Hence, the weighted model count of this formula is $weight(\neg P_{0.6}) \times weight(\neg P_{0.5})$ $= 0.6 \times 0.5 = 0.3$, which is just the entry in the penultimate row of the full CPT shown in Example 4.4.

Towards converting Equation 4.7 into CNF, I also introduce an auxiliary indicator variable w_i for each conjunction such that $w_i \Leftrightarrow I_{X_i} \wedge P_{q_i}$. This dramatically reduces the number of clauses generated. Equation 4.7 is then transformed into,

$$\begin{aligned}
(\neg I_Y \quad \vee \quad & ((w_1 \vee \ldots \vee w_n) \wedge \\
& (\neg I_{X_1} \vee \neg P_{q_1} \vee w_1) \wedge \\
& (I_{X_1} \vee \neg w_1) \wedge \\
& (P_{q_1} \vee \neg w_1) \\
& \wedge \ldots \wedge \\
& (\neg I_{X_n} \vee \neg P_{q_n} \vee w_n) \wedge \\
& (I_{X_n} \vee \neg w_n) \wedge \\
& (P_{q_n} \vee \neg w_n))) \wedge \\
(I_Y \quad \vee \quad & ((\neg I_{X_1} \vee \neg P_{q_1}) \\
& \wedge \ldots \wedge \\
& (\neg I_{X_n} \vee \neg P_{q_n})))
\end{aligned} \tag{4.8}$$

The formula is not in CNF, but can be easily transformed into CNF using the distribu-

tive law. It can be seen that the WMC1 encoding can also easily encode evidence—i.e, if $I_Y = 0$ or $I_Y = 1$, the formula can be further simplified—before the final translation into CNF.

Example 4.6. *Consider once again the Bayesian network shown in Figure 4.2. To illustrate the encoding of evidence, suppose that nausea is present (i.e., Nausea = 1) and headache is not present (i.e., Headache = 0). The corresponding constraints for the evidence are as follows.*

$$(I_C \wedge P_{0.6}) \vee (I_F \wedge P_{0.5}) \vee (I_M \wedge P_{0.4}) \Leftrightarrow 1 \tag{4.9}$$

$$(I_C \wedge P_{0.3}) \vee (I_F \wedge P_{0.2}) \vee (I_M \wedge P_{0.1}) \Leftrightarrow 0 \tag{4.10}$$

Using Equation 4.8, the above constraints can be converted into CNF clauses. Constraint Equation 4.9 gives the clauses,

$$\begin{aligned} &(w_1 \vee w_2 \vee w_3) \\ \wedge\ &(\neg I_C \vee \neg P_{0.6} \vee w_1) \wedge (I_C \vee \neg w_1) \wedge (P_{0.6} \vee \neg w_1) \\ \wedge\ &(\neg I_F \vee \neg P_{0.5} \vee w_2) \wedge (I_F \vee \neg w_2) \wedge (P_{0.5} \vee \neg w_2) \\ \wedge\ &(\neg I_M \vee \neg P_{0.4} \vee w_3) \wedge (I_M \vee \neg w_3) \wedge (P_{0.4} \vee \neg w_3) \end{aligned}$$

and constraint Equation 4.10 gives the clauses,

$$(\neg I_C \vee \neg P_{0.3}) \wedge (\neg I_F \vee \neg P_{0.2}) \wedge (\neg I_M \vee \neg P_{0.1}).$$

Correctness

To show the correctness of encoding WMC1 of a noisy-OR, I first show that each entry in the full CPT representation of a noisy-OR relation can be determined using the weighted model count of the encoding. As always, let there be causes $X_1, \ldots, X_n$ leading to an effect Y and let there be a noisy-OR relation at node Y, where all random variables have Boolean-valued domains.

Lemma 4.1. *Each entry in the full CPT representation of a noisy-OR at a node Y, $P(Y = y \mid X_1 = x_1, \ldots, X_n = x_n)$, can be determined using the weighted model count of Equation 4.7 created using the encoding WMC1.*

Proof. Let F_Y be the encoding of the noisy-OR at node Y using WMC1 and let s be the set of assignments to the indicator variables I_Y, $I_{X_1}, \ldots, I_{X_n}$ corresponding to the desired entry in the CPT (e.g., if $Y = 0$, I_Y is instantiated to false; otherwise it is instantiated to true). For each $X_i = 0$, the disjunct $(I_{X_i} \wedge P_{q_i})$ in Equation 4.7 is false and would be removed in the residual formula $F_Y|_s$; and for each $X_i = 1$, the disjunct reduces to (P_{q_i}). If $I_Y = 0$, each of the disjuncts in Equation 4.7 must be false and there is only a single model of the formula. Hence,

$$weight(F_Y|_s) = \prod_{i \in T_x} weight(\neg P_{q_i}) = \prod_{i \in T_x} q_i = P(Y = 0 \mid \boldsymbol{X}),$$

where $T_x = \{i \mid X_i = 1\}$ and $P(Y = 0 \mid \boldsymbol{X}) = 1$ if T_x is empty. If $I_Y = 1$, at least one of the disjuncts in Equation 4.7 must be true and there are, therefore, $2^{|T_x|} - 1$ models. It can be seen that if we sum over all $2^{|T_x|}$ possible *assignments*, the weight of the formula is 1. Hence, subtracting off the one possible assignment which is not a model gives,

$$weight(F_Y|_s) = 1 - \prod_{i \in T_x} weight(\neg P_{q_i}) = 1 - \prod_{i \in T_x} q_i = P(Y = 1 \mid \boldsymbol{X}).$$

□

A *noisy-OR Bayesian network* over a set of random variables $Z_1, \ldots, Z_n$ is a Bayesian network where there are noisy-OR relations at one or more of the Z_i and full CPTs at the remaining nodes. The next step in the proof of correctness is to show that each entry in the joint probability distribution represented by a noisy-OR Bayesian network can be determined using weighted model counting. In what follows, I assume that noisy-OR nodes are encoded using WMC1 and the remaining nodes are encoded using Sang et al.'s general encoding discussed in Section 2.2.4. Similar results can be stated using Darwiche's general encoding.

Lemma 4.2. *Each entry in the joint probability distribution, $P(Z_1 = z_1, \ldots Z_n = z_n)$, represented by a noisy-OR Bayesian network can be determined using weighted model counting and encoding WMC1.*

Proof. Let F be the encoding of the Bayesian network using WMC1 for the noisy-OR nodes and let s be the set of assignments to the indicator variables $I_{Z_1}, \ldots, I_{Z_n}$ corresponding to the desired entry in the joint probability distribution. By Equation 2.1, the entry in the joint probability distribution is determined by multiplying the corresponding CPT entries at each node in the network. For those nodes with full CPTs, s determines the correct entry in each CPT by Lemma 2 in Sang et al. [89] and for those nodes with noisy-ORs, s determines the correct probability by Lemma 4.1 above. Thus, $weight(F \wedge s)$ is the multiplication of the corresponding CPT entries; i.e., the entry in the joint probability distribution. □

The final step in the proof of correctness is to show that queries of interest can be correctly answered.

Theorem 4.1. *Given a noisy-OR Bayesian network, general queries of the form $P(\boldsymbol{Q} \mid \boldsymbol{E})$ can be determined using weighted model counting and encoding WMC1.*

Proof. Let F be the CNF encoding of a noisy-OR Bayesian network. A general query $P(\boldsymbol{Q} \mid \boldsymbol{E})$ on the network can be answered by,

$$\frac{P(\boldsymbol{Q} \wedge \boldsymbol{E})}{P(\boldsymbol{E})} = \frac{weight(F \wedge Q \wedge E)}{weight(F \wedge E)},$$

where Q and E are propositional formulas which enforce the appropriate values for the indicator variables that correspond to the known values of the random variables.

By definition, the function *weight* computes the weighted sum of the solutions of its argument. By Lemma 4.2, this is equal to the sum of the probabilities of those sets of assignments that satisfy the restrictions $Q \wedge E$ and E, respectively, which in turn is equal to the sum of the entries in the joint probability distribution that are consistent with $\boldsymbol{Q} \wedge \boldsymbol{E}$ and $\boldsymbol{E}$, respectively. □

As Sang et al. [89] note, the weighted model counting approach supports queries and evidence in arbitrary propositional form and such queries are not supported by any other exact inference method.

4.3.2 Weighted CNF Encoding 2: A Multiplicative Encoding

Again, let there be causes $X_1, \ldots, X_n$ leading to an effect Y and let there be a noisy-OR relation at node Y (see Figure 4.1(a)), where all random variables are assumed to have Boolean-valued domains.

My second weighted model encoding method (WMC2) takes as its starting point Díez and Galán's [35] directed auxiliary graph transformation of a Bayesian network with a noisy-OR/MAX relation[1]. Díez and Galán note that for the noisy-OR relation, Equation (4.6) can be represented as a product of matrices,

$$\begin{pmatrix} P(Y = 0 \mid \boldsymbol{X}) \\ P(Y = 1 \mid \boldsymbol{X}) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} P(Y \leq 0 \mid \boldsymbol{X}) \\ P(Y \leq 1 \mid \boldsymbol{X}) \end{pmatrix}.$$

Based on this factorization, one can integrate a noisy-OR node into a regular Bayesian network by introducing a hidden node for each Y' for each noisy-OR node Y. The transformation first creates a graph with the same set of nodes and arcs as the original network. Then, for each node Y with a noisy-OR relation,

- add a hidden node Y' with the same domain as Y,
- add an arc $Y' \rightarrow Y$,
- redirect each arc $X_i \rightarrow Y$ to $X_i \rightarrow Y'$, and
- associate with Y a factorization table,

	$Y' = 0$	$Y' = 1$
$Y = 0$	1	0
$Y = 1$	−1	1.

This auxiliary graph is not a Bayesian network as it contains parameters which are less than 0. So the CNF encoding methods for general Bayesian networks (see Chapter 2) cannot be applied here.

[1]The Díez and Galán [35] transformation is a generalization to noisy-MAX of the noisy-OR transformation of Takikawa and D'Ambrosio [98].

I introduce indicator variables $I_{Y'}$ and I_Y for Y' and Y, and an indicator variable I_{X_i} for each parent of Y'. The weights of these variables are as follows,

$$weight(I_{Y'}) \;=\; weight(I_Y) \;=\; weight(I_{X_i}) \;=\; 1.$$

For each arc $X_i \rightarrow Y'$, $1 \leq i \leq n$, I create two parameter variables $P^0_{X_i,Y'}$ and $P^1_{X_i,Y'}$. The weights of these variables are as follows,

$$\begin{array}{rclcrcl} weight(P^0_{X_i,Y'}) & = & 1, & \quad & weight(P^1_{X_i,Y'}) & = & q_i, \\ weight(\neg P^0_{X_i,Y'}) & = & 0, & & weight(\neg P^1_{X_i,Y'}) & = & 1-q_i. \end{array}$$

For each factorization table, I introduce two variables, u_Y and w_Y, where the weights of these variables are given by,

$$\begin{array}{rclcrcl} weight(u_Y) & = & 1, & \quad & weight(\neg u_Y) & = & 0, \\ weight(w_Y) & = & -1, & & weight(\neg w_Y) & = & 2. \end{array}$$

For the first row of a factorization table, we generate the clause,

$$(\neg I_{Y'} \vee I_Y), \tag{4.11}$$

and for the second row, we generate the clauses,

$$(\neg I_{Y'} \vee \neg I_Y \vee u_Y) \wedge (I_{Y'} \vee \neg I_Y \vee w_Y). \tag{4.12}$$

Finally, for every parent X_i of Y', we generate the clauses,

$$(I_{Y'} \vee I_{X_i} \vee P^0_{X_i,Y'}) \wedge (I_{Y'} \vee \neg I_{X_i} \vee P^1_{X_i,Y'}). \tag{4.13}$$

We now have a conjunction of clauses; i.e., CNF.

Example 4.7. *Consider once again the Bayesian network shown in Figure 4.2 and the parameters for the noisy-ORs shown in Table 4.1. The auxiliary graph transformation is shown in Figure 4.6. The WMC2 encoding introduces the seven Boolean indicator variables I_C, I_F, I_M, I'_N, I_N, I'_H, and I_H; the twelve parameter variables,*

$$\begin{array}{cccc} P^0_{C,N'} & P^1_{C,N'} & P^0_{C,H'} & P^1_{C,H'} \\ P^0_{F,N'} & P^1_{F,N'} & P^0_{F,H'} & P^1_{F,H'} \\ P^0_{M,N'} & P^1_{M,N'} & P^0_{M,H'} & P^1_{M,H'}; \end{array}$$

and the four factorization variables u_N, w_N, u_H, and w_H. The noisy-OR at node Nausea can be encoded as the set of clauses,

$$\begin{array}{lll} \neg I_{N'} \vee I_N & I_{N'} \vee I_C \vee P^0_{C,N'} & I_{N'} \vee \neg I_C \vee P^1_{C,N'} \\ \neg I_{N'} \vee \neg I_N \vee u_N & I_{N'} \vee I_F \vee P^0_{F,N'} & I_{N'} \vee \neg I_F \vee P^1_{F,N'} \\ I_{N'} \vee \neg I_N \vee w_N & I_{N'} \vee I_M \vee P^0_{M,N'} & I_{N'} \vee \neg I_M \vee P^1_{M,N'} \end{array}$$

To illustrate the weighted model counting of the formula, suppose that nausea and malaria are absent and cold and flu are present (i.e., Nausea = 0, Malaria = 0,

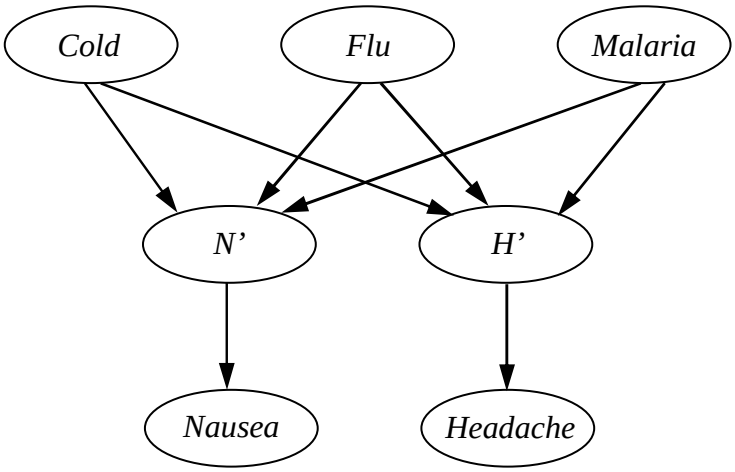


Figure 4.6: Díez and Galán's [35] transformation of a noisy-OR relation applied to the Bayesian network shown in Figure 4.2.

Cold $= 1$*, and Flu* $= 1$*; and for the corresponding indicator variables* I_N *and* I_M *are false and* I_C *and* I_F *are true). The formula can be simplified to,*

$$P^1_{C,N'} \wedge P^1_{F,N'} \wedge P^0_{M,N'}.$$

(To see this, note that clauses that evaluate to true are removed and literals that evaluate to false are removed from a clause. As a result of simplifying the first clause, $I_{N'}$ *is forced to be false and is removed from the other clauses.) There is just one model for this formula, the model that sets each of the conjunctions to true. Hence, the weighted model count of this formula is* $weight(P^1_{C,N'}) \times weight(P^1_{F,N'}) \times weight(P^0_{M,N'})$ $= 0.6 \times 0.5 \times 1.0 = 0.3$*, which is just the entry in the penultimate row of the full CPT shown in Example 4.4.*

Once again, it can be seen that WMC2 can also easily encode evidence into the CNF formula; i.e., if $I_Y = 0$ or $I_Y = 1$, the formula can be further simplified.

Example 4.8. *Consider once again the Bayesian network shown in Figure 4.2. To illustrate the encoding of evidence, suppose that nausea is present (i.e.,* $Nausea = 1$*) and headache is not present (i.e.,* $Headache = 0$*). The WMC2 encoding results in the following set of clauses,*

$$\begin{array}{lll}
 & I_{N'} \vee I_C \vee P^0_{C,N'} & I_{N'} \vee \neg I_C \vee P^1_{C,N'} \\
\neg I_{N'} \vee u_N & I_{N'} \vee I_F \vee P^0_{F,N'} & I_{N'} \vee \neg I_F \vee P^1_{F,N'} \\
I_{N'} \vee w_N & I_{N'} \vee I_M \vee P^0_{M,N'} & I_{N'} \vee \neg I_M \vee P^1_{M,N'} \\
 & & \\
\neg I_{H'} & I_C \vee P^0_{C,H'} & \neg I_C \vee P^1_{C,H'} \\
 & I_F \vee P^0_{F,H'} & \neg I_F \vee P^1_{F,H'} \\
 & I_M \vee P^0_{M,H'} & \neg I_M \vee P^1_{M,H'}
\end{array}$$

Correctness

To show the correctness of encoding WMC2 of a noisy-OR, I first show that each entry in the full CPT representation of a noisy-OR relation can be determined using the weighted model count of the encoding. As always, let there be causes $X_1, \ldots, X_n$ leading to an effect Y and let there be a noisy-OR relation at node Y, where all random variables have Boolean-valued domains.

