PROBABILISTIC TECHNIQUES FOR CONSTRAINT SATISFACTION PROBLEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

In Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Lukas Kroc

August 2009

PROBABILISTIC TECHNIQUES FOR CONSTRAINT SATISFACTION PROBLEMS

Lukas Kroc, Ph.D.

Cornell University 2009

Constraint satisfaction problems (CSPs) are at the core of many tasks with direct practical relevance, such as hardware and software verification, planning, and scheduling to name a few. The two main solution paradigms for solving such problems are based on backtrack-style search and local search. However, recently, a new powerful technique, called survey propagation (SP), was introduced. SP can solve certain classes of problem instances with millions of variables and constraints. This discovery raised a question of whether the traditional techniques are the best one can do when tackling large and important constraint satisfaction problems.

This dissertation discusses non-traditional approaches to solving CSPs. The techniques we use include probabilistic inference, statistical tools and a combination of systematic and local search. We provide a new derivation of SP based on purely combinatorial insights. Our method also enables us to derive SP-style procedures for a diverse range of constraint satisfaction problems, and provides insights into structure of solution spaces. The second part of the dissertation describes approaches to counting number of solutions of a CSP with the help of Belief Propagation and statistics. We also provide a novel hybrid algorithm for minimizing the number of violated constraints, which employs both systematic and local search, harnessing their complementary strengths. These methods are the most scalable algorithms for certain hard to solve classes of Boolean satisfiability (SAT), model counting (#SAT), and maximum satisfiability (MaxSAT) problems.

BIOGRAPHICAL SKETCH

Lukas Kroc has earned his PhD degreed in computer science at Cornell University in 2009. His research interests lie in the field of artificial intelligence, with focus on using techniques of probabilistic inference in combinatorial reasoning. Prior to his PhD time he worked at Los Alamos National Laboratory on large scale simulation design and development. Lukas received a MSc in computer science from Charles University, Prague, in 2004.

To Beatrice, for being an eternal inspiration.

Thus spoke Ulysses to Dante the Pilgrim about his journey to Ithaca [Alighieri and Musa, 2002]:

> *Not sweetness of a son, not reverence*
> *for an aging father, not the debt of love*
> *I owed Penelope to make her happy,*
>
> *could quench deep in myself the burning wish*
> *to know the world and have experience*
> *of all man's vices, of all human worth.*

ACKNOWLEDGEMENTS

I would like to extend my sincerest gratitude towards people who have helped me with this dissertation, and have supported me during my time at Cornell. This, of course, includes the members of my committee, Michael Spivey, John Hopcroft, and my advisor Bart Selman. Bart's very encouraging and supportive attitude in particular was a great help during my studies. Above all, I appreciated very much his friendly approach to me, and his good understanding of basic human needs, the search for happiness being one of the more important ones.

My stay in Ithaca would not have been possible without the constant support from many of my friends, who I dare not list here fearing that I might unintentionally omit some. I want to thank them all from the bottom of my heart for sharing parts of their lives with me, and for showing me many a thing that I did not even know I had been missing before. However, one name must be mentioned, as he not only supported me as a friend, but also very much as a researcher: Ashish Sabharwal. Ashish was a post-doctoral researcher at Cornell at the time, and all the research described in this dissertation was discussed with him and made possible only through his help.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . vi
Table of Contents . . . vii
List of Tables . . . ix
List of Figures . . . x

1 Introduction **1**
1.1 Problem Classes . . . 4
1.2 Solution Techniques . . . 5
1.3 Technical Contributions . . . 7

2 Preliminaries **9**
2.1 Boolean Satisfiability and Graph Coloring Problems . . . 9
2.2 Factor Graphs and Probabilistic Inference . . . 11
2.3 Approximate Probabilistic Inference Using Belief Propagation . . . 13

3 Message-Passing and Local Heuristics as Decimation Strategies for Satisfiability **21**
3.1 Solving SAT by Decimation . . . 24
3.1.1 Belief Propagation & Survey Propagation . . . 25
3.1.2 Convergent BP . . . 27
3.2 Decimation Strategies . . . 32
3.3 Observable Differences In Decimation Strategies . . . 34
3.3.1 BP and SP Avoid Creating Unit Clauses . . . 34
3.3.2 Evolution of Problem Hardness . . . 37
3.4 Discussion . . . 39

4 Survey Propagation Revisited **41**
4.1 Covers of CNF Formulas . . . 44
4.2 Problem Reformulation: From Solutions To Covers . . . 47
4.2.1 Inference Over Covers . . . 52
4.3 Derivation of the SP Equations . . . 53
4.4 Experimental Results . . . 57
4.4.1 Existence of Covers . . . 57
4.4.2 SP, BP, and Marginals . . . 61
4.5 Discussion . . . 64

5 Reasoning About Clusters **66**
5.1 A Pseudo-Partition Function for Counting Clusters . . . 69
5.2 Understanding $\mathbf{Z}_{(-\mathbf{1})}$ and Proving Exactness . . . 74
5.2.1 General Proof Framework . . . 76
5.2.2 $\mathbf{Z}_{(-\mathbf{1})}$ for Graph Coloring . . . 84

5.2.3 $\mathbf{Z}_{(-1)}$ for 2-SAT . . . 89
5.2.4 $\mathbf{Z}_{(-1)}$ for Recursively Monotone Spaces and SAT . . . 89
5.2.5 Marginals of $\mathbf{Z}_{(-1)}$ . . . 94
5.3 Exact Computation of Number of Clusters and $\mathbf{Z}_{(-1)}$ . . . 99
5.3.1 Computing number of clusters . . . 101
5.3.2 Evaluating $\mathbf{Z}_{(-1)}$ . . . 102
5.4 Belief Propagation Inference for Clusters . . . 103
5.4.1 Assumptions . . . 103
5.4.2 Kullback-Leibler divergence . . . 106
5.4.3 Minimizing divergence . . . 106
5.4.4 BP equations for $Z_{(-1)}$ . . . 110
5.5 Comparison with SP . . . 111
5.5.1 Derivation of SP Equations from $\mathrm{BP}_{(-1)}$ for SAT . . . 112
5.5.2 Simplifying $\mathrm{BP}_{(-1)}$ for 3-COL . . . 113
5.6 Empirical Validation . . . 114
5.6.1 $\mathbf{Z}_{(-1)}$ vs. the number of clusters . . . 114
5.6.2 $\mathrm{BP}_{(-1)}$ Approximation of $\mathbf{Z}_{(-1)}$ . . . 117
5.6.3 Estimates on very large problems and for various constraint densities . . . 119
5.6.4 $\mathbf{Z}_{(-1)}$ vs. the number of clusters for structured instances . . . 120
5.6.5 $\mathbf{Z}_{(-1)}$ marginals vs. clusters marginals . . . 123
5.7 Discussion . . . 124

6 Leveraging Belief Propagation, Backtrack Search, and Statistics for Model Counting 126
6.1 Lower Bounds Using BP Marginal Estimates: **BPCount** . . . 131
6.2 Upper Bound Estimation Using Backtrack Search: **MiniCount** . . . 137
6.2.1 Estimating the Upper Bound Using Statistical Methods . . . 143
6.3 Experimental Results . . . 147
6.4 Discussion . . . 152

7 Integrating Systematic and Local Search Paradigms: A New Strategy for MaxSAT 154
7.1 Preliminaries . . . 161
7.2 Hybrid Solver: Using DPLL to Guide Local Search . . . 163
7.3 Relaxed Solver: Tolerating Contradictions . . . 166
7.4 Experimental Results . . . 168
7.4.1 Hybrid Search Pattern . . . 172
7.5 Discussion . . . 176

8 Conclusions and Future Research Directions 177

LIST OF TABLES

6.1 Performance of `BPCount` and `MiniCount` 148
6.2 (Continued from Table 6.1.) Performance of `BPCount` and `MiniCount`.149
6.3 Performance of `MiniCount` without restarts. 150

7.1 Comparison of MaxSAT results for exact, local search, and currently proposed hybrid, and relaxed DPLL methods. 169
7.2 (Continued from Table 7.1.) Comparison of MaxSAT results. . . . 170

LIST OF FIGURES

2.1 An example of a factor graph 12
2.2 Belief propagation update equations for SAT problem. 19

3.1 Modified belief propagation update equations, BP^{κ}. 29
3.2 Quality of solution marginals computed by BP (left) and BP^{κ} (right). 31
3.3 Various decimation heuristics on random 3-SAT, 4-SAT, and 5-SAT formulas 33
3.4 Cumulative number of unit clauses generated during decimation . . 36
3.5 Hardness of subformulas created by BP^{κ} and SP 38

4.1 BP equations for F_x 55
4.2 BP equations for F_a 55
4.3 The peeling experiment, showing the evolution of the number of stars as $*$-propagation proceeds. 58
4.4 Non-trivial covers in random formulas. Left: existence probability. Right: average number. 59
4.5 Left: Expected number of solutions leading to non-trivial covers (log-log scale). Right: Probability of a solution leading to a non-trivial cover. 61
4.6 Magnetization plots. Left: SP vs. covers. Middle: covers vs. solutions. Right: BP vs. solutions. 63

5.1 Illustration for the proof of Lemma 4 87
5.2 Left: $Z_{(-1)}$ vs. number of clusters in random 3-COL problems with 20, 50 and 100 vertices, and average vertex degree between $1.0 - 4.7$. Right: Cluster marginals vs. $Z_{(-1)}$-marginals for one instance of random 3-COL problem with 100 vertices. 115
5.3 $Z_{\mathrm{BP}_{(-1)}}$ compared to $Z_{(-1)}$ for 3-COL problem on random graphs with 50, 100 and 200 vertices and average vertex degree in the range $1.0 - 4.7$. 117
5.4 Left: $Z_{(-1)}$ vs. $Z_{\mathrm{BP}_{(-1)}}$ for random 3-SAT with varying n and $\alpha = 4.0$. Right: $Z_{(-1)}(x = 1)$ vs. $Z_{\mathrm{BP}_{(-1)}}(x = 1)$ marginals for one instance of random 3-SAT with $n = 200, \alpha = 4.0$. 118
5.5 Average $Z_{\mathrm{BP}_{(-1)}}$ and $Z_{(-1)}$ for 3-COL vs. average vertex degrees for small and large random graphs. 119
5.6 Average $Z_{\mathrm{BP}_{(-1)}} = Z_{SP}$ and $Z_{(-1)}$ for 3-SAT vs. clause-to-variable ratio for small and large random formulas. 121
5.7 Left: number of clusters vs. $Z_{(-1)}$ and number of covers in random 3-SAT for $n = 90, \alpha = 4.0$. Right: positive cluster marginals vs. $Z_{(-1)}(x = 1)$ marginals for one instance of random 3-SAT with $n = 200, \alpha = 4.0$. 124

6.1 Sampled and theoretical quantiles for formulas described in the experimental section (top: `alu2_gr_rcs_w8`, `lang19`; bottom: `2bitmax_6`, `wff-3-150-525`, `ls11-norm`). 145

7.1 Qualitative search behavior in terms of the number of unsatisfied clauses (y-axis, log scale) as runtime progresses (x-axis). Both state-of-the-art pure local search methods, unguided `Walksat` and `MiniWalk` are shown. Note the deep drops of `MiniWalk`, which distinguish it from the other techniques. Instance: babic-dspam-vc973.cnf. 173

7.2 Qualitative search behavior in terms of the number of unsatisfied clauses (y-axis, log scale) vs. runtime (x-axis). Both state-of-the-art pure local search methods, unguided `Walksat` and `MiniWalk` are shown. Note the unique steep drops of `MiniWalk`. Instance: ibm-2002-31_1r3-k30.cnf. 174

7.3 Number of unsatisfied clauses in `MiniWalk` and DPLL search depth vs. time in babic-dspam-vc973.cnf. 175

Chapter 1

Introduction

Many problems occurring in science, engineering and life in general have the form of making several interdependent decisions, so that a certain set of conditions is satisfied. For example, an airline company makes decisions about which airplanes fly from where and when, subject to the conditions that the planes are available and its customers are properly served. These kinds of problems can be conveniently modeled in a framework of Constraints Satisfaction Problems (CSPs). The choices and other unknowns in the problem are represented by *variables*, typically with a discrete domain, and the conditions that need to be satisfied in a solution of the problem are given as a set of *constraints*, limiting the set of values each variable can take given values of other variables. A solution to a CSP is then an assignment to all variables such that all constraints are satisfied. This framework allows for natural modeling of many practical problems, and devising solution techniques for CSP is thus a very important area of research. In the worst case, solving a CSP problem is an NP-hard task, because Boolean Satisfiability (SAT), the first problem to be shown NP-hard, is a special case of a CSP with binary boolean variables and constraints specified as logical disjunctions. It is also the case that many practical CSP problems are hard to solve, and a lot of research effort goes into

improving efficiency of CSP and SAT solvers. The solvers have become gradually more and more powerful, and in the past 15 years they evolved from a "toy-problem solvers" to solvers that help with important industrial tasks. Nevertheless, scalability remains an issue, and new and more advanced techniques are being sought.

The material contained in this dissertation touches on several possible extensions to current techniques. A large portion is devoted to examining and analyzing an algorithm that was introduced in the statistical physics community by Mézard, Parisi, and Zecchina [2002], called Survey Propagation (SP). This algorithm is able to solve hard instances from a family of so called "random SAT problems" with millions of variables, far outperforming other kinds of solvers. The solution technique being employed is *decimation*, that is setting variables one by one, where the information about which variables to set to which values comes from the SP. Being developed in a vastly different field, the algorithm remained hard to understand in the computer science community for many years. Braunstein and Zecchina [2004] and Maneva et al. [2005] showed that SP algorithm can be viewed as an instance of the well known Belief Propagation algorithm (BP) [Pearl, 1988], which has strengths and limitations that are well established in the CS community. Both BP and SP are called message passing algorithms, since they operate by converging on a solution by letting the variables and constraints exchange information (messages). The details are reviewed in the following chapter.

The first part of the dissertation compares performance of SP and BP with other decimation heuristics, with the aim of identifying features of the decimation process that allow the SP algorithm to outperform the other approaches. Survey propagation is then analyzed in more detail in the subsequent chapter, where solid empirical evidence is provided to complement the original derivation of SP as an

instance of the BP algorithm. The part of the dissertation dealing primarily with SP is completed by Chapter 5, where a novel derivation of the survey propagation message passing algorithm is provided. This chapter also discusses insights into solution space structure of CSPs that are closely related to the performace of SP.

The second part of the dissertation, Chapter 6, proposes a way to incorporate BP into a probabilistic scheme to efficiently estimate number of solutions of a CSP. We also propose a statistical framework suitable to reason about number of solutions. Finally, the last chapter deals with the problem of finding a variable assignment that satisfies the largest number of constraints (in cases where not all constrains can be satisfied simultaneously). We propose a novel approach of harnessing the power of both traditional solution techniques, local and systematic search, by letting the two search processes exchange information, benefiting from their complementary strengths. Therefore, the dissertation touches, in some way, on three different solution techniques (local search, systematic search, and decimation guided by message passing) for three classes of problems (solution search, solution count, and search for assignment violating fewest possible constraints).

The rest of the introduction briefly reviews each of the problem classes and solution techniques used in the dissertation. The focus here is on getting the intuitive picture between their differences and relative strengths and weaknesses, a more in-depth treatment is left for subsequent chapters where the techniques are being discussed. Finally, the technical contributions detailed in this dissertation are outlined at the end of the introduction. The dissertation is organized in such way that each chapter can be read independently, with the knowledge of basic terminology and concepts reviewed in Chapter 2.

1.1 Problem Classes

Without going into a formal definition, let us informally introduce a Constraint Satisfaction Problem (CSP). Such problem has a set of *variables* with discrete domains, and a set of *constraints*, each specifying a set of allowed value tuples for a certain subset of variables. We can now describe the three problem classes that are discussed in this dissertation:

Solution Search: A value assignment to all variables is being sought such that every constraint is satisfied, that is the assignment comprises of allowed value tuples for all constraints. Such an assignment is a *solution*. A special case of this class is the well studied Boolean Satisfiability problem (SAT).

Solution Count: Problem of counting the number of solutions of a CSP. A special case is a problem of counting number of solutions to SAT, a so called #SAT problem.

Most Satisfying Assignment: Problem of finding an assignment that violates the fewest possible constraints (usually with the assumption that there is no solution that satisfies all constraints). A special case is Maximum Satisfiability (MaxSAT), a problem of minimizing the number of violated constraints in SAT.

These are all hard problems in the worst case. Solution search (also a special case of minimizing number of violated constraints) is an NP-hard problem, due to SAT; solution count is #P-hard, due to #SAT [Valiant, 1979].

1.2 Solution Techniques

The two most widely used solution techniques for CSP are *systematic search* and *local search*. This part briefly describes the two approaches, contrasting their way of search. Details can be found for example in Biere et al. [2009]. In addition, a third solution technique that became promising with the success of Survey propagation, so called *decimation*, is reviewed.

Systematic Search: The basic idea of systematic search is to keep assigning variables one at a time, until a solution is found, or some constraint becomes violated. In the latter case, the search unassigns some previous assignments (backtracks), making the violated constraint possible to satisfy, and continues the search with other variable assignments. The most important decisions to be made are which variables to assign values first (variable selection), and which values to choose (value selection). In case of SAT, there are only two possible values for each variable (TRUE or FALSE), and together with the simplicity of the constraints (logical or), this has led to development of highly efficient heuristics, capable of solving relatively large instances of SAT encoded problems. The systematic solution search in SAT dates back several decades, and modern solvers still use an enhanced version of the Davis-Putnam-Logemann-Loveland (DPLL) procedure [Davis and Putnam, 1960, Davis et al., 1962]. The two main ingredients of modern systematic SAT solvers are fast *unit propagation* (the ability to discover variables that surely have to assume one particular value after some other variables have been set) and *clause learning* (the ability to learn inferred constraints that better guide the search).

To contrast with the other approaches, the main characteristic of systematic search is the following: it works with an incomplete, but consistent assignment. Incomplete because not all variables are assigned a value, and consistent because

no constraints are allowed to be violated at any time. A solution is, of course, both a complete and consistent assignment. Systematic search is also able to determine when a problem has no solution. The downside is that the solution search may take too long.

Local Search: In contrast, local search starts with a complete assignment to variables (often random), which will in general violate some (often many) constraints. It then proceeds to modify the variable assignment "locally" (that is, only few variable assignments change at a time) such that fewer constraints are violated. This is often done by changing variables that occur in some violated constraint to a configuration that satisfies that particular constraint. This, of course, can cause other constrains to become violated, but the hope is that overall the number of violated constraints decreases as the search progresses. The local search paradigm became popular when it turned out that it was vastly more efficient than systematic search on certain families of problems, most notably on random SAT problems [Selman et al., 1996], a family of problems where constraints are drawn uniformly at random from some predefined set, see Section 2.1 for details.

In a nutshell, local search works with a complete assignment to variables, but one that is inconsistent. Through small changes to the assignment, it seeks a consistent one. This is in a sense complementary to what systematic search is doing. Unfortunately, a standard local search is not able to determine unsatisfiability of a problem.

Decimation: This approach can be viewed as a radically different from the previous two. It is not a search, but rather a *construction* of a solution. Just like in solving many other complex problems (e.g. building a house), a lot of time is spent on making sure each step taken is a correct one, and will not have to be undone.

The decimation step itself is rather simple: determine which variable-value assignment is the most "prominent" one, and fix it. This simplifies the problem and the procedure can be repeated. We leave the exact meaning of "prominent" unspecified at this point, but it will usually be something like "an assignment that is compatible with most solutions". This is of course a very expensive piece of information (in fact, more expensive than obtaining any one solution!), but sufficiently accurate estimates are obtainable using probabilistic inference with message passing algorithms, like belief or survey propagation. The details of how decimation works are covered in Chapter 3.

On the other hand, decimation could also be viewed as a systematic search, with very expensive variable and value ordering heuristics. However, if a violated constraint is found due to incorrect variable setting, the algorithm fails rather than performing a backtrack.

1.3 Technical Contributions

The main technical contributions of the dissertation are summarized below.

- Comparison of performance of various decimation heuristics, including SP and BP, based on work published in the proceedings of the 24th Annual ACM Symposium on Applied Computing [Kroc et al., 2009b].
- Empirical evidence showing remarkable accuracy of SP when reasoning about "covers" of large random SAT formulas. Covers are combinatorial objects devised to derive survey propagation for SAT as as instance of belief propagation. This work appears in the proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence [Kroc et al., 2007].
- Theoretical framework for efficient reasoning about clusters of solutions in

CSP. A cluster is a set of solutions that are similar to each other in terms of the values of the variables. This work results in new insights about the solution space structure of CSPs, and provides a novel derivation of SP for SAT as a BP used to reason about clusters.

- A new message passing algorithm for estimating number of clusters in graph coloring problems. This is based on work published in the proceedings of the 22nd Conference on Advances in Neural Information Processing Systems [Kroc et al., 2008a].

- Modification of belief propagation to be used in a framework for lowerbounding number of solutions of a SAT problem. Part of that work is also first efficient upperbounding technique for number of solutions, but only under some statistical assumptions. This work was published in the proceedings of the 5th International Conference on Integration of AI and OR Techniques in Constraint Programming [Kroc et al., 2008b], and is under review for Annals of Operations Research.

- Combining systematic and local search for maximum satisfiability problems, where both searches run in parallel and exchange information, harnessing their complementary strengths. This is the most recent work, based on publications in the proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing [Kroc et al., 2009c] and the 21st International Joint Conference on Artificial Intelligence [Kroc et al., 2009a].

Chapter 2

Preliminaries

This chapter reviews basic concepts and terminology used throughout the dissertation.

2.1 Boolean Satisfiability and Graph Coloring Problems

The two constraint satisfaction problems most frequently studied in this dissertation are Boolean Satisfiability (SAT) and Graph Coloring (COL) problems, described below.

SAT: The Boolean satisfiability problem (SAT) is well-known to be NP-complete and asks the following question: Given a logical formula F over n Boolean variables $x_1, \ldots, x_n$, is there a TRUE/FALSE (equivalently, 1/0) assignment to each x_i such that F evaluates to TRUE? Such an assignment is referred to as a *satisfying assignment* for or *solution* of F. If no such truth assignment exists, F is called unsatisfiable.

We are interested in formulas in the Conjunctive Normal Form (CNF), where F is specified as a conjunction (AND, $\wedge$) of *clauses*, each clause is a disjunction (OR, $\vee$) of *literals*, and each literal is a variable x_i or its negation $\neg x_i$. An example of a CNF formula is $F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3)$. A clause containing k literals is called a k-clause. 1-clauses are often referred to as unit clauses. When every clause of a formula is a k-clause, we have a k-CNF formula, and the corresponding satisfiability problem is referred to as the k-SAT problem.

Next to SAT problems that arise as encodings of various problems from other domains, we will also work with *random k-SAT* formulas F, characterized by an important parameter α defined as the ratio of the number of clauses to the number of variables in F. A random k-SAT formula over n variables at ratio α is generated by selecting αn clauses uniformly at random from the set of all k-clauses over the n variables. The resulting distribution is often denoted by $\mathbb{F}(n, \alpha n)$. Random 3-SAT formulas, in particular, have been studied extensively in both theory and practice. They empirically exhibit a phase transition at the ratio $\alpha_c \approx 4.26$ [Mertens et al., 2006] in the sense that nearly all formulas below this threshold are found to be satisfiable, while those above are found to be unsatisfiable. This phase transition also coincides with a sharp peak in the computational complexity of finding solutions using state-of-the-art methods around this critical ratio. An interesting and well-researched area, then, is to device techniques that push the limit of feasibility of various solution methods as close to the phase transition region as possible.