Lemma 4.3. *Each entry in the full CPT representation of a noisy-OR at a node Y, $P(Y = y \mid X_1 = x_1, \ldots, X_n = x_n)$, can be determined using the weighted model count of Equations 4.11–4.13 created using the encoding WMC2.*

Proof. Let F_Y be the encoding of the noisy-OR at node Y using WMC2 and let s be the set of assignments to the indicator variables $I_Y, I_{X_1}, \ldots, I_{X_n}$ corresponding to the desired entry in the CPT. For each $X_i = 0$, the clauses in Equation 4.13 reduce to $(I_{Y'} \vee P^0_{X_i,Y'})$, and for each $X_i = 1$, the clauses reduce to $(I_{Y'} \vee P^1_{X_i,Y'})$. If $I_Y = 0$, the clauses in Equations 4.11 & 4.12 reduce to $(\neg I_{Y'})$. Hence,

$$\begin{aligned} weight(F_Y|_s) &= weight(\neg I_{Y'}) \prod_{i \notin T_x} weight(P^0_{X_i,Y'})) \prod_{i \in T_x} weight(P^1_{X_i,Y'})) \\ &= \prod_{i \in T_x} q_i \\ &= P(Y = 0 \mid \boldsymbol{X}), \end{aligned}$$

where $T_x = \{i \mid X_i = 1\}$ and $P(Y = 0 \mid \boldsymbol{X}) = 1$ if T_x is empty. If $I_Y = 1$, the clauses in Equations 4.11 & 4.12 reduce to $(\neg I_{Y'} \vee u_Y) \wedge (I_{Y'} \vee w_Y)$. Hence,

$$\begin{aligned} weight(F_Y|_s) &= weight(\neg I_{Y'})weight(\neg u_Y)weight(w_Y) \prod_{i \in T_x} q_i + \\ &\quad weight(\neg I_{Y'})weight(u_Y)weight(w_Y) \prod_{i \in T_x} q_i + \\ &\quad weight(I_{Y'})weight(u_Y)weight(\neg w_Y) + \\ &\quad weight(I_{Y'})weight(u_Y)weight(w_Y) \\ &= 1 - \prod_{i \in T_x} q_i \\ &= P(Y = 1 \mid \boldsymbol{X}). \end{aligned}$$

□

The remainder of the proof of correctness for encoding WMC2 is similar to that of encoding WMC1.

Lemma 4.4. *Each entry in the joint probability distribution, $P(Z_1 = z_1, \ldots Z_n = z_n)$, represented by a noisy-OR Bayesian network can be determined using weighted model counting and encoding WMC2.*

Theorem 4.2. *Given a noisy-OR Bayesian network, general queries of the form* $P(\boldsymbol{Q} \mid \boldsymbol{E})$ *can be determined using weighted model counting and encoding WMC2.*

4.4 Efficient Encodings of Noisy-MAX into CNF

Let there be causes $X_1, \ldots, X_n$ leading to an effect Y and let there be a noisy-MAX relation at node Y (see Figure 4.1(a)), where the random variables may have multi-valued (non-Boolean) domains. Let d_X be the number of values in the domain of some random variable X.

The WMC2 multiplicative encoding above can be extended to noisy-MAX by introducing more indicator variables to represent variables with multiple values. In this section, I explain the extension and present two noisy-MAX encodings based on two different weight definitions of the parameter variables. The two noisy-MAX encodings are denoted MAX1 and MAX2, respectively. I begin by presenting those parts of the encodings that MAX1 and MAX2 have in common. As with WMC2, these two noisy-MAX encodings take as their starting point Díez and Galán's [35] directed auxiliary graph transformation of a Bayesian network with noisy-OR/MAX. Díez and Galán show that for the noisy-MAX relation, Equation (4.6) can be factorized as a product of matrices,

$$P(Y = y \mid \boldsymbol{X}) = \sum_{y'=0}^{y} M_Y(y, y') \cdot P(Y \leq y' \mid \boldsymbol{X}) \tag{4.14}$$

where M_Y is a $d_Y \times d_Y$ matrix given by,

$$M_Y(y, y') = \begin{cases} 1, & \text{if } y' = y, \\ -1, & \text{if } y' = y - 1, \\ 0, & \text{otherwise.} \end{cases}$$

For each noisy-MAX node Y, I introduce d_Y indicator variables I_{Y_0} ... $I_{Y_{d_Y-1}}$, to represent each value in the domain of Y, and $\binom{d_Y}{2} + 1$ clauses to ensure that exactly one of these variables is true. As in WMC2, I introduce a hidden node Y' with the same domain as Y, corresponding indicator variables to represent each value in the domain of Y', and clauses to ensure that exactly one domain value is selected in each model. For each parent X_i, $1 \leq i \leq n$, of Y, I define indicator variables $I_{i,x}$, where $x = 0, \ldots, d_{X_i} - 1$, and add clauses that ensure that exactly one of the indicator variables corresponding to each X_i is true. Each indicator variable and each negation of an indicator variable has weight 1.

Example 4.9. *Consider once again the Bayesian network shown in Figure 4.2 and the parameters for the noisy-MAX shown in Table 4.2. As the node Nausea has domain* $\{$*absent = 0, mild = 1, severe = 2*$\}$ *and the parents Cold, Flu, and Malaria are Boolean valued, both the MAX1 and MAX2 encodings introduce the Boolean indicator variables* I_{N_a}, I_{N_m}, I_{N_s}, $I_{N'_a}$, $I_{N'_m}$, $I_{N'_s}$, I_{C_0}, I_{C_1}, I_{F_0}, I_{F_1}, I_{M_0}, *and* I_{M_1}. *The weights*

of these variables and their negations are 1. Four clauses are added over the indicator variables for Nausea,

$$\begin{array}{lll} (I_{N_a} \vee I_{N_m} \vee I_{N_s}) & \wedge & (\neg I_{N_a} \vee \neg I_{N_m}) \\ & \wedge & (\neg I_{N_a} \vee \neg I_{N_s}) \\ & \wedge & (\neg I_{N_m} \vee \neg I_{N_s}). \end{array}$$

Similar clauses are added over the indicator variables for the hidden node N' and over the indicator variables for the parents Cold, Flu, and Malaria, respectively.

For each factorization table, I introduce two auxiliary variables, u_Y and w_Y, where the weights of these variables are given by,

$$\begin{array}{rclrcl} weight(u_Y) & = & 1, & weight(\neg u_Y) & = & 0, \\ weight(w_Y) & = & -1, & weight(\neg w_Y) & = & 2. \end{array}$$

For each factorization table, a clause is added for each entry in the matrix,

$$M_Y(y, y') = \begin{cases} 1, & \text{add } (\neg I_{y'} \vee \neg I_y \vee u_Y) \quad \text{if } y' = y, \\ -1, & \text{add } (\neg I_{y'} \vee \neg I_y \vee w_Y) \quad \text{if } y' = y - 1, \\ 0, & \text{add } (\neg I_{y'} \vee \neg I_y \vee \neg u_Y) \quad \text{otherwise.} \end{cases}$$

Example 4.10. *Consider once again the Bayesian network shown in Figure 4.2 and the parameters for the noisy-MAX shown in Table 4.2. As Nausea has domain {absent = 0, mild = 1, severe = 2}, the factorization table M_N is given by,*

	$N' = absent$	$N' = mild$	$N' = severe$
$N = absent$	1	0	0
$N = mild$	−1	1	0
$N = severe$	0	−1	1.

Auxiliary variables u_N and w_N are introduced and the following clauses, shown in row order, would be added for the factorization table M_N,

$$\begin{array}{lll} \neg I_{N_a} \vee \neg I_{N'_a} \vee u_N & \neg I_{N_a} \vee \neg I_{N'_m} \vee \neg u_N & \neg I_{N_a} \vee \neg I_{N'_s} \vee \neg u_N \\ \neg I_{N_m} \vee \neg I_{N'_a} \vee w_N & \neg I_{N_m} \vee \neg I_{N'_m} \vee u_N & \neg I_{N_m} \vee \neg I_{N'_s} \vee \neg u_N \\ \neg I_{N_s} \vee \neg I_{N'_a} \vee \neg u_N & \neg I_{N_s} \vee \neg I_{N'_m} \vee w_N & \neg I_{N_s} \vee \neg I_{N'_s} \vee u_N. \end{array}$$

That completes the description of those parts of the encodings that are common to both MAX1 and MAX2.

4.4.1 Weighted CNF Encoding 1 for Noisy-MAX

My first weighted model counting encoding for noisy-MAX relations (MAX1) is based on an additive definition of noisy-MAX. Recall the decomposed probabilistic model

for the noisy-MAX relation discussed at the end of Section 4.1. It can be shown that for the noisy-MAX, $P(Y \leq y \mid X_1, \ldots, X_n)$ can be determined using,

$$P(Y \leq y \mid X_1, \ldots, X_n) = \sum_{Y_i \leq y} \prod_{i=1}^{n} P(Y_i \mid X_i) = \sum_{Y_i \leq y} \prod_{\substack{i=1 \\ X_i \neq 0}}^{n} q_{i,Y_i}^{X_i} \tag{4.15}$$

where the $q_{i,Y_i}^{X_i}$ are the parameters to the noisy-MAX, and the sum is over all the configurations or possible values for $Y_1, \ldots, Y_n$, such that each of these values is less than or equal to the value y. Note that the outer operator is summation; hence, I refer to MAX1 as an additive encoding. Substituting the above into Equation 4.14 gives,

$$P(Y = y \mid X_1, \ldots, X_n) = \sum_{y'=0}^{y} M_Y(y, y') \cdot \left(\sum_{Y_i \leq y'} \prod_{\substack{i=1 \\ X_i \neq 0}}^{n} q_{i,Y_i}^{X_i} \right). \tag{4.16}$$

It is this equation that I encode into CNF. The encoding of the factorization table M_Y is common to both encodings and has been explained above. It remains to encode the computation for $P(Y \leq y \mid X_1, \ldots, X_n)$.

For each parent X_i, $1 \leq i \leq n$, of Y I introduce d_Y indicator variables, $I_{i,y}$, to represent the effect of X_i on Y, where $0 \leq y \leq d_Y - 1$, and add clauses that ensure that exactly one of the indicator variables correspond to each X_i is true. Note that these indicators variables are in addition to the indicator variables common to both encodings and explained above. As always with indicator variables, the weights of $I_{i,y}$ and $\neg I_{i,y}$ are both 1.

For each parameter $q_{i,y}^x$ to the noisy-MAX, I introduce a corresponding parameter variable $P_{i,y}^x$. The weight of each parameter variable is given by,

$$weight(P_{i,y}^x) = q_{i,y}^x \qquad\qquad weight(\neg P_{i,y}^x) = 1$$

where $1 \leq i \leq n$, $0 \leq y \leq d_Y - 1$, and $1 \leq x \leq d_{X_i} - 1$. The relation between X_i and Y is represented by the parameter clauses[2],

$$(I_{i,x} \wedge I_{i,y}) \Leftrightarrow P_{i,y}^x$$

where $1 \leq i \leq n$, $0 \leq y \leq d_Y - 1$, and $1 \leq x \leq d_{X_i} - 1$.

Example 4.11. *Consider once again the Bayesian network shown in Figure 4.2 and the parameters for the noisy-MAX shown in Table 4.2. For the noisy-MAX at node Nausea, the encoding introduces the indicator variables* I_{C,N_a}, I_{C,N_m}, I_{C,N_s}, I_{F,N_a}, I_{F,N_m}, I_{F,N_s}, I_{M,N_a}, I_{M,N_m}, *and* I_{M,N_s}, *all with weight 1, and the clauses,*

[2]To improve readability, in this section the propositional formulas are sometimes written in a more natural but non-clausal form. I continue to refer to them as clauses when the translation to clause form is straightforward.

$$
\begin{array}{lll}
I_{C,N_a} \vee I_{C,N_m} \vee I_{C,N_s} & I_{F,N_a} \vee I_{F,N_m} \vee I_{F,N_s} & I_{M,N_a} \vee I_{M,N_m} \vee I_{M,N_s} \\
\neg I_{C,N_a} \vee \neg I_{C,N_m} & \neg I_{F,N_a} \vee \neg I_{F,N_m} & \neg I_{M,N_a} \vee \neg I_{M,N_m} \\
\neg I_{C,N_a} \vee \neg I_{C,N_s} & \neg I_{F,N_a} \vee \neg I_{F,N_s} & \neg I_{M,N_a} \vee \neg I_{M,N_s} \\
\neg I_{C,N_m} \vee \neg I_{C,N_s} & \neg I_{F,N_m} \vee \neg I_{F,N_s} & \neg I_{M,N_m} \vee \neg I_{M,N_s}
\end{array}
$$

As well, the following parameter variables and associated weights would be introduced,

$$
\begin{array}{lll}
weight(P^1_{C,N_a}) = 0.7 & weight(P^1_{F,N_a}) = 0.5 & weight(P^1_{M,N_a}) = 0.1 \\
weight(P^1_{C,N_m}) = 0.2 & weight(P^1_{F,N_m}) = 0.2 & weight(P^1_{M,N_m}) = 0.4 \\
weight(P^1_{C,N_s}) = 0.1 & weight(P^1_{F,N_s}) = 0.3 & weight(P^1_{M,N_s}) = 0.5,
\end{array}
$$

along with the following parameter clauses,

$$
\begin{array}{lll}
(I_{C_1} \wedge I_{C,N_a}) \Leftrightarrow P^1_{C,N_a} & (I_{F_1} \wedge I_{F,N_a}) \Leftrightarrow P^1_{F,N_a} & (I_{M_1} \wedge I_{M,N_a}) \Leftrightarrow P^1_{M,N_a} \\
(I_{C_1} \wedge I_{C,N_m}) \Leftrightarrow P^1_{C,N_m} & (I_{F_1} \wedge I_{F,N_m}) \Leftrightarrow P^1_{F,N_m} & (I_{M_1} \wedge I_{M,N_m}) \Leftrightarrow P^1_{M,N_m} \\
(I_{C_1} \wedge I_{C,N_s}) \Leftrightarrow P^1_{C,N_s} & (I_{F_1} \wedge I_{F,N_s}) \Leftrightarrow P^1_{F,N_s} & (I_{M_1} \wedge I_{M,N_s}) \Leftrightarrow P^1_{M,N_s}
\end{array}
$$

It remains to relate (i) the indicator variables, $I_{i,x}$, which represent the value of the parent variable X_i, where $x = 0, \ldots, d_{X_i} - 1$; (ii) the indicator variables, $I_{i,y}$, which represent the effect of X_i on Y, where $y = 0, \ldots, d_Y - 1$; and (iii) the indicator variables, $I_{Y'_{y'}}$, which represent the value of the hidden variable Y', where $y' = 0, \ldots, d_Y - 1$. Causal independent clauses define the relation between (i) and (ii) and assert that if the cause X_i is absent ($X_i = 0$), then X_i has no effect on Y; i.e.,

$$I_{i,x_0} \Rightarrow I_{i,y_0}$$

where $1 \leq i \leq n$. Value constraint clauses define the relation between (ii) and (iii) and assert that if the hidden variable Y' takes on a value y', then the effect of X_i on Y cannot be that Y takes on a higher degree or more severe value y; i.e.,

$$I_{Y'_{y'}} \Rightarrow \neg I_{i,Y_y}$$

where $1 \leq i \leq n$, $0 \leq y' \leq d_Y - 1$, and $y' < y \leq d_Y - 1$.

Example 4.12. *Consider once again the Bayesian network shown in Figure 4.2 and the parameters for the noisy-MAX shown in Table 4.2. For the noisy-MAX at node Nausea, the encoding introduces the causal independence clauses,*

$$I_{C_0} \Rightarrow I_{C,N_a} \qquad I_{F_0} \Rightarrow I_{F,N_a} \qquad I_{M_0} \Rightarrow I_{M,N_a}$$

the value constraint clauses for $N' =$ absent,

$$
\begin{array}{lll}
I_{N'_a} \Rightarrow \neg I_{C,N_m} & I_{N'_a} \Rightarrow \neg I_{F,N_m} & I_{N'_a} \Rightarrow \neg I_{M,N_m} \\
I_{N'_a} \Rightarrow \neg I_{C,N_s} & I_{N'_a} \Rightarrow \neg I_{F,N_s} & I_{N'_a} \Rightarrow \neg I_{M,N_s}
\end{array}
$$

and the value constraint clauses for $N' =$ mild,

$$I_{N'_m} \Rightarrow \neg I_{C,N_s} \qquad I_{N'_m} \Rightarrow \neg I_{F,N_s} \qquad I_{N'_m} \Rightarrow \neg I_{M,N_s}$$

4.4.2 Weighted CNF Encoding 2 for Noisy-MAX

My second weighted model counting encoding for noisy-MAX relations (MAX2) is based on a multiplicative definition of noisy-MAX. Equation 4.5 states that $P(Y \leq y \mid X_1, \ldots, X_n)$ can be determined using,

$$P(Y \leq y \mid \boldsymbol{X}) = \prod_{\substack{i=1 \\ x_i \neq 0}}^{n} \sum_{y'=0}^{y} q_{i,y'}^{x_i}. \tag{4.17}$$

Note that the outer operator is multiplication; hence I refer to MAX2 as a multiplicative encoding. Substituting the above into Equation 4.14 gives,

$$P(Y = y \mid X_1, \ldots, X_n) = \sum_{y'=0}^{y} M_Y(y, y') \cdot \left(\prod_{\substack{i=1 \\ x_i \neq 0}}^{n} \sum_{y''=0}^{y'} q_{i,y''}^{x_i} \right). \tag{4.18}$$

It is this equation that I encode into CNF. The encoding of the factorization table M_Y is common to both encodings and has been explained above. It remains to encode the computation for $P(Y \leq y \mid X_1, \ldots, X_n)$.

For each parameter $q_{i,y}^x$ to the noisy-MAX, I introduce a corresponding parameter variable $P_{i,y}^x$. The weight of each parameter variable pre-computes the summation in Equation 4.17,

$$weight(P_{i,y}^x) = \sum_{y'=0}^{y} q_{i,y'}^x \qquad\qquad weight(\neg P_{i,y}^x) = 1$$

where $1 \leq i \leq n$, $0 \leq y \leq d_Y - 1$, and $1 \leq x \leq d_{X_i} - 1$. The relation between X_i and Y' is represented by the parameter clauses,

$$(I_{i,x} \wedge I_{y'}) \Leftrightarrow P_{i,y}^x,$$

where $0 \leq y \leq d_Y - 1$ and $0 \leq x \leq d_{X_i} - 1$.

Example 4.13. *Consider once again the Bayesian network shown in Figure 4.2 and the parameters for the noisy-MAX shown in Table 4.2. For the noisy-MAX at node Nausea, the following parameter variables and associated weights would be introduced,*

$$\begin{array}{lll} weight(P_{C,N_a}^1) = 0.7 & weight(P_{F,N_a}^1) = 0.5 & weight(P_{M,N_a}^1) = 0.1 \\ weight(P_{C,N_m}^1) = 0.9 & weight(P_{F,N_m}^1) = 0.7 & weight(P_{M,N_m}^1) = 0.5 \\ weight(P_{C,N_s}^1) = 1 & weight(P_{F,N_s}^1) = 1 & weight(P_{M,N_s}^1) = 1, \end{array}$$

along with the following parameter clauses,

$$\begin{array}{lll} (I_{C_1} \wedge I_{N'_a}) \Leftrightarrow P_{C,N_a}^1 & (I_{F_1} \wedge I_{N'_a}) \Leftrightarrow P_{F,N_a}^1 & (I_{M_1} \wedge I_{N'_a}) \Leftrightarrow P_{M,N_a}^1 \\ (I_{C_1} \wedge I_{N'_m}) \Leftrightarrow P_{C,N_m}^1 & (I_{F_1} \wedge I_{N'_m}) \Leftrightarrow P_{F,N_m}^1 & (I_{M_1} \wedge I_{N'_m}) \Leftrightarrow P_{M,N_m}^1 \\ (I_{C_1} \wedge I_{N'_s}) \Leftrightarrow P_{C,N_s}^1 & (I_{F_1} \wedge I_{N'_s}) \Leftrightarrow P_{F,N_s}^1 & (I_{M_1} \wedge I_{N'_s}) \Leftrightarrow P_{M,N_s}^1 \end{array}$$

As stated so far, the encoding is sufficient for correctly determining each entry in the full CPT representation of a noisy-MAX relation using weighted model counting. However, to improve the efficiency of the encoding, I add redundant clauses. The redundant clauses do not change the set of solutions to the encoding, and thus do not change the weighted model count. They do, however, increase the propagation and thus the overall speed of computation in the special case where all of the causes are absent. To this end, for each noisy-MAX node Y, I introduce an auxiliary variable I_{v_Y} with weights given by,

$$weight(I_{v_Y}) \quad = \quad 1, \qquad weight(\neg I_{v_Y}) \quad = \quad 0,$$

and I introduce the clauses,

$$\left(\bigwedge_i^n I_{i,0}\right) \Rightarrow (I_{Y_0'} \Rightarrow I_{v_Y}), \qquad \left(\bigwedge_i^n I_{i,0}\right) \Rightarrow (I_{Y_0} \Rightarrow I_{v_Y}),$$

and the clauses,

$$\left(\bigwedge_i^n I_{i,0}\right) \Rightarrow (I_{y'} \Rightarrow \neg I_{v_Y}), \qquad \left(\bigwedge_i^n I_{i,0}\right) \Rightarrow (I_{y} \Rightarrow \neg I_{v_Y}),$$

where $1 \leq y' \leq d_Y - 1$ and $1 \leq y \leq d_Y - 1$.

Example 4.14. *Consider once again the Bayesian network shown in Figure 4.2. For the noisy-MAX at node Nausea, an auxiliary variable I_{v_N} is introduced with $weight(I_{v_N}) = 1$ and $weight(\neg I_{v_N}) = 0$ along with the following redundant clauses,*

$$\begin{array}{ll}
(I_{C_0} \wedge I_{F_0} \wedge I_{M_0}) \Rightarrow (I_{N_a'} \Rightarrow I_{v_N}) & (I_{C_0} \wedge I_{F_0} \wedge I_{M_0}) \Rightarrow (I_{N_a} \Rightarrow I_{v_N}) \\
(I_{C_0} \wedge I_{F_0} \wedge I_{M_0}) \Rightarrow (I_{N_m'} \Rightarrow \neg I_{v_N}) & (I_{C_0} \wedge I_{F_0} \wedge I_{M_0}) \Rightarrow (I_{N_m} \Rightarrow \neg I_{v_N}) \\
(I_{C_0} \wedge I_{F_0} \wedge I_{M_0}) \Rightarrow (I_{N_s'} \Rightarrow \neg I_{v_N}) & (I_{C_0} \wedge I_{F_0} \wedge I_{M_0}) \Rightarrow (I_{N_s} \Rightarrow \neg I_{v_N}).
\end{array}$$

4.5 Experimental Evaluation

In this section, I empirically evaluate the effectiveness of my encodings. I use the Cachet solver[3] as it is currently recognized as one of the fastest weighted model counting solvers.

I compare against ACE2 [17]. We also implemented Díez and Galán's [35] approach, which consists of variable elimination applied to an auxiliary network that permits exploitation of causal independence. Our implementation uses algebraic decision diagrams [6] as the base data structure to represent conditional probability tables. Algebraic decision diagrams permit a compact representation by aggregating identical probability values. They also speed up computation by exploiting context-specific independence [14], taking advantage of determinism and caching intermediate results to avoid duplicate computation. The variable elimination heuristic that we used is a greedy one that first eliminates all variables that appear in deterministic potentials of one variable (this is equivalent to unit propagation) and then eliminates the variable that creates the smallest algebraic decision diagram with respect to the eliminated algebraic decision diagrams. In order to avoid creating an algebraic decision diagram for each variable when searching for the next variable to eliminate, the size of a new algebraic decision diagram is estimated by the smallest of two upper bounds: (i) the cross product of the domain size of the variables of the new algebraic decision diagram and (ii) the product of the sizes (e.g., the number of nodes) of the eliminated algebraic decision diagrams.

Good variable ordering heuristics play an important role in the success of modern DPLL-based model counting solvers. Here, I evaluate two heuristics: *Variable State Aware Decaying Sum* (VSADS) and *Tree Decomposition Variable Group Ordering* (dTree). The VSADS heuristic is one of the current best performing dynamic heuristics designed for DPLL-based model counting engines [90]. It can be viewed as a scoring system that attempts to satisfy the most recent conflict clauses and makes its branching decisions based on the number of occurrences of a variable at the same time. Compared with the VSADS heuristic, the dTree heuristic [64] can be described as a mixed variable ordering heuristic. DTree first uses a binary tree decomposition to generate ordered variable groups. The decomposition is done prior to search. The order of the variables within a group is then decided dynamically during the backtracking search using a dynamic heuristic.

All of the experiments were performed on a Pentium workstation with 3GHz hyper-threading CPU and 2GB RAM.

4.5.1 Experiment 1: Random Two-Layer Networks

In my first set of experiments, I used randomly generated two-layer networks to compare the space efficiency and complexity of the WMC1 and WMC2 encodings.

Both the WMC1 and WMC2 encodings can answer probabilistic queries using Equation 2.2. Both encodings lead to quick factorization given evidence during the

[3] `http://www.cs.rochester.edu/u/kautz/Cachet/index.htm`

encoding. The clauses from negative evidence can be represented compactly in the resulting CNF, even with a large number of parents. In the WMC2 encoding, positive evidence can be represented by three Boolean variables, whereas the WMC1 encoding requires n Boolean variables, one for each parent. In the WMC2 encoding, we use two parameter variables ($P^0_{X_i,Y'}$ and $P^1_{X_i,Y'}$) to represent every arc, while the WMC1 encoding only needs one.

Table 4.3: Binary, two layer, noisy-OR networks with 500 diseases and 500 symptoms. Effect of increasing amount of positive evidence (P+) on number of variables in encoding (#var.), treewidth of the encoding (width), average time to solve (sec.), and number of instances solved within a cutoff of one hour (solv.), where the test set contained a total of 30 instances. The value N/A means the average is undefined because of timeouts.

	WMC1				WMC2				ACE	
P+	#var	width	sec.	solv.	#var	width	sec.	solv.	sec.	solv.
30	3686	10	0.2	30	6590	11	0.1	30	32	30
35	3716	11	0.6	30	6605	11	0.2	30	33	30
40	3746	13	21	30	6620	11	0.5	30	33	30
45	3776	14	39	30	6635	13	2	30	36	30
50	3806	19	75	30	6650	13	6	30	41	30
55	3836	22	175	30	6665	16	71	30	166	30
60	3916	24	N/A	17	6680	16	N/A	27	N/A	21

Each random network contains 500 diseases and 500 symptoms. Each symptom has six possible diseases uniformly distributed in the disease set. Table 4.3 shows the treewidth of the encoded CNF for the WMC1 and WMC2 encodings. The first column shows the amount of positive evidence in the symptom variables. The remainder of the evidence variables are negative symptoms. It can be seen that although the WMC1 encoding generates fewer variables than the WMC2 encoding, the CNF created by the WMC2 encoding has smaller width. The probability of evidence (PE) is computed using the tree decomposition guided variable ordering [64] and the results are compared against ACE2[4] (a more detailed experimental analysis is given in the next section). ACE2 is the latest package which applies the general Bayesian networks encoding method to noisy-OR model and then compiles the CNF into an arithmetic circuit [17].

4.5.2 Experiment 2: QMR-DT

In my second set of experiments, I used a Bayesian network called QMR-DT. Compared with randomly generated problems, the QMR-DT presents a real-world inference task with various structural and sparsity properties. For example, in the

[4] `http://reasoning.cs.ucla.edu/ace/`

empirical distribution of diseases, a small proportion of the symptoms are connected with a large number of diseases (see Figure 4.7).

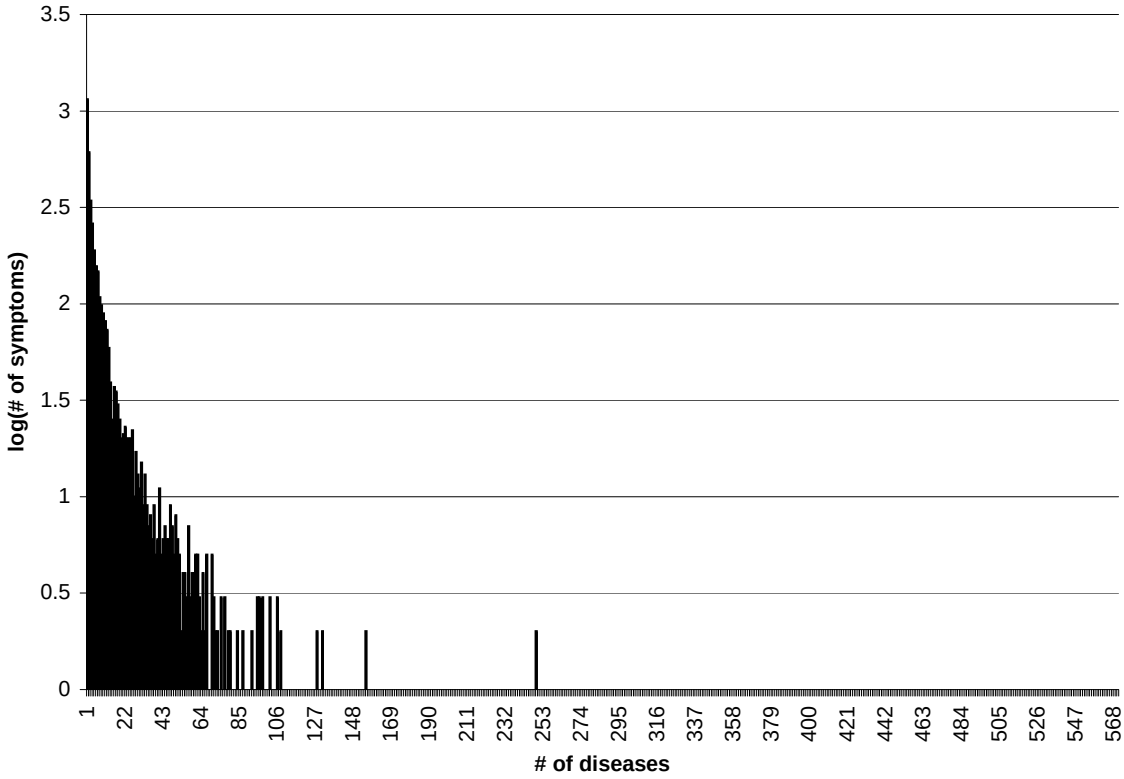


Figure 4.7: Empirical distribution of diseases in the QMR-DT Bayesian network. Approximately 80% of the symptoms are connected with less than 50 diseases.

The network I used was aQMR-DT, an anonymized version of QMR-DT[5]. Symptom vectors with k positive symptoms were generated for each experiment. For each evidence vector, the symptom variables were sorted into ascending order by their parent (disease) number, the first k variables were chosen as positive symptoms, and the remaining symptom variables were set to negative.

I report the runtime to answer the probability of evidence (PE) queries. I also experimented with an implementation of Quickscore[6], but found that it could not solve any of the test cases shown in Figure 4.8. The approach based on weighted model counting also outperforms variable elimination on QMR-DT. The model counting time for 2560 positive symptoms, when using the WMC1 encoding and the VSADS dynamic variable ordering heuristic, is 25 seconds. This same instance could not be solved within one hour by variable elimination.

I tested two different heuristics on each encoding: the VSADS dynamic variable order heuristic and dTree [64], the semi-static tree decomposition-based heuristic. The runtime using an encoding and the dTree heuristic is the sum of two parts: the

[5] `http://www.utoronto.ca/morrislab/aQMR.html`

[6] `http://www.cs.ubc.ca/~murphyk/Software/BNT/bnt.html`

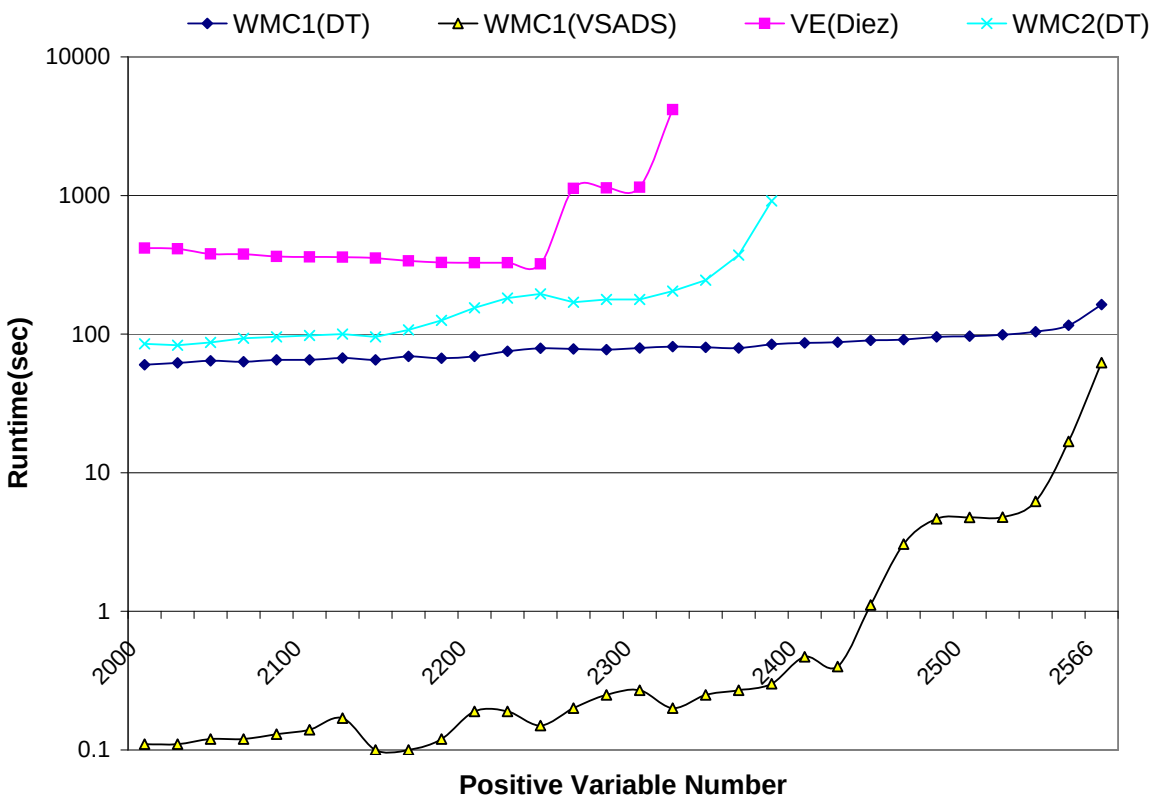


Figure 4.8: The QMR-DT Bayesian network with 4075 symptoms and 570 diseases. Effect of amount of positive symptoms on the time to answer probability of evidence queries, for the WMC1 encoding and the dTree variable ordering heuristic, the WMC1 encoding and the VSADS variable ordering heuristic, the WMC2 encoding and the dTree variable ordering heuristic, and Díez and Galán's [35] approach using variable elimination.

preprocessing time by dTree and the runtime of model counting on the encoding. In this experiment, dTree had a faster runtime than VSADS in the model counting process. However, the overhead of preprocessing for large size networks is too high to achieve better overall performance.

The WMC2 encoding generates twice as many variables as the WMC1 encoding. Although the WMC2 encoding is more promising than the WMC1 encoding on smaller size networks (see Table 4.3), here the WMC2 encoding is less efficient than the WMC1 encoding. The overhead of the tree decomposition ordering on the WMC2 encoding is also higher than on the WMC1 encoding. Our results also show that dynamic variable ordering does not work well on the WMC2 encoding. Model counting using the WMC2 encoding and the VSADS heuristic cannot solve networks when the amount of positive evidence is greater than 1500 symptoms.

The experimental results also show that my approach is more efficient than ACE2. For example, using ACE2, a CNF of QMR-DT with 30 positive symptoms creates 2.8×10^5 variables, 2.8×10^5 clauses and 3.8×10^5 literals. Also, it often requires more than 1GB of memory to finish the compilation process. With the WMC1 encoding,

the same network and the same evidence create only 4.6×10^4 variables, 4.6×10^4 clauses and 1.1×10^5 literals. Cachet, the weighted model counting engine, needs less than 250MB of memory in most cases to solve these instances. And in our experiments, ACE2 cannot solve QMR-DT with more than 500 positive evidence within an hour.