COL: The Graph k-Coloring problem is the following: given an undirected graph G, decide whether each vertex of G can be assigned one of k colors such that no two adjacent vertices have the same color. It is well known that that for $k \geq 3$ the

problem is NP-complete, and we will most often work with 3-COL.

Like in the case of SAT, one can define a family of *random k-COL problems*. What is being randomized here is the input graph, which is taken to be an Erdös-Rényi random graph $G(n, m)$. In this model, n is the number of vertices, and m is the number of edges added to the graph. The m edges are selected uniformly at random without replacement from the set of all $\binom{n}{2}$ possible edges, i.e. all pairs of distinct vertices. Similarly to random 3-SAT, random 3-COL problems exhibits a sharp threshold of colorability at a certain constraint density c, which is in this case specified by average vertex degree, that is the number of edges per vertex in the input graph G. This threshold is estimated to be close to $c \approx 4.7$ [Braunstein et al., 2003].

2.2 Factor Graphs and Probabilistic Inference

An arbitrary Constraint Satisfaction Problem (CSP) can be expressed in the form of a *factor graph*, a bipartite graph with two kinds of nodes [Kschischang et al., 2001]. The *variable nodes*, denoted $\vec{x} = (x_1, \ldots, x_n)$, represent the variables in the CSP with their discrete domain Dom[1]. The *factor nodes*, denoted $\alpha, \ldots,$ with associated *factor functions*, denoted $f_\alpha, \ldots,$ represent the constraints of the CSP. Each factor function is a Boolean-valued function with arguments $\vec{x}_\alpha$ (denoting a subset of the variables in $\vec{x}$) and range $\{0, 1\}$, and evaluates to 1 if and only if the associated constraint is satisfied. For efficiency of valuating the factor functions, we assume they only have a constant number of arguments. The factor graph has an edge between a variable node x_i and a factor node α if and only if the variable x_i appears in the constraint represented by f_α; we denote this event by $i \in \alpha$.

[1]For simplicity of notation, we assume that all variables have the same domain. However, this is not necessary.

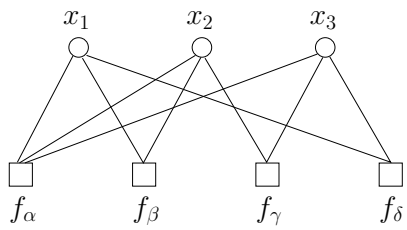


Figure 2.1: A factor graph representing the SAT problem $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3)$.

Figure 2.1 depicts an example factor graph for the SAT problem of deciding whether the following formula is satisfiable:

$$\underbrace{(x_1 \vee x_2 \vee \neg x_3)}_{\alpha} \wedge \underbrace{(\neg x_1 \vee x_2)}_{\beta} \wedge \underbrace{(\neg x_2 \vee x_3)}_{\gamma} \wedge \underbrace{(x_1 \vee \neg x_3)}_{\delta}$$

The factors $f_\alpha, f_\beta, \ldots$ represent predicates ensuring that the corresponding clauses have at least one satisfying literal.

The factor representation weighs all variable assignments $\vec{x}$ according to the product of values of all factors. We denote this product by $F(\vec{x}) := \prod_\alpha f_\alpha(\vec{x}_\alpha)$. In our case, the weight of an assignment $\vec{x}$ evaluates to 1 if and only if all of the factors have value of 1, otherwise it evaluates to 0. The assignments with weight 1 correspond exactly to satisfying assignments, or solutions, of the CSP. The number of satisfying assignments can thus be expressed as the weighted sum across all possible value assignments to $\vec{x}$. We denote this quantity by Z, the so called *partition function*:

$$Z := \sum_{\vec{x} \in Dom^n} F(\vec{x}) = \sum_{\vec{x} \in Dom^n} \prod_\alpha f_\alpha(\vec{x}_\alpha) \tag{2.1}$$

Factor graphs can be viewed as a succinct representation of a probability space where each state $\vec{x}$ has probability $\frac{1}{Z}F(\vec{x})$. Factor graphs are very similar (and equivalent in their expressive power) to other graphical models, like Bayesian Networks and Markov Random Fields [Yedidia et al., 2003].

Once a factor graph is constructed, there are several questions that are commonly of interest:

Marginal probabilities: Computation of marginal probability $p_i(x_i = v_i)$ of any one variable, that is computing the cumulative probability of all states where the variable x_i is fixed to value v_i. Formally: $p_i(x_i = v_i) := \frac{1}{Z}\sum_{\vec{x}_{-i} \in Dom^{n-1}} F(\vec{x})$. The sum is over all valuations of all variables except x_i.

Partition function: Computing the value of the partition function Z (very much related to the marginal probabilities).

Most likely state: Finding a state $\vec{x}$ with the highest weight, $F(\vec{x})$, or maximum likelihood state. Formally: $\text{argmax}_{\vec{x} \in Dom^n} F(\vec{x})$.

Probabilistic inference is the act of finding answers to these questions. Unfortunately, exact inference is in general #P-hard, so a common way of performing inference in large instances of factor graphs is to use *approximate inference*. One such technique is described below.

2.3 Approximate Probabilistic Inference Using Belief Propagation

Belief Propagation (BP) is an algorithm for approximate inference in graphical models [Pearl, 1988]. It can be used to estimate answers to all three common questions: values of marginal probabilities (the estimates are called "beliefs"), value of the partition function, and finding most likely state. The heart of the algorithm lies in solving a set of iterative equations derived specifically for a given problem instance (the variables in the system are called "messages", thus BP is

called a message passing algorithm). These equations are designed to provide accurate answers if applied to problems with no circular dependencies, such as constraint satisfaction problems with no loops in the corresponding factor graph, but in general give results of uncertain quality.

One way to derive the BP equations is to define a problem where one wants to minimize difference between two probability distributions, one of which is of interest but hard to reason about, and another which is easy to reason about. By minimizing the difference between the two, one arrives at a distribution which (hopefully) approximates well the original one, and at the same time allows for efficient reasoning. This so called variational derivation of BP by Yedidia, Freeman, and Weiss [2003, 2005] is briefly reviewed in this section. It is a starting point for the variational derivation of the survey propagation algorithm discussed in Chapter 5, but it is not necessary for understanding of the main findings in the dissertation.

Original Probability Distribution Given a factor graph, we are interested in a probability distribution over a set of states $\vec{x}$ of its variables defined as:

$$p(\vec{x}) := \frac{1}{Z} F(\vec{x}) = \frac{1}{Z} \prod_{\alpha} f_{\alpha}(\vec{x}_{\alpha})$$

where $\vec{x}_{\alpha}$ are all variables adjacent to factor α, and Z is a normalization constant, the partition function, $Z = \sum_{\vec{x}} \prod_{\alpha} f_{\alpha}(\vec{x}_{\alpha})$

We define *energy* of a configuration as:

$$E(\vec{x}) := -\sum_{\alpha} \ln f_{\alpha}(\vec{x}_{\alpha})$$

and therefore $p(\vec{x}) = \frac{1}{Z} e^{-E(\vec{x})}$. We can introduce *temperature* T by raising every f_{α} in $p(\vec{x})$ to the power of $\frac{1}{T}$, therefore $p_T(\vec{x}) = \frac{1}{Z(T)} e^{-\frac{E(\vec{x})}{T}}$.

Since $p_T(\vec{x}) = \frac{1}{Z(T)} \prod_{\alpha} f_{\alpha}(\vec{x}_{\alpha})^{\frac{1}{T}}$, the standard factor function is obtained for $T = 1$, and in the limit $T \to 0$, the only configurations with $p_T(\vec{x}) > 0$ will be

those with $p(\vec{x}) = \max_{\vec{y}} p(\vec{y})$. Thus, with $T \to 0$, one is performing maximum likelihood inference. We assume $T = 1$ for the rest of the derivation.

Trial Distribution We would like to compute values of $p(\vec{x})$, but since that is too expensive (due to having to compute Z), we try to approximate it, with a "trial" distribution $b(\vec{x})$. The trial distribution is some (unknown) distribution for which we will assume marginalization happens as if the factor graph was a tree (see Proposition 1), and which we will be able to parametrize with polynomially many parameters (as opposed to having to specify probability individually for each of the exponentially many states).

Kullback-Leibler Divergence To assess how well $b(.)$ approximates $p(.)$, we look at the Kullback-Leibler (KL) divergence between the two probability mass functions:

$$D(b \parallel p) := \sum_{\vec{x}} b(\vec{x}) \ln \frac{b(\vec{x})}{p(\vec{x})}$$

which is known to be non-negative, and zero iff $b \equiv p$.

Using the definition of p, we get:

$$D(b \parallel p) = \sum_{\vec{x}} b(\vec{x}) \ln b(\vec{x}) + \frac{1}{T} \sum_{\vec{x}} b(\vec{x}) E(\vec{x}) + \ln Z \tag{2.2}$$

which we want to minimize in order to well approximate p by b.

Free Energies Define "average energy" of $\vec{x}$ to be $U(b) := \sum_{\vec{x}} b(\vec{x}) E(\vec{x})$, and "entropy" to be $S(b) := -\sum_{\vec{x}} b(\vec{x}) \ln b(\vec{x})$. Then

$$G(b) := \frac{1}{T} U(b) - S(b) \equiv D(b \parallel p) - \ln Z$$

is "Gibbs Free Energy", and is minimized iff $b \equiv p$ (see Equation 2.2).

Moreover, when $G(b)$ is minimized (say at b_{min}), we also get an estimate of the value of the normalization constant $Z = e^{-G(b_{min})}$ by assuming that $D(b \parallel p) = 0$.

Shortening the Expressions The expressions for $D(b \parallel p)$ contains two exponential sums (over the whole state space), which is intractable to work with. We will massage the expressions to get tractable ones, using at one point the (possibly false) assumption of the factor graph being a tree, thus trading in accuracy for efficiency.

$$\begin{aligned} U(b) &= -\textstyle\sum_{\vec{x}} b(\vec{x}) \sum_{\alpha} \log f_\alpha(\vec{x}_\alpha) \\ &= -\textstyle\sum_{\alpha} \sum_{\vec{x}_\alpha} \left(\sum_{\vec{x}_{-\alpha}} b(\vec{x}) \right) \log f_\alpha(\vec{x}_\alpha) \\ &= -\textstyle\sum_{\alpha} \sum_{\vec{x}_\alpha} b_\alpha(\vec{x}_\alpha) \log f_\alpha(\vec{x}_\alpha) \end{aligned}$$

where $b_\alpha(\vec{x}_\alpha) = \sum_{\vec{x}_{-\alpha}} b(\vec{x})$ is the marginal probability of b on variables participating in the factor α. This construction needs no assumption. The tree-graph assumption is used to simplify the expression for $S(b)$. For this, we also need the following proposition:

Proposition 1. *If G is a tree, then the following holds:*

$$b(\vec{x}) = \frac{\prod_\alpha b_\alpha(\vec{x}_\alpha)}{\prod_i b_i(x_i)^{d_i - 1}}$$

where b_α and b_i are marginal probabilities of b and d_i is a degree of variable i.

This proposition thus links marginal probabilities to the full probability. It is also the point where the tree assumption is used. If we could find such expression for general graphs, BP would be exact. We have replaced exponentially many values of $b(\vec{x})$ by polynomially many $b_i(x_i), b_\alpha(\vec{x}_\alpha)$ (assuming bounded factor degree), and these are now the "parameters" of the probability mass function $b(.)$.

Given the proposition, we can rewrite:

$$\begin{aligned} S(b) &= -\sum_x b(\vec{x}) \log b(\vec{x}) \\ &= -\sum_x b(\vec{x}) \left(\sum_\alpha \log b_\alpha(\vec{x}_\alpha) - \sum_i (d_i - 1) \log b_i(x_i) \right) = \\ &= -\sum_\alpha \sum_{\vec{x}_\alpha} b_\alpha(\vec{x}_\alpha) \log b_\alpha(\vec{x}_\alpha) + \sum_i (d_i - 1) \sum_{x_i} b_i(x_i) \log b_i(x_i) \end{aligned}$$

Therefore, we have completely eliminated the $b(\vec{x})$ expressions and will from now on work only with the marginals $b_i(x_i)$ and $b_\alpha(\vec{x}_\alpha)$ (again, polynomially many instead of exponentially many).

Minimizing Gibbs Free Energy (or, equivalently, the KL-divergence between b and p)

From the previous section, the problem now becomes:

Minimize:

$$\begin{aligned} \tilde{D}(b \parallel p) \quad := \quad & -\textstyle\sum_\alpha \sum_{\vec{x}_\alpha} b_\alpha(\vec{x}_\alpha) \log f_\alpha(\vec{x}_\alpha) \\ & +\textstyle\sum_\alpha \sum_{\vec{x}_\alpha} b_\alpha(\vec{x}_\alpha) \log b_\alpha(\vec{x}_\alpha) \\ & -\textstyle\sum_i (d_i - 1) \sum_{x_i} b_i(x_i) \log b_i(x_i) \end{aligned} \tag{2.3}$$

as an expression in variables $b_i(x_i)$ and $b_\alpha(\vec{x}_\alpha)$ (for every state of x_i or $\vec{x}_\alpha$ separate variable! These should not be thought of as functions, but as parameters).

Subject to:

$$\sum_{x_i} b_i(x_i) = 1 \qquad \forall i \tag{2.4}$$

$$\sum_{\vec{x}_\alpha} b_\alpha(\vec{x}_\alpha) = 1 \qquad \forall \alpha \tag{2.5}$$

$$b_i(x_i) = \sum_{x_{\alpha \setminus i}} b_\alpha(\vec{x}_\alpha) \qquad \forall \alpha, i \in \alpha, x_i \tag{2.6}$$

The conditions (2.4,2.5) assure that the parameters are marginal probabilities, and condition (2.6) (consistency) assures that the variable and factor marginals are 'compatible' (in the natural way). The $\tilde{D}(b \parallel p)$ is KL-divergence of probability mass functions b and p *assuming* the factor graph G is a tree.

Lagrangian Minimization We will now minimize the expression (2.3) (constrained minimization) using the technique of Lagrangian multipliers (to turn it into an unconstrained minimization problem). We thus introduce Lagrangian multipliers λ_i for each condition (2.4), λ_α for each condition (2.5), and $\lambda_{\alpha i}(x_i)$ for each

condition (2.6).

The Lagrangian then becomes:

$$\begin{aligned}
&\Lambda(b_i, b_\alpha, \lambda_i, \lambda_\alpha, \lambda_{\alpha i}(x_i)) = \\
&\quad - \textstyle\sum_\alpha \sum_{\vec{x}_\alpha} b_\alpha(\vec{x}_\alpha) \log f_\alpha(\vec{x}_\alpha) + \\
&\quad + \textstyle\sum_\alpha \sum_{\vec{x}_\alpha} b_\alpha(\vec{x}_\alpha) \log b_\alpha(\vec{x}_\alpha) - \sum_i (d_i - 1) \sum_{x_i} b_i(x_i) \log b_i(x_i) \\
&\quad + \textstyle\sum_i \lambda_i \left(\sum_{x_i} b_i(x_i) - 1 \right) \\
&\quad + \textstyle\sum_\alpha \lambda_\alpha \left(\sum_{\vec{x}_\alpha} b_\alpha(\vec{x}_\alpha) - 1 \right) \\
&\quad + \textstyle\sum_\alpha \sum_{i \in \alpha} \sum_{x_i} \lambda_{\alpha i}(x_i) \left(b_i(x_i) - \sum_{x_{\alpha \setminus i}} b_\alpha(\vec{x}_\alpha) \right)
\end{aligned} \tag{2.7}$$

Now the set of stationary points of Λ (that is where $\nabla \Lambda = 0$) is exactly the set of (possible) fixed points of BP equations, as is shown below.

Setting the partial derivatives of Λ w.r.t. $b_i(x_i)$ and $b_\alpha(\vec{x}_\alpha)$ to zero then yields:

$$\frac{\partial \Lambda}{\partial b_i(x_i)} : \qquad -(d_i - 1)(\log b_i(x_i) + 1) + \textstyle\sum_{\alpha \ni i} \lambda_{\alpha i}(x_i) + \lambda_i = 0 \tag{2.8}$$

$$\frac{\partial \Lambda}{\partial b_\alpha(\vec{x}_\alpha)} : \quad -\log f_\alpha(\vec{x}_\alpha) + \log b_\alpha(\vec{x}_\alpha) + 1 - \textstyle\sum_{i \in \alpha} \lambda_{\alpha i}(x_i) + \lambda_\alpha = 0 \tag{2.9}$$

which can be rewritten as:

$$\begin{aligned}
\log b_i(x_i) &= \tfrac{1}{d_i - 1} \textstyle\sum_{\alpha \ni i} \lambda_{\alpha i}(x_i) + \tfrac{\lambda_i}{d_i - 1} - 1 \\
\log b_\alpha(\vec{x}_\alpha) &= \log f_\alpha(\vec{x}_\alpha) + \textstyle\sum_{i \in \alpha} \lambda_{\alpha i}(x_i) - \lambda_\alpha - 1
\end{aligned}$$

The terms involving λ_i and λ_α plus the constants serve as normalizing constants (since the marginal probabilities b_i and b_α must sum to one). We will therefore use the "$\propto$" symbol in the following, dropping these terms. So we get:

$$\begin{aligned}
b_i(x_i) &\propto \left(\textstyle\prod_{\alpha \ni i} e^{\lambda_{\alpha i}(x_i)} \right)^{\frac{1}{d_i - 1}} \\
b_\alpha(\vec{x}_\alpha) &\propto f_\alpha(\vec{x}_\alpha) \textstyle\prod_{i \in \alpha} e^{\lambda_{\alpha i}(x_i)}
\end{aligned} \tag{2.10}$$

The BP Update Equations We will work with equations (2.10) to get them into a message-update form. We need a set of closed equations that we can solve in an iterative fashion.

Notation used in the figure: $V(a)$ is the set of all variables in clause a. For a variable i appearing in clause a, $C_a^u(i)$ is the set of all clauses in which i appears with the *opposite* sign than it does in a. $C_a^s(i)$ is the set of all clauses in which i appears with the *same* sign as it does in a except for a (thus $C_a^u(i)$, $C_a^s(i)$ and $\{a\}$ are disjoint and their union is the set of all clauses where i appears). $C^+(i)$ is the set of all clauses in which i appears unnegated. $C^-(i)$ is the set of all clauses in which i appears negated.

Messages from clauses to variables:

$$\eta_{a\to i} = \prod_{j\in V(a)\setminus i} \left[\frac{\Pi^u_{j\to a}}{\Pi^u_{j\to a} + \Pi^s_{j\to a}}\right]$$

Messages from variables to clauses:

$$\Pi^u_{i\to a} = \prod_{b\in C^s_a(i)} (1-\eta_{b\to i})$$
$$\Pi^s_{i\to a} = \prod_{b\in C^u_a(i)} (1-\eta_{b\to i})$$

Computing marginals from a fixed point η^* of the message equations:

$$\mu_i(1) \propto \prod_{b\in C^-(i)} (1-\eta^*_{b\to i})$$
$$\mu_i(0) \propto \prod_{b\in C^+(i)} (1-\eta^*_{b\to i})$$

$\mu_i(1)$ is the probability that variable i is positive in a random satisfying assignment, and $\mu_i(0)$ is the probability that it is negative.

Figure 2.2: Belief propagation update equations for SAT problem.

First define $n_{i\to\alpha}(x_i) := e^{\lambda_{\alpha i}(x_i)}$ to simplify the expressions. To deal with the $(.)^{\frac{1}{d_i-1}}$ form of $b_i(x_i)$, find $\nu_{\beta i}(x_i)$ such that $\lambda_{\alpha i}(x_i) = \sum_{\beta\ni i\setminus\alpha} \nu_{\beta i}(x_i)$. That is always possible since we are solving $A\vec{\nu}_{.i}(x_i) = \vec{\lambda}_{.i}(x_i)$ where $\vec{\lambda}_{.i}(x_i) = (\lambda_{\alpha i}(x_i))_{\alpha\ni i}$ and similarly for $\vec{\nu}_{.i}(x_i)$. The matrix A is all ones with zero on diagonal. It is clearly invertible, and thus $\vec{\nu}_{.i}(x_i)$ is uniquely defined by $\vec{\lambda}_{.i}(x_i)$. Now define $m_{\beta\to i}(x_i) := e^{\nu_{\beta i}(x_i)}$, and thus $n_{i\to\alpha}(x_i) = \prod_{\beta\ni i\setminus\alpha} m_{\beta\to i}(x_i)$.

With these definitions, we have

$$\begin{aligned} b_i(x_i) &\propto \left(\prod_{\alpha \ni i} n_{i\to\alpha}(x_i)\right)^{\frac{1}{d_i-1}} \\ &= \left(\prod_{\alpha \ni i} \prod_{\beta \ni i \setminus \alpha} m_{\beta\to i}(x_i)\right)^{\frac{1}{d_i-1}} \\ &= \prod_{\alpha \ni i} m_{\alpha\to i}(x_i) \\ &= n_{i\to\alpha}(x_i) m_{\alpha\to i}(x_i) \quad \text{for any } \alpha \ni i \\ b_\alpha(\vec{x}_\alpha) &\propto f_\alpha(\vec{x}_\alpha) \prod_{i\in\alpha} n_{i\to\alpha}(x_i) \end{aligned} \tag{2.11}$$

Now we need to solve for $n_{i\to\alpha}$ and $m_{\alpha\to i}$. The consistency condition (2.6) dictates that for any $\alpha, i \in \alpha, x_i \in Dom$:

$$b_i(x_i) = \sum_{\vec{x}_{\alpha\setminus i} \in Dom^{|\alpha|-1}} b_\alpha(\vec{x}_\alpha)$$

and therefore:

$$n_{i\to\alpha}(x_i) m_{\alpha\to i}(x_i) \propto \sum_{\vec{x}_{\alpha\setminus i}} f_\alpha(\vec{x}_\alpha) \prod_{j\in\alpha} n_{j\to\alpha}(x_j)$$

and if $n_{i\to\alpha}(x_i) \neq 0$:

$$m_{\alpha\to i}(x_i) \propto \sum_{\vec{x}_{\alpha\setminus i}} f_\alpha(\vec{x}_\alpha) \prod_{j\in\alpha\setminus i} n_{j\to\alpha}(x_j)$$

So the final BP update equations are, for $x_i \in Dom$:

$$n_{i\to\alpha}(x_i) \propto \prod_{\beta \ni i \setminus \alpha} m_{\beta\to i}(x_i) \tag{2.12}$$

$$m_{\alpha\to i}(x_i) \propto \sum_{\vec{x}_{\alpha\setminus i} \in Dom^{|\alpha|-1}} f_\alpha(\vec{x}_\alpha) \prod_{j\in\alpha\setminus i} n_{j\to\alpha}(x_j) \tag{2.13}$$

The final beliefs (that is marginals of the trial distribution $b(\vec{x})$) are obtained from the fixed point by (2.11). The BP update equations can be yet simplified if the factor graph models a SAT problem. The BP equations for SAT are given in Figure 2.2.

Chapter 3

Message-Passing and Local Heuristics as Decimation Strategies for Satisfiability

In 2002, Mézard, Parisi, and Zecchina [2002] introduced the Survey Propagation (SP) method to solve hard Boolean satisfiability (SAT) problem instances. The method is remarkably effective, capable of solving certain million variable instances that were far beyond the reach of previous techniques. Even more importantly, SP, which can be viewed as a form of belief propagation (BP), is the first successful example of the use of a probabilistic reasoning technique to solve a purely combinatorial search problem. SP is based on the so-called 1-RSB cavity method from statistical physics. The method is still far from well-understood, but in recent years, we are starting to see results that provide important insights into its workings.