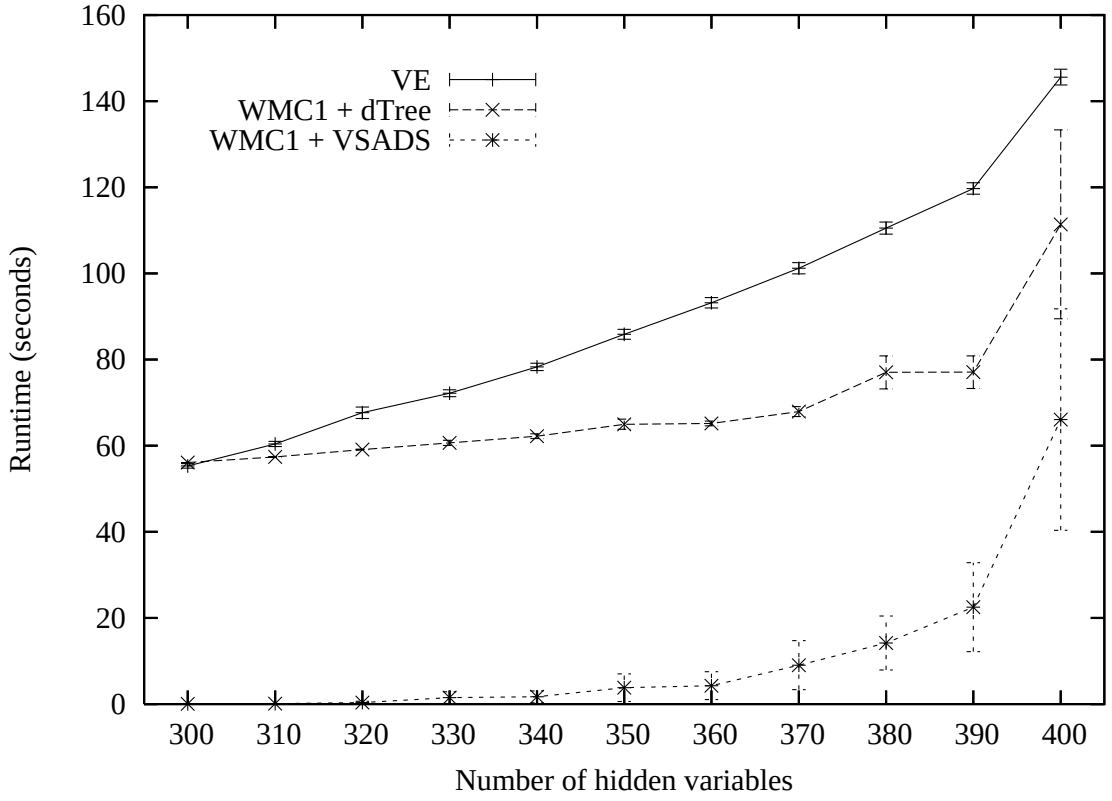


Figure 4.9: Random noisy-OR Bayesian networks with 3000 random variables. Effect of number of hidden variables on average time to answer probability of evidence queries, for the WMC1 encoding and the VSADS variable ordering heuristic, the WMC1 encoding and the dTree variable ordering heuristic, and Díez and Galán's [35] approach using variable elimination.

4.5.3 Experiment 3: Random Multi-Layer Networks

In my third set of experiments, I used randomly generated multi-layer Bayesian networks. To test randomly generated multi-layer networks, I constructed a set of acyclic Bayesian networks using the same method as Díez and Galán [35]: create n binary variables; randomly select m pairs of nodes and add arcs between them, where an arc is added from X_i to X_j if $i < j$; and assign a noisy-OR distribution or a noisy-MAX distribution to each node with parents.

Figure 4.9 shows the effect of the number of hidden variables on the average time to answer probability of evidence (PE) queries for random noisy-OR Bayesian

networks. Each data point is an average over 30 randomly generated instances, where each instance had 3000 nodes in total. The "spikes" in the runtime at the end of the curves indicates that the memory limits of our computer were reached (2GB RAM) and the process had started swapping between memory and the hard disk.

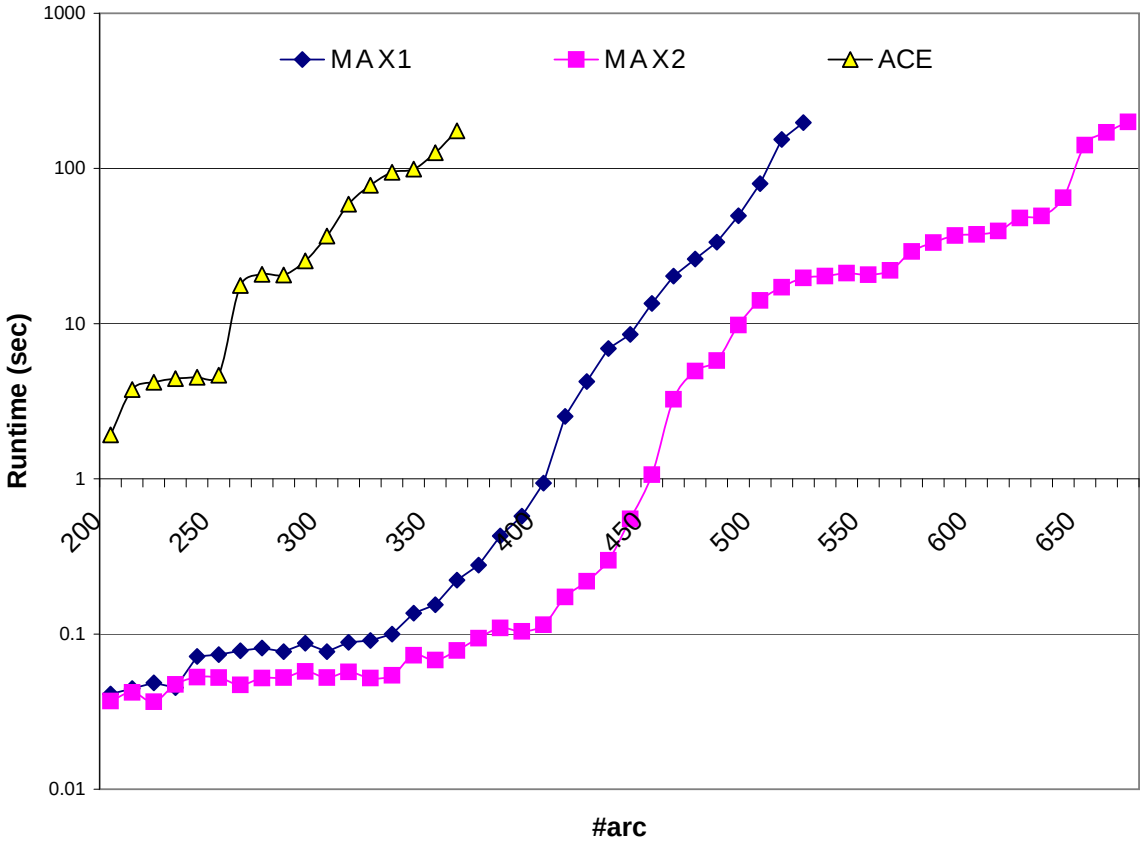


Figure 4.10: Random noisy-MAX Bayesian networks with 100 five-valued random variables. Effect of number of arcs on average time to answer probability of evidence queries, for the MAX1 encoding for noisy-MAX, the MAX2 encoding for noisy-MAX, and Chavira, Allen, and Darwiche's ACE2 [17].

The results from the two layer QMR-DT and the multiple layer random noisy-OR show that on average, the approach based on weighted model counting performed significantly better than variable elimination and significantly better than ACE2. All the approaches benefit from the large amount of evidence, but the weighted model counting approach explores the determinism more efficiently with dynamic decomposition and unit propagation (resolution). In comparison to variable elimination, the weighted model counting approach encodes the local dependencies among parameters and the evidence into clauses/constraints. The topological or structural features of the CNF, such as connectivity, can then be explored dynamically during DPLL's simplification process.

Heuristics based on conflict analysis have been successfully applied in modern SAT solvers. However, conflicts rarely occur in model counting problems with large

numbers of solutions, as arise in encodings of Bayesian networks. In situations where there are few conflicts, the VSADS heuristic essentially makes random decisions. But here, for large Bayesian networks with large amounts of evidence, VSADS work very well because the constraints that were generated from the evidence limits the number of solutions. DTree is also a good choice due to its divide-and-conquer nature. However, when we use dTree to decompose the CNF generated from QMR-DT, usually the first variable group contains more than 500 disease variables. As well, the overhead of preprocessing affects the overall efficiency of this approach.

Similarly, I performed an experiment with 100 five-valued random variables. Figure 4.10 shows the effect of the number of arcs on the average time to answer probability of evidence (PE) queries for random noisy-MAX Bayesian networks. Each data point is an average over 50 randomly generated instances. It can be seen that on these instances our CNF encoding MAX2 out performs our encoding MAX1 and significantly outperforms Chavira, Allen, and Darwiche's ACE2 [17]. It has been recognized that for noisy-MAX relations, the multiplicative factorization has significant advantages over the additive factorization [98, 35]. Hence, one would expect that the CNF encoding based on the multiplicative factorization (encoding MAX2) would perform better than the CNF encoding based on the additive factorization (encoding MAX1). The primary disadvantage of encoding MAX1 is that it must encode in the CNF summing over all configurations. As a result, MAX1 generates much larger CNFs than MAX2, including more variables and and more clauses. In encoding MAX2, the weight of a parameter variable represents the maximum effect of each cause and hence minimizes the add computations.

4.6 Summary

Large graphical models, such as QMR-DT, are often intractable for exact inference when there is a large amount of positive evidence. I presented space efficient CNF encodings for noisy-OR/MAX relations. I also explored alternative search ordering heuristics for the DPLL-based backtracking algorithm on these encodings. In my experiments, I showed that together my techniques extend the model counting approach for exact inference to networks that were previously intractable for the approach. As well, my techniques gave speedups of up to two orders of magnitude over the best previous approaches for Bayesian networks with noisy-OR/MAX relations and scaled up to networks with larger numbers of random variables.

In the next chapter, I continue to consider how to exploit structure to improve efficiency and scalability when the knowledge base is expressed as a Bayesian network and the inference engine is based on weighted model counting. In particular, I discuss techniques for taking advantage of previous computations when solving an incrementally updated Bayesian network.

Chapter 5

Exploiting Structure in Probabilistic Reasoning: Incremental Reasoning

Many real world Bayesian network applications need to update their networks incrementally as new data becomes available. For example, the capability of updating a Bayesian network is crucial for building adaptive systems. Many methods for refining a network have been proposed, both for improving the conditional probability parameters and for improving the structure of the network (see, e.g., [15, 73, 43] and references therein). However, little attention has been directed toward improving the efficiency of exact inference in incrementally updated Bayesian networks.

In this chapter, I propose techniques for improving the efficiency of exact inference in incrementally updated Bayesian networks by exploiting common structure. In particular, I propose and formalize the concept of *dynamic model counting* and present an algorithm for performing dynamic model counting, which I call DynaMC. The philosophy behind dynamic model counting is intuitive. Bayesian networks can be efficiently encoded into CNFs [23, 89]. When two CNF formulas share many common clauses, it is possible to use the knowledge learned while counting the solutions of one formula to simplify the solving and counting of the next formula (see Figure 5.1).

The updating of a Bayesian network, such as adding an edge or deleting an edge, is presented as a sequence of regular model counting problems. I extend dynamic decomposition and component caching (good learning) [4] to multiple runs on a series of changing instances with similar structure. In each run, the previous instance evolves through local changes, and the number of models of the problem can be re-counted quickly based on previous results after each modification.

The techniques I propose provide a general approach for reusing partial results generated from answering previous queries based on the same or a similar Bayesian network. My focus is to improve the efficiency of exact inference when the network structure or the parameters or the evidence is updated. I show that my approach can be used to significantly improve inference on multiple challenging Bayesian network instances and other problems encoded as dynamic model counting problems.

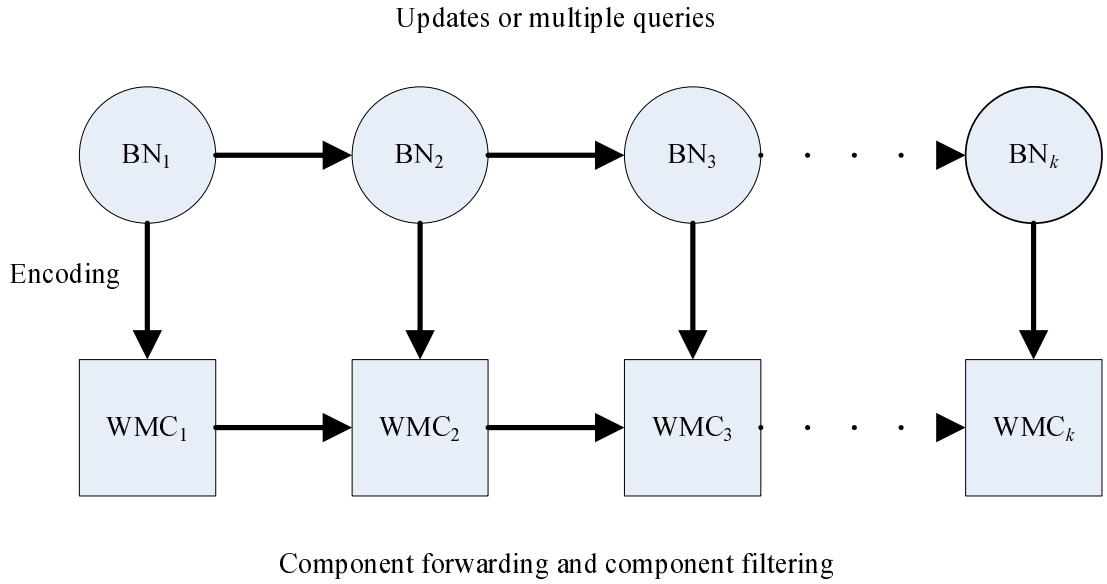


Figure 5.1: Schematic of dynamic model counting. A sequence of Bayesian networks $\mathrm{BN}_1, \ldots, \mathrm{BN}_k$ leads to a sequence of weighted model counting instances $\mathrm{WMC}_1, \ldots, \mathrm{WMC}_k$ that may share much common structure.

5.1 Applications of Dynamic Model Counting

In this section, I motivate the introduction of dynamic model counting. I do so by listing three types of important applications or tasks that can take advantage of the concept to potentially speed up the resolution of the tasks.

- Solving a sequence of probabilistic queries that are based on the same Bayesian network but different evidence.

  Algorithms based on junction trees have two steps: building a junction tree (compilation is done once and for all at the start before any queries are posed) and then answering queries. Usually once the junction tree has been built, one can answer a query about any variable with the following algorithm: instantiate evidence in the potentials of the density and then pass messages according to a message passing protocol. Although a junction tree can be thought of as storing the subjoints computed during elimination to avoid repeated computations, if given new evidence, this algorithm must repeat the instantiation and the message passing process.

  I demonstrated in the previous chapter that weighted model counting outperform junction tree based algorithms for large Bayesian networks with high density and width, such as BN2O. In many cases, the junction tree's compilation process is not able to complete because it requires to much space to build.

  For inference methods based on weighted model counting, the process of solving the encoded weighted model counting problem is usually viewed as a whole process; a new DPLL-based model counting has to be performed for different

queries. For example, to compute the posterior probability of a disease based on a set of symptoms in a QMR-DT network, we encode this QMR-DT network into a CNF and solve this CNF with DPLL, and we would need to run DPLL to solve a new CNF if certain symptoms of the patient change. The concept of dynamic model counting and the DynaMC algorithm that I present avoid recomputing such new queries from scratch.

- Parameter learning and sensitivity analysis in Bayesian networks.

  Determining accurate or reasonable probabilities for a Bayesian network can be difficult when the knowledge must be acquired from a human expert. The parameter learning problem is the problem of learning the conditional probabilities for a given network structure from (possibly incomplete) data and various algorithms and methods have been proposed (e.g., [96, 54, 74, 86]). For example, Russell et al. [86] show how gradient descent can be used to compute an approximate maximum likelihood estimate of the parameter vector of a Bayesian network from a collection of independent data cases. However, many Bayesian network inferences are required in the evaluation functions and gradient calculations. These inferences are the most expensive operations in all of the iterative learning methods; all other operations are trivial compared to the probabilistic inferences. Therefore, the efficiency of these iterative learning methods heavily depend on the inference algorithm they require.

  As well, my dynamic model counting approach can also be used to speed up sensitivity analysis in Bayesian networks [16, 66, 82]. In sensitivity analysis, the derivative of queries relative to parameters measures the sensitivity of a Bayesian network to parameter variations, and the derivative can be computed using queries with the same evidence but various parameters.

- Solving a sequence of probabilistic queries when the Bayesian network has updated parameters or structure.

  As noted earlier, many real world Bayesian network applications need to update their networks incrementally as new data becomes available and many methods for refining a network over time have been proposed (see, e.g., [15, 73, 43] and references therein). However, only limited attention has been directed toward improving the efficiency of exact inference in incrementally updated Bayesian networks.

  Inference methods based on junction trees can efficiently update the results for different queries for a fixed network structure. However, when the network structure changes, the junction tree usually must be reconstructed from scratch. Inference methods based on weighted model counting are designed for answering a single query of a fixed Bayesian network. Each new network needs to be recompiled, even though a change (be it evidence or structure) may only affect a small part of the network. With dynamic model counting, after each network is encoded into CNF, partial inference results can be maintained.

5.2 Related Work

In this section, I relate my work to previously proposed exact methods for inference in Bayesian networks and to previously proposed exact methods in satisfiability and model counting. My focus is on related work on probabilistic inference that attempts to accommodate changing queries, changing parameters, and changing structure in Bayesian networks and on related work on good learning and caching in satisfiability and model counting.

For probabilistic inference, junction tree based algorithms [75, 92] have a compilation step that transforms a Bayesian network into a secondary structure called a junction tree. A junction tree is built before any observations have been received. When the Bayesian network is reused, the cost of building the secondary structure can be amortized over multiple queries. However, when the network structure changes, the junction tree usually must be reconstructed from scratch. The variable elimination algorithm (VE) processes one query at a time (see, e.g., [72] and references therein). If a user wants the posterior probabilities of several variables, or for a sequence of observations, they need to run VE for each of the variables and observation sets. Evidence can simplify the inference of VE by eliminating the observed variables at the start of the algorithm, but each observation in the junction tree requires propagation of evidence. Because VE is query oriented, one can prune nodes that are irrelevant to specific queries [7, 46]. The junction tree structure is kept static at run time, and hence does not allow pruning of irrelevant nodes. Based on this space-time trade off, the junction tree is particularly suited to the case where observations arrive incrementally and where the cost of building the junction tree can be amortized over many cases. VE is suited for one-off queries, where there is a single query variable and all of the observations are given at once. Because VE permits pruning of irrelevant variables, VE can answer many of the possible queries of large real-world networks, which cannot be processed by junction tree algorithm. But currently VE does not have the ability to reuse the previous inference result for different queries in Bayesian networks (there has been work on reuse in VE for more generalized graphical models; see, e.g., [59, 93]).

Darwiche [21] proposes dynamic junction trees, but in his framework the network structure is fixed and the query changes over time. Flores, Gámez, and Olesen [36] propose the incremental compilation of a junction tree in the case where the structure of the network changes. Their idea is to identify the parts of the junction tree that are affected by the changes and only reconstruct those parts. However, junction tree algorithms do not perform well on Bayesian networks with high density and large width. Also, those improved methods cannot achieve the same result as VE and WMC for large scale real world Bayesian networks with large amounts of evidence.

Dynamic versions of CSPs [30, 91] and SAT [61, 67] have been proposed. However, the focus of previous work has been on the satisfiability problem—finding one solution or showing that no solution exists—and the learned knowledge is expressed in terms of conflict clauses (or nogoods). In nogood learning, a constraint that captures the reason for a failure is computed from every failed search path.

In contrast to learning from failure, component caching (or good learning) is learn-

ing from success. Both component caching and nogood learning collect information during search. However, it has been shown experimentally that good learning can be more effective than nogood learning for model counting [88]. In my work, to maintain the partial solutions of the problems having similar structure, I generalize component caching (good learning) to multiple runs.

Bayardo and Miranker [8] were the first to propose component caching and demonstrated that component caching improves the worst-case complexity of tree-structured problems. A good learning algorithm keeps track of domain values that were previously used to completely instantiate an entire subtree of the problem. By skipping over the variables in these subtrees, good learning avoids solving the same subproblems multiple times. Bayardo and Pehoushek [9] present the first implementation of good learning in the model counting system called Relsat. Their system identifies connected components dynamically during search, in contrast to previous work which as based on a static method [8]. Bacchus, Dalmao, and Pitassi [4] show that DPLL augmented with component caching can solve Bayesian inference and model counting with worst case time complexity that is theoretically competitive with the best methods and can be exponentially better. Sang et al. [88] show that good learning plus component caching dramatically improves the performance of their model counting solver Cachet.

Three of the most prominent model counting engines based on DPLL are Relsat [9], Cachet [88], and sharpSAT [100]. Bayardo and Pehoushek [9] describe two difficulties for an effective implementation of good learning: the space complexity of recording the defining set and the adaptability of goods with dynamic variable orderings. These problems are caused by the original design of good learning. Modern DPLL-based model counting engines have effectively solved these problems. For example, to decrease the space complexity, Cachet [88] removes all cached values of child components once their parent component's value has been computed. To solve the second problem, Cachet implements a dynamic component detection using a simple depth-first search after a component's decision literal is chosen. So the branching heuristics can be separated into two parts: the choice of decision variable and the choice of component. Both choosing decision literal and choosing branching component are based on the dynamic variable ordering. Cachet [88] also shows how to combine component caching and nogood learning in an implementation to preserve correctness. Thurley's sharpSAT [100] proposes several ideas that let components be stored more compactly, simplifies the search space, and improves the cache management. Different variants of caching schemes are also considered in [4].

In the CSP domain, Jégou and Terrioux [65] also proposed structural goods for solving CSPs. They showed that exploiting goods led to realize a "forward-jump" in the search tree, analogous to but in the reverse direction of backjumping. The main idea of their approach is that backtracking search will be guided for the choice of variables by the structure of the networks tree-decomposition.

5.3 Incremental Inference using Dynamic Model Counting

In this section, I propose and formalize the concept of *dynamic model counting* and present an algorithm for performing dynamic model counting, which I call DynaMC.

As has been extensively discussed in previous chapters, a Bayesian network can be described as a weighted model counting (WMC) problem [23, 89]. The conventional model counting problem (#SAT) asks, given a Boolean formula F in CNF, how many of its assignments are satisfying? There are natural polynomial-time reductions between the Bayesian inference problem and model counting problems [4]. In some situations, we need to recalculate certain parts of the preceding instance. For example, when WMC is used to perform Bayesian inference and we adjust the CPTs of a hidden variable, we get a sequence of model counting instances. The instances have the same clauses but some variables have updated weights. In this case, we need to recalculate the part of the original instance where the weights are updated. It is always possible to solve each WMC from scratch, using the usual algorithms. But inefficiency can become a problem in real world applications. Hence, my interest in defining dynamic model counting and developing dynamic model counting algorithms that take advantage of the incremental changes.

In what follows, I use the phrases static model counting problem and conventional model counting problem interchangeably. In both cases, I mean the problem where the propositional formula remains fixed or is static.

Definition 5.1 (Dynamic model counting problem)**.** *A* dynamic model counting problem *is a sequence M_0, M_1, M_2, ..., of conventional model counting (MC) problems, each one resulting from a change in the preceding one. As a result of such an incremental change, the number of solutions of the propositional formula may decrease (in which case it is considered a restriction) or increase (in which case it is a relaxation).*

Solving a dynamic model counting problem consists of sequentially computing the number of models of each of the conventional model counting problems (MCs) in the sequence. A naive approach to solving a dynamic model counting problem is to successively apply an existing static model counting algorithm to each MC. Compared with updating model counts from scratch, a more efficient solution is to maintain the model counts for the subproblems of the previous MCs so that one only re-computes the part affected by the insertion or deletion of constraints. The hope is that the number of assignments that satisfy a component in formula M_{i-1} can be used to solve a new formula M_i, which has the same component.

Definition 5.2 (Component)**.** *A* component *of a CNF formula F is a set of clauses Φ, the variables of which are disjoint from the variables in the remaining clauses $F - \Phi$.*

The partial inference result of model counting can be represented by a set of components. In current weighted model counting engines, disjoint components of the

formula, generated dynamically during a DPLL search, are cached so that they only have to be solved once [88].

5.3.1 Component Forwarding

In the DPLL-based model counting algorithm (Algorithm 5.1), a component is defined relative to the residual formula after all possible unit propagations have been performed (the reduced formula after unit propagation of the current partial assignment). The algorithm takes the original CNF formula as the first component and keeps creating new components and counting the models in each component until the number of satisfying assignments for each component has been stored in the cache.

When DPLL decomposes the initial MC instance into subproblems recursively and solves each problem, most of the variables on the implication chains are instantiated after making a relatively small number of decisions, and the internal structure often changes dramatically in different parts of the search tree. Thus, we expect this *dynamic* decomposition method can be used to effectively identify subproblems and, compared with static decomposition, dynamic decomposition should capture more internal structure. After DPLL branches (instantiates a variable) and has performed unit propagation, a separate depth-first search (DFS) is used to identify connected components [99]. The depth-first search is over the primal graph representation of the CNF formulate.

Definition 5.3 (Primal graph)**.** *In the primal graph for a CNF formula F, there is a vertex for each propositional variable and there is an edge between two vertices if the corresponding pair of variables appear together in some clause.*

Using DFS for dynamically detecting components (goods) has the advantage that we have more options for variable ordering heuristics inside each component.

Example 5.1. *As an example for component analysis, consider the propositional CNF formula,*

$$\begin{array}{rcl} F & = & (\neg A \vee \neg B \vee \neg L) \wedge (B \vee \neg L \vee M) \wedge (M \vee \neg N \vee O) \\ & & \wedge\ (B \vee \neg D \vee M) \wedge (D \vee \neg M \vee \neg N) \wedge (\neg C \vee E \vee F) \\ & & \wedge\ (E \vee F \vee G) \wedge (E \vee G \vee \neg D) \wedge (D \vee G) \\ & & \wedge\ (J \vee \neg G \vee H) \wedge (\neg H \vee I \vee \neg K) \wedge (G \vee D \vee \neg H) \\ & & \wedge\ (D \vee H \vee K) \wedge (\neg D \vee H). \end{array}$$

Figure 5.2 gives the primal constraint graph of the formula F. Figure 5.3 shows the search tree of regular DPLL on F up to depth 3. A complete decomposition tree of F is shown in Figure 5.4. Figure 5.5 shows the decomposition trees at leaf node F_3 in Figure 5.3. The decompositions at leaf nodes are different due to the partial assignments and related unit propagations. In those diagrams, it can be seen that F is decomposed dynamically along each branch of the search tree.

As presented in Algorithm 5.1, the components are organized by *component_stack*. At each decision-making point, the new components are detected in the top component of the branchable component stack. If the new components are found in the

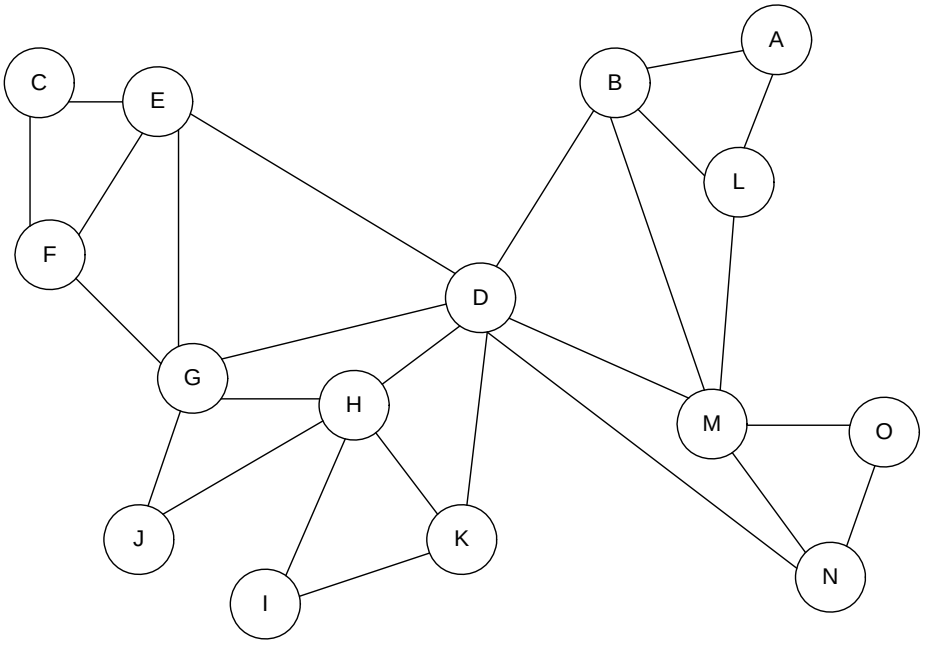


Figure 5.2: The primal graph of the formula F in Example 5.1.

cache, then we can percolate up and update the total counting number (see Figure 5.4). Otherwise, we keep analyzing the new component until its model count is calculated and we save the count in the cache and component database for our next problem in the sequence. Here, to make it easier to present, we assume that the updated problem is solved with the same variable ordering as the original one. My implementation, however, is based on a dynamic variable ordering heuristic, which has been proven to be very important for DPLL.

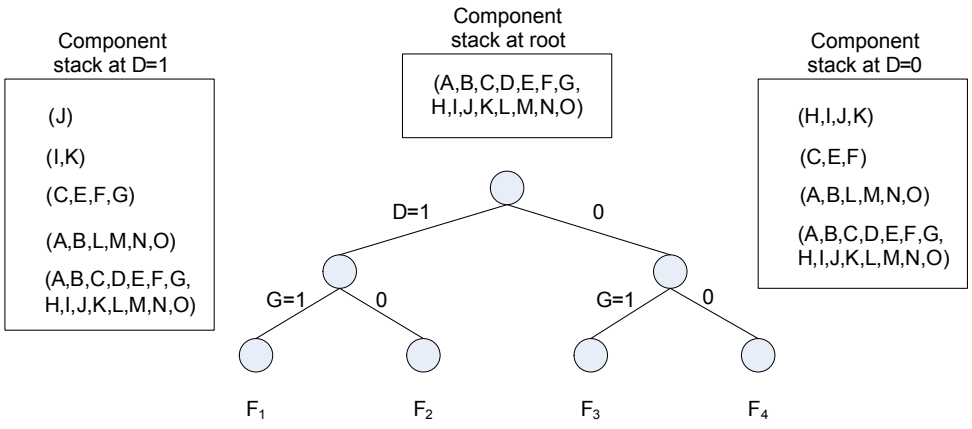


Figure 5.3: The search tree and component stacks when branching on the variables D and G of the formula F in Example 5.1.

Example 5.2. *Consider once again the CNF formula F given in Example 5.1 and*

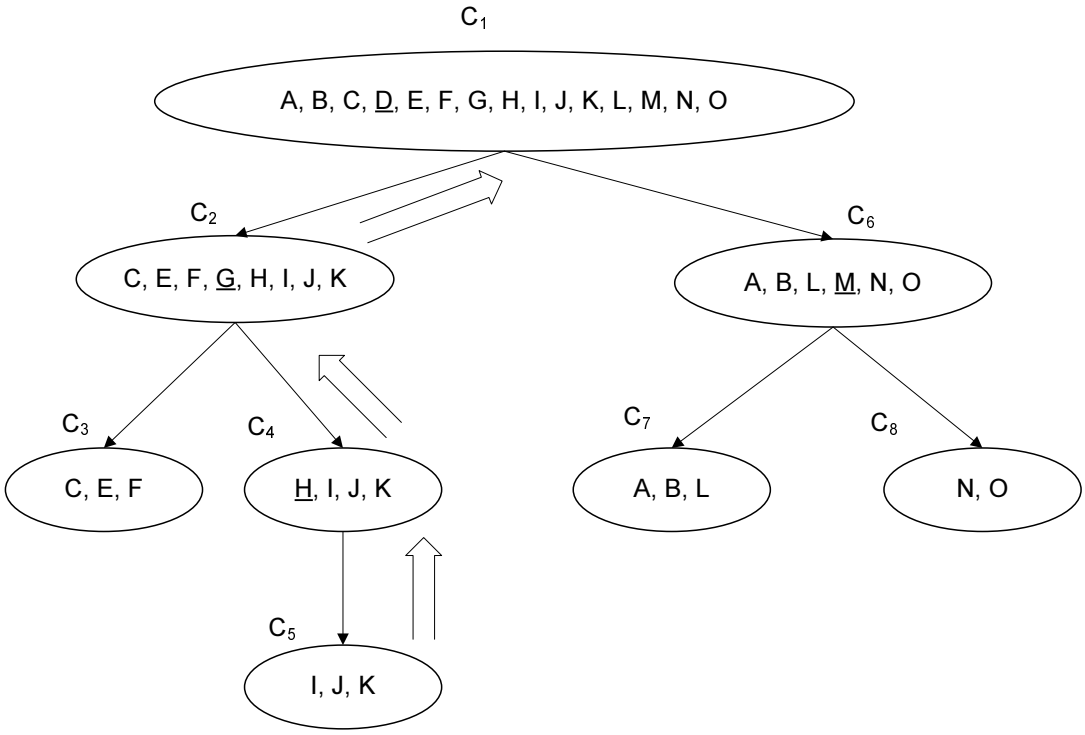


Figure 5.4: A possible decomposition tree of the formula F in Example 5.1.

the partial search tree for F shown in Figure 5.3. The component stack starts with a single component that consists of all variables in F. Suppose that D is chosen to be the first variable to be instantiated in this component.