The SP method solves the SAT problem using the "decimation" process, which assigns a truth value to one variable (or a few variables) of F and simplifies F, obtaining a smaller formula on $n-1$ variables. SP repeatedly decimates the formula in this manner, until a simplified instance is obtained that is easily solved by ex-

isting SAT solvers, such as `Walksat` [Selman et al., 1996]. The decimation process can fail in that it may assign a variable the wrong value, inadvertently eliminating all remaining satisfying assignments. The remarkable property of SP is that it can take a million variable, hard random 3-SAT instance and set 40%-50% of the variables, without making any "mistake." The resulting (satisfiable) subformula is then easily solved by `Walksat`. How does SP select a variable for decimation? Intuitively speaking, it considers the space of all satisfying assignments, and estimates the marginal probability of each variable being TRUE (or FALSE). It then selects the variable and the value assignment that has the highest marginal probability, thus, in a sense, preserving the largest number of satisfying assignments in the subproblem. As it turns out, SP does not directly consider the space of satisfying assignments but rather the space of "covers," which are special combinatorial objects representing clusters of solutions. It was shown by Braunstein and Zecchina [2004] and Maneva et al. [2005] that the SP method can be viewed as a form of belief propagation (BP) on a new constraint satisfaction problem derived from the original SAT instance, in which the objects of interest correspond to the covers of the original SAT problem (the derivation is also presented in Chapter 4).

The introduction of SP with its remarkable effectiveness has created the impression that to solve hard combinatorial instances via decimation, SP is required and BP would have little success. This was not explicitly stated in the literature but has become a fairly common view. However, we show that BP can be almost as effective as SP. In particular, we show that for random k-SAT formulas over a wide range of parameter settings, BP works just as well as SP when performing decimation. For a preview of these results, see Figure 3.3. (To remedy some of the convergence difficulties of BP on loopy factor graphs, we use a damped version BP^{κ}.) This finding is good news in terms of the use of probabilistic techniques for

solving combinatorial problems. SP equations need to be developed in a specialized manner for each individual combinatorial problem. For example, different equations have been developed for k-SAT, graph coloring, and vertex cover problems over a series of papers. At this point there is no generic recipe for writing the SP equations for arbitrary combinatorial problems, and one needs an understanding of the 1-RSB cavity method from statistical physics to do this for each problem. The BP equations on the other hand are quite generic and can be written down directly from the factor graph representing the combinatorial problem in question. In practical terms, our first message is that in further exploring probabilistic methods for solving combinatorial problems, BP holds a promise that is quite similar to that of SP. In particular, in the development of hybrid search methods, e.g., to boost SAT solvers, one should not rule out BP methods before going to SP.

We also compare the decimation strategies of BP, BP$^\kappa$, and SP, with those found in more traditional approaches towards SAT solving. A highly successful class of modern SAT solvers is based on the so-called DPLL procedure [Davis and Putnam, 1960, Davis et al., 1962]. Such solvers perform essentially a backtrack search through the space of all truth assignments searching for a satisfying one. In the search, heuristics are used to select which variable to set next and to what value. Each branch of the backtrack search tree corresponds to a decimation strategy. Our experiments show that the heuristics employed in modern SAT solvers can in fact solve some random k-SAT instances through pure decimation (no backtracking) but the process, as one might expect, quickly fails on harder instances. We thus obtain a strict order in terms of decimation strategies, from the least effective one, which is a random variable setting, to local heuristics used in modern SAT solvers, to BP, BP$^\kappa$, and finally, SP.

As noted above, the decimation process can fail because one may inadvertently

create an unsatisfiable subproblem. Consider, for example, the Boolean constraint C defined as (x_1 `or` $\neg x_2$ `or` x_3). If the decimation heuristic assigns a value to both x_1 and x_2, C is either satisfied and disappears, or reduced to the "unit" clause x_3. If another clause C' similarly creates the unit clause $\neg x_3$, the subproblem becomes unsatisfiable. This observation suggests that unit clauses may play an important role in determining the success or failure of a decimation strategy. Indeed, as we show, what distinguishes message passing based heuristics from standard search heuristics on the hardest random 3-SAT instances is that the former virtually avoid the creation of unit clauses.

Finally, we do see that despite its promising performance, BP$^\kappa$ doesn't quite match SP's performance on the hardest set of random instances. We investigate this behavior in terms of an indirect measure: the evolution of the hardness of the resulting formula as decimation proceeds. Our results reveal an interesting phenomenon, where both BP$^\kappa$ and SP start by gradually making the formula easier, but while SP continues to do so, BP$^\kappa$ soon makes the formula much harder to solve than it originally was. Such studies have not been performed earlier to our knowledge, and shed light into the not-so-well understood differences between BP and SP for solving CSPs.

3.1 Solving SAT by Decimation

The computational hardness of SAT coupled with its numerous applications have led to the development of dozens of progressively faster SAT solvers, which essentially fall into two categories: branch-and-backtrack search (i.e., DPLL-based methods) and local search. Both of these work with partial or sub-optimal candidate solutions, and attempt to iteratively improve these candidate solutions until a

satisfying assignment is found. A key aspect of both is the ability to make possibly wrong decisions early on in the search and later rectify the mistakes as the search progresses.

A conceptually much simpler method of finding a solution to a (satisfiable) formula F is **decimation**: select a literal ℓ of F according to some heuristic, set $\ell = \text{TRUE}$, simplify F by removing all clauses containing ℓ and shrinking those containing $\neg\ell$, and repeat. Decimation is said to *fail* if one ends up with a subformula F' of F that contains an empty (and thus unsatisfiable) clause. To be of any interest, the heuristic used for decimation must, in particular, satisfy the property that if there is a unit clause ℓ in F', then the heuristic suggests that ℓ (or some other unit literal) be set to TRUE. Since there is no "repair" mechanism in decimation, its success as a solution technique for SAT relies critically on the correctness of every single variable setting made during the process until a solution is found. As one might expect, decimation does not work very well in solving many formulas of interest. However, when one considers random 3-SAT formulas, the picture is quite positive. Analysis of decimation with various local heuristics has provided formal lower bounds on the satisfiability threshold [e.g. Achlioptas et al., 2005, Hajiaghayi and Sorkin, 2003, Kaporis et al., 2006]. Further, decimation with SP as the guiding heuristic [Mézard et al., 2002] has turned out to not only work surprisingly well for 3-SAT, it is the best method that we know of for this problem.

3.1.1 Belief Propagation & Survey Propagation

Belief propagation (BP) [Pearl, 1988] is an iterative algorithm for computing marginal probabilities of the nodes of a graphical model such as a Bayesian network. It works by iteratively solving a set of mutually recursive equations on variables that represent *messages* among the nodes of the graphical model. In the

context of SAT, one can think of a CNF formula F with n variables and m clauses as a factor graph G_F with n variable nodes taking value in $\{0, 1\}$ and m factors. Each variable node of G_F corresponds naturally to a variable of F. Each factor f_C corresponds to a clause C of F and is connected to the variable nodes corresponding to the variables appearing in C. f_C evaluates to 1 for a certain valuation of the variable nodes iff C evaluates to TRUE for the corresponding truth assignment to the variables.

We are interested, for example, in computing $\Pr[x_1 = 1]$ for our example formula F: this represents the marginal probability of x_1 being 1 when all solutions to F are chosen with equal probability. Equivalently, it is the fraction of solutions of F that have $x_1 = 1$. We use the term **magnetization** to refer to the difference between the marginal probabilities of a variable or literal being TRUE and it being FALSE. We refer the reader to standard texts [e.g. Pearl, 1988] for details of the iterative equations used to compute such marginal probabilities.

The BP iterations have been proved to converge and provide accurate answers essentially only when applied to problems with no circular dependencies, such as SAT instances with no loops in the factor graph. Empirically, BP has been shown to provide very good approximations even in some domains which do not satisfy this condition [Murphy et al., 1999]. Unfortunately, the number of domains where this is true is rather small, and BP fails on many problems of interest as well as on hard random k-SAT instances. Making progress in this direction is an open problem with active interest [e.g., Hsu and McIlraith, 2006, Pretti, 2005].

Survey propagation (SP) is a similar iterative method designed specifically for solving hard SAT instances, and is in fact the most successful method when dealing with random k-SAT instances. While introduced initially from a very different perspective, namely the cavity method in statistical physics [Mézard et al., 2002],

it was later shown to be an instance of BP applied to a modified problem [Braunstein and Zecchina, 2004, Maneva et al., 2005]. This modified problem is that of finding "covers" of the formula rather than satisfying assignments. The notion of covers [Maneva et al., 2005] is based on generalized assignments to Boolean variables, i.e., assignments in $\{0, 1, *\}$, and each such cover is supposed to represent a whole set of assignments that are close to each other in Hamming distance (loosely called "clusters"). The iterative equations of SP thus compute marginal probabilities of a variable being 0, 1, or $*$ in the covers of F. Covers have many interesting properties; we refer the reader to Maneva et al. [2005], Kroc et al. [2007] for details.

3.1.2 Convergent BP

As mentioned above, BP equations often do not converge to a fixpoint for hard enough problem instances. Moreover, problem instances that we study have tens or hundreds of thousands of variables and clauses, which is several orders of magnitude more than what off-the-shelf probabilistic inference programs can handle. Even improvements of the belief propagation technique that allow it to be used on a wider set of problems, such as Generalized Belief Propagation [Yedidia et al., 2000] or Loop Corrected Belief Propagation [Mooij et al., 2007], are not scalable enough for our purposes. The problem of very slow convergence on hard instances seems to plague also approaches based on other methods for solving the BP equations than the simple iteration scheme, such as the convex-concave procedure introduced by Yuille [2002]. Finally, in our context, the speed requirement is accentuated by the need to use marginal estimation repeatedly during the decimation process.

We consider a parameterized variant of BP which is guaranteed to converge when this parameter is small enough, and which imposes no additional computa-

tional cost over the traditional BP[1]. As we will shortly see, this "damped" variant of BP provides much more useful information than BP iterations terminated without convergence, and, surprisingly, performs very well as a decimation heuristic for random k-SAT. We refer to this particular way of damping the BP equations as $\mathbf{BP}^\kappa$, where $\kappa \geq 0$ is a real parameter that controls the extent of damping in the iterative equations. The iterative update equations for BP^κ are given in Figure 3.1. The only difference from the normal BP equations is the exponent κ in the updates for the Π messages.

The role of the parameter κ is to damp oscillations of the message values by pushing the variable-to-clause messages, Π, closer to 1. Intuitively speaking, the damping is realized by the function $y = x^\kappa$ for $\kappa < 1$. For inputs x that are positive and less than one, the function increases their value, or sets them to 1 in the case of $\kappa = 0$. As a result, after normalization, the Π values are less extreme. For $\kappa = 0$, we can obtain a probabilistic interpretation of the algorithm reminiscent of a local heuristic for SAT solving:

Proposition 2. *The system of BP^κ equations for $\kappa = 0$ converges in one iteration for any starting point, and the following holds for the resulting values μ_i (see Figure 3.1 for notation):*

$$\mu_i(1) \propto \prod_{b \in C^-(i)} \left(1 - 2^{-(|V(b)|-1)}\right)$$

$$\mu_i(0) \propto \prod_{b \in C^+(i)} \left(1 - 2^{-(|V(b)|-1)}\right)$$

Proof. For any initial starting point η_0 (with values in $[0, 1]$), the first iteration sets all $\Pi^u = 1$ and $\Pi^s = 1$. This means $\eta_{a \to i} = (\frac{1}{2})^{|V(a)|-1}$ for all clauses a containing variable i. This is the fixed point η^*, because applying the updates again yields the same values. The rest follows directly from the form of the μ_i equations in

[1]Similar but distinct parametrization was proposed by Pretti [2005].

Notation used in the figure: $V(a)$ is the set of all variables in clause a. For a variable i appearing in clause a, $C_a^u(i)$ is the set of all clauses in which i appears with the *opposite* sign than it does in a. $C_a^s(i)$ is the set of all clauses in which i appears with the *same* sign as it does in a except for a (thus $C_a^u(i)$, $C_a^s(i)$ and $\{a\}$ are disjoint and their union is the set of all clauses where i appears). $C^+(i)$ is the set of all clauses in which i appears unnegated. $C^-(i)$ is the set of all clauses in which i appears negated.

Messages from clauses to variables:

$$\eta_{a\to i} = \prod_{j\in V(a)\setminus i} \left[\frac{\Pi^u_{j\to a}}{\Pi^u_{j\to a} + \Pi^s_{j\to a}}\right]$$

Messages from variables to clauses:

$$\Pi^u_{i\to a} = \left(\prod_{b\in C^s_a(i)} (1-\eta_{b\to i})\right)^\kappa$$

$$\Pi^s_{i\to a} = \left(\prod_{b\in C^u_a(i)} (1-\eta_{b\to i})\right)^\kappa$$

Computing marginals from a fixed point η^* of the message equations:

$$\mu_i(1) \propto \prod_{b\in C^-(i)} (1-\eta^*_{b\to i})$$

$$\mu_i(0) \propto \prod_{b\in C^+(i)} (1-\eta^*_{b\to i})$$

$\mu_i(1)$ is the probability that variable i is positive in a random satisfying assignment, and $\mu_i(0)$ is the probability that it is negative.

Figure 3.1: Modified belief propagation update equations, BP^κ.

Figure 3.1. □

The intuitive interpretation of the values of μ_i in Proposition 2 is the following: assuming independence of variable occurrences in clauses, the value $2^{-(|V(b)|-1)}$ can be interpreted as the probability that clause $b \in C(i)$ is unsatisfied by a random truth assignment if variable i is not considered. $\mu_i(1)$ is thus the probability that all clauses b in which variable i appears negated are satisfied, and analogously for $\mu_i(0)$. Since clauses not considered in the expressions for μ_i are satisfied by i itself, the resulting values of μ_i are proportional to the probability of all clauses where i appears being satisfied with the particular setting of variable i, when all other variables are set randomly. This is very local information, and depends only on what clauses the variable appears in negatively or positively. The parameter κ can thus be used to tune the tradeoff between the ability of the iterative system to converge and the locality of the information obtained from the resulting fixed point.

When $\kappa = 1$, BP^κ is identical to regular belief propagation. On the other hand, when $\kappa = 0$, the equations surely coverge in one step to a unique fixed point and the marginal estimates obtained from this fixed point have a clear probabilistic interpretation in terms of a local property of the variables (see Proposition 2 above). The κ parameter thus allows one to interpolate between regimes where the equations correspond to the original BP providing global information if the iterations converge ($\kappa = 1$), and regime where the iterations surely converge but provide only local information (for $\kappa = 0$).

The resulting marginals for $\kappa \neq 1$ do not correspond to the actual BP marginal estimates, but we believe that the information obtained retains relevance to the true marginals. This is shown in Figure 3.2, where the quality of the estimate of solution marginals obtained by running standard BP and cutting it off because of

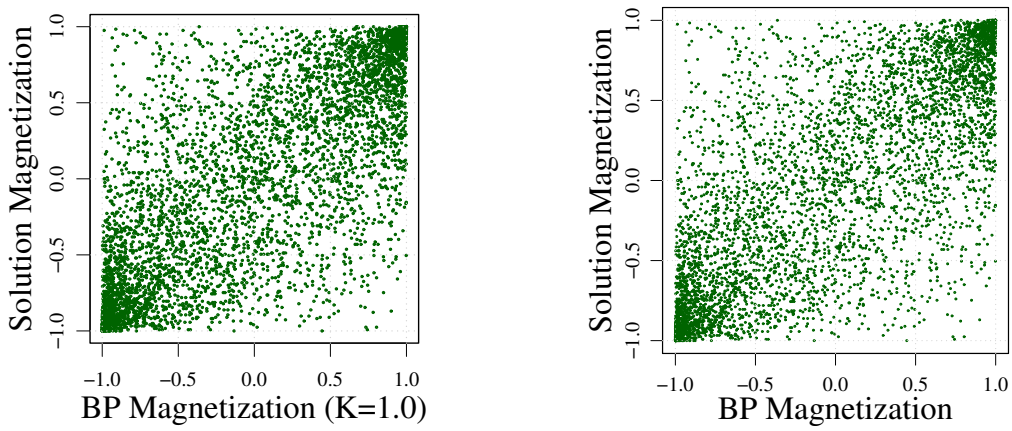


Figure 3.2: Quality of solution marginals computed by BP (left) and BP$^{\kappa}$ (right).

non-convergence (left pane) is compared to the quality of marginals as obtained by BP$^{\kappa}$ for $\kappa = 0.9$ (right pane), for a hard random 3-SAT formula with $5,000$ variables and $21,000$ clauses. The scatter-plots in both cases contrast the computed marginals with marginals obtained by *sampling* satisfying assignments for the formula and computing empirical marginals. What is actually plotted in Figure 3.2 is the magnetization for each variable, i.e., the difference between the probability of a variable being TRUEand the probability of it being FALSE. If the computed marginals did correspond exactly to the sampled solution marginals, the plot would be a diagonal line. The further away from the diagonal the points lie, the less accurate the estimate is. The figure shows that BP$^{\kappa}$ is more accurate in estimating the solution marginals than unconverged BP. This is especially true in the *extreme magnetization* regions at the top-right and bottom-left of the scatter-plots, which is the region of interest when using these estimates as a decimation heuristic. We refer the reader to Kroc et al. [2007] for a similar comparison between BP and SP, where SP is shown to also compute very good marginal estimates in the extreme magnetization regions.

3.2 Decimation Strategies

In this section, we consider various strategies that can be used as a heuristic for the decimation process for solving SAT instances. All of these heuristics give the highest priority to setting to TRUE any literal appearing in a *unit clause*; such a literal much be satisfied in any case in order to obtain a solution. When no unit clauses are present, we consider five alternatives:

1. Random: Choose a variable x uniformly at random and set it to TRUE or FALSE uniformly at random. We note that on random formulas, this is equivalent to the heuristic employed by the state-of-the-art SAT solver `Minisat` [Eén and Sörensson, 2005] before it encounters its first contradiction.

2. DPLL: Follow the heuristic used by the DPLL-based SAT solver `zChaff` [Moskewicz et al., 2001] before the first backtrack. This heuristic prefers the variable that occurs the most often, and sets it to TRUE or FALSE based on majority occurrence. If positive and negative occurrences are equally numerous, it goes for the polarity that creates the *most* unit literals.

3. BP: Compute variable magnetizations using BP (report failure if it does not converge). Set the variable with the most extreme magnetization accordingly.

4. BP$^\kappa$: Similar to BP, but with magnetizations obtained from BP$^\kappa$ (we used $\kappa = 0.9$).

5. SP: Similar to BP, but with magnetizations obtained from SP. In case SP finds the formula in a "paramagnetic" state where it can provide no useful information [see Mézard et al., 2002], the formula is easily solved by the local search SAT solver `Walksat`.

We evaluate these heuristics on random 3-SAT, 4-SAT, and 5-SAT problems, parameterized by the clause-to-variable ratio α. Recall that these instances become progressively harder as α increases; the computationally hardest instances

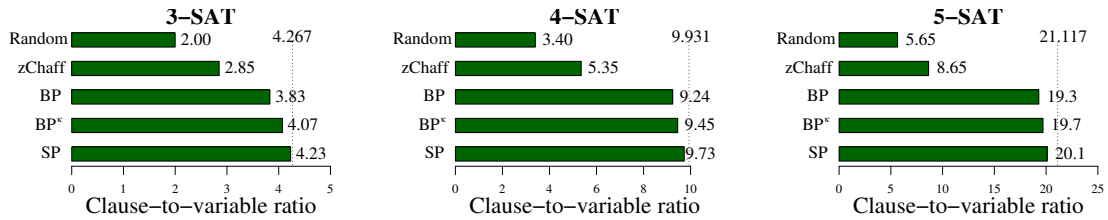


Figure 3.3: Various decimation heuristics on random 3-SAT, 4-SAT, and 5-SAT formulas

are found near the α value of 4.26 for 3-SAT, 9.93 for 4-SAT, and 21.12 for 5-SAT [Mertens et al., 2006]. For each heuristic and each $k = 3, 4, 5$, we measure the value of α up to which decimation with this heuristic solves at least 50% of the random formulas generated (we use formulas with $10,000$ variables). Figure 3.3 gives a summary of the results.

As one would expect, all five heuristics can solve these formulas for small enough α, but start to break down as α grows. The three propagation-based methods — BP, BP$^\kappa$, and SP — go much farther than the two local information heuristics — random and zChaff. While the zChaff heuristic solves 3-SAT instances for $\alpha \leq 2.85$, BP pushes this to 3.83 and SP as far as 4.23. A surprising finding is that κ-damping improves the performance of BP considerably, bringing it half-way from BP to SP. As k grows, i.e., one goes from 3-SAT to 4-SAT and 5-SAT, the difference in performance between propagation-based methods and local information methods becomes even more extreme. In particular, the random and zChaff heuristics stop working relatively much earlier.

Among various propagation-based heuristics themselves, BP interestingly appears to get closer to SP in performance as k increases. BP$^\kappa$ is able to push the limits of BP significantly further, for example, to $\alpha \leq 4.07$ for 3-SAT. Nevertheless, in all cases, SP is able to solve instances harder than the hardest instances solved by any other technique, such as those at $\alpha = 4.2$ for 3-SAT. This naturally raises

the question: what is it that SP is doing differently than these other heuristics? In the next section we will focus on identifying measures that distinguish SP from the other techniques, in particular, from BP^{κ}.

3.3 Observable Differences In Decimation Strategies

In this section we focus on identifying key differences between various heuristics as decimation progresses. Despite the obvious difference that one strategy solves a given instance while other does not, it is surprisingly difficult to identify features of the decimation process that would clearly distinguish between the strategies. For example, simple measures like the ratio of 2-clauses to 3-clauses in the decimated formula, or the frequency of positive vs. negative occurrences of variables, do not show a significant difference, especially amongst various message passing heuristics. We identify two measures that do show a clear difference, namely, how many unit clauses the heuristic generates and how hard-to-solve does the decimated formula become.

3.3.1 BP and SP Avoid Creating Unit Clauses

Quite surprisingly, our experiments reveal that a key difference between decimation based on message passing heuristics (both SP and BP) and other heuristics considered is that message passing heuristics avoid the creation of unit clauses. Since unit clauses allow for simplification of the formula, DPLL solvers and decimation strategies perform *unit propagation*, that is setting literals in unit clauses to true, possibly creating new unit clauses, until all unit clauses are resolved. This chaining

effect may lead to a contradiction, and while this is fine for a DPLL solver with a possibility of backtracking, the increased chance of contradiction may be fatal for decimation strategies. Minimizing the number of unit clauses is thus a desirable property of decimation strategies. Figure 3.4 shows the cumulative number of unit clauses generated as decimation progresses, for a random 3-SAT formula with $10,000$ variables at ratio $\alpha = 4.2$. The decimation performed by randomly setting variables is not careful about avoiding unit clauses, and indeed after setting 10% of the variables the decimation is dominated by unit propagation, which can be seen by observing that the slope of the "Random" curve in the plot is one. That there is actually a contradiction (i.e., the decimated formula has already become unsatisfiable) is detected only after nearly 50% of the variables have been set, marked by the cross in the figure where the "Random" curve ends. The situation is similar for the DPLL heuristic, `zChaff`, but the contradiction is detected much sooner. BP inspired decimation results in a significantly lower number of unit clauses (BP^κ was used for $\kappa = 0.9$ because standard BP does not converge for such α), at least till roughly 40% of the variables have been set, which is the point where SP usually hands the formula over to `Walksat`. Of course, when enough variables are fixed, it becomes nearly impossible to avoid unit propagation using any heuristic, and BP^κ shows this trend after 40% of the variables are set. Finally, SP inspired decimation results in virtually no unit propagation. There were only three unit clauses created, and this number diminishes for larger formulas. Note that the ordering of the curves from left to right corresponds to the order in Figure 3.3, confirming that avoiding unit propagation is desirable when solving formulas by decimation (in contrast to solving by search with backtracking).