Assigning $D = 1$ and forming the residual formula $F|_{\{D\}}$ gives,

$$\begin{array}{lll} (\neg A \vee \neg B \vee \neg L) & \wedge\ (B \vee \neg L \vee M) & \wedge\ (M \vee \neg N \vee O) \\ \wedge\ (B \vee M) & & \wedge\ (\neg C \vee E \vee F) \\ \wedge\ (E \vee F \vee G) & \wedge\ (E \vee G) & \\ \wedge\ (J \vee \neg G \vee H) & \wedge\ (\neg H \vee I \vee \neg K) & \\ & \wedge\ (H) & \end{array}$$

The literal H is unit and forced to be true. Performing unit propagation gives,

$$\begin{array}{lll} (\neg A \vee \neg B \vee \neg L) & \wedge\ (B \vee \neg L \vee M) & \wedge\ (M \vee \neg N \vee O) \\ \wedge\ (B \vee M) & & \wedge\ (\neg C \vee E \vee F) \\ \wedge\ (E \vee F \vee G) & \wedge\ (E \vee G) & \\ & \wedge\ (I \vee \neg K) & \end{array}$$

All disconnected components are identified using DFS and are pushed onto the component stack. In the example, the following components are identified: (A, B, L, M, N, O), (C, E, F, G), (I, K), and (J). Note that J does not appear in the simplified formula, but is an isolated vertex in the primal graph and is thus a separate, unconstrained component. The new components are pushed onto the component stack (see Figure

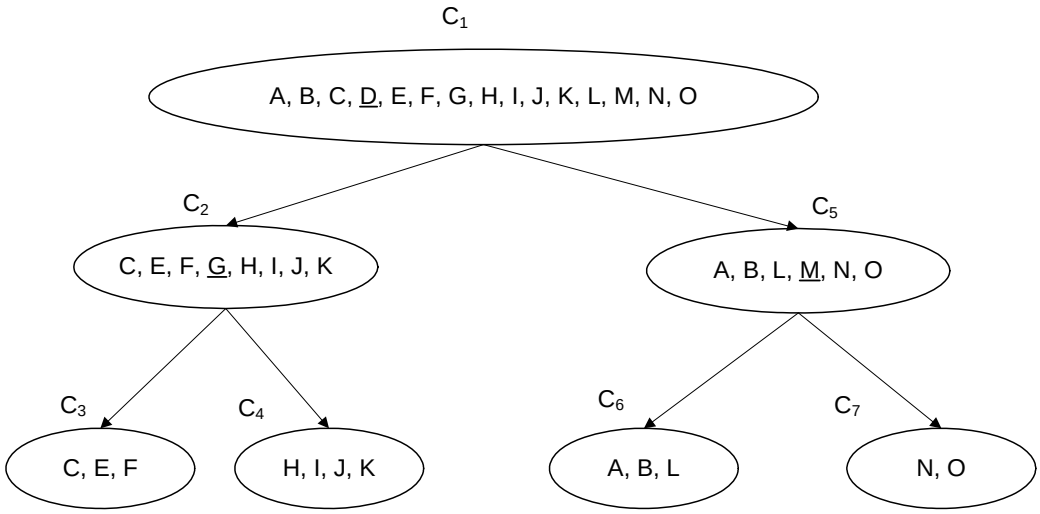


Figure 5.5: The decomposition tree of F at leaf node 3 of the search tree in Figure 5.3.

5.3). (In practice in the implementation, because the components (I, K) and (J) both have two variables or less, these two components would not be kept on the stack. Instead, their weights would be calculated right away.) If no equivalent component is found in the component hash table, (A, B, L, M, N, O) and (C, E, F, G) would stay on the component stack until their weights were fully computed from their children. In our example, the variable G would be branched on next, further components would be identified and looked up in the cache or solved, and so on.

Assigning $D = 0$ and forming the residual formula $F|_{\{\neg D\}}$ gives,

$$\begin{array}{lll} (\neg A \vee \neg B \vee \neg L) & \wedge (B \vee \neg L \vee M) & \wedge (M \vee \neg N \vee O) \\ & \wedge (\neg M \vee \neg N) & \wedge (\neg C \vee E \vee F) \\ \wedge (E \vee F \vee G) & & \wedge (G) \\ \wedge (J \vee \neg G \vee H) & \wedge (\neg H \vee I \vee \neg K) & \wedge (G \vee \neg H) \\ \wedge (H \vee K) & & \end{array}$$

The literal G is unit and forced to be true. Performing unit propagation gives,

$$\begin{array}{lll} (\neg A \vee \neg B \vee \neg L) & \wedge (B \vee \neg L \vee M) & \wedge (M \vee \neg N \vee O) \\ & \wedge (\neg M \vee \neg N) & \wedge (\neg C \vee E \vee F) \\ & & \\ \wedge (J \vee H) & \wedge (\neg H \vee I \vee \neg K) & \\ \wedge (H \vee K) & & \end{array}$$

The following components are identified using DFS: (A, B, L, M, N, O), (C, E, F), and (H, I, J, K). Only (C, E, F) and (H, I, J, K) will be left on the component stack. Since the weight of the component (A, B, L, M, N, O) has already been calculated and stored in the hashtable, it is not on the stack.

In each instance, new components are detected dynamically using a simple depth-first search on components. Once the decision literal is chosen and unit propagation

Algorithm 5.1: Weighted Model Count (WMC)

```
input  : CNF formula F
output: Return the weighted model count of formula F

Push F onto component_stack;
while component_stack is not empty do
    Select a variable in top(component_stack) and begin branching;
    BCP();
    if Find conflict then
        Backtrack;
    Detect components;
    for each new component C detected do
        if C has only 2 literals then
            pass weight(C) to the parent Component;
        else if in_cache(C) then
            pass weight(C) to the parent Component;
        else
            Push(C);
return weight(F);
```

is finished, the original component and newly generated component are pushed into the component stack. And the solving process stops when the component stack is empty.

In Algorithm 5.2, the components of each instance are indexed by a hash function and saved in a component database. Following Sang et al. [89], I use a hash function based on a short, fixed-length binary sequence known as the cyclic redundancy checksum (CRC) code. The hash function takes as an argument a component. The CRC code is generated based on variables in each clause inside the component. This CRC code is used as the index of a component library. When a new component is detected the calculation is repeated; if the new CRC does not match any index calculated earlier, then the storage saves the new component at a new indexed place. Otherwise, if two components are exactly matched, the weight of old component is returned as the weight of new component. The components, which are still valid for the updates, are imported to a new hash table before processing the next instance. This table is checked when a new component is created to see if its value is already in the hash table.

Algorithm 5.2: Dynamic Weighted Model Counting (DynaMC)

input : Database of components from solving CNF formulas $F_0 \dots F_{i-1}$; and a CNF formula F_i

output: Return the weighted model count of formula F_i

Import stable components from component database;
WMC(F_i);
Save new components in database;

5.3.2 Component Filtering

To bound the cache space between instances that share an updated clause set, the validity of each cached component is tracked using two properties: *clean* and *stable*.

Definition 5.4 (clean component)**.** *A* clean component *is a component that does not include any variable from the set of clauses that have changed. A component is said to be* unclean *if it is not clean.*

Definition 5.5 (stable component)**.** *A* stable component *is a component that keeps the same clauses and has the same number of solutions after the update operations. A component C is* stable *if all of the following conditions are satisfied:*

1. *C is a clean component;*
2. *C is not on a path from the root to any unclean component.*

A component is said to be unstable *if it is not stable.*

Example 5.3. *Suppose that, in Figure 5.5, component C_7 is not a clean component and all of the other components are clean components. Applying the above definition, components C_7, C_5 and C_1 are unstable components and the rest of the components are stable components.*

After updating the previous problem, all the unstable components in the cache are deleted. The remaining stable components are imported into the component database of the new instance.

We record the model count of every component and their parent-child relation for each run. If F is updated—for example, a clause is removed from the original component C_6—instead of recalculating every component, we only need to recalculate those components that belong to $ancestors(C_6)$. So, except for C_6, we need to recalculate C_5 and C_1 (see Figure 5.5).

5.3.3 Updates in CNF

Once the model count of the original problem has been determined, a relaxation will create more models, while a restriction will decrease the model count. In the following,

Table 5.1: Updates to a CNF formula that do not add or delete variables.

Update is a restriction	Remove literals from existing clauses Add clauses of existing variables
Update is a relaxation	Remove clauses Add literals of existing variables to existing clauses

I discuss the possible updates of a CNF formula. There are several basic operations that could be used to update a CNF formula to get the next model counting problem in the sequence (see Table 5.1). Each of the possible modifications of a Bayesian network—add a node, delete a node, add an edge, delete an edge, reverse an edge, and change a CPT entry—can be expressed as a combination of these basic update operations on the CNF encoding of the network.

Remove literals from existing clauses.

One basic modification of a CNF is to remove a literal from a clause. Removing literals from clauses decreases the number of models of the original formula. If, for example, all the variables of a clause are in a component, the removal of a literal in that clause will lead to changes in the primal graph of this component. Since the structure of this component's children in the decomposition tree may also be changed, we not only need to recount the number of models of this component, but the number of models of those children components should be updated as well. Considering the disconnectivity among its child components, only those components that have connection with the removed variable need to be recounted.

For purposes of explanation, assume in the next two examples that we follow the same variable ordering in each run and we remove only one literal from an existing clause.

Example 5.4. *Consider the example in Figure 5.5. Both C_2 and C_5 are identified after variable D is instantiated. If the literal $\neg B$ is removed from the clause $(\neg A \vee \neg B \vee \neg L)$ in C_5, the structure of C_2 remains the same. If there is any structural change in C_2, it must be caused either by the existing unit propagation chain connecting the updated clause and C_2 or by the instantiation of variable D. Since D is not in the updated clause and C_2 and C_5 are disconnected in the search tree, C_2 is "stable".*

Example 5.5. *For another example, if we delete literal D from the clause $(B \vee \neg D \vee M)$. After $\neg D$ becomes instantiated, C_2 (see Figure 5.5) is a clean component. If C_2 or C_3 are unstable due to the removal of $\neg D$, there must be at least one path, which does not include D, between C_2 (or C_3) and the other literals (B, M) in the updated clause. However, if such a path exists, C_2 and C_3 will not be disconnected from C_5 after $\neg D$ becomes instantiated.*

Remove clauses

Another basic modification of a CNF is to remove a clause. Removing a clause increases the number of models of the original formula. This process only affects those components that include this clause and their children components. For example, the component C_5, C_6 and C_7 are stable components if clause $(D \vee G)$ is removed.

Add clauses of existing variables

Another basic modification of a CNF is to add clauses that consist only of existing variables. New clauses of existing variables add new constraints to the original formula and hence decreases the number of models of the original formula. When separated components are connected by the new clauses, only those child components that have variables in the new clauses become unstable. For example, in Figure 5.5, if a new clause $(K \vee M)$ is added into F, the components C_3, C_6 and C_7 are still stable, while C_4's model count changes because of the new constraint.

Add literals of existing variables to existing clauses.

Another basic modification of a CNF is to add a literal of an existing variable to an existing clause. Adding a literal of an existing variable to a clause increases the number of models of the original formula. Since the new literals may connect components in the different branches of the decomposition tree, the related parts of the decomposition tree need to be updated. We get a similar conclusion as adding a new clause. Here, the clean component does not include any of the original literals of the updated clause and the literals that share the same clause with the new literal. In the search tree of an updated CNF, if either the new literal or any related literal has been instantiated in the search tree, all the existing clean components are stable.

Problem expansion: Adding literals of new variables.

Adding new variables creates more satisfiable solutions. As discussed above, all clean components are stable.

5.3.4 Encoding Bayesian Networks Updates

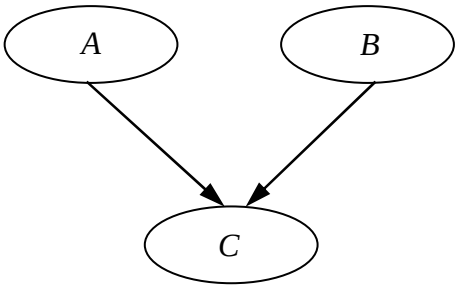
In this section, I show how the possible modifications of a Bayesian network can be expressed by a combination of the basic update operations on the CNF formula discussed in the previous section.

Evidence updates.

The possible evidence updates are as follows: (i) change a positive evidence to a negative evidence; (ii) change a negative evidence to a positive evidence; (iii) add a new evidence; and (iv) remove an evidence.

$P(A = a_1)$	$P(A = a_2)$
0.1	0.9

$P(B = b_1)$	$P(B = b_2)$
0.2	0.8

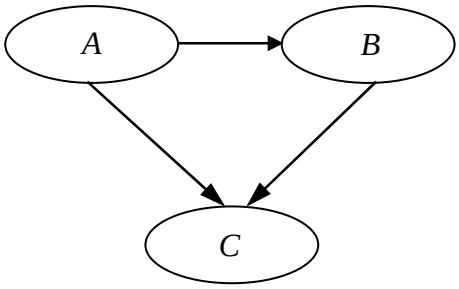


A	B	$P(C = c_1 \mid A, B)$	$P(C = c_2 \mid A, B)$
a_1	b_1	0.9	0.1
a_1	b_2	0.8	0.2
a_2	b_1	0.3	0.7
a_2	b_2	0.1	0.9

Figure 5.6: A Bayesian network over random variables A, B, and C, where $dom(A) = \{a_1, a_2\}$, $dom(B) = \{b_1, b_2\}$, and $dom(C) = \{c_1, c_2\}$.

$P(A = a_1)$
0.1

$P(A = a_2)$
0.9



A	$P(B = b_1 \mid A)$
a_1	0.2
a_2	0.4

A	$P(B = b_2 \mid A)$
a_1	0.8
a_2	0.6

A	B	$P(C = c_1 \mid A, B)$	$P(C = c_2 \mid A, B)$
a_1	b_1	0.9	0.1
a_1	b_2	0.8	0.2
a_2	b_1	0.3	0.7
a_2	b_2	0.1	0.9

Figure 5.7: The Bayesian network shown in Figure 5.6 updated with an arc from A to B and an expanded conditional probability table at node B.

We encode each evidence as a unit clause. For example, one possible general CNF encoding of the original Bayesian network in Figure 5.6 is the following.

$$
\begin{array}{lll}
A & : & (I_{a_1} \vee I_{a_2}) \wedge (\neg I_{a_1} \vee \neg I_{a_2}) \\
B & : & (I_{b_1} \vee I_{b_2}) \wedge (\neg I_{b_1} \vee \neg I_{b_2}) \\
C & : & (I_{c_1} \vee I_{c_2}) \wedge (\neg I_{c_1} \vee \neg I_{c_2}) \\
A & : & P_{a_1} \Rightarrow I_{a_1} \qquad\qquad \neg P_{a_1} \Rightarrow I_{a_2} \\
B & : & P_{b_1} \Rightarrow I_{b_1} \qquad\qquad \neg P_{b_1} \Rightarrow I_{b_2} \\
C & : & I_{a_1} \wedge I_{b_1} \wedge P_{c_1|a_1,b_1} \Rightarrow I_{c_1} \qquad I_{a_1} \wedge I_{b_1} \wedge \neg P_{c_1|a_1,b_1} \Rightarrow I_{c_2} \\
 & & I_{a_1} \wedge I_{b_2} \wedge P_{c_1|a_1,b_2} \Rightarrow I_{c_1} \qquad I_{a_1} \wedge I_{b_2} \wedge \neg P_{c_1|a_1,b_2} \Rightarrow I_{c_2} \\
 & & I_{a_2} \wedge I_{b_1} \wedge P_{c_1|a_2,b_1} \Rightarrow I_{c_1} \qquad I_{a_2} \wedge I_{b_1} \wedge \neg P_{c_1|a_2,b_1} \Rightarrow I_{c_2} \\
 & & I_{a_2} \wedge I_{b_2} \wedge P_{c_1|a_2,b_2} \Rightarrow I_{c_1} \qquad I_{a_2} \wedge I_{b_2} \wedge \neg P_{c_1|a_2,b_2} \Rightarrow I_{c_2}.
\end{array}
$$

If we add a new evidence, for example $I_C = true$ (or $I_C = false$), a new unit clause I_C (or $\neg I_C$) can be added to F.

For Noisy-OR/MAX relations, when evidence is updated, one often needs to re-encode the network by adding/removing auxiliary variables and updating clauses. For example, with the additive encoded noisy-OR (Figure 4.2), if *Nausea* goes from positive to negative, this update of evidence can be encoded as replacing a set of clauses Equation (4.10) with Equation (4.9). In this additive encoding, fully connected symptom variables are all disconnected and independent.

Parameter updates.

Many Bayesian network applications need to learn parameters from newly available data, such as weblogs and sensors. It is a practical need that the inference results can be reused for answering the same query using new parameters. For a WMC encoded from a Bayesian network and a query, we only need to update changed parameter variables.

Structure updates.

In Bayesian networks, we can easily add a new causal relation by connecting two nodes with a directed arc and expanding the existing CPT. For example, in Figure 5.7, a new arc is added in Figure 5.6 to present that node A effects node B and the CPT at node B is changed to specify the conditional probability of the new cause.

We add the following clauses in order to specify the new CPT at node B in Figure 5.7. Here, the parameter variables P_{b0} and P_{b1} are also created for the conditional probability at node B.

$$
\begin{array}{r}
(I_A \vee I_B \vee \neg P_{b0}) \\
\wedge (I_A \vee \neg I_B \vee P_{b0}) \\
\wedge (\neg I_A \vee I_B \vee \neg P_{b1}) \\
\wedge (\neg I_A \vee \neg I_B \vee P_{b1})
\end{array}
$$

$$
\begin{array}{lcll}
\text{old } B & : & P_{b_1} \Rightarrow I_{b_1} & \neg P_{b_1} \Rightarrow I_{b_2} \\
\text{new } B & : & I_{a_1} \wedge P_{b_1|a_1} \Rightarrow I_{b_1} & I_{a_1} \wedge \neg P_{b_1|a_1} \Rightarrow I_{b_2} \\
 & & I_{a_2} \wedge P_{b_1|a_2} \Rightarrow I_{b_1} & I_{a_2} \wedge \neg P_{b_1|a_2} \Rightarrow I_{b_2}.
\end{array}
$$

When a new independent cause is added into a noisy-OR/MAX relation, the change of the CNF is based on different encodings. For example, in Figure 4.2, if a new disease $SwineFlu$ also causes $Headache$, we update the additive encoding as

$$(I_C \wedge P_{0.6}) \vee (I_F \wedge P_{0.5}) \vee (I_M \wedge P_{0.4}) \vee (I_S \wedge P_S) \Leftrightarrow I_H \tag{5.1}$$

If $I_H = 0$, none of the old component need to be updated since I_S and P_S form a new independent component; however, if $I_H = 1$, all the components related with this noisy-OR relation need to be updated.

5.4 Experimental Evaluation

In this section, I empirically evaluate the effectiveness of my proposed dynamic weighted model counting (DynaMC) algorithm.