Creating no unit clauses means that no trivial contradiction (i.e., two unit clauses requiring a variable to be set opposing ways) is derived. But how do BP^κ

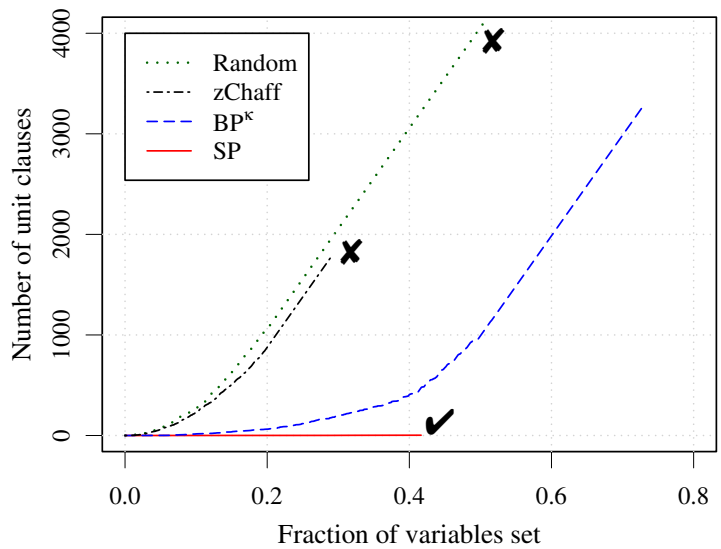


Figure 3.4: Cumulative number of unit clauses generated during decimation

and SP manage to avoid unit propagation for such a long time during decimation? The following proposition studies this question in the ideal case, and shows that if the message passing method was able to compute marginals exactly, unit propagation would be naturally avoided. Since BP/SP inspired decimation always fixes variables with the highest magnetization, Proposition 3 suggests that unit clauses are highly likely to be avoided. Of course, marginals computed by BP and SP on complex problems are not exact, but as remarked in Section 3.1.2, both BP$^\kappa$ and SP do a very good job of estimating the extreme marginals.

Proposition 3. *Let μ be the exact solution marginals of a formula F. That is, for each variable x, $\mu(x = 0) = P[\sigma(x) = 0 \mid \sigma$ is a solution$]$, where σ is some variable valuation with a uniform prior, and analogously for $\mu(x = 1)$. If the maximum marginal for 0 or 1 is unique, i.e., the most magnetized literal is unique, then setting this literal to* TRUE *will not create any unit clauses.*

Proof. Recall that magnetization of a literal is the difference between its marginal probabilities of being TRUE and FALSE. Let x be the unique most magnetized

literal of F. For contradiction, assume that setting $x = 1$ creates a unit clause (y). This means that there must have been a binary clause $(\neg x \vee y)$ in F. Now any solution σ of F must, by definition, have the property that every clause has at least one satisfying literal under valuation σ. In particular, because of our binary clause, it must be the case that $\sigma(x) = 1 \Rightarrow \sigma(y) = 1$. But this means that $\mu(x = 1) \leq \mu(y = 1)$ and $\mu(y = 0) \leq \mu(x = 0)$, and thus $\mu(y = 1) - \mu(y = 0) \geq \mu(x = 1) - \mu(x = 0)$. This is a contradiction with x being the unique most magnetized literal. $\square$

An analogous proposition can be stated for any Constrain Satisfaction Problem, not only SAT. In particular, it follows by a similar analysis that decimation by exact cover marginals tries to avoid creation of unit clauses. This, coupled with the observation in Kroc et al. [2007] that SP is remarkably accurate in computing the cover marginals, explains why SP creates so few unit clauses.

3.3.2 Evolution of Problem Hardness

So far we have seen that quite clearly, decimation based on probabilistic reasoning is better than decimation based on standard DPLL heuristics in the domain of random SAT instances. Furthermore, compared to BP$^\kappa$, SP inspired decimation is able to solve formulas much closer to the phase transition threshold, where the problem hardness quickly increases with α. We now focus on what happens during the decimation process on these hard formulas, digging deeper into the difference between BP$^\kappa$ and SP.

In order to understand the gradual effect of decimation as more and more variables are set, we study the evolution of the problem hardness in terms of how difficult it is to solve the decimated formula using `Walksat`, a local search SAT solver suitable for random SAT instances. The hardness is measured as

the average number of variable-flips required by `Walksat` to solve the decimated formula generated by SP or BP$^\kappa$, with $\kappa = 0.9$. (Standard BP equations did not converge, and the other heuristics quickly made the problem much harder.) Figure 3.5 summarizes the general trend through the results on a 5000 variable 3-SAT formula at $\alpha = 4.2$. The plot shows the hardness measure on the y-axis, and the fraction of variables set on the x-axis. The solid lines correspond to decimation by BP$^\kappa$ and SP only, while the dashed lines correspond to first setting 20% of the variables using one strategy, and then continuing with the other strategy.

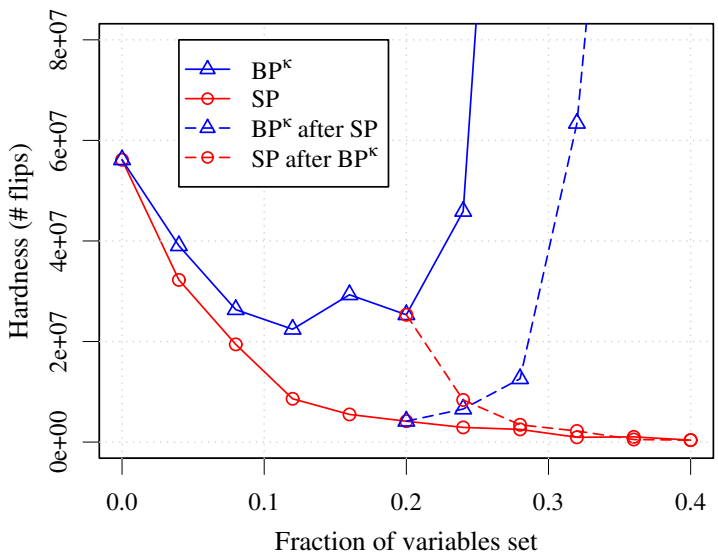


Figure 3.5: Hardness of subformulas created by BP$^\kappa$ and SP

We see that the decimated formulas generated by BP$^\kappa$ and SP behave quite differently with respect to their computational hardness. Initially, both SP and BP$^\kappa$ keep making the formula easier to solve, in this case till approximately 12% of the variables have been set. After this point, SP continues to make the formula easier to solve, while BP$^\kappa$ starts increasing the hardness, which eventually surpasses the hardness of the original formula. Finally, the dashed lines show that this behavior is quite robust: running BP$^\kappa$ on an SP-decimated formula quickly makes the formula much harder, while running SP on a BP$^\kappa$-decimated formula reduces

the hardness, and eventually solves the formula.

As seen in the figure, BP^κ starts going "wrong" when approximately 12% of the variables are set. At that point, it would be helpful to be able to identify variables in the formula that SP sets, making the formula yet easier. It turns out that the proper variables are indistinguishable by several measures commonly used in local heuristics, such as the number of positive and negative occurrences in 2- and 3-clauses. This suggests that local heuristics, looking no deeper than these easy-to-obtain statistics, are unlikely to follow SP's path in making the formula easier by further decimation. On the other hand, the SP- and BP^κ-decimated formulas do eventually get more and more different in terms of their computational hardness. At this point it is unclear exactly why this happens.

3.4 Discussion

Survey Propagation was introduced by statistical physicists as a fairly complex technique for solving random k-SAT instances very efficiently, and was later understood as essentially a form of BP, albeit on somewhat more complex combinatorial objects than solutions. This raised a natural question: can the original BP framework, more well-known to computer scientists, itself provide a good solution technique for hard random k-SAT instances? Our experimental results provide an affirmative answer to this question. Simply terminating the iterative equations of BP when they don't converge and using the current values as estimates of true marginals already pushes the limit of decimation much further than local information based heuristics commonly employed in backtrack search SAT solvers. By using a parameterized convergent form of BP, the limit is pushed even further. The gap between BP and SP narrows as k increases, and heuristics based on message-

passing in general begin to perform substantially better than other heuristics.

We also looked into concrete observable differences between heuristics as they decimation proceeds. While many simple measures like the ratio of 2-clauses to 3-clauses or the fraction of positive vs. negative occurrences do not show a significant difference, especially amongst various message-passing heuristics, SP reveals its robustness in terms of how hard it makes the formulas as variables are fixed. While an SP-decimated formula continuously becomes easier as decimation proceeds, other heuristics eventually make the simplified formula in fact much harder to solve (by `Walksat`). Another interesting property we identified is that message passing heuristics, unlike standard SAT heuristics, avoid unit propagation. This, we think, is a very unique feature and its success motivates investigation into a new class of variable selection strategies that may be of value to the SAT and CSP research communities.

Chapter 4

Survey Propagation Revisited

Survey Propagation (SP) is a new exciting algorithm for solving hard combinatorial problems. It was discovered by Mézard, Parisi, and Zecchina [2002], and is so far the only known method successful at solving random Boolean satisfiability (SAT) problems with 1 million variables and beyond in near-linear time in the hardest region. The SP method is quite radical in that it tries to approximate certain marginal probabilities related to the set of satisfying assignments. It then iteratively assigns values to variables with the most extreme probabilities. In effect, the algorithm behaves like the usual backtrack search methods for SAT (DPLL-based), which also assign variable values incrementally in an attempt to find a satisfying assignment. However, quite surprisingly, SP almost never has to backtrack. In other words, the "heuristic guidance" from SP is almost always correct. Note that, interestingly, computing marginals on satisfying assignments is actually believed to be much harder than finding a single satisfying assignment (#P-complete vs. NP-complete). Nonetheless, SP is able to efficiently approximate certain marginals and uses this information to successfully find a satisfying assignment.

SP was derived from rather complex statistical physics methods, specifically,

the so-called cavity method developed for the study of spin glasses. Close connections to belief propagation (BP) methods were subsequently discovered. In particular, it was discovered by Braunstein and Zecchina [2004] (later extended by Maneva, Mossel, and Wainwright [2005]) that SP equations are equivalent to BP equations for obtaining marginals over a special class of combinatorial objects, called covers. Intuitively, a cover provides a representative generalization of a cluster of satisfying assignments. The discovery of a close connection between SP and BP via the use of covers laid an exciting foundation for explaining the success of SP. Unfortunately, subsequent experimental evidence suggested that hard random 3-SAT formulas have, with high probability, only one (trivial) cover [Maneva et al., 2005]. This would leave all variables effectively in an undecided state, and would mean that marginals on covers cannot provide any useful information on how to set variables. Since SP clearly sets variables in a non-trivial manner, it was conjectured that there must be another explanation for the good behavior of SP; in particular, one that is not based on the use of marginal probabilities of variables in the covers.

In this chapter, we revisit the claim that hard random 3-SAT formulas do not have interesting non-trivial covers. In fact, we show that such formulas have large numbers of non-trivial covers. The main contribution discussed in this chapter is the first clear empirical evidence showing that in random 3-SAT problems near the satisfiability and hardness threshold, (1) a significant number of non-trivial covers exist; (2) SP is remarkably good at computing variable marginals based on covers; and (3) these cover marginals closely relate to solution marginals at least in the extreme values, where it matters the most for survey inspired decimation. As a consequence, we strongly suspect that explaining SP in terms of covers may be the correct path after all.

Note that (2) above is quite surprising for random 3-SAT formulas because such formulas have many loops. The known formal proof that SP computes cover marginals only applies to tree-structured formulas, which in fact have only a single (trivial) cover. Further, it's amazing that while SP computes such marginals in a fraction of a second, the next best methods of computing these marginals that we know of (via exact enumeration, or sampling followed by "peeling") require over 100 CPU hours.

Our experiments also indicate that cover marginals are more "conservative" than solution marginals in the sense that variables that are extreme with respect to cover marginals are almost certainly also extreme with respect to solution marginals, but not vice versa. This sheds light on why it is safe to set variables with extreme cover marginals in an iterative manner, as is done in the survey inspired decimation process for finding a solution using the marginals computed by SP.

In addition to these empirical results, we also revisit the derivation of the SP equations themselves, with the goal of presenting the derivation in an insightful form purely within the realm of combinatorial constraint satisfaction problems (CSPs). We describe how one can reformulate in a natural step-by-step manner the problem of finding a satisfying assignment into one of finding a cover, by considering related factor graphs on larger state spaces. The BP equations for this reformulated problem are exactly the SP equations for the original problem, as shown in Section 4.3.

4.1 Covers of CNF Formulas

A truth assignment to n variables can be viewed as a string of length n over the alphabet $\{0, 1\}$, and extending this alphabet to include a third letter "$*$" leads to a *generalized assignment*. A variable with the value $*$ can be interpreted as being "undecided," while variables with values 0 or 1 can be interpreted as being "decided" on what they want to be. We will be interested in certain generalized assignments called *covers*. Our formal definition of covers follows the one given by Achlioptas and Ricci-Tersenghi [2006]. Let variable x be called a *supported variable* under a generalized assignment σ if there is a clause C such that x is the only variable that satisfies C and all other literals of C are FALSE. Otherwise, x is called *unsupported*.

Definition 1. A generalized assignment $\sigma \in \{0, 1, *\}^n$ is a **cover** of a CNF formula F iff

1. every clause of F has at least one satisfying literal or at least two literals with value $*$ under σ, and
2. σ has no unsupported variables assigned 0 or 1.

The first condition ensures that each clause of F is either already satisfied by σ or has enough undecided variables to not cause any undecided variable to be forced to decide on a value (no "unit propagation"). The second condition says that each variable that is assigned 0 or 1 is set that way for a reason: there exists a clause that relies on this setting in order to be satisfied. For example, consider the formula $F \equiv (x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z)$. F has exactly two covers: 111 and $***$. This can be verified by observing that whenever some variable is 0 or $*$, then all non-$*$ variables are unsupported. Notice that the string

of all $*$'s always satisfies the conditions in Definition 1; we refer to this string as the *trivial cover*.

Covers were introduced by Maneva et al. [2005] as a useful concept to analyze the behavior of SP, but their combinatorial properties are much less known than those of solutions. A cover can be thought of as a *partial assignment* to variables, where the variables assigned $*$ are considered unspecified. In this sense, each cover is a representative of a potentially large set of complete truth assignments, satisfying as well as not satisfying. This motivates further differentiation:

Definition 2. A cover $\sigma \in \{0, 1, *\}^n$ of F is a **true cover** iff there exists a satisfying assignment $\tau \in \{0, 1\}^n$ of F such that σ and τ agree on all values where σ is not a $*$, i.e., $\forall i \in \{1, \ldots, n\}(\sigma_i \neq * \implies \sigma_i = \tau_i)$. Otherwise, σ is a **false cover**.

A true cover thus generalizes at least one satisfying assignment. True covers are interesting to study when trying to satisfy a formula, because if there exists a true cover with variable x assigned 0 or 1, then there must also exist a satisfying assignment with the same setting of x.

One can construct a true cover $\sigma \in \{0, 1, *\}^n$ of F by starting with any satisfying assignment $\tau \in \{0, 1\}^n$ of F and generalizing it using a simple procedure called **$*$-propagation**.[1] The procedure starts by initially setting $\sigma = \tau$. It then repeatedly chooses an arbitrary variable unsupported under σ and turns it into a $*$, until there are no more unsupported variables. The resulting string σ is a true cover, which can be verified as follows. The satisfying assignment τ already satisfies the first condition in Definition 1, and $*$-propagation does not destroy this property. In particular, a variable on which some clause relies is never turned into a $*$. The second condition in Definition 1 is also clearly satisfied when $*$-propagation halts,

[1] This was introduced under different names as the peeling procedure or coarsening, e.g., by Maneva et al. [2005].

so that σ must be a cover. Moreover, since σ generalizes τ, it is a true cover. Note that $*$-propagation can, in principle, be applied to an arbitrary generalized assignment. However, unless we start with one that satisfies the first condition in the cover definition, $*$-propagation may not lead to a cover.

We end with a discussion of two insightful properties of covers. The first relates to "self-reducibility" and the second to covers for tree-structured formulas.

No self-reducibility. Consider the relation between the decision and search versions of the problem of finding a solution of a CNF formula F. In the decision version, one needs an algorithm that determines whether or not F has a solution, while in the search version, one needs an algorithm that explicitly finds a solution. The problem of finding a solution for F is *self-reducible*, i.e., given an oracle for the decision version, one can efficiently solve the search version by iteratively fixing variables to 1 or 0, testing whether there is still a solution, and continuing in this way. Somewhat surprisingly, this strategy does *not* work for the problem of finding a cover. In other words, an oracle for the decision version of this problem does not immediately provide an efficient algorithm for finding a cover. (The lack of self-reducibility makes it very hard to find covers as we will see below.) As a concrete example, consider the formula F described right after Definition 1. To construct a cover of F, we could ask whether there exists a cover with x set to 1. Since 111 is a cover (yet unknown to us), the decision oracle would say yes. We could then fix x to 1, simplify the formula to $(y \vee \neg z) \wedge (\neg y \wedge z)$, and ask whether there is a cover with y set to 0. This residual formula indeed has 00 as a cover, and the oracle would say yes. With one more query, we will end up with 100 as the values of x, y, z, which is in fact *not* a cover of F.

Tree-structured formulas. For tree-structured formulas without unit clauses, i.e., formulas whose factor graph does not have a cycle, the only cover is the trivial all-$*$ cover. We argue this using the connection between covers and SP shown by Braunstein and Zecchina [2004], which says that when generalized assignments have a uniform prior, SP on a tree formula F provably computes probability marginals of variables being 0, 1, and $*$ in covers of F. Moreover, it can be verified from the iterative equations for SP that with no unit clauses, zero marginals for any variable being 0 or 1, and full marginals for any variable being a $*$ is a fixed point of SP. Since SP provably has exactly one fixed point on tree formulas, it follows that the only cover of such formulas is the trivial all-$*$ cover.

4.2 Problem Reformulation: From Solutions To Covers

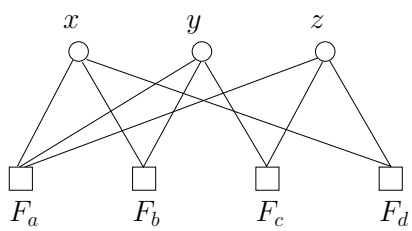
We now show that the concept of covers can be quite naturally arrived at when trying to find solutions of a CNF formula, thus motivating the study of covers from a purely generative perspective. Starting with a CNF formula F, we describe how F is transformed step-by-step into the problem of finding covers of F, motivating each step.

Although our discussion applies to any CNF formula F, we will be using the following example formula with 3 variables and 4 clauses to illustrate the steps:

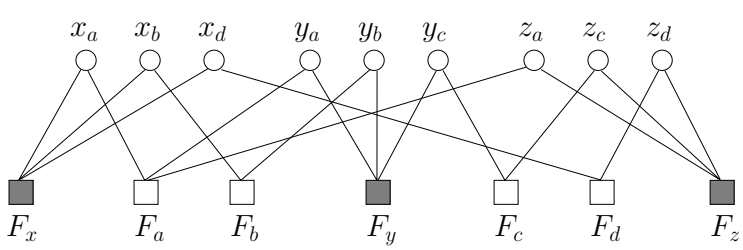
$$\underbrace{(x \vee y \vee \neg z)}_{a} \wedge \underbrace{(\neg x \vee y)}_{b} \wedge \underbrace{(\neg y \vee z)}_{c} \wedge \underbrace{(x \vee \neg z)}_{d}$$

Let N denote the number of variables, M the number of clauses, and L the number of literals of F.

Original problem. The problem is to find an assignment in the space $\{0,1\}^N$ that satisfies F. The factor graph for F has N variable nodes and M function nodes, corresponding directly to the variables $x, y, \ldots$ and clauses $a, b, \ldots$ in F (see e.g. Kschischang et al. [2001]). The factor graph for the example formula is depicted below. Here factors $F_a, F_b, \ldots$ represent predicates ensuring that the corresponding clause has at least one satisfying literal.



Variable occurrences. The first step in the transformation is to start treating every *variable occurrence* $x_a, x_b, y_a, y_b, \ldots$ in F as a separate unit that can be either 0 or 1. This allows for more flexibility in the process of finding a solution, since a variable can decide what value to assume in each clause separately. Of course, we need to add constraints to ensure that the occurrence values are eventually consistent: for every variable x in F, we add a constraint F_x that all occurrences of x have the same value. Now the search space is $\{0,1\}^L$, and the corresponding factor graph contains L variable nodes and $M + N$ function nodes (the original clause factors $F_a, F_b, \ldots$ and the new constraints $F_x, F_y, \ldots$).



At this point, we have not relaxed solutions to the original problem F: solutions to the modified problem correspond precisely to the original solutions, because variable occurrences are forced to be consistent. However, we moved this consistency

check from the syntactic level (variables could not be inconsistent simply by the problem definition) to the semantic level (we have special constraints to guarantee consistency).

Relaxing assignments. The next step is to relax the problem by allowing variable nodes to assume the special value "$*$". The semantics of $*$ is "undecided," meaning that the variable node is set neither to 0 nor to 1. The new search space is $\{0, 1, *\}^L$, and we must specify how our constraints handle the value $*$. Variable constraints $F_x, \ldots$ have the same meaning as before, namely, all variable nodes $x_a, x_b, \ldots$ have the same value for every variable x. Clause constraints $F_a, \ldots$ now have a modified meaning: a clause is satisfied if it contains at least one satisfying literal or at least *two* literals with the value $*$. The motivation here is to either satisfy a clause or leave enough "freedom" in the form of at least two undecided variables. (A single undecided variable would be forced to take on a particular value if all other literals in the clause were falsified.) With this transformation, the factor graph remains structurally the same, while the set of possible values for variable nodes changes.

The solutions to this modified problem do not necessarily correspond directly to solutions of the original one. In particular, if there are no unit clauses and all variables are set to $*$, the problem is already "solved" without providing any useful information.

Reducing freedom of choice. To distinguish variables that could assume the value $*$ from those that truly need to be fixed to either 0 or 1, we require that every non-$*$ variable has a clause that needs the variable to be 0 or 1 in order to be satisfied. The search space does not change, but we need to add constraints to implement the reduction in the freedom of choice.

Notice that this requirement is equivalent to "no unsupported variables" in the definition of a cover, and that the first requirement in that definition is fulfilled by the clause constraints. Therefore, we are now searching for covers of F. A natural way to represent the "no unsupported variable" constraint in the factor graph is to add for each variable x a new function node F'_x, connected to the variable nodes for x as well as for all other variables sharing a clause with x. This, of course, creates many new links and introduces additional short cycles, even if the original factor graph was acyclic. The following transformation step alleviates this issue.

Reinterpreting variable nodes. As the final step, we change the semantics of the variable nodes' values and of the constraints so that the "no unsupported variable" condition can be enforced without additional function nodes. The reasoning is that the simple $\{0, 1, *\}$ domain creates a bottleneck for how much information can be communicated between nodes in the factor graph. By altering the semantics of the variable nodes' values, we can improve on this.

The new value of a variable node x_a will be a pair $(r_{a\to x}, w_{x\to a}) \in \{(0,0),(0,1),(1,0)\}$, so that the size of the search space is still 3^L. We interpret the value $r_{a\to x}$ as a *request* from clause a to variable x with the meaning that a relies on x to satisfy it, and the value $w_{x\to a}$ as a *warning* from variable x to clause a that x is set such that it does not satisfy a. The values 1 and 0 indicate presence and absence, resp., of the request or warning. We can recover the original $\{0, 1, *\}$ values from these new values as follows: if $r_{a\to x} = 1$ for some a, then x is set to satisfy clause a; if there is no request from any clause where x appears, then x is undecided (a value of $*$ in the previous interpretation). The variable constraints $F_x, \ldots$ not only ensure consistency of the values of $x_a, x_b, \ldots$ as before, but also ensure the second cover condition as described below. The clause constraints $F_a, \ldots$ remain unchanged.

The variable constraint F_x is a predicate ensuring that the following two conditions are met:

1. if $r_{a\to x} = 1$ for *any* clause a where x appears, then $w_{x\to b} = 0$ for all clauses b where x appears with the *same* sign as in a, and $w_{x\to b} = 1$ for all b where x appears with the *opposite* sign. Since x must be set to satisfy a, this ensures that clauses that are unsatisfied by x do receive a warning.
2. if $r_{a\to x} = 0$ for *all* clauses a where x appears, then $w_{x\to a} = 0$ for all of them, i.e., no clause receives a warning from x.

To evaluate F_x, values $(r_{a\to x}, w_{x\to a})$ are needed only for clauses a in which x appears, which is exactly the set of variable nodes the factor F_x is connected to. Notice that the case $(r_{a\to x}, w_{x\to a}) = (1, 1)$ cannot happen due to condition 1 above. The conditions also imply that the variable occurrences of x are consistent, and in particular that two clauses where x appears with opposite signs (say a and b) cannot simultaneously request to be satisfied by x. This is because either $r_{a\to x} = 0$ or $r_{b\to x} = 0$ must hold due to condition 1.

The clause constraint F_a is a predicate stating that clause a issues a request to its variable x if and only if it receives warnings from all its other variables: $r_{a\to x} = 1$ iff $w_{y\to a} = 1$ for all variables $y \neq x$ in a. Again, F_a can be evaluated using exactly values from the variable nodes it is connected to.

When clause a issues a request to variable x (i.e., $r_{a\to x} = 1$), x must be set to satisfy a, thus providing a satisfying literal for a. If a does not issue any request, then according to the condition of F_a, at least two of a's variables, say x and y, must not have sent a warning. In this case, F_x and F_y state that each of x and y is either undecided or satisfies a. Thus the first condition in the cover definition holds in any solution of this new constraint satisfaction problem. The second condition also holds, because every variable x that is not undecided must have received a

request from some clause a, so that x is the only literal in a that is not FALSE. Therefore x is supported.

Let us denote this final constraint satisfaction problem by $P(F)$. (It is a function of the original formula F.) Notice that the factor graph of $P(F)$ has the same topology as the factor graph of F. In particular, if F has a tree factor graph, so does $P(F)$. Further, by the construction of $P(F)$ described above, its solutions correspond precisely to the covers of F.

4.2.1 Inference Over Covers

This section discusses an approach for solving the problem $P(F)$ with probabilistic inference using belief propagation (BP). It arrives at the survey propagation equations for F by applying BP equations to $P(F)$.

Since the factor graph of $P(F)$ can be easily viewed as a Bayesian Network [cf. Pearl, 1988], one can compute marginal probabilities over the set of satisfying assignments of the problem, defined as

$$\Pr[x_a = v \mid \text{all constraints of } P(F) \text{ are satisfied}]$$

for each variable node x_a and $v \in \{(0,0),(0,1),(1,0)\}$. The probability space here is over all assignments to variable nodes with uniform prior.

Once these solution marginals are known, we know which variables are most likely to assume a particular value, and setting these variables simplifies the problem. A new set of marginals can be computed on this simplified formula, and the whole process repeated. This method of searching for a satisfying assignment is called the **decimation procedure**. The problem, of course, is to compute the marginals (which, in general, is much harder than finding a satisfying assignment). One possibility for computing marginals is to use the belief propagation algorithm

[cf. Pearl, 1988]. Although provably correct essentially only for formulas with a tree factor graph, BP provides a good approximation of the true marginals in many problem domains in practice [Murphy et al., 1999]. Moreover, as shown by Maneva et al. [2005], applying the BP algorithm to the problem of searching for covers of F results in the SP algorithm. Thus, on formulas with a tree factor graph, the SP algorithm provably computes marginal probabilities over covers of F, which are equivalent to marginals over satisfying assignments of $P(F)$. When the formula contains loops, SP computes a loopy approximation to the cover marginals. The derivation of SP equations from the problem $P(F)$ is detailed in the next section.

4.3 Derivation of the SP Equations

Previous section shows how to formulate a constraint satisfaction problem $P(F)$ such that its solutions are exactly the covers of a formula F. Here we proceed to show how the belief propagation formalism applied to $P(F)$ (as described in Section 4.2.1) results in the survey propagation equations.

Review of BP. We assume familiarity with the general form of BP equations, as used for example by Neapolitan [2004] in Theorem 3.2. In short, BP uses *messages* to communicate information between nodes of the factor graph (between variable nodes $x_a, \ldots$ and function nodes $F_x, F_a, \ldots$). Each message is a function of one argument, which takes on the same values as the variable node end-point of the message. There are two kinds of messages: from variable nodes to function nodes (denoted by $\pi_{x \to F}(.)$), and from function nodes to variable nodes (denoted by $\lambda_{F \to x}(.)$). In a two-level Bayesian Network, π messages are computed by (piecewise) multiplying together the λ messages received on all other links. The λ messages are more complicated: they are sums across all possible worlds (values

for arguments of received π messages) of products of all-but-one π messages with the chosen arguments. In case of a deterministic system (which is our case: every world has probability of either 1 or 0), this is equivalent to sum of products of π messages with arguments that are compatible with each other as judged by the corresponding function node, F_x or F_a. Moreover, if a variable node only has two neighboring function nodes, then it merely passes received λ messages from one neighbor to the other. Since all variable nodes in $P(F)$ have degree two, we can safely ignore the existence of π messages and only focus on λ messages. Thus, every F_x node receives messages from F_a nodes (which we will denote by $\lambda_{a\to x}$) and and every F_a node receives messages from F_x nodes (denoted by $\lambda_{x\to a}$), both of which are functions of one argument, $(r_{a\to x}, w_{x\to a}) \in \{(0,0),(0,1),(1,0)\}$. The BP equations are constructed by considering the set of compatible variable node values given the one fixed value in the argument.

Let $C(x)$ be the set of all clauses containing variable x, and $V(a)$ the set of all variables appearing in clause a. Further, let $C_a^s(x)$ be the set of all clauses other than a where x occurs with the same sign as in a. Similarly define $C_a^u(x)$ to be the set of clauses where x occurs with the opposite sign as in a. Note that $C_a^s(x) \cup C_a^u(x) \cup \{a\} = C(x)$.

Equations for F_x. The equations for messages sent from a factor node F_x are given in Figure 4.1. For the argument value of $(1,0)$ the set of compatible values, as judged by F_x when $x_a = (1,0)$, is one where there can be requests from clauses where x appears with the same sign as in a (and no warnings sent to them), but there must be no requests from opposite clauses (and warning must be sent). Similarly for $(0,1)$, but here the roles of $C_a^s(x)$ and $C_a^u(x)$ are exchanged, plus the fact that x sends a warning to a means that it must be receiving a request from some opposite clause (which is accounted for by the "$-$" term). Finally, for the

$$\lambda_{x\to a}(1,0) = \prod_{b\in C_a^s(x)} (\lambda_{b\to x}(0,0) + \lambda_{b\to x}(1,0)) \prod_{b\in C_a^u(x)} \lambda_{b\to x}(0,1)$$

$$\lambda_{x\to a}(0,1) = \prod_{b\in C_a^s(x)} \lambda_{b\to x}(0,1) \left[\prod_{b\in C_a^u(x)} (\lambda_{b\to x}(0,0) + \lambda_{b\to x}(1,0)) - \prod_{b\in C_a^u(x)} \lambda_{b\to x}(0,0) \right]$$

$$\lambda_{x\to a}(0,0) = \left[\prod_{b\in C_a^s(x)} (\lambda_{b\to x}(0,0) + \lambda_{b\to x}(1,0)) - \prod_{b\in C_a^s(x)} \lambda_{b\to x}(0,0) \right] \prod_{b\in C_a^u(x)} \lambda_{b\to x}(0,1) + \prod_{b\in C(x)\setminus a} \lambda_{b\to x}(0,0)$$

Figure 4.1: BP equations for F_x

$$\lambda_{a\to x}(1,0) = \prod_{y\in V(a)\setminus x} \lambda_{y\to a}(0,1)$$

$$\lambda_{a\to x}(0,0) = \prod_{y\in V(a)\setminus x} (\lambda_{y\to a}(0,0) + \lambda_{y\to a}(0,1)) - \prod_{y\in V(a)\setminus x} \lambda_{y\to a}(0,1)$$

$$\lambda_{a\to x}(0,1) = \prod_{y\in V(a)\setminus x} (\lambda_{y\to a}(0,0) + \lambda_{y\to a}(0,1)) - \prod_{y\in V(a)\setminus x} \lambda_{y\to a}(0,1) + \sum_{y\in V(a)\setminus x} (\lambda_{y\to a}(1,0) - \lambda_{y\to a}(0,0)) \prod_{y'\in V(a)\setminus\{x,y\}} \lambda_{y'\to a}(0,1)$$

Figure 4.2: BP equations for F_a

value $(0,0)$, there are two possibilities: either at least one request is received from $C_a^s(x)$ (the first term in the sum, analogous to the expression for $(0,1)$), or there are no requests at all (the second term in the sum).

Equations for F_a. Figure 4.2 shows equations for messages sent from a factor node F_a. The argument value of $(1,0)$ is the easiest: a sends out a request if and only if all other variables send it a warning, so that the only compatible values are all $(0,1)$. The case of $(0,0)$ is the complement: a cannot receive a warning from all other variables, and since x does not send a warning, a does not send a request anywhere. The last case, $(0,1)$, is a little more complicated. The first part

is the same as before, but there needs to be a correction term to account for two extra possibilities: first it is now possible that a issues a request to some y if all other variables also send a warning (the positive term in the sum), and second it is not possible that all-but-one variable send a a warning and yet a does not issue a request to the last one (the negative term in the sum).

Deriving the SP equations. The expression for $\lambda_{a\to x}(0,1)$ can be simplified by assuming that $\lambda_{y\to a}(1,0) = \lambda_{y\to a}(0,0)$ for all y, in which case it reduces to the expression for $\lambda_{a\to x}(0,0)$. This assumption is crucial, but not very restrictive. Notice that it then follows that $\lambda_{x\to a}(1,0) = \lambda_{x\to a}(0,0)$ (by inspecting the appropriate expressions in Figure 4.1), and therefore the assumption keeps holding when iteratively solving the BP equations, provided it was true at the beginning.

The last step in the derivation is to rename and normalize the terms appropriately so as to "recognize" the SP equations in Figures 4.1 and 4.2. Let us define

$$\eta_{a\to x} \triangleq \prod_{y\in V(a)\setminus x} \frac{\lambda_{y\to a}(0,1)}{\lambda_{y\to a}(0,0)+\lambda_{y\to a}(0,1)}$$

and

$$\begin{aligned}
\Pi^u_{x\to a} &\triangleq \lambda_{x\to a}(0,1)\\
\Pi^0_{x\to a} &\triangleq \prod_{b\in C(x)\setminus a} \lambda_{b\to x}(0,0)\\
\Pi^s_{x\to a} &\triangleq \lambda_{x\to a}(0,0) - \Pi^0_{x\to a}
\end{aligned}$$

Notice that $\eta_{a\to x}$ is just a rescaled $\lambda_{a\to x}(1,0)$, and that the scaling factor $\lambda_{y\to a}(0,0) + \lambda_{y\to a}(0,1)$ equals $\Pi^u_{y\to a} + \Pi^s_{y\to a} + \Pi^0_{y\to a}$. Rescaling $\lambda_{a\to x}(0,0)$ and

$\lambda_{a\to x}(0,1)$ in the same way (and using the assumption that they are equal) yields $\lambda_{a\to x}(0,0) = \lambda_{a\to x}(0,1) = 1 - \eta_{a\to x}$. Finally, writing down the BP equations for $\lambda_{x\to a}(0,1)$ and $\lambda_{x\to a}(0,0)$ in terms of these new variables results in the familiar SP equations established in Braunstein et al. [2005]:

$$\eta_{a\to x} = \prod_{y\in V(a)\setminus x} \frac{\Pi^u_{y\to a}}{\Pi^u_{y\to a} + \Pi^s_{y\to a} + \Pi^0_{y\to a}}$$

$$\Pi^u_{x\to a} = \prod_{b\in C^s_a(x)} (1-\eta_{b\to x}) \left[1 - \prod_{b\in C^u_a(x)} (1-\eta_{b\to x})\right]$$

$$\Pi^s_{x\to a} = \prod_{b\in C^u_a(x)} (1-\eta_{b\to x}) \left[1 - \prod_{b\in C^s_a(x)} (1-\eta_{b\to x})\right]$$

$$\Pi^0_{x\to a} = \prod_{b\in C(x)\setminus a} (1-\eta_{b\to x})$$

In addition, the expressions for marginal probabilities computed by BP from a fixed point of the above equations can be shown, in a similar way, to be equivalent to the SP "bias" expressions.

4.4 Experimental Results

This section presents our main contributions. We begin by demonstrating that non-trivial covers do exist in large numbers in random 3-SAT formula, and then explore connections between SP, BP, and variable marginals computed from covers as well as solutions, showing in particular that SP approximates cover marginals surprisingly well.

4.4.1 Existence of Covers

Motivated by theoretical results connecting SP to covers of formulas, Maneva et al. [2005] suggested an experimental study to test whether non-trivial covers even ex-

ist in random 3-SAT formulas. They proposed a seemingly good way to do this (the "peeling experiment"), namely, start with a uniformly random satisfying assignment of a formula F and, while it has unsupported variables, $*$-propagate the assignment. When the process terminates, one obtains a (true) cover of F. Unfortunately, what they observed is that this process repeatedly hits the trivial all-$*$ cover, from which they concluded that non-trivial covers most likely do not exist for such formulas. However, it is known that near-uniformly sampling solutions of such formulas to start with is a hard problem in itself and that most sampling methods obtain solutions in a highly non-uniform manner [Wei et al., 2004]. Consequently, one must be careful in drawing conclusions from relatively few and possibly biased samples.

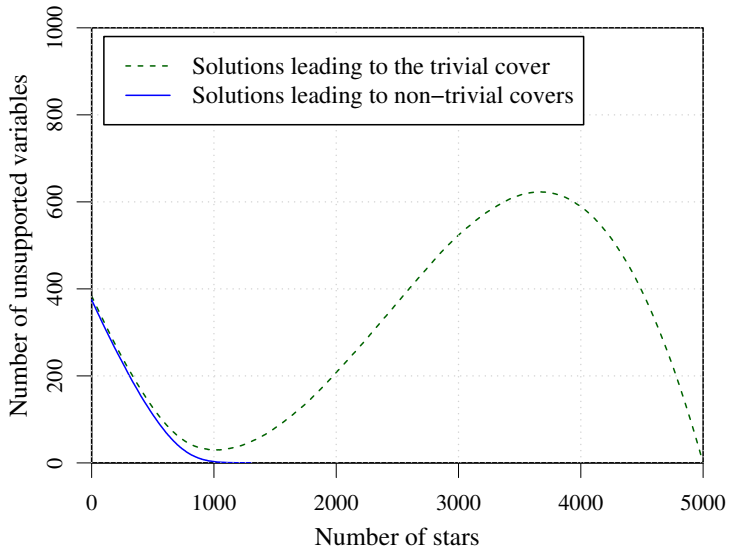


Figure 4.3: The peeling experiment, showing the evolution of the number of stars as $*$-propagation proceeds.

To understand this issue better, we ran the same peeling experiment on a 5000 variable random 3-SAT formula at clause-to-variable ratio 4.2 (which is close to the hardness threshold for random 3-SAT problems), but used `SampleSat` [Wei et al.,

2004] to obtain samples, which is expected to produce fairly uniform samples. Figure 4.3 shows the evolution of the number of unsupported variables at each stage as $*$-propagation is performed starting from a solution. Here, the x-axis shows the number of stars, which monotonically increases by $*$-propagation. The y-axis shows the number of unsupported variables present at each stage. As one moves from left to right following the $*$-propagation process, one hits a cover if the number of unsupported variables drops to zero (so that $*$-propagation terminates). The two curves in the plot correspond to solutions that $*$-propagated to the trivial cover and those that did not. In our experiment, out of 500 satisfying assignments used, nearly 74% led to the trivial cover; their average is represented by the top curve. The remaining 26% of the sampled solutions actually led to non-trivial covers; their average is represented by the bottom curve. Thus, when solutions are sampled near-uniformly, a substantial fraction of them lead to non-trivial covers.[2]

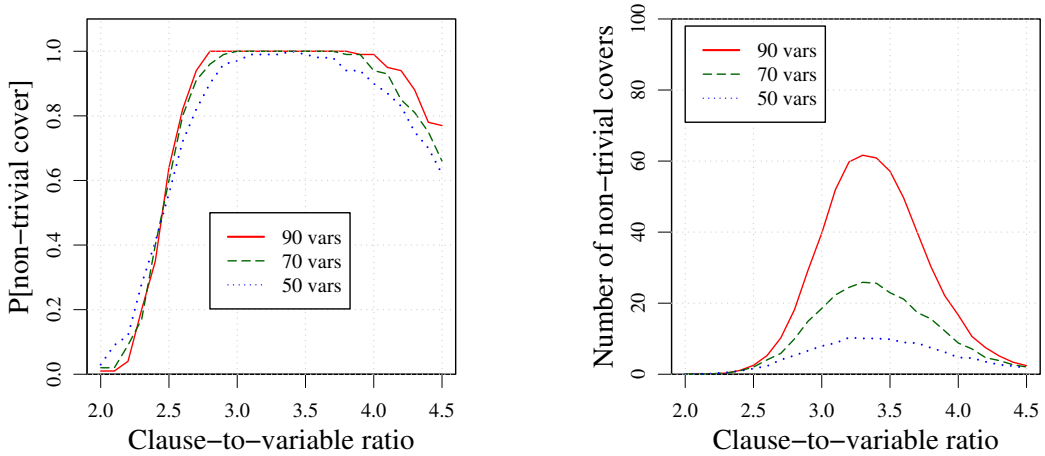


Figure 4.4: Non-trivial covers in random formulas. Left: existence probability. Right: average number.

An alternative method of finding covers is to create a new Boolean formula G whose solutions correspond go the covers of F. It turned out to be extremely

[2]That this was not observed by Maneva et al. [2005] can be attributed to the fact that SP was used to find satisfying assignments [Mossel, April 2007], resulting in highly non-uniform samples.

hard to solve G to find *any* non-trivial cover using state-of-the-art SAT solvers for number of variables as low as 150. So we confined our experiments to small formulas, with 50, 70 and 90 variables. We found all covers for such formulas with varying clause-to-variable ratios α. The results are shown in Figure 4.4, where each data point corresponds to statistics obtained from 500 formulas. The left pane shows the probability that a random formula, for a given clause-to-variable ratio, has at least one non-trivial cover (either true or false). The figure shows a nice phase transition where covers appear, at around $\alpha = 2.5$, which is surprisingly sharp given the small formula sizes. Also, the region where covers surely exist is widening on both sides as the number of variables increases, supporting the claim that non-trivial covers exist even in large formulas. The right pane of Figure 4.4 shows the actual number of non-trivial covers, with a clear trend that the number increases with the size of the formula, for all values of the clause-to-variable ratio. It is worth noting that the number of covers is very small compared to the number of satisfying assignments; e.g. for 90 variables and $\alpha = 4.2$, the expected number of satisfying assignments is $150,000$, while there are only 8 covers on average. Somewhat surprisingly, the number of false covers is almost negligible, around 2 at the peak, and does not seem to be growing nearly as fast as the total number of covers. This might explain why SP, although approximating marginals over all covers, is successful in finding satisfying assignments.

We also consider how the number of solutions that lead to non-trivial covers changes for larger formulas, as the number of variables N increases from 200 to 4000. The left pane of Figure 4.5 shows that an estimate of this number, in fact, *grows exponentially* with N. For each N, the estimate is obtained by averaging over 200 formulas at ratio 4.2 the following quantity: the fraction $p(N)$ of 20,000 sampled solutions that lead to a non-trivial cover, scaled up by the expected number of

solutions for N-variable formulas at this ratio, which is $(2 \times (7/8)^{4.2})^N \approx 1.1414^N$. The resulting number, $p(N) \times 1.1414^N$, is plotted on the y-axis of the left pane, with N on the x-axis. The right pane of Figure 4.5 shows the data used to estimate $p(N)$ along with its fit on the y-axis, with N on the x-axis again.

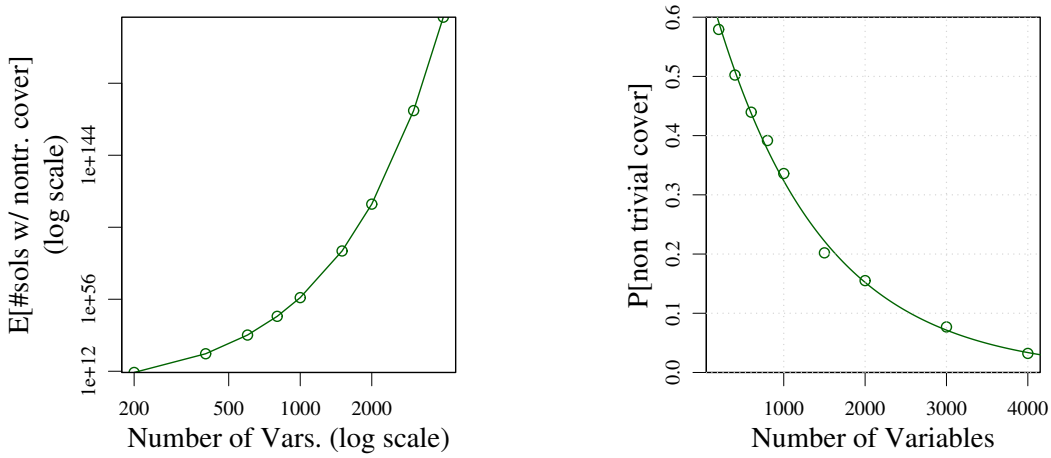


Figure 4.5: Left: Expected number of solutions leading to non-trivial covers (log-log scale). Right: Probability of a solution leading to a non-trivial cover.

Notice that the left pane is in log-scale for both axes, and clearly increases faster than a linear function. Thus, the estimated number of solutions that lead to non-trivial covers grows super-polynomially. In fact, performing a best fit for this curve suggests that this number grows exponentially, roughly as 1.1407^N. This number is indeed a vanishingly small fraction of the expected number of solutions (1.1414^N) as observed by Maneva et al. [2005], but nonetheless exponentially increasing. The existence of covers for random 3-SAT also aligns with what Achlioptas and Ricci-Tersenghi [2006] recently proved for k-SAT with $k \geq 9$.

4.4.2 SP, BP, and Marginals

We now study the behavior of SP and BP on a random formula in relation to solutions and covers of that formula. While theoretical work has shown that SP, viewed

as BP on a related combinatorial problem, provably computes cover marginals on tree-structured formulas, we demonstrate that even on random 3-SAT instances, which are far from tree-like, SP approximates cover marginals surprisingly well. We also show that cover marginals, especially in the extreme range, are closely related to solution marginals in an intriguing "conservative" fashion. *The combination of these two effects, we believe, plays a crucial role in the success of SP.* Our experiments also reveal that BP performs poorly at computing any marginals of interest.

Given marginal probabilities, we define the **magnetization** of a variable to be the difference between the marginals of the variable being positive and it being negative. For the rest of our experiments, we start with a random 3-SAT formula F with 5000 variables and 21000 clauses (clause-to-variable ratio of 4.2), and plot the magnetization of the variables of F in the range $[-1, +1]$.[3] The marginals for magnetization are obtained from four different sources, which are compared and contrasted against each other: (1) by running SP on F till the iterations converge; (2) by running BP on F but terminating it after 10,000 iterations because the equations do not converge; (3) by sampling solutions of F using `SampleSat` and computing an estimate of the positive and negative marginals from the sampled solutions (the solution marginals); and (4) by sampling solutions of F using `SampleSat`, $*$-propagating them to covers, and computing an estimate of the positive and negative marginals from these covers (the cover marginals). Note that in (4), we are sampling true covers and obtaining an estimate. An alternative approach is to use SP itself on F to try to sample covers of F, but the issue here is that the problem of finding (non-trivial) covers is not self-reducible to the decision problem of whether covers exist, as shown in Section 4.1. Therefore, it is not clear whether SP can be used to actually find a cover, despite it approximating the cover

[3]For clarity, the plots show magnetizations for one such formula, although the trend is generic.

marginals very well.

Recall that the SP-based decimation process works by identifying variables with *extreme magnetization*, fixing them, and iterating. We will therefore be interested mostly in what happens in the extreme magnetization regions in these plots, namely, the lower left corner $(-1, -1)$ and the upper right corner $(+1, +1)$.

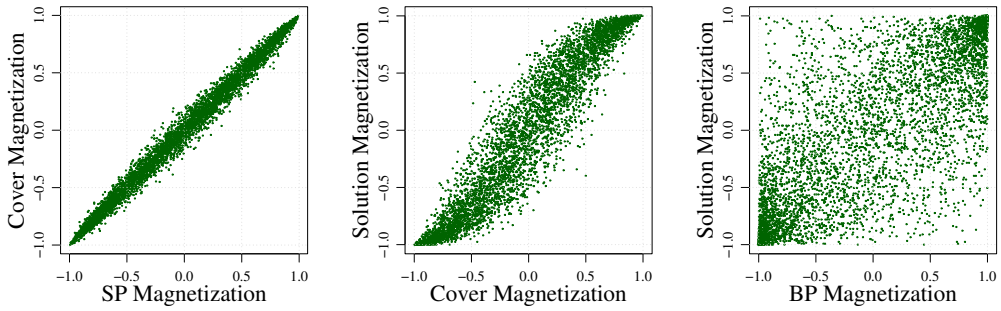


Figure 4.6: Magnetization plots. Left: SP vs. covers. Middle: covers vs. solutions. Right: BP vs. solutions.

In the left pane of Figure 4.6 we plot the magnetization computed by SP on the x-axis and the magnetization obtained from cover marginals on the y-axis. The scatter plot has exactly 5000 data points, with one point for each variable of the formula F. If the magnetizations on the two axes matched perfectly, all points would fall on a single diagonal line from the bottom-left corner to the top-right corner. The plot shows that *SP is highly accurate at computing cover marginals, especially in the extreme regions* at the bottom-left and top-right.

The middle pane of Figure 4.6 compares the magnetization based on cover marginals with the magnetization based on solutions marginals. This will provide an intuition for why it might be better to follow cover marginals rather than solution marginals when looking for a satisfying assignment.[4] We see an interesting "s-shape" in this plot, which can be interpreted as follows: fixing variables with ex-

[4]Of course, if solution marginals could be computed perfectly, this would not be an issue. In practice, however, the best we can hope is to approximately estimate marginals.

treme cover magnetizations is more conservative compared to fixing variables with extreme solution magnetizations. Which means that variables that are extreme w.r.t. cover-based magnetization are also extreme w.r.t. solution-based magnetization (but not necessarily vice-versa). Recall that the extreme region is exactly where decimation-based algorithms, that often fix a small set of extreme variables per iteration, need to be correct. Thus, etimates of cover marginals provide a safer heuristic for fixing variables than estimates of solution marginals.

As a comparison with BP, the right pane of Figure 4.6 shows BP magnetization vs. magnetization based on solution marginals for the same 5000 variable, 21000 clause formula. Since BP almost never converges on such formulas, we terminated BP after 10,000 iterations (SP took roughly 50 iterations to converge) and used the partially converged marginals obtained so far for computing magnetization. The plot shows that BP provides very poor estimates for the magnetizations based on solution marginals. (The points are equally scattered when BP magnetization is plotted against cover magnetization.) In fact, BP appears to identify as extreme many variables that have the opposite solution magnetization. Thus, when magnetization obtained from BP is used as a heuristic for identifying variables to fix, mistakes are often made that eventually lead to a contradiction, i.e. unsatisfiable reduced formula.

4.5 Discussion

A comparison between left and right panes of Figure 4.6 suggests that approximating statistics over covers (as done by SP) is much more accurate than approximating statistics over solutions (as done by BP). This appears to be because covers are much more coarse grained than solutions; indeed, even an exponentially large

cluster of solutions will have only a single cover as its representative. This cover still captures critical properties of the cluster necessary for finding solutions, such as backbone variables, which is what SP appears to exploit.

We also saw that the extreme magnetization based on cover marginals is more conservative than that based on solution marginals (as seen in the "s-shape" of the plot in the middle pane of Figure 4.6). This suggests that while SP, based on approximating cover marginals, may miss some variables with extreme magnetization, when it does find a variable to have extreme magnetization, it is quite likely to be correct. This provides an intuitive explanation of why the decimation process based on extreme SP magnetization succeeds with high probability on random 3-SAT problems without having to backtrack, while the decimation process based on BP magnetizations more often fails to find a satisfying assignment in practice.

BP and SP have been proven to compute exact marginals on solutions and covers, respectively, only for tree-structured formulas (with some simple exceptional cases). For BP, solution marginals on tree formulas are already non-trivial, and it is reasonable to expect it to compute a fair approximation of marginals on loopy networks (formulas). However, for SP, cover marginals on tree formulas are trivial: the only cover here is the all-$*$ cover. Cover marginals become interesting only when one goes to loopy formulas, such as random 3-SAT. In this case, as seen in our experiments, *it is remarkable that the SP computes a good approximation of non-trivial cover marginals for non-tree formulas.*

Chapter 5

Reasoning About Clusters

Message passing algorithms, in particular Belief Propagation (BP), have been very successful in efficiently computing interesting properties of succinctly represented large spaces, such as joint probability distributions. Recently, these techniques have also been applied to compute properties of discrete spaces, in particular, properties of the space of solutions of combinatorial problems. For example, for propositional satisfiability (SAT) and graph coloring (COL) problems, marginal probability information about the uniform distribution over solutions (or similar combinatorial objects) has been the key ingredient in the success of BP-like algorithms for these problems. Most notably, the survey propagation (SP) algorithm utilizes this information to solve very large hard random instances of these problems very close to the so-called phase transition region [Mézard et al., 2002, Braunstein et al., 2003]. The performance of SP is unmatched by any other known algorithm for these challenging combinatorial problems. Belief propagation can also be used to directly obtain, in addition to marginal probabilities, an estimate of the number of solutions, and has recently been incorporated into a more robust method for obtaining bounds on the solution count [Kroc et al., 2008b]. Given the importance and challenge of this problem, the AI community has invested a

substantial effort in developing good algorithms for solution counting, with nearly a dozen competing methods around today.

Earlier work on random ensembles of Constraint Satisfaction Problems (CSPs) has shown that the computationally hardest instances occur near phase boundaries, where instances go from having solutions to having no globally satisfying solution (a "solution focused picture"). In recent years, this picture has been refined and it was found that a key factor in determining the hardness of instances in terms of search algorithm (or sampling algorithm) is the question of how are the solutions distributed throughout the search space. This has made the structure of the solution space in terms of its clustering properties a key factor in determining the performance of combinatorial search methods ("cluster focused picture"). Can BP-like algorithms be used to provide such cluster-focused information? For example, how many clusters are there in a solution space? How big are the clusters? Answers to such questions will shed further light into our understanding of these hard combinatorial problems and lead to better algorithmic approaches for reasoning about them, be it for finding one solution or answering queries of probabilistic inference about the set of solutions. The study of the solution space geometry has indeed been the focus of a number of recent papers (e.g., Mézard et al. [2002], Braunstein et al. [2003], Achlioptas and Ricci-Tersenghi [2006], Maneva [2006], Krzakala et al. [2007], Ardelius et al. [2007], Hartmann et al. [2008]), especially by the statistical physics community, which has developed extensive theoretical tools to analyze such spaces under certain structural assumptions and large size limits. We provide a purely combinatorial method for counting the number of clusters, which is applicable even to small size problems and can be approximated very well by message passing techniques.

Solutions can be thought of as 'neighbors' if they differ in the value of one

variable, and the transitive closure of the neighbor relation defines clusters in a natural manner. Counting the number of clusters is a challenging problem. To begin with, it is not even clear what is the best succinct way to represent clusters. One relatively crude but useful way is to represent a cluster by the set of 'backbone' variables in that cluster, i.e., variables that take a fixed value in all solutions within the clusters. Interestingly, while it is easy (polynomial time) to verify whether a variable assignment is indeed a solution of CSP, the same check is much harder for a candidate cluster represented by the set of its backbone variables. Some clusters may not even have a useful succinct representation, e.g. if it has no backbone variables. These difficulties explain why counting the number of clusters is likely to be much harder in practice than counting the number of solutions, as it indeed is.

We propose one of the first scalable methods for estimating the number of clusters of solutions of combinatorial problems using a belief propagation like algorithm. Our technique applies to discrete constraint satisfaction problems in general, and we evaluate it here on the boolean satisfiability and graph coloring problems. While the naïve method, based on enumeration of solutions and pairwise distances, scales to graph coloring problems with 50 or so nodes and a recently proposed local search based method provides estimates up to a few hundred node graphs [Hartmann et al., 2008], our approach—being based on BP—easily provides fast estimates for graphs with $100,000$ nodes. We validate the accuracy of our approach by also providing a fairly non-trivial exact counting method for clusters, utilizing advanced knowledge compilation techniques. A comparison with the exact counts for graphs with a few hundred nodes suggests that our BP-based method is remarkably accurate in counting the number of clusters of random graph coloring problems in a wide range of parameter settings.

Our approach works with the factor graph representation of a constraint satisfaction problem. Yedidia, Freeman, and Weiss [2005] showed that if one can write the so-called "partition function", Z, for a quantity of interest in a factor graph with non-negative weights, then there is a fairly mechanical variational method derivation that yields belief propagation equations for estimating Z. Under certain assumptions, we derive a partition function style quantity, $Z_{(-1)}$, which has negations and relies on an inclusion-exclusion argument to count the number of clusters. We prove that $Z_{(-1)}$ exactly counts the number of clusters in various problems, for example for any 2-SAT problem, or for 3-COL problems where each connected component of the input graph has at least one triangle. In addition, we observe empirically that the quantity is extremely accurate for random 3-SAT and 3-COL problems, as well as for certain encoded industrial SAT instances. We then use the variational method (with negations, in this case) to obtain BP equations for estimating $Z_{(-1)}$. Our experiments show that the approximation obtained from our BP equations, $Z_{\mathrm{BP}_{(-1)}}$, is very accurate for random problems.

5.1 A Pseudo-Partition Function for Counting Clusters

We will need some additional formal definitions in this chapter. We define the *solutionspace* of a CSP to be the set of all its satisfying assignments. Two variable assignments are called *neighbors* if they differ in the value of exactly one variable.

Definition 3. A set $\mathcal{C} \subseteq Dom^n$ is a **cluster** if it is a maximal subset such that any two elements of $\mathcal{C}$ can be connected by a sequence of elements from $\mathcal{C}$ where consecutive elements are neighbors. When $\mathcal{C}$ is a subset of a solutionspace S of a

CSP, it is referred to as a **solution cluster** for that CSP.

We will often work with an extended domain consisting *subsets* of the original domain of a Boolean-valued function.

Definition 4. Let $f : Dom^n \rightarrow \{0, 1\}$ be a Boolean-valued function. The **extended domain** for f is defined as $DomExt := \mathcal{P}(Dom) \setminus \emptyset$, where $\mathcal{P}$ denotes powerset. Elements $\vec{y} \in DomExt^n$ are called **generalized assignments**. The **extended function** $f^{ext} : DomExt^n \rightarrow \{0, 1\}$ is defined as $f^{ext}(\vec{y}) := \prod_{\vec{x} \in \vec{y}} f(\vec{x})$, where $\vec{x} \in \vec{y}$ denotes point-wise inclusion, i.e., $\vec{x} \in \vec{y}$ iff $x_i \in y_i$ for every i.

Each generalized assignment $\vec{y} \in DomExt^n$ thus associates a non-empty set of values with each original variable, defining a *hypercube* of size $\prod_i |y_i|$ in Dom^n. The extended f^{ext} evaluates to 1 on this hypercube $\vec{y}$ iff f evaluates to 1 on all points $\vec{x}$ within this hypercube. This valuation can be efficiently implemented because length of each $\vec{y}_\alpha$ is only a constant. Note that $DomExt^n$ is a strict subset of all subspaces of Dom^n; specifically, $DomExt^n$ containts only hypercubes within Dom^n.

Consider a space $S \in Dom^n$, possibly the solutionspace of a CSP. When S is not connected (i.e., does not form a single cluster), we can think of S as composed of multiple smaller non-adjacent subspaces. Let $dist_H$ denote the Hamming distance between two vectors in Dom^n (i.e., the number of positions in which they differ).

Definition 5. Two subspaces $S, S' \subseteq Dom^n$ are called **separated** if for every $\vec{x} \in S$ and $\vec{x}' \in S'$, $dist_H(\vec{x}, \vec{x}') \geq 2$.

Now we start by developing an expression for estimating the number of solution clusters of a CSP defined on domain Dom^n by counting certain objects in the

corresponding extended domain $DomExt^n$. We will assume here that the solution-space of the CSP F we work with decomposes into a set of *separated hypercubes*, so that solution clusters correspond exactly to these hypercubes. This will allow us to develop a surprisingly simple expression for counting the number of clusters, and we will later see, both theoretically and empirically, that the same expression applies with high precision to much more complex solutionspaces as well.

Let us start by writing the indicator function $\chi(\vec{y})$ for the property that $\vec{y} \in DomExt^n$ forms a *maximal* solution hypercube for F, which corresponds exactly to a cluster under our assumption of hypercube clusters. We do this by utilizing the extended interpretation F^{ext} of F. Here $\vec{y}[y_i \leftarrow y_i']$ denotes the *syntactic substitution* of y_i' into y_i in $\vec{y}$.

$$\chi(\vec{y}) := \underbrace{F^{ext}(\vec{y})}_{\vec{y} \text{ is legal}} \cdot \underbrace{\prod_i \prod_{v_i \notin y_i} \Big(1 - F^{ext}(\vec{y}[y_i \leftarrow y_i \cup \{v_i\}])\Big)}_{\text{no point-wise generalization of } \vec{y} \text{ is legal}}$$

This expression does not yet yield itself to a factor graph representation, because it is not a product of factor functions (note that the $F^{ext}(.)$ function inside the parenthesis is itself a product). We thus need to massage the expression further to obtain one that factorizes.

Since the solution clusters are assumed to be perfect hypercubes, the dimensions of a (non-maximal) hypercube that can be extended independently while remaining within the solutionspace can also be extended simultaneously (i.e., $F^{ext}(\vec{y}[y_i \leftarrow y_i \cup \{v_i\}]) = 1$ and $F^{ext}(\vec{y}[y_j \leftarrow y_j \cup \{v_j\}]) = 1$ implies $F^{ext}(\vec{y}[y_i \leftarrow y_i \cup \{v_i\}\,,\ y_j \leftarrow y_j \cup \{v_j\}]) = 1$). Moreover, $F^{ext}(\vec{y}[y_i \leftarrow y_i \cup \{v_i\}]) = 1$ for any j implies $F^{ext}(\vec{y}) = 1$. Using these observations, $\chi(\vec{y})$ can be reformulated by factoring out the product

as:

$$F^{ext}(\vec{y})\left(\sum_{\substack{\vec{y}'\in\\(\mathcal{P}(Dom))^n\setminus\vec{y}}}(-1)^{\#o(\vec{y}')}\underbrace{\prod_i\prod_{v_i\in y'_i}F^{ext}(\vec{y}[y_i\leftarrow y_i\cup\{v_i\}])}_{=F^{ext}(\vec{y}\cup\vec{y}')\text{ by the hypercube assumption}}\right)$$

$$\overset{\vec{z}:=\vec{y}\cup\vec{y}'}{=}\sum_{\vec{z}\supseteq\vec{y}}(-1)^{\#o(\vec{z}\setminus\vec{y})}F^{ext}(\vec{z})\;=\;(-1)^{\#e(\vec{y})}\sum_{\vec{z}\supseteq\vec{y}}(-1)^{\#e(\vec{z})}F^{ext}(\vec{z})$$

where $\#o(\vec{y})$ denotes the number of *odd-size elements* of $\vec{y}$, and $\#e(\vec{y})$ the number of *even-size elements* of $\vec{y}$.

To count the number of maximal hypercubes in the solutionspace, we sum $\chi(\vec{y})$ across all vectors $\vec{y}\in DomExt^n$:

$$\begin{aligned}\sum_{\vec{y}}\chi(\vec{y})&=\sum_{\vec{y}}(-1)^{\#e(\vec{y})}\sum_{\vec{z}\supseteq\vec{y}}(-1)^{\#e(\vec{z})}F^{ext}(\vec{z})\\&=\sum_{\vec{z}}(-1)^{\#e(\vec{z})}F^{ext}(\vec{z})\left(\sum_{\emptyset\notin\vec{y}\subseteq\vec{z}}(-1)^{\#e(\vec{y})}\right)\\&=\sum_{\vec{z}}(-1)^{\#e(\vec{z})}F^{ext}(\vec{z})\left(\prod_i\underbrace{\sum_{\emptyset\notin\vec{y}_i\subseteq\vec{z}_i}(-1)^{\delta_e(y_i)}}_{=1}\right)\\&=\sum_{\vec{z}}(-1)^{\#e(\vec{z})}F^{ext}(\vec{z})\end{aligned}$$

This gives us the mZ expression.

The expression above is central to our study and we denote it by $Z_{(-1)}$. Formally,

Definition 6. For a function $F: Dom^n\to\{0,1\}$, the pseudo-partition function $Z_{(-1)}$ is defined as:

$$\boxed{Z_{(-1)}(F):=\sum_{\vec{y}\in DomExt^n}(-1)^{\#e(\vec{y})}F^{ext}(\vec{y})}\tag{5.1}$$

$Z_{(-1)}$ is called **exact on** F if $Z_{(-1)}(F)=\#clusters(F)$.

When F^{ext} factors as $F^{ext}(\vec{y}) := \prod_\alpha f_\alpha^{ext}(\vec{y}_\alpha)$, we can equivalently write $Z_{(-1)} = \sum_{\vec{y} \in DomExt^n} (-1)^{\#e(\vec{y})} \prod_\alpha f_\alpha^{ext}(\vec{y}_\alpha)$. The notation is chosen so as to emphasize the relatedness of $Z_{(-1)}$ to the usual partition function (eqn. (2.1)) denoted by Z, and indeed the expressions differ only in the (-1) term and the extension of domain.

Starting with the expression (5.1), it is relatively easy to verify that it indeed counts number of clusters if each cluster is a separated hypercube in the solution-space. To see this, notice that the sum $\sum_{\vec{y} \in DomExt^n}$ can be expressed as a double sum across the clusters and across hypercubes $\vec{y}$ inside each cluster (any $\vec{y}$ that is not fully within some cluster evaluates to 0 under $F^{ext}(.)$). What remains to observe is that every hypercube has exactly one extra sub-hypercube with *even* number of dimensions of even size (positive hypercube), as compared to sub-hypercubes with *odd* number of dimensions of even size (negative hypercube). We will do this by pairing together all-but-one positive with all negative sub-hypercubes of some hypercube cluster $\vec{h}$. Fix a point $\vec{x} \in \vec{h}$. This point corresponds to a positive sub-hypercube containing exactly $\vec{x}$, and every other sub-hypercube $\vec{y} \subseteq \vec{h}$ can be paired with $\vec{y}'$ through the following operation. Let i be the smallest such that $y_i \neq \{x_i\}$; let $\vec{y}' = \vec{y}$ on all elements except the i-th one, and $y'_i = y_i \setminus \{x_i\}$ if $x_i \in y_i$ and $y'_i = y_i \cup \{x_i\}$ if $x_i \notin y_i$. Since $\vec{h}$ is a hypercube, $\vec{y}'$ is its sub-hypercube and differs from $\vec{y}$ by one in size of exactly one element. One hypercube in the pair is thus positive, the other negative, and they cancel in the expression for $Z_{(-1)}$. The only unpaired hypercube is the one corresponding to the single point $\vec{x}$, and thus $Z_{(-1)}$ for $\vec{h}$ is 1.

This argument shows that if the solutionspace comprises of a set of separated hypercubes, then $Z_{(-1)}$ is exact (as each solution hypercube is a cluster). Surprisingly, $Z_{(-1)}$ is remarkably accurate even for random 3-SAT and nearly all 3-COL instances as we will see in the following sections.

5.2 Understanding $\mathbf{Z}_{(-1)}$ and Proving Exactness

This section is devoted to proving the exactness of $Z_{(-1)}$ on two problems: 2-SAT, and a broad subset of 3-COL. For concreteness, **2-SAT** is the Boolean satisfiability problem where each constraint (a "clause") is the disjunction of at most two variables or their negations. **3-COL** is the problem where given an undirected graph G as input, one must color each vertex of G with one of three colors c_1, c_2, or c_3 such that no two vertices sharing an edge get the same color. The **generalized 3-COL** problem is an extension of 3-COL where each vertex x_i comes with a pre-specified color domain $C_i \subseteq \{c_1, c_2, c_3\}$ and must be colored with a color from C_i.

We prove the following two results[1]:

Theorem 1 (2-SAT). *$Z_{(-1)}$ is exact on the 2-SAT problem.*

Theorem 2 (3-COL with a restricted vertex). *$Z_{(-1)}$ is exact on the (generalized) 3-COL problem restricted to graphs where every solution cluster has at least one vertex that takes at most two colors.*

It turns out that a fairly simple structural property of graphs can ensure that the solution space property required by the above theorem holds. Specifically, consider a connected graph G that has a triangle (i.e., a 3-clique) T. In any cluster of legal 3-colorings of G, each vertex of T is "locked" to a single color—it cannot flip its color without at least one other vertex of T also simultaneously flipping its color. Thus any vertex of T acts as a single witness vertex satisfying the required property for every solution cluster. This is formalized by the following corollary.

[1]The proofs in this chapter were developed in close collaboration with Ashish Sabharwal.

Corollary 1 (3-COL with triangles). *$Z_{(-1)}$ is exact on the 3-COL problem restricted to graphs in which every connected component has at least one triangle.*

Proof. Both $Z_{(-1)}$ and the number of clusters for a graph with multiple connected components are simply the product of the $Z_{(-1)}$ values and the number of clusters, respectively, for the individual connected components of the graph. We will therefore focus, w.l.o.g., on graphs with a single connected component and show that $Z_{(-1)}$ is exact on such graphs when the triangle condition is satisfied.

Consider a connected graph G with a triangle T. Let x be a vertex of T. We will argue that in any solution cluster $\mathcal{C}$, x takes at most one color, thereby satisfying the precondition in Theorem 2 and finishing the proof. To see this, assume w.l.o.g. that there is a coloring in $\mathcal{C}$ in which x has color c_1. The other two vertices in T must then have colors c_2 and c_3, respectively. This "locks" all three vertices of T into their current coloring throughout $\mathcal{C}$, because neither of these three vertices can flip its color without at least one other vertex of T also simultaneously flipping its color. □

Having a triangle is not the only way to ensure that Theorem 2 holds. In fact, one can start with any graph G that is not 3-colorable (for instance the triangle-free *Mycielski graphs* [Mycielski, 1955]) and apply the process of removing one edge at a time until it becomes 3-colorable. Let G' be the resulting graph and $\{u, u'\}$ be the last edge removed in this process. Both u and u' must have identical colors in any legal 3-coloring of G' (otherwise the graph would have been 3-colorable even before removing $\{u, u'\}$). This implies that in every solution cluster for G', both u and u' assume only one fixed color, so that Theorem 2 applies and $Z_{(-1)}$ is exact on G'.