My DynaMC program is built on Cachet, which is currently the fastest model counting solver. The difference between DynaMC and Cachet is that new components learned in each instance are saved in a database and the valid portion of the database is imported into the new instance before the beginning of a new run. Also, I implemented a DynaMC compiler based on JavaBayes. A consistent variable numbering system must be maintained among compiled CNFs. In this way, adding or deleting variables or links in the original Bayesian network only generates local changes in each compiled CNF. The experiments were run on a Linux desktop with a 2.6GHz P4 processor and 1GB of memory, except for Experiment 2 that used a 3.0GHz P4 processor with 3GB of memory.

I compared DynaMC against Cachet. In my experiments, both programs computed the weight of the formula. I tested my approach over several kinds of networks: real networks taken from the repository of the Hebrew University[1] and QMR-DT, deterministic QMR networks, and artificially generated grid networks[2].

The experiments overall show that DynaMC method works well with scaling (Experiment 1), Bayesian network structure changes (Experiments 2 & 3), and Bayesian network parameter changes (Experiment 4). In Experiment 1, I made changes to the encoded weighted model counting problem. In Experiments 2–4, I made changes to the actual Bayesian networks. In all my experiments, the reported runtimes assume that the component cache is memory-resident.

[1] http://compbio.cs.huji.ac.il/Repository/networks.html
[2] http://www.cs.washington.edu/homes/kautz/Cachet/

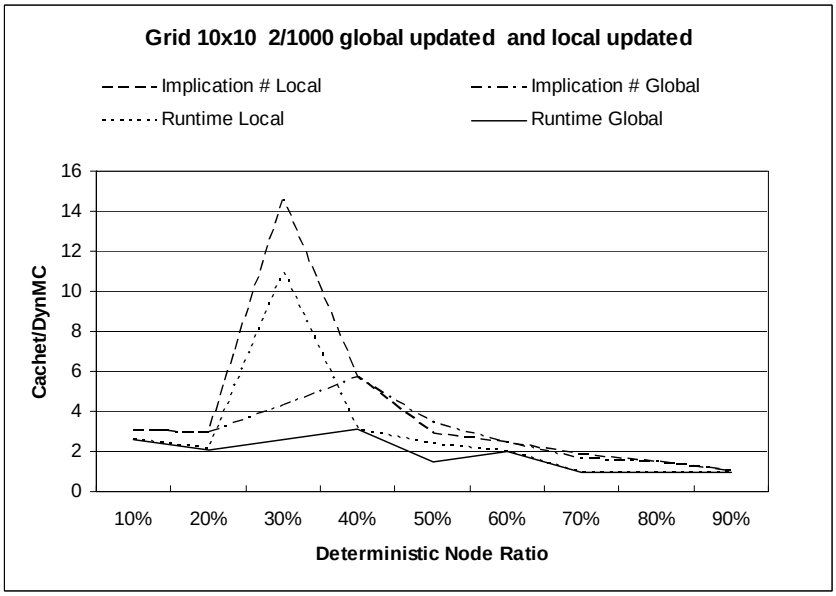


Figure 5.8: The ratio (Cachet/DynaMC) of runtime and implication number on 10 $\times$ 10 grid problems. 10 instances are tested on different percentage of deterministic nodes. We globally and locally delete 2/1000 literals on each instance.

5.4.1 Experiment 1: Grid Networks

In my first set of experiments, I used compiled grid problems to study the effect of the number of changed literals and their relative positions (whether the changes occurred globally or locally). Following Sang et al. [89], a grid Bayesian network is a network with $N \times N$ random variables $X_{i,j}$, $1 \leq i, j \leq N$. Each node $X_{i,j}$ has parents $X_{i-1,j}$ and $X_{i,j-1}$, whenever those indices are greater than zero. The fraction of the nodes that are assigned deterministic conditional probability tables (CPTs) is a parameter, the deterministic ratio. The CPTs for such nodes are randomly filled in with 0 or 1; in the remaining nodes, the CPTs are randomly filled with values chosen uniformly in the interval (0, 1) [89]. A *compiled* grid problem is a CNF that is generated from a grid Bayesian network using Sang's encoding [89]. My approach to generating global modifications was to randomly select a clause from the original CNF and randomly add/delete a literal. To generate local modifications, I created a series of CNFs, $M_0, M_1, \ldots, M_n$. In M_i $(0 < i \leq n)$, only the clauses that share at least one variable with the modified clauses in M_{i-1} are selected. In order to compare DynaMC and traditional MC, I collected the ratio of median runtime and the ratio of number of unit propagations (implications).

Figure 5.8 and 5.9 show the results obtained for 0.2% and 1% literal deletion of a 10 $\times$ 10 grid network compiled into a dynamic model counting problem problem. As discussed earlier, the larger the portion that is shared between successive problems, the more components we can save for achieving speed-ups. Now, as can be seen, the distribution of changes also plays an important role. DynaMC works much better

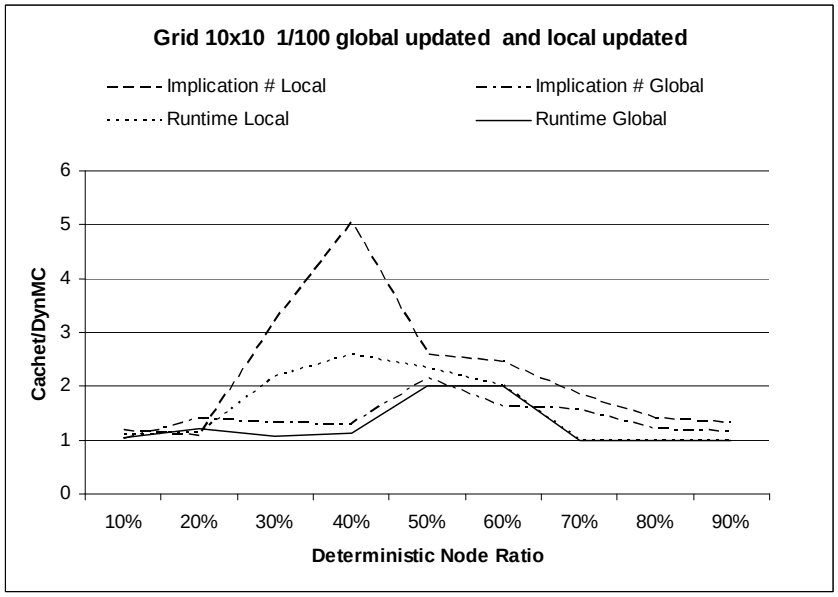


Figure 5.9: The ratio (Cachet/DynaMC) of runtime and implication number on 10 $\times$ 10 grid problems. 10 instances are tested on different percentage of deterministic nodes. We globally and locally delete 1/100 literals on each instance.

when local changes are performed. Intuitively, we need to recompute more model counts for independent subproblems if modifications are distributed evenly in more components. At the low end of the deterministic ratio, the constraint graphs of compiled CNFs have very high width and density. So there is a low possibility of finding disconnected components while executing DPLL. The problems at the high end of deterministic ratio are relatively easy, so they can be solved without checking the component database.

In Figure 5.10, I mixed both insert literal and delete literal operations: the ten modifications included five insertions and five deletions. I also fixed the deterministic ratio as 75% and tested different problem sizes from 10 $\times$ 10 to 44 $\times$ 44. The experimental results show that DynaMC can be solved more efficiently than a set of independent regular MC problems.

5.4.2 Experiment 2: Grid Networks

In my second set of experiments, I again used grid networks to study the effect of structure changes in Bayesian networks. The grid networks in this experiment have 90% deterministic ratio. My approach for generating modifications in Bayesian networks is based on the assumption that the updates usually would be concentrated on a limited region of the model. I use the following procedure to create $2m$ modifications in a sequence Seq. The procedure is similar to the procedure used by Flores, Gámez, and Olesen [37] in their experiments on incremental junction tree compilation.

1. $DeleteSeq = \{\}$, $AddSeq = \{\}$

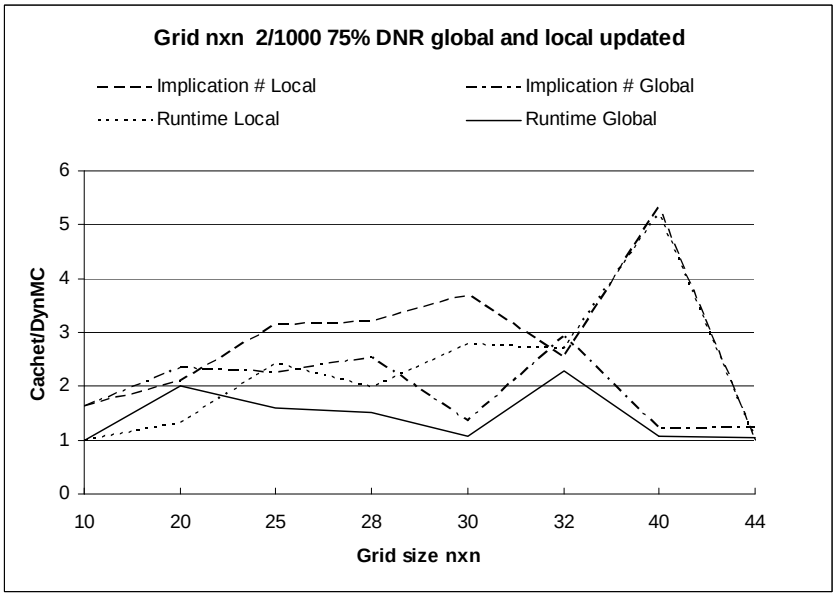


Figure 5.10: The ratio (Cachet/DynaMC) of runtime and implication number on $N \times N$ grid problems. 10 instances are tested on each problem size. We globally and locally insert and delete 2/1000 literals on each instance.

2. Randomly select a node V_i from the network B. Then remove all the edges $e_i = \{E \mid V_{pi} \rightarrow V_i\}$ between V_i and its parents V_{pi} from B and add modification $delete(e_i)$ onto the end of DeleteSeq

3. Insert modification $add(e_i)$ into the front of AddSeq

4. All the remaining nodes linked to V_i are included in a set N_i. The next node V_{i+1} is randomly selected from N_i.

5. Return to Step 2, until we have run m loops

6. $Seq = DeleteSeq$ concatenate $AddSeq$

In Table 5.2, I tested 10 modifications for each problem size. Each modification includes one or two edges in the grid network, depending on the location of the randomly selected node. The total runtime is the sum of the runtime for solving the 10 modified networks. DynaMC is two times faster than Cachet in the best case.

5.4.3 Experiment 3: Solving Noisy-OR/MAX Incrementally

In my third set of experiments, I used randomly generated QMR-DT like networks to study the effect of structure changes in Bayesian networks.

I also perform Bayesian inference incrementally with WMC1+VSADS. I randomly generated QMR-DT like networks. Each random network contains 500 diseases and 500 symptoms. Each symptom has 6 possible diseases uniformly distributed in the

Table 5.2: Total runtime of DynaMC and Cachet and percentage reduction of DynaMC over Cachet, for 10 modifications of $N \times N$ grid networks.

Grid	Total Runtime (sec.) DynaMC	Cachet	Improvement %
10×10	34	31	-9.7%
12×12	103	170	39.4%
14×14	182	228	20.2%
16×16	229	368	37.8%
18×18	292	597	51.1%
20×20	299	370	19.2%
21×21	488	616	20.8%
22×22	596	710	16.1%
24×24	1558	2067	24.6%

disease set. The prior and conditional probabilities are also uniformly generated from the interval (0, 1).

I tested a sequence of evidence q_i ($30 \leq i \leq 48$) on 10 random generated QMR-DT-like networks. Each q_{i+1} has one more positive symptom that is randomly picked from the negative symptoms of q_i. I directly encoded every evidence into CNFs and solved each of the 10 evidence sequences independently and incrementally. Figure 5.11 shows the average runtime of the two methods. Incremental inference empirically subsumes independent inference, and is at times an order of magnitude faster.

DQMR is a simplified representation of the QMR-DT [89]. Each DQMR problem is a two-level multiply connected Bayesian network in which the top layer consists of diseases and the bottom layer consists of symptoms. If a disease may yield a symptom, there is an edge from the disease to the symptom. I test networks with 50 to 100 diseases and symptoms. The edges of the bipartite graph are randomly chosen. Each symptom is caused by three randomly chosen diseases. The problem consists of computing the weight of the encoded formula given a set of consistent observations of symptoms. In each instance of 50+50 networks, 10% disease nodes are randomly selected as observed. The observed nodes are 50% in 100+100 networks. Table 5.3 shows that the modifications on real Bayesian networks can be translated into dynamic model counting problems and solved more efficiently using DynaMC.

5.4.4 Experiment 4: Other Bayesian Networks

In my fourth set of experiments, I studied the effect of using DynaMC when network determinism changed (see Table 5.4). I generated a sequence of 10 instances $M_0, M_1, \ldots, M_{10}$ for each network. M_k ($0 < k \leq 10$) is generated by randomly selecting a node and setting one entry of its CPT to the value 1. The component library imported into M_k is generated in M_{k-1}.

Due to the memory resource required by the large component library I skipped

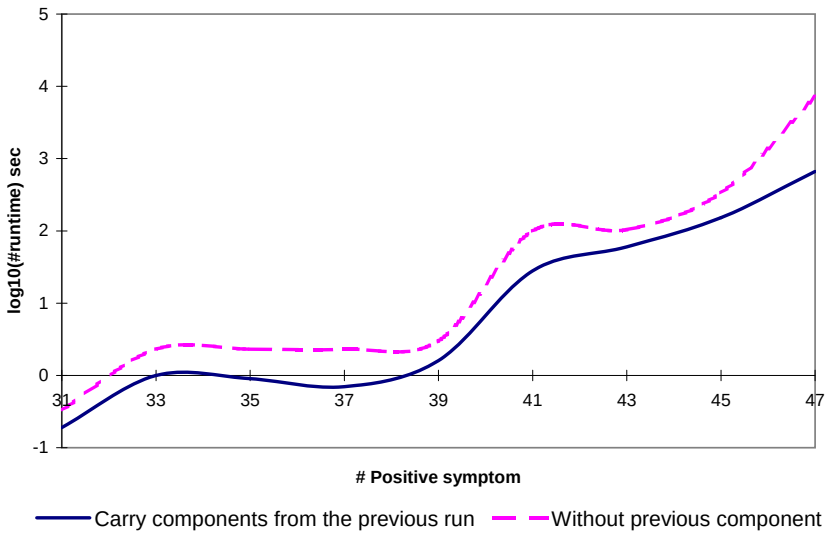


Figure 5.11: Log of average runtime (seconds) for DynaMC (carry components from the previous run) and Cachet (without previous components) on a sequence of ten QMR-DT like networks with 500 disease and 500 symptoms.

networks that could not be solved by both Cachet and DynaMC. It has been noted that for Bayesian networks that have variables with large cardinalities, large CPTs, or a small amount of determinism, the general encoding method does not work well [18]. Those encoded CNFs are simply too large to be quickly decomposed and quickly use up all available memory. In the same paper, Chavira and Darwiche propose a more efficient encoding.

For a few test instances, my method was slower than Cachet due to the overhead of querying the cache. In those "failed" cases, I found that the components imported from previous runs were extremely small. Usually, the average size of imported components in those cases was less than 10 literals. In many successful instances, the average number of literals of imported components was more than 100 literals. When

Table 5.3: Total runtime and number of implications of 10 DQMR instances for each network, where (e) indicates that an edge from a disease to a symptom was randomly selected and removed from the original DQMR network, and (n) indicates removing a randomly selected symptom node.

	Total Runtime (sec.)		Implication#	
DQMR	DynaMC	Cachet	DynaMC	Cachet
50+50 (e)	194	334	4.3×10^6	8.0×10^6
100+100 (e)	22	48	1.6×10^6	3.7×10^6
50+50 (n)	34	63	8.2×10^5	1.3×10^6
100+100 (n)	101	172	5.6×10^6	1.1×10^7

Table 5.4: Total runtime and implication number of a sequence of 10 instances for real networks.

	Total Runtime (sec.)		Implication#	
BN	DynaMC	Cachet	DynaMC	Cachet
Alarm	17	43	1.5×10^6	8.4×10^6
Insurance	360	1082	1.2×10^8	3.9×10^8
Asia	0.01	0.01	169	840
Car-starts	0.02	0.02	370	1690
Water	94	465	2.7×10^7	1.4×10^8

the component database is full of small components, the overhead of checking each new generated component increases. Even if the correct component is found, only a few variables can be skipped in the search tree. If I limit the size components to import only "big" components, I can improve the performance of most "fail" instances. In practice, one would expect the imported components to have at least 20-50 literals. However, I did not set a component limit in any of the experiments reported above. Another possible solution for the problem of overhead is to design a more accurate hash function to increase the hitting rate, so that when searching for a component in the database the query can return more quickly.

5.5 Summary

In applications where there is uncertainty about the state of the world and our own observations of that world, a knowledge representation and reasoning system based on Bayesian networks can be used to advantage. In many such applications, queries to the Bayesian network will be interleaved with updates and refinements to the network. So there is an obvious need for improving the performance of probabilistic inference as incremental local change happens. I presented techniques for improving the efficiency of exact inference in incrementally updated Bayesian networks. My approach takes as its starting point previous work that has shown that probabilistic inference in Bayesian networks can be reduced to weighted model counting of a CNF encoding of the network [23, 76, 89]. I defined the concept of *dynamic model counting* and presented an algorithm for performing dynamic model counting. I showed that by maintaining the partial solutions of similar instances, great speedups can be achieved over current model counting algorithms. Cachet is currently the fastest model counting solver available. In grid, DQMR and QMR-DT problems, Cachet dominates both junction tree and previous state-of-the-art conditioning algorithms. As compared with Cachet, I obtained significant improvements in runtime for most networks when the networks are incrementally updated.

In the next chapter, I summarize the contributions of this thesis and discuss some potential future work.

Chapter 6

Conclusion and Future Work

In this chapter, I summarize the work presented in this thesis and discuss possible future work.