5.2.1 General Proof Framework

For the most part, we will work directly with subspaces $S \subseteq Dom^n$ without worrying about how S is syntactically represented; e.g., S may be the solution space of a Boolean formula F or a graph coloring problem or of any CSP. We will write $S(\vec{x})$ to be the 0-1 indicator of whether $\vec{x} \in S$, and $\S(\vec{y})$ as an indicator of whether the whole hypercube represented by $\vec{y} \in DomExt^n$ is contained in S. In this notation, $Z_{(-1)}$ may be re-written as

$$Z_{(-1)}(S) = \sum_{\vec{y} \in DomExt^n} (-1)^{\#e(\vec{y})} \S(\vec{y}) \tag{5.2}$$

We start with a straightforward property of $Z_{(-1)}$ on separated subspaces that follows directly from the definition of $Z_{(-1)}$ and was alluded to above:

Proposition 4 (Additivity of $\mathbf{Z_{(-1)}}$). *Let S be the union of (pairwise) separated subspaces $S_i, 1 \leq i \leq t$. Then*

$$Z_{(-1)}(S) \;=\; \sum_{i=1}^{t} Z_{(-1)}(S_i).$$

Proof. By definition, the value of $Z_{(-1)}$ depends only on hypercubes $\vec{y}$ that are entirely contained in the space S. For each such $\vec{y}$, there exists a unique separated subspace S_i that contains it, which implies that $S(\vec{y}) = \sum_i S_i(\vec{y})$. This allows us to write:

$$\begin{aligned} Z_{(-1)}(S) = \sum_{\vec{y} \in DomExt^n} (-1)^{\#e(\vec{y})} S(\vec{y}) \\ &= \sum_{\vec{y} \in DomExt^n} (-1)^{\#e(\vec{y})} \sum_i S_i(\vec{y}) \\ &= \sum_i \sum_{\vec{y} \in DomExt^n} (-1)^{\#e(\vec{y})} S_i(\vec{y}) \\ &= \sum_i Z_{(-1)}(S_i) \end{aligned}$$

□

Thus, in order to analyze whether $Z_{(-1)}$ is exact on a space S, it suffices to focus on one cluster $\mathcal{C}$ at a time and analyze whether $Z_{(-1)}$ is exact on $\mathcal{C}$.

We will think of the space Dom^n as the domain of n variables $x_1, x_2, \ldots, x_n$. In order to understand $Z_{(-1)}$, it will help to see how it behaves on smaller subspaces when one fixes the value (or the domain) of any variable x_i. This will let us describe $Z_{(-1)}$ on a space in terms of its behavior on simpler subspaces.

Which simpler spaces should one consider? For $u \in Dom$, a space $S \subseteq Dom^n$ **restricted** to $x_i = u$, denoted $S|_{x_i=u}$, is a subspace of Dom^{n-1} constructed by taking points in S where x_i equals u and projecting out x_i to get an $n-1$ dimensional subspace. Note that such restrictions are easy to apply computationally; we simply "fix" the value of x_i to a value u and get a simpler problem.

More generally, we can restrict the domain of x_i to any subset $v \in DomExt$, in which case we use the notion of a stronger restriction where we keep only those points in S for which *all* values of x_i from v are allowed. Formally,

Definition 7. For a space $S \subseteq Dom^n$, $v \in DomExt$, and a variable x_i, the *simultaneously restricted subspace* $S|_{x_i \sim v} \subseteq Dom^{n-1}$ is defined as $\bigcap_{u \in v} S|_{x_i=u}$.

We call this a strong restriction because a natural weaker restriction would keep all points in S for which *some* value of x_i from v is allowed. This weak restriction, unfortunately, does not relate well to the notion of hypercubes on which our $Z_{(-1)}$ expression is based. In fact, the hypercube notion gives us a natural way to test whether, say, $S_{x_i \sim v}$ is non-empty: this is equivalent to checking whether the hypercube formed by v is contained in S.

As an example, when S is the solution space of a Boolean formula F, then $S|_{x_i \sim v}$ for $v \in \{\{0\}, \{1\}, \{0,1\}\}$ corresponds to the solution subspace of the modified formula F' where the variable x_i is replaced by the constant 0, replaced by the constant 1, or syntactically deleted, respectively (deleting a variable ensures that

any remaining solutions can be extended by both truth values of x_i). Similarly, for a graph 3-coloring problem, $S_{x_i \sim \{c_1,c_2\}}$ corresponds to the 3-colorings where all neighbors of the i-th vertex have color c_3 so that vertex i is free to take either of colors c_1 or c_2.

We are now ready to formulate some interesting properties of $Z_{(-1)}$ and clusters. The following property allows us to write $Z_{(-1)}$ as a signed sum of $Z_{(-1)}$ on subspaces of one smaller dimension. Specifically, we fix any variable x_i and look at all possible subspaces formed by restricting x_i to various values $v \in \mathit{DomExt}$. $Z_{(-1)}$ on S can then be decomposed as an alternating sum of the $Z_{(-1)}$ values on all such subspaces. For brevity of notation, for $v \in \mathit{DomExt}$, let $\mathit{even}(v)$ be 1 if $|v|$ is even and 0 otherwise.

Proposition 5 (Decomposability of $\mathbf{Z_{(-1)}}$). *For $S \subseteq \mathit{Dom}^n$ and any variable x_i,*

$$Z_{(-1)}(S) \;=\; \sum_{v \in \mathit{DomExt}} (-1)^{\mathit{even}(v)} Z_{(-1)}(S|_{x_i \sim v}).$$

Proof. From expression (5.2), we have

$$\begin{aligned}
Z_{(-1)}(S) &= \sum_{\vec{y} \in \mathit{DomExt}^n} (-1)^{\#e(\vec{y})} S(\vec{y}) \\
&= \sum_{v \in \mathit{DomExt}} \sum_{\substack{\vec{z} \in \mathit{DomExt}^{n-1} \\ \vec{y} = (v,\vec{z})}} (-1)^{\#e(\vec{y})} S(\vec{y}) \\
&= \sum_{v \in \mathit{DomExt}} \sum_{\substack{\vec{z} \in \mathit{DomExt}^{n-1} \\ \vec{y} = (v,\vec{z})}} (-1)^{\#e(\vec{z}) + \#e(v)} S[y_i \leftarrow v](\vec{z}) \\
&= \sum_{v \in \mathit{DomExt}} (-1)^{\#e(v)} \sum_{\vec{z} \in \mathit{DomExt}^{n-1}} (-1)^{\#e(\vec{z})} S[y_i \leftarrow v](\vec{z}) \\
&= \sum_{v \in \mathit{DomExt}} (-1)^{1+|v|} Z_{(-1)}(S[y_i \leftarrow v])
\end{aligned}$$

This finishes the proof. □

Since $Z_{(-1)}$ is supposed to capture the number of clusters, it is natural to define a syntactically similar decomposability property for the number of clusters. This property, as we will shortly see, turns out to be extremely helpful in understanding the behavior of $Z_{(-1)}$, and does hold for the solution spaces of interesting problems.

Definition 8 (Cluster decomposability). For a cluster $\mathcal{C} \subseteq Dom^n$ and variable x_i, $\mathcal{C}$ is called i**-decomposable** if

$$\sum_{v \in DomExt} (-1)^{even(v)} \#clusters(\mathcal{C}|_{x_i \sim v}) \;=\; 1. \tag{5.3}$$

$\mathcal{C}$ is called **decomposable** if it is i-decomposable for some i.

We are interested in exploring the following question: for which problems P is $Z_{(-1)}$ exact? By a "problem" we mean a collection of specific instances, e.g., 3-SAT, 3-COL, 3-COL restricted to graphs with at least one triangle, a strict subclass of 3-SAT, etc. We prove a general property that will let us answer this question for specific problems. We will call a problem P **closed under value restrictions** if for any instance F of P, variable x with domain Dom, and values $v \subset Dom$, the simplified problem instance $F|_{x \in v}$ formed by restricting x to take values only in v is also in P.

The following lemma shows that in order to understand for which problems is $Z_{(-1)}$ exact, one may equivalently study the decomposability of solution clusters for that problem.

Lemma 1. *Let P be a problem that is closed under value restrictions. Let $cl(P)$ denote the set of solution clusters of P. $Z_{(-1)}$ is exact on every $\mathcal{C} \in cl(P)$ if and only if every $\mathcal{C} \in cl(P)$ is decomposable.*

Proof. Suppose first that every $\mathcal{C} \in cl(P)$ is decomposable. We prove that $Z_{(-1)}$ is exact on every such cluster. The argument is by induction on the cluster size

$|\mathcal{C}|$. For the base case of $|\mathcal{C}| = 1$, $Z_{(-1)}(\mathcal{C}) = 1$ and $\mathcal{C}$ is trivially i-decomposable for every i (exactly one term in the decomposition expression, equation (5.3), is 1; the rest are 0). For the inductive case of $|\mathcal{C}| \geq 2$, fix any i such that the i-th variable x_i takes at least two different values in $\mathcal{C}$. For $v \in \mathit{DomExt}$, each cluster in $S' := \mathcal{C}|_{x_i \sim v}$ forms a subspace strictly smaller than $\mathcal{C}$. (Note that S' may not necessarily be a single connected cluster). By the inductive hypothesis and the additivity of $Z_{(-1)}$, $Z_{(-1)}(S') = \#\mathit{clusters}(S')$. Using this fact and the decomposability of $Z_{(-1)}$, we obtain

$$\begin{aligned} Z_{(-1)}(\mathcal{C}) &= \sum_{v \in \mathit{DomExt}} (-1)^{\mathit{even}(v)} Z_{(-1)}(\mathcal{C}|_{x_i \sim v}) \\ &= \sum_{v \in \mathit{DomExt}} (-1)^{\mathit{even}(v)} \#\mathit{clusters}(\mathcal{C}|_{x_i \sim v}) \end{aligned}$$

By assumption, $\mathcal{C}$ is decomposable, so that the last summation equals 1, yielding $Z_{(-1)}(\mathcal{C}) = 1$. Thus, $Z_{(-1)}$ is exact on $\mathcal{C}$.

For the converse, suppose that some $\mathcal{C} \in \mathit{cl}(P)$ is not decomposable. We show that there must exist a solution cluster on which $Z_{(-1)}$ is inexact. If $Z_{(-1)}(\mathcal{C}) \neq 1$, then $Z_{(-1)}$ is inexact on $\mathcal{C}$ and the proof is complete. On the other hand, if $Z_{(-1)}(\mathcal{C}) = 1$, notice that the non-decomposability of $\mathcal{C}$ implies that the summation in equation (5.3) does not equal one. It follows from the decomposability of $Z_{(-1)}$ that for at least one $v \in \mathit{DomExt}$, we have $Z_{(-1)}(\mathcal{C}|_{x_i \sim v}) \neq \#\mathit{clusters}(\mathcal{C}|_{x_i \sim v})$. In other words, we have found a witness subspace $S' := \mathcal{C}|_{x_i \sim v}$ on which $Z_{(-1)}$ is inexact. Because of the additivity of $Z_{(-1)}$, it follows that $Z_{(-1)}$ must be inexact on at least one cluster $\mathcal{C}'$ within S'. Note finally that due to P being closed under value restrictions, $\mathcal{C}'$ is also a solution cluster in $\mathit{cl}(P)$, on which we have found $Z_{(-1)}$ to be inexact. This finishes the proof. □

When are clusters decomposable? A good first step in understanding the **inclusion-exclusion style summation** in equation (5.3) is to think of a

simplified case where for all $v \in DomExt$, $\#clusters(\mathcal{C}|_{x_i \sim v})$ is either 0 or 1, and is 1 if and only if in $\mathcal{C}$, x_i can assume every value from v. If this simplified interpretation holds, then each term in the inclusion-exclusion style summation is 0, 1, or -1; the summation adds up 1 for each value u_1 that can be taken by x_i in $\mathcal{C}$, subtracts away the overcount by adding -1 for each value pair $\{u_1, u_2\}$ that can be taken by x_i in $\mathcal{C}$, adds back as compensation for over-subtraction a 1 for each triple of values that can be taken by x_i, and so on. This, by the standard inclusion-exclusion principle, yields precisely 1 (as long as $\mathcal{C}$ is non-empty). There are, unfortunately, two issues with this simplified interpretation: **simultaneous valuation** and **cluster fragmentation**, arising because the extended domain computational framework only allows us to efficiently test for the inclusion of hypercubes in $\mathcal{C}$, rather than testing whether x_i can take several distinct values in $\mathcal{C}$, not necessarily forming a full hypercube.

By simultaneous valuation, we mean that rather than testing whether a variable x_i appears with, say, three possible values u_1, u_2, and u_3 in $\mathcal{C}$ (i.e., whether $\forall j \in \{1, 2, 3\} \,.\, \mathcal{C}|_{x_i = u_j} \neq \emptyset$), our machinery only allows us to test whether a single partial assignment in $\mathcal{C}$ is extendible to a solution by all three values $x_i = u_j, j \in \{1, 2, 3\}$ *simultaneously* (i.e., $\mathcal{C}|_{x_i \sim \{u_1, u_2, u_3\}} \neq \emptyset$, meaning $\cap_{j=1}^{3} \mathcal{C}|_{x_i = u_j} \neq \emptyset$). This is a strictly stronger requirement in general and much harder to satisfy. We would thus often be *undercounting* the contribution for values $\{u1, u2, u3\}$ in the inclusion-exclusion summation.

By cluster fragmentation, we mean that, furthermore, due to the spatial structure of the cluster under consideration, it is very possible for the value restriction $\mathcal{C}|_{x_i \sim v}$ to fragment $\mathcal{C}$ into multiple clusters, so that $\#clusters(\mathcal{C}|_{x_i \sim v}) > 1$. In this case, we will be incorrectly *overcounting* the contribution to the inclusion-exclusion summation for values v, even if $Z_{(-1)}$ was exact on the subspace $\mathcal{C}|_{x_i \sim v}$.

Interestingly, we show that there are broad classes of problems for which simultaneous valuation and cluster fragmentation issues can be circumvented, resulting in decomposable clusters. We explore two specific ways in which the decompoability condition, equation (5.3), can hold. These may appear too restrictive at first glance, but we will show that they provably encompass *all* connected 3-COL instances with at least one triangle and can be empirically demonstrated to hold for nearly all solution clusters of random 3-SAT instances.

The following lemma capitalizes on the fact that if a variable takes only two values in a cluster $\mathcal{C}$, there must exist a "shared region" in $\mathcal{C}$ where that variable is free to assume both values. In such a scenario, testing a simultaneous restriction for non-emptiness is as good as testing two individual restrictions for non-emptiness.

Lemma 2. *If a variable x_i takes at most two values in a cluster $\mathcal{C}$, then the simultaneous valuation issue does not arise for $\mathcal{C}$, i.e., $\mathcal{C}|_{x_i \sim \{u_1,u_2\}} \neq \emptyset$ iff x_i takes both values u_1 and u_2 in $\mathcal{C}$.*

Proof. The statement of the lemma is trivially true if x_i takes only one value in $\mathcal{C}$. We therefore assume that x_i takes exactly two values in $\mathcal{C}$.

By the definition of simultaneous restriction, if $\mathcal{C}|_{x_i \sim \{u_1,u_2\}} \neq \emptyset$, then x_i must take both values, u_1 and u_2, in $\mathcal{C}$. The converse is more interesting. Suppose x_i takes values u_1, u_2 in vectors $\vec{u}_1, \vec{u}_2 \in \mathcal{C}$, respectively. Since $\mathcal{C}$ is connected, there exists a path in $\mathcal{C}$ from $\vec{u}_1$ to $\vec{u}_2$, i.e., a sequence of vectors differing in the value of exactly one variable. At some point, therefore, the path must involve a vector $\vec{u}$ in which x_i initially equals u_1 and is then flipped to u_2. $\vec{u}$ is then a witness for the non-emptiness of $\mathcal{C}|_{x_i \sim \{u_1,u_2\}}$. □

The next lemma shows that the key component determining whether a cluster $\mathcal{C}$ remains connected when a variable is restricted to one or both of its two possible

values is that it remains connected when applying the simultaneous restriction to both values.

Lemma 3. *If a variable x_i takes exactly two values u_1 and u_2 in a cluster $\mathcal{C}$ and $\mathcal{C}|_{x_i \sim \{u_1,u_2\}}$ is a connected cluster, then both $\mathcal{C}|_{x_i=u_1}$ and $\mathcal{C}|_{x_i=u_2}$ are also connected clusters.*

The idea is that if there is a path between two points in $\mathcal{C}|_{x_i=u_1}$ that goes through a region in $\mathcal{C}$ where x_i can only be u_2, then this path must enter and exit this "u_2 only" region through the "shared" region $\mathcal{C}|_{x_i \sim \{u_1,u_2\}}$. Since this shared region is itself connected, one can redirect this portion of the path so that it stays completely within the shared region, which can then be modified so that the path uses only $x_i = u_1$. We should note that even if the starting problem is completely symmetric (e.g., in 3-COL instances where the colors are symmetric), this doesn't mean that $\mathcal{C}|_{x_i=u_1}$ and $\mathcal{C}|_{x_i=u_2}$ are isomorphic. Therefore, even in this symmetric case, the proof of the lemma would not be trivial. The shared region, of course, is symmetric w.r.t. u_1 and u_2, which is what the reasoning exploits.

The formal proof goes as follows:

Proof. We will show that $\mathcal{C}|_{x_i=u_1}$ forms a connected cluster; the result for $\mathcal{C}|_{x_i=u_2}$ follow by a symmetric argument.

For brevity define $\mathcal{C}_1 := \mathcal{C}|_{x_i=u_1}$ and $\mathcal{C}_{\{1,2\}} := \mathcal{C}|_{x_i \sim \{u_1,u_2\}}$. Consider any two vectors $\vec{u}_a, \vec{u}_b \in \mathcal{C}_1$, so that $x_i = u_1$ in both of these vectors. We will show that there is a path fully within $\mathcal{C}_1$ that connects $\vec{u}_a$ and $\vec{u}_b$. Since $\mathcal{C}$ is connected, there must be a path p in $\mathcal{C}$ from $\vec{u}_a$ to $\vec{u}_b$. If p lies completely in $\mathcal{C}_1$, the proof is complete. If not, at some point p must go outside $\mathcal{C}_1$, by changing the value of x_i from u_1 to u_2. This flip, by definition, must occur within $\mathcal{C}_{\{1,2\}}$. Let $\vec{w}_c \in \mathcal{C}_{\{1,2\}}$ be the vector in p at this last point, where x_i still equals u_1. The path p now traverses various solutions in $\mathcal{C}_{\{1,2\}}$ and possibly even in $\mathcal{C}_2$. Nonetheless, p must eventually

return to $\vec{u}_b \in \mathcal{C}_1$, and thus pass through another point $w_d \in \mathcal{C}_{\{1,2\}}$ with $x_i = u_1$, where the value of x_i is flipped back from u_2 to u_1. Now, since the precondition of the lemma says that $\mathcal{C}_{\{1,2\}}$ is connected, we can replace this portion of p between w_c and w_d with a path that lies completely within $\mathcal{C}_{\{1,2\}}$. Furthermore, since every solution in $\mathcal{C}_{\{1,2\}}$, by definition, allows both values u_1 and u_2 for x_i, this new path can be made to have $x_i = u_1$ throughout. Thus we have a path from u_a to w_b to w_c purely within $\mathcal{C}_1$. Repeating this path reconfiguration process for any further points where p again goes outside $\mathcal{C}_1$, we obtain a modified path connecting $\vec{u}_a$ and $\vec{u}_b$ and lying entirely within $\mathcal{C}_1$. □

5.2.2 $\mathbf{Z}_{(-1)}$ for Graph Coloring

Armed with these generic notions of cluster fragmentation and simultaneous valuation, we now utilize special properties of the clustering structure of the solution space of the 3-COL problem to fill in the missing pieces needed for proving Theorem 2. In particular, we will show that every solution cluster under consideration is decomposable.

Recall the generalized 3-COL problem, where each vertex has its own color domain $C_i \subseteq \{1, 2, 3\}$. This problem is clearly closed under value restrictions, which allows us to apply Lemma 1.

Lemma 4 (Main 3-COL Lemma). *For the generalized 3-COL problem, if a vertex x_i takes exactly two colors c_1 and c_2 in a cluster $\mathcal{C}$, then the cluster fragmentation issue does not arise for $\mathcal{C}$, i.e., for $c \subseteq \{c_1, c_2\}$, $\#clusters(\mathcal{C}|_{x_i \sim c}) = 1$.*

Proof. By Lemma 3, it suffices to show that $\mathcal{C}$ remains connected when x_i is simultaneously assigned colors $\{c_1, c_2\}$. The argument relies on the special properties of solution clusters of (generalized) 3-COL instances and is the central combinatorial

argument best thought of in terms of the vertices "borrowing" and "lending" colors when connecting points within $\mathcal{C}$.

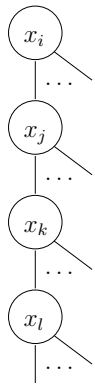
Consider two points p and p' in $\mathcal{C}|_{x_i \sim \{c_1,c_2\}}$. For the sake of contradiction, suppose that p and p' are *not* connected by a path lying entirely within $\mathcal{C}|_{x_i \sim \{c_1,c_2\}}$. The two points, of course, are connected within $\mathcal{C}$ as $\mathcal{C}$ is a single cluster. It follows that every path $p_0 = p, p_1, p_2, \ldots, p_\ell = p'$ connecting p and p' within $\mathcal{C}$ must be such that it cannot be converted into a path where x_i takes both of the colors c_1 and c_2 simultaneously throughout the path (otherwise p and p' would also be connected within $\mathcal{C}|_{x_i \sim \{c_1,c_2\}}$). In other words, each such path must have a point p_s such that x_i has, without loss of generality, color c_1 at this point but cannot be simultaneously assigned both c_1 and c_2. This implies that at least one neighbor of x_i, say x_j, itself has color c_2 at the point p_s, preventing x_i from having color c_2. Note that since the end points of the path, p_0 and p_ℓ, are within $\mathcal{C}|_{x_i \sim \{c_1,c_2\}}$, x_i *can* take both colors at these end points, implying that every neighbor of x_i, in particular x_j, must have had color c_3 at both of these end points. In summary, we have argued that if $\mathcal{C}|_{x_i \sim \{c_1,c_2\}}$ consists of multiple clusters with p and p' in different clusters, then every path connecting p and p' in $\mathcal{C}$ must necessarily involve a neighbor x_j of x_i changing color from c_3 to, w.l.o.g., c_2 and then back to c_3 along the path.

For ease of illustration, we will think of this process as x_i temporarily "lending" one of its two colors, namely c_2, to x_j along the path and then retrieving the color c_2 back; from another point of view, we can think of x_j starting out with color c_3, temporarily "borrowing" c_2 from x_i, and finally "releasing" c_2 to fall back to its original color, c_3, at the end of the path. In the following, we will argue that if p_s is the point along the path at which x_j borrows c_2 from x_i and p_t is the point along the path at which x_j releases c_2 and makes it again avail-

able for x_i, then $p_{s-1} = p_t$. That is, there is no advantage for x_j to borrow a color from x_i in order to help connect p and p' in $\mathcal{C}$; the strictly shorter sequence $p_0 = p, p_1, \ldots, p_{s-1}, p_{t+1}, p_{t+2}, \ldots, p_\ell = p'$ itself is a path connecting p and p' in $\mathcal{C}$. This fact—that we can start out with any path connecting p and p' in $\mathcal{C}$ and convert it into a strictly shorter path connecting the same two points—is clearly a contradiction as it cannot be true for, in particular, the shortest path connecting p and p' in $\mathcal{C}$. Hence, our initial assumption that $\mathcal{C}|_{x_i \sim \{c_1, c_2\}}$ has multiple clusters must be false, finishing the proof.

We now prove the last remaining claim that under the above conditions, p_{s-1}, the point in the path immediately before x_j borrows c_2, must be identical to p_t, the point at which x_j releases c_2. The proof relies critically on the pre-condition that x_i is restricted to take only two colors, c_1 and c_2. The way this condition is used is the following. Consider the point p_s at which x_i lends color c_2 to its neighbor x_j. At this point, x_i, not having access to color c_3, is "frozen" into color c_1 until x_j makes color c_2 available to x_i. Also, at point p_{s-1}, x_j could not have had color either c_1 or c_2 as x_i was allowed to have both of these colors; x_j must therefore change its color from c_3 to c_2 when borrowing c_2 from x_i. Note that at this point, x_i is frozen into color c_1 and x_j has switched from c_3 to c_2.

Now, since x_j changed colors from c_3 to c_2, it must be the case that all its neighbors have color c_1. The only "reason" for x_j to change color from c_3 to c_2 can be to make c_3 "available" for one of its neighbors, say x_k, to change into; of course, all neighbors of x_j at this point must have color c_1 as x_j is flipping from c_3 to c_2. Without loss of generality, assume that at the next point in the path, x_k, originally of color c_1, in turn borrows color c_3 from x_j. At this point, x_j itself is frozen into color c_2 until x_k makes c_3 available for x_j (recall that x_i is already frozen into c_1 and cannot make c_1 available for x_j without x_j releasing c_2 to begin



	p_{s-1}		p_s		p_{s+1}		p_{s+2}	
x_i	c_1, c_2	c_1	(c_1)	(c_1)	(c_1)	(c_1)	(c_1)	(c_1)
x_j	(c_3)	c_3	c_2, c_3	c_2	(c_2)	(c_2)	(c_2)	(c_2)
x_k	?	?	(c_1)	c_1	c_1, c_3	c_3	(c_3)	(c_3)
x_l	?	?	?	?	(c_2)	c_2	c_1, c_2	c_1

Figure 5.1: Illustration for the proof of Lemma 4: a small fragment of a graph is shown, with colors assigned to each vertex in the table next to it. The coloring states progress from left to right on the path between p_{s-1} and p_{s+2}, and the adjacent nodes "lock" each other into their current colors. The locked states are denoted by parentheses.

with—a dead-lock situation). The situation is depicted in Figure 5.1.

In summary, x_i lends one of its two colors, c_2, to x_j in order to enable x_j lend c_3 to x_k, in order to enable x_k lend c_1 to its neighbor, and so on. In the process, x_i is frozen to color c_1 until x_j releases c_2, x_j is frozen to c_2 until x_k releases c_3, x_k is frozen to c_3 until its chosen neighbor releases c_1, and the chain goes on with one vertex lending its original color (i.e., its color at point p_{s-1}) to a neighboring vertex and waiting in a dead-lock state for this neighbor to release the lent color. Recall, however, that as the path connecting p and p' continues and we reach the point p_t, x_i is supposedly again allowed to have both of the colors c_1 and c_2. In other words, by the time we reach p_t in the path, x_j must have released c_2. Since x_i was frozen into c_1, x_j must have fallen back to its original color, c_3, when releasing c_2. Similarly, for x_j to fall back from c_2 to c_3, x_k must have released c_3 and fallen back to its original color c_1, and for this to happen, x_k's chosen neighbor must have released c_1 and fallen back to its original color c_2, and so on. In other words, the whole chain of vertices whose change in color was triggered by x_i lending c_2 to

x_j at point p_s must all have fallen back to their original colors (i.e., their colors at point p_{s-1}) by the time we reach p_t. It follows that p_{s-1} is identical to p_t, proving the desired claim. $\square$

With this, we have all the ingredients needed to prove Theorem 2. The reasoning is that if a vertex takes at most two colors in a solution cluster $\mathcal{C}$, then we can resort to Lemmas 2 and 4 to conclude that neither simultaneous valuation nor cluster fragmentation is an issue for $\mathcal{C}$, which implies, in a vain similar to the "simplified case" discussion earlier, that $\mathcal{C}$ is decomposable. The exactness of $Z_{(-1)}$ then follows from Lemma 1. Following is the full proof.

Proof of Theorem 2. We prove that cluster decomposability holds for all solution clusters of the generalized 3-COL problem restricted to graphs where every solution cluster has at least one vertex that takes at most two colors. This, from Lemma 1, implies that $Z_{(-1)}$ must be exact on this problem (it can be easily verified that this problem is closed under variable restrictions).

To reason about cluster decomposability in such graphs, consider any solution cluster $\mathcal{C}$ of this problem and let x_i be a vertex that takes at most two colors in $\mathcal{C}$. Clearly, if x_i takes only one color in $\mathcal{C}$, then $\mathcal{C}$ is trivially i-decomposable. If x_i takes two colors, then Lemmas 2 and 4 imply, respectively, that neither simultaneous valuation nor cluster fragmentation is an issue for $\mathcal{C}$. It follows by the standard inclusion-exclusion principle that the sum in equation (5.3) evaluates to exactly 1 for $\mathcal{C}$, as discussed earlier for the "simplified case". Therefore, $\mathcal{C}$ is i-decomposable. $\square$

5.2.3 $\mathbf{Z}_{(-1)}$ for 2-SAT

The result for 2-SAT can also be derived in a similar manner as for 3-COL above. In particular, we will show that every solution cluster of a 2-SAT problem is decomposable, and will thus prove Theorem 1.

The 2-SAT problem is clearly closed under value restrictions, so that Lemma 1 applies. Further, since we are working with binary domains, both of Lemmas 2 and 3 also apply. The only missing piece then is a result equivalent to Lemma 4 for 3-COL above, stating that solution clusters of 2-SAT do not fragment when a variable is restricted to values from $\{0, 1\}$. (As remarked earlier, restricting simultaneously to $\{0, 1\}$ corresponds to simply syntactically deleting the variable from the formula.) Formally, capitalizing on Lemma 3 as before and on the structural properties of 2-SAT solution clusters, we can show:

Lemma 5 (Main 2-SAT Lemma). *For the 2-SAT problem, the cluster fragmentation issue does not arise, i.e., for any cluster $\mathcal{C}$ and $v \subseteq \{0, 1\}$, $\#clusters(\mathcal{C}|_{x_i \sim v}) = 1$.*

From this, Theorem 1 follows by the exact same reasoning as above for Theorem 2.

5.2.4 $\mathbf{Z}_{(-1)}$ for Recursively Monotone Spaces and SAT

For the k-SAT problem, $Z_{(-1)}$ is not provably exact on essentially all instances as was the case for 3-COL. Nonetheless, we formulate a general framework of *monotone variables* and *recursively monotone spaces* which establishes generic sufficient conditions for $Z_{(-1)}$ to be exact. This framework includes many simple natural spaces such as hypercubes, solution spaces of polytree formulas, and solution spaces of formulas where each literal occurs with only one polarity (a "pure" lit-

eral). We will later empirically demonstrate that nearly all solution clusters of the random k-SAT problem satisfy this property. Since many of the statements that follow hold for arbitrary discrete finite domains, we use Dom instead of $\{0, 1\}$ despite the fact that we are focusing on SAT in this section.

Definition 9. For a space $S \subseteq Dom^n$, $u \in Dom$, and a variable x_i, x_i is **u-monotone** in S if $\vec{x} \in S$ implies $\vec{x}[x_i \leftarrow u] \in S$. u is called a **dominating value** for x_i.

In words, this says that any point in S where $x_i \neq u$ has a corresponding point in S where $x_i = u$ (not necessarily vice versa). Notice that pure literals in CNF formulas are natural examples of monotone variables, while backbone variables and "don't care" variables are also extreme cases of monotone variables. In general, monotone variables need not be pure, backbone, or "don't care" variables.

Definition 10 (Recursively Monotone Spaces). A space $S \subseteq Dom^n$ is **recursively monotone** if S is non-empty and respects the following: either $n = 1$ or there exists a u-monotone variable x_i such that $S|_{x_i=u}$ is recursively monotone.

Proposition 6. *The following spaces in $\{0, 1\}^n$ are recursively monotone:*

- *hypercubes,*
- *solution space of satisfiable polytree formulas on n variables, and*
- *solution space of monotone formulas (i.e., formulas where every variable occurs in only one polarity; not necessarily polytrees) on n variables.*

Proof. This follows from Proposition 9 (below), which is a strictly stronger statement. □

We first state a basic property of monotone variables, that setting such variables to their dominating value does not split clusters.

Lemma 6. *Let x_i be a u-monotone variable in a space S. Then for every cluster $\mathcal{C}$ in S, $\#clusters(\mathcal{C}|_{x_i=u}) = 1$. In particular, $\#clusters(S) = \#clusters(S|_{x_i=u})$.*

Proof. For brevity, let $\mathcal{C}'$ denote the restricted space $\mathcal{C}|_{x_i=u}$. We assume here that $\mathcal{C}$, being a cluster, must be non-empty. Since u is a dominating value for x_i in S, x_i must take value u in at least half the points in any cluster within S. In particular, $\mathcal{C}'$ must be non-empty, so that $\#clusters(\mathcal{C}') \geq 1$.

We will utilize u-monotonicity of x_i to show that any two points $\vec{a}$ and $\vec{b}$ in $\mathcal{C}'$ are connected by a path lying within $\mathcal{C}'$, finishing the proof that $\#clusters(\mathcal{C}')$ in fact equals 1. Think of $\vec{a}$ and $\vec{b}$ as points in the potentially larger cluster $\mathcal{C}$; they must be connected by a path p within $\mathcal{C}$ since $\mathcal{C}$ is a cluster. Transform p into a new path p' by replacing the value of the component x_i in each point in p by u. Due to u-monotonicity of x_i in S, p' lies completely in S and in fact also in $\mathcal{C}$ as each point of p' is away from a point in p by a Hamming distance of exactly zero or one (depending on whether or not the x_i component equals u at that point in the path p). Since $x_i = u$ for all points in p', this new path p' connecting $\vec{a}$ and $\vec{b}$ lies fully within $\mathcal{C}'$, finishing the proof. □

Interestingly, cluster decomposability does hold for any space with even a single monotone variable, but for a very different reason than it does for solution spaces of 3-COL problems. In particular, it is *not* that case that clusters don't fragment for any value assignment to a monotone variables; rather even if a cluster fragments when assigning non-dominating values to a monotone variable, the effect cancels out because of monotonicity. We state this as the following proposition.

Proposition 7. *If a space has a monotone variable, then every cluster in it is decomposable.*

Proof. Let S be the space in Dom^n under consideration and x_i be a variable that is u-monotone in S. We will show that every cluster $\mathcal{C}$ in S is i-decomposable.

The terms on the left hand side of the cluster decomposability condition, equation (5.3), can be grouped together and cancel out as follows:

$$\begin{aligned}
&\sum_{v \in DomExt} (-1)^{even(v)} \#clusters(\mathcal{C}|_{x_i \sim v}) \\
&= \#clusters(\mathcal{C}|_{x_i=u}) \\
&\quad + \sum_{\substack{v \in DomExt \\ u \notin v}} (-1)^{even(v)} \#clusters(\mathcal{C}|_{x_i \sim v}) \\
&\quad - \sum_{\substack{v \in DomExt \\ u \notin v}} (-1)^{even(v)} \#clusters(\mathcal{C}|_{x_i \sim v \cup \{u\}}) \\
&= \#clusters(\mathcal{C}|_{x_i=u}).
\end{aligned}$$

The two summations in the middle expression above cancel out term-for-term because for every $v \in DomExt$ with $u \notin v$, u-monotonicity of x_i implies that the space $\mathcal{C}|_{x_i \sim v}$ is in fact structurally identical to the space $\mathcal{C}|_{x_i \sim v \cup \{u\}}$. Thus, all that remains to show for proving i-decomposbility of $\mathcal{C}$ is that $\#clusters(\mathcal{C}|_{x_i=u}) = 1$. This, however, follows immediately from Lemma 6. □

We call the above result a proposition rather than a lemma because, unfortunately, we won't be able to use it as a building block to immediately apply Lemma 1 and conclude that $Z_{(-1)}$ is exact on monotone spaces. The problem is that the subclass of SAT restricted to formulas whose solution spaces have a monotone variable is *not* closed under value restrictions, so that Lemma 1 does not apply. In fact, even the stronger subclass of SAT restricted for formulas whose solution spaces are recursively monotone is not closed under value restrictions; $S|_{x_i=u}$ is guaranteed to be recursively monotone only when u is a dominating value for x_i. We therefore resort to a key cancellation property of recursively monotone spaces, as used in the proof of the following lemma: the subspaces $S|_{x_i \sim v}$ and $S|_{x_i \sim v \cup \{u\}}$ are *structurally identical* when x_i is u-monotone in S, so that their contributions to $Z_{(-1)}$ cancel

each other out.

Lemma 7. *Let x_i be a u-monotone variable in a space S. Then for every cluster $\mathcal{C}$ in S, $Z_{(-1)}(\mathcal{C}) = Z_{(-1)}(\mathcal{C}|_{x_i=u})$. In particular, $Z_{(-1)}(S) = Z_{(-1)}(S|_{x_i=u})$.*

Proof. Recall the terms on the right hand side of the decomposability equation in Proposition 5, as applied to the cluster $\mathcal{C}$. These terms can be grouped together and cancel out as follows:

$$\begin{aligned}
&\sum_{v \in DomExt} (-1)^{even(v)} Z_{(-1)}(\mathcal{C}|_{x_i \sim v}) \\
&= Z_{(-1)}(\mathcal{C}|_{x_i=u}) \\
&\quad + \sum_{\substack{v \in DomExt \\ u \notin v}} (-1)^{even(v)} Z_{(-1)}(\mathcal{C}|_{x_i \sim v}) \\
&\quad - \sum_{\substack{v \in DomExt \\ u \notin v}} (-1)^{even(v)} Z_{(-1)}(\mathcal{C}|_{x_i \sim v \cup \{u\}}) \\
&= Z_{(-1)}(\mathcal{C}|_{x_i=u}).
\end{aligned}$$

The two summations in the middle expression above cancel out term-for-term because for every $v \in DomExt$ with $u \notin v$, u-monotonicity of x_i implies that the space $\mathcal{C}|_{x_i \sim v}$ is in fact structurally identical to the space $\mathcal{C}|_{x_i \sim v \cup \{u\}}$. The result for $Z_{(-1)}(S)$ follows from this by using the additivity of $Z_{(-1)}$ on separated subspaces. (Alternatively, the result for $Z_{(-1)}(S)$ can be directly derived from Proposition 5 as above.) □

We are now ready to prove our second main theorem:

Theorem 3 (Union of Recursively Monotone Spaces). *If S is the union of (pairwise) separated recursively monotone spaces, then $Z_{(-1)}$ is exact on S.*

Proof. From the additivity of $Z_{(-1)}$ on separated subspaces, it suffices to prove that $Z_{(-1)}$ is exact when S is a single recursively monotone space. We will, in

fact, prove that in this case, $Z_{(-1)}(S) = \#clusters(S) = 1$. We use induction on n, the dimension of the space Dom^n in which S lives. For the base case of $n = 1$, $\#clusters(S)$ equals 1 because all points in S are trivially 1-flip away from each other, and $Z_{(-1)}(S)$ equals 1 because it is simply the difference between the number of odd and even size non-empty subsets of S, which can be easily verified to be 1. For $n \geq 2$, let x_i be a u-monotone variable in S. From Lemma 6, we have that $\#clusters(S) = \#clusters(S|_{x_i=u})$. Further, from Lemma 7, we have that $Z_{(-1)}(S) = Z_{(-1)}(S|_{x_i=u})$. Thus, $Z_{(-1)}(S) = \#clusters(S) = 1$ if and only if $Z_{(-1)}(S|_{x_i=u}) = \#clusters(S|_{x_i=u}) = 1$. Since $S|_{x_i=u}$ is a recursively monotone space of dimension $n-1$, this last fact follows from the inductive hypothesis, finishing the proof. □

The following proposition highlights an insightful property proved as part of the above proof:

Proposition 8. *If S is recursively monotone, then $\#clusters(S) = Z_{(-1)}(S) = 1$.*

This implies, for example, that the solution spaces of satisfiable polytree formulas and monotone formulas have a single cluster.

5.2.5 Marginals of $\mathbf{Z_{(-1)}}$

For a given problem instance, we can define a *cluster marginal* of a variable x_i to be the fraction of solution clusters in which x_i only appears with one particular value (i.e. x_i is a backbone of the clusters). Since $Z_{(-1)}$ counts well the number of clusters, it is natural to ask whether it is also possible to obtain the marginal information. Indeed, $Z_{(-1)}$ provides an estimate for cluster marginals, and we call them $Z_{(-1)}$ *marginals*. Moreover, for binary domains, one can prove that the $Z_{(-1)}$ marginals can be used to compute the cluster marginals if a generalized

notion of recursive monotonicity holds. This section discusses the details of this construction.

Let us use $Z_{(-1)}(y_i = v)$ to denote the value of $Z_{(-1)}$ on a subspace $S \subseteq Dom^n$ where the extended variable y_i is fixed to $v \in DomExt$. That is $Z_{(-1)}(y_i = v) = \sum_{\substack{\vec{y} \in DomExt^n \\ y_i = v}} (-1)^{\#e(\vec{y})} \S(\vec{y})$. Note that there is a tight relationship between $Z_{(-1)}$ on restricted subspaces, and $Z_{(-1)}$ with fixed variables. In particular, $Z_{(-1)}(S|_{x_i=u}) = Z_{(-1)}(y_i = \{u\})$ for $u \in Dom$.

Definition 11 ($Z_{(-1)}$-Marginal). The **$Z_{(-1)}$-marginals** of a solution space $S \subseteq Dom^n$ for some varible i is a $|DomExt|$-tuple $(m_i^v)_{v \in DomExt}$ defined as:

$$m_i^v := \frac{Z_{(-1)}(y_i = v)}{Z_{(-1)}} = \frac{\sum_{\substack{\vec{y} \in DomExt^n \\ y_i = v}} (-1)^{\#e(\vec{y})} \S(\vec{y})}{\sum_{\vec{y} \in DomExt^n} (-1)^{\#e(\vec{y})} \S(\vec{y})} \tag{5.4}$$

for $v \in DomExt$.

This definition reflects exactly the concept of marginal probability in factor graphs representing a probability space. However, our factor graph can assign negative weights to assignments (and thus some items in the marginal vector will be negative), so it does not capture a probability mass function in general. But as the theorem proved at the end of this section shows, if the solution space S has favorable properties, the $Z_{(-1)}$-marginals $(m_i^v)_{v \in DomExt}$ can be used to compute the fraction of clusters where x_i is fixed to some value. We begin with a natural generalization of the concept of recursively monotone spaces, which will be needed for the proof of the theorem.

Definition 12 (k-Recursively Monotone Spaces). A space $S \in Dom^n$ is **k-recursively monotone** for $k \in \{0, \ldots, n\}$ if it is nonempty and for any $\ell, 0 \leq \ell \leq k$, variables $I \subseteq \{1, \ldots, n\}, |I| = \ell$, and every assignment $\vec{v}_I \in DomExt^\ell$ to $\vec{x}_I$, $S|_{\vec{x}_I \sim \vec{v}_I}$ is either recursively monotone or empty.

Definition 12 itself shows that recursive monotonicity is indeed a monotone property: if a space S is k-recursively monotone, it is also $(k-1)$-recursively monotone. Also, 0-recursive monotonicity is equivalent to recursive monotonicity.

Corollary 2. *Let S be a union of k-recursively monotone (pairwise) separated subspaces $S_i, 1 \leq i \leq t$, i.e. $S = \bigcup_{i=1}^{t} S_i$. Then for any $I \subseteq \{1, \ldots, n\}, |I| \leq k$, and any $\vec{v}_I \in DomExt^{|I|}$, it holds that $S|_{\vec{x}_I \sim \vec{v}_I} = \bigcup_{i=1}^{t} S_i|_{\vec{x}_I \sim \vec{v}_I}$ is a union of empty or recursively monotone (pairwise) separated subspaces $S_i|_{\vec{x}_I \sim \vec{v}_I}$.*

Proof. By definition of k-recursive monotonicity, we know that $S_i|_{\vec{x}_I \sim \vec{v}_I}$ is either empty or recursively monotone for any i. Moreover, any two nonempty spaces $S_i|_{\vec{x}_I \sim \vec{v}_I}$ and $S_j|_{\vec{x}_I \sim \vec{v}_I}$, $i \neq j$, must necessarily be separated: if they were not, there would exist $\vec{x}'_i \in S_i|_{\vec{x}_I \sim \vec{v}_I}$ and $\vec{x}'_j \in S_j|_{\vec{x}_I \sim \vec{v}_I}$ such that $dist(\vec{x}'_i, \vec{x}'_j) \leq 1$. Extending these vectors with $\vec{v}_I$, we have $(\vec{x}'_i, \vec{v}_I) \in S_i$, $(\vec{x}'_j, \vec{v}_I) \in S_j$ and $dist((\vec{x}'_i, \vec{v}_I), (\vec{x}'_j, \vec{v}_I)) = dist(\vec{x}'_i, \vec{x}'_j) \leq 1$. This is a contradiction to S_i and S_j being separated. □

Proposition 9. *For every $k \in \{0, 1, \ldots, n\}$, the following nonempty spaces are k-recursively monotone:*

- *hypercubes, in dom^n for any domain Dom,*
- *solution space of satisfiable polytree boolean formulas on n variables, and*
- *solution space of monotone boolean formulas (i.e., formulas where every variable occurs in only one polarity; not necessarily polytrees) on n variables.*

Proof. Hypercubes. Let us start with the simplest case: recursive monotonicity of $S = \{\vec{u}\}$, a single point in Dom^n. Each x_i is trivially u_i-monotone by definition, and by induction on n, $S|_{x_i = u_i}$ is recursively monotone. This is so by definition for $n = 1$, and for $n > 1$ it is true by induction hypothesis because $S|_{x_i = u_i}$ is still a single point (in Dom^{n-1}). For general hypercubes with at least two points, we will

represent S by a vector of generalized assignments $\vec{y} \in DomExt^n$. If $n = 1$, then S is recursively monotone by definition. Otherwise, since $|S| > 1$, there is at least one i for which $|y_i| > 1$. Let us pick an arbitrary point in the hypercube $\vec{u} \in S = \vec{y}$. Now x_i is u_i-monotone in S, because variable values are assigned independently for each variable, and thus $\vec{x} \in S$ implies $\vec{x}[x_i \leftarrow u_i] \in S$. Moreover, $S|_{x_i=u_i}$ is also a hypercube, represented by $\vec{y}[y_i \leftarrow \{u_i\}]$, and it is strictly smaller than S. By induction on $|S|$, we get that $S|_{x_i=u_i}$ is recursively monotone.

This shows that S is recursively monotone. Now, for k-recursive monotonicity for $k \geq 1$, observe that for any k variables $I \subseteq \{1, \ldots, n\}, |I| = k$ and any values $v_I \in DomExt^k$, $S|_{\vec{x}_I=\vec{v}_I}$ is either a hypercube (which is recursively monotone by the above reasoning) or empty. Therefore S is also k-recursively monotone.

Polytrees. Consider the factor graph G_F of a polytree formula F whose solution space is S. If F is the empty formula, then S is the full hypercube and is therefore k-recursively monotone for every k. Otherwise, F has at least one variable. Consider any leaf of G_F (which must exist because G_F is a forest). If the leaf is a clause node, we have a unit clause which yields a backbone variable and thus a monotone variable. If the leaf is a variable node, this variable occurs in only one clause