6.1 Conclusion

Knowledge representation and reasoning (KRR) have been central to the field of artificial intelligence since its inception. In this thesis, my focus has been on KRR systems based on propositional logic and logical inference and KRR systems based on Bayesian networks and probabilistic inference. The general problem that I addressed was improving the efficiency of inference in these types of KRR systems. The specific solutions that I proposed involved techniques and algorithms for exploiting the structure of real-world instances within backtracking algorithms for query answering.

For the task of determining whether a propositional CNF formula is satisfiable, the main results are as follows. Previous studies have demonstrated that backtracking search algorithms can be considerably improved if they take advantage of the internal structure of propositional formulas to decompose the an instance into independent subproblems. However, most existing decomposition techniques are static and performed prior to search. I propose a dynamic decomposition method based on hypergraph separators. Integrating the separator decomposition into the variable ordering of a modern SAT solver leads to speedups on large real-world satisfiability problems. In comparison to static decomposition based variable orderings, my approach does not need time to construct the full decomposition prior to search, which sometimes needs more time than the solving process itself. Furthermore, my dynamic method can solve hard instances not solvable by previous static approaches within an acceptable amount of time.

For the task of answering a general probabilistic query of the from $P(Q \mid E)$ from a Bayesian network, the main results are as follows. Previous studies have demonstrated that encoding a Bayesian network into a SAT formula and then performing weighted model counting using a DPLL-based algorithm can be an effective method for exact inference, where DPLL is a backtracking algorithm specialized for SAT that includes unit propagation, conflict recording and backjumping [89]. I present tech-

niques for improving this weighted model counting approach for Bayesian networks with noisy-OR and noisy-MAX relations. In particular, I present space efficient CNF encodings for noisy-OR and noisy-MAX which exploit their structure or semantics. In my encodings, I pay particular attention to reducing the treewidth of the CNF formula and to directly encoding the effect of unit propagation on evidence into the CNF formula, without actually performing unit propagation. I also explore alternative search ordering heuristics for the DPLL-based backtracking algorithm. I experimentally evaluated my techniques on large-scale real and randomly generated Bayesian networks. On these benchmarks, my techniques gave speedups of up to two orders of magnitude over the best previous approaches for Bayesian networks with noisy-OR relations and scaled up to networks with larger numbers of random variables. My techniques extend the model counting approach for exact inference to networks that were previously intractable for the approach.

Further, many real world Bayesian network applications need to update their networks incrementally as new data becomes available. For example, the capability of updating a Bayesian network is crucial for building adaptive systems. I present techniques for improving the efficiency of exact inference in incrementally updated Bayesian networks by exploiting common structure. In particular, I propose and formalize the concept of dynamic weighted model counting and present an algorithm for performing dynamic model counting. The techniques I propose provide a general approach for reusing partial results generated from answering previous queries based on the same or a similar Bayesian network. My focus is to improve the efficiency of exact inference when the network structure or the parameters or the evidence is updated. I show that my approach can be used to significantly improve inference on multiple challenging Bayesian network instances and other problems encoded as dynamic model counting problems.

Together my results increase the efficiency and improve the scalability of query answering in knowledge representation and reasoning systems, and so increases their applicability in practice.

6.2 Future Work

I structure my discussion of possible future work according to the three main tasks addressed in the thesis.

I considered a knowledge base expressed in propositional logic and an inference engine based on model finding and proposed methods for dynamically decomposing propositional formulas during the backtracking search (see Chapter 3). Future work could include additional ways to combine a structure-guided ordering heuristic and a conflict analysis-guided ordering heuristic. One possibility that may be worthy of future exploration is to dynamically recognize during the search that a small separator exists, before any decision is made on which variable to branch on next. The idea would be that if the residual primal graph was decomposable by some small separator(s), we would record all of these separators and instantiate those variables first. If the current residual primal graph was not decomposable, we would keep using

Table 6.1: Dynamic k-set separator decomposition methods, where n and m denote the number of vertices and the number of edges, respectively, of the primal graph representation of a propositional formula.

Separator size	Algorithm	Complexity	Property
0	DFS/WFS	$O(n+m)$	connected
1	Articulation Point	$O(n+m)$	biconnected
2	SPQR Tree	$O(n+m)$	triconnected
3	A. Kanevsky [68]	$O(n^2)$	four-connected
k	Z. Galil [44]	$O(max(k, n^{1/2})kmn^{1/2})$	$(k+1)$-connected

the conflict analysis-guided ordering heuristics to find the next decision variable. For small values of k, efficient algorithms are available (see Table 6.1).

I presented techniques for extending the weighted model counting approach to exact inference to Bayesian networks that contain the widely used noisy-OR and noisy-MAX relations (see Chapter 4). Future work could include developing SAT-encodings of other causal independence relations such as noisy-AND/MIN, noisy-XOR, and noisy-ADD (see, e.g., [34]).

I presented techniques for improving the efficiency of exact inference in incrementally updated Bayesian networks (see Chapter 6). Future work could include applying my framework to specific applications. One exciting possible application is bounded model checking (see, e.g., [10]). In bounded model checking one formally verifies that a system, such as a hardware CPU or a software project, satisfies a property such as "liveness" or "safety". The basic idea is to iteratively search for a counter-example of length bounded by k, $k = 0, \ldots$, where a counter-example represents a bug and the search continues until some known upper bound is reached or computational resources are exhausted. The bounded model checking problem can be reduced to propositional satisfiability and the important feature in this context is that the satisfiability instances that arise at iteration k and $k+1$ are intricately related. The fit here is not exact as my framework was for dynamic model counting and in this context we are only interested in satisfiability. Nevertheless, this avenue appears to be worth exploring. Another possible application of my framework is in learning Bayesian networks from data (see, e.g., [54]). Many proposals have been given for learning or boosting the performance of a Bayesian network so as to improve accuracy over time. These approaches involve modifying both the network structure and the network parameters. Hence, my framework may also be useful in this context.

Bibliography

[1] I. Adler, G. Gottlob, and M. Grohe. Hypertree width and related hypergraph invariants. *Eur. J. Comb.*, 28(8):2167–2181, 2007.

[2] F. A. Aloul, I. L. Markov, and K. A. Sakallah. MINCE: A static global variable-ordering for SAT and BDD. In *International Workshop on Logic and Synthesis*. University of Michigan, June 2001.

[3] E. Amir and S. McIlraith. Solving satisfiability using decomposition and the most constrained subproblem. In *Proceedings of the 4th Workshop on Theory and Applications of Satisfiability Testing (SAT-01)*, 2001.

[4] F. Bacchus, S. Dalmao, and T. Pitassi. DPLL with caching: A new algorithm for #SAT and Bayesian inference. *Electronic Colloquium on Computational Complexity*, 10(3), 2003.

[5] F. Bacchus, S. Dalmao, and T. Pitassi. Value elimination: Bayesian inference via backtracking search. In *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI-03)*, pages 20–28, 2003.

[6] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93)*, pages 188–191, 1993.

[7] M. Baker and T. E. Boult. Pruning Bayesian networks for efficient computation. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence (UAI-90)*, pages 225–232, 1991.

[8] R. J. Bayardo Jr. and D. P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 298–304, 1996.

[9] R. J. Bayardo Jr. and J. D. Pehoushek. Counting models using connected components. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, pages 157–162, 2000.

[10] A. Biere, A. Cimatti, E. Clarke, O. Shtrichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58, 2003.

[11] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th ACM/IEEE Design Automation Conference (DAC-99)*, pages 317–320, 1999.

[12] P. Bjesse, J. Kukula, R. Damiano, T. Stanion, and Y. Zhu. Guiding SAT diagnosis with tree decompositions. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-03)*, pages 315–329, 2003.

[13] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–23, 1993.

[14] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specific independence in Bayesian networks. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence (UAI-96)*, pages 115–123, 1996.

[15] W. L. Buntine. Theory refinement of Bayesian networks. In *Proceedings of the Seventh Conference on Uncertainty in Artificial Intelligence (UAI-91)*, pages 52–60, 1991.

[16] H. Chan and A. Darwiche. Sensitivity analysis in Bayesian networks: From single to multiple parameters. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence (UAI-04)*, pages 67–75, 2004.

[17] M. Chavira, D. Allen, and A. Darwiche. Exploiting evidence in probabilistic inference. In *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence (UAI-05)*, pages 112–127, 2005.

[18] M. Chavira and A. Darwiche. Compiling Bayesian networks with local structure. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 1306–1312, 2005.

[19] M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, 2007.

[20] B. D'Ambrosio. Symbolic probabilistic inference in large BN2O networks. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence (UAI-94)*, pages 128–135, 1994.

[21] A. Darwiche. Dynamic jointrees. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 97–104, 1998.

[22] A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 126(1-2):5–41, 2001.

[23] A. Darwiche. A logical approach to factoring belief networks. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR-02)*, pages 409–420, 2002.

[24] A. Darwiche. A differential approach to inference in Bayesian networks. *J. ACM*, 50(3):280–305, 2003.

[25] A. Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge, 2009.

[26] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[27] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[28] R. Dechter. Learning while searching in constraint satisfaction problems. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, pages 178–183, 1986.

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

[30] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 37–42, 1988.

[31] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[32] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[33] F. J. Díez. Parameter adjustement in Bayes networks. The generalized noisy OR-gate. In *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence (UAI-93)*, pages 99–105, 1993.

[34] F. J. Díez and M. J. Druzdzel. Canonical probabilistic models for knowledge engineering. Technical Report CISIAD-06-01, UNED, Madrid, 2006.

[35] F. J. Díez and S. F. Galán. Efficient computation for the noisy MAX. *International J. of Intelligent Systems*, 18:165–177, 2003.

[36] M. J. Flores, J. A. Gámez, and K. G. Olesen. Incremental compilation of Bayesian networks. In *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI-03)*, pages 233–240, 2003.

[37] M. J. Flores, J. A. Gámez, and K. G. Olesen. Incremental compilation of Bayesian networks in practice. In *Proceedings of the Fourth International Conference On Intelligent Systems Design and Applications (ISDA 2004)*, pages 843–848, 2004.

[38] J. W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

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

[40] E. C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 32(4):755–761, 1985.

[41] E. C. Freuder. In pursuit of the holy grail. *ACM Computing Surveys*, 28A(4):63, 1996. Constraint Programming Position Paper for ACM Workshop on Strategic Directions in Computing Research.

[42] E. C. Freuder and M. J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 1076–1078, 1985.

[43] N. Friedman and M. Goldszmidt. Sequential update of Bayesian network structure. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI-97)*, pages 165–174, 1997.

[44] Z. Galil. Finding the vertex connectivity of graphs. *SIAM J. Comput*, 9:197–199, 1980.

[45] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[46] D. Geiger, T. Verma, and J. Pearl. d-separation: From theorems to algorithms. In *Proceedings of the Fifth Conference on Uncertainty in Artificial Intelligence (UAI-89)*, pages 139–148, 1990.

[47] E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT solver. *Discrete Applied Mathematics*, 155(12):142–149, 2002.

[48] S. W. Golomb and L. D. Baumert. Backtrack programming. *J. ACM*, 12(4):516–524, 1965.

[49] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, chapter 20. IOS Press, 2009.

[50] I. J. Good. A causal calculus. *The British Journal for the Philosophy of Science*, 12(45):43–51, 1961.

[51] M. Gyssens, P. Jeavons, and D. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66:57–90, 1994.

[52] D. Heckerman. A tractable inference algorithm for diagnosing multiple diseases. In *Proceedings of the Fifth Conference on Uncertainty in Artificial Intelligence (UAI-89)*, pages 163–172, 1989.

[53] D. Heckerman. Causal independence for knowledge acquisition and inference. In *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence (UAI-93)*, 1993.

[54] D. Heckerman. A tutorial on learning Bayesian networks. Technical Report MSR-TR-95-06, Microsoft Research, 1995.

[55] D. Heckerman and J. Breese. Causal independence for probability assessment and inference using Bayesian networks. *IEEE, Systems, Man, and Cyber.*, 26:826–831, 1996.

[56] M. Henrion. Some practical issues in constructing belief networks. In *Proceedings of the Third Conference on Uncertainty in Artificial Intelligence (UAI-87)*, pages 132–139, 1987.

[57] M. Herbstritt. Improving propositional satisfiability algorithms by dynamic selection of branching rules. Technical report, University of Freiburg, 2001.

[58] hMETIS. www-users.cs.umn.edu/~karypis/metis/hmetis.

[59] J. Hoey, R. St. Aubin, A. Hu, and C. Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 279–288, 1999.

[60] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *J. of Automated Reasoning*, 15(3):359–383, 1995.

[61] H. H. Hoos and K. O'Neill. Stochastic local search methods for dynamic SAT-an initial investigation. In *AAAI-2000 Workshop Leveraging Probability and Uncertainty in Computation*, pages 22–26, 2000.

[62] H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In I. Gent, H. van Maaren, and T. Walsh, editors, *SAT2000: Highlights of Satisfiability Research in the Year 2000*, Frontiers in Artificial Intelligence and Applications, pages 283–292. Kluwer Academic, 2000.

[63] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Elsevier / Morgan Kaufmann, 2004.

[64] J. Huang and A. Darwiche. A structure-based variable ordering heuristic for SAT. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 1167–1172, 2003.

[65] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146:43–75, 2003.

[66] F. V. Jensen. *Bayesian Networks and Decision Graphs*. Information Science and Statistics. Springer, July 2001.

[67] H. Jin and F. Somenzi. An incremental algorithm to check satisfiability for bounded model checking. *Electr. Notes Theor. Comput. Sci.*, 119:51–65, 2005.

[68] A. Kanevsky and V. Ramachandran. Improved algorithms for graph four-connectivity. *J. Comput. Syst. Sci.*, 42(3):288–306, 1991.

[69] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Applications in VLSI domain. Technical report, University of Minnesota, 1997.

[70] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference (DAC-99)*, pages 343–348. ACM, 1999.

[71] H. Kautz and B. Selman. Blackbox: A new approach to the application of theorem proving to problem solving, 1998.

[72] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

[73] W. Lam and F. Bacchus. Using new data to refine a Bayesian network. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence (UAI-94)*, pages 383–390, 1994.

[74] S. L. Lauritzen. The EM algorithm for graphical association models with missing data. *Computational Statistics and Data Analysis*, 19:191–201, 1995.

[75] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. In *Readings in Uncertain Reasoning*, pages 415–448. Morgan Kaufmann, 1990.

[76] M. L. Littman. Initial experiments in stochastic satisfiability. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 667–672, 1999.

[77] S. O. Memik and F. Farzan. Accelerated SAT-based scheduling of control/data flow graphs. In *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD-02)*, page 395, 2002.

[78] R. A. Miller, F. E. Masarie, and J. D. Myers. Quick medical reference for diagnostic assistance. *Medical Computing*, 3:34–48, 1986.

[79] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[80] K. G. Olesen, U. Kjaerulff, F. Jensen, F. V. Jensen, B. Falck, S. Andreassen, and S. K. Andersen. A MUNIN network for the median nerve: A case study on loops. *Appl. Artificial Intelligence*, 3(2-3):385–403, 1989.

[81] R. Parker and R. Miller. Using causal knowledge to create simulated patient cases: The CPCS project as an extension of INTERNIST-1. In *The 11th Symposium Computer Applications in Medical Care*, pages 473–480, 1987.

[82] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[83] D. Poole, A. Mackworth, and R. Goebel. *Computational Intelligence: A Logical Approach*. Oxford, 1998.

[84] N. Robertson and P. D. Seymour. Graph minors II: algorithmic aspects of tree-width. *Journal Algorithms*, 7:309–322, 1986.

[85] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[86] S. J. Russell, J. Binder, D. Koller, and K. Kanazawa. Local learning in probabilistic networks with hidden variables. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1146–1152, 1995.

[87] D. Sabin and E. C. Freuder. Understanding and improving the MAC algorithm. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP 1997)*, pages 167–181, 1997.

[88] T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT-04)*, 2004.

[89] T. Sang, P. Beame, and H. Kautz. Solving Bayesian networks by weighted model counting. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 1–10, 2005.

[90] T. Sang, P. Beame, and H. A. Kautz. Heuristics for fast exact model counting. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, pages 226–240, 2005.

[91] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International J. on Artificial Intelligence Tools*, 3:1–15, 1994.

[92] G. R. Shafer and P. P. Shenoy. Probability propagation. *Annals of Mathematics and Artificial Intelligence*, 2:327–351, 1990.

[93] G. Shani, P. Poupart, R. I. Brafman, and S. E. Shimony. Efficient ADD operations for point-based algorithms. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 330–337, 2008.

[94] L. Simon, D. L. Berre, and E. A. Hirsch. The SAT 2002 competition. citeseer.ist.psu.edu/simon02sat.html.

[95] C. Sinz. Visualizing the internal structure of SAT instances. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT-04)*, May 2004.

[96] C. Spiegelhalter and R. G. Cowell. Learning in probabilistic expert systems. In J. M. Bernardo, J. O. Berger, A. P. Dawid, and A. F. M. Smith, editors, *Bayesian Statistics 4*, pages 447–465. Oxford University Press, 1993.

[97] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[98] M. Takikawa and B. D'Ambrosio. Multiplicative factorization of noisy-MAX. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 622–630, 1999.

[99] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput*, 1:146–160, 1972.

[100] M. Thurley. sharpSAT: Counting models with advanced component caching and implicit BCP. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT-06)*, pages 424–429, 2006.

[101] P. van Beek. Backtracking search algorithms. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 4. Elsevier, 2006.

[102] F. van Harmelen, V. Lifschitz, and B. Porter, editors. *Handbook of Knowledge Representation*. Elsevier, 2008.

[103] A. Zagorecki and M. J. Druzdzel. Knowledge engineering for Bayesian networks: How common are noisy-MAX distributions in practice? In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 482–489, 1992.

[104] zChaff. www.ee.princeton.edu/~chaff/zchaff.php.

[105] N. L. Zhang and D. Poole. Exploiting causal independence in Bayesian network inference. *J. of Artificial Intelligence Research*, 5:301–328, 1996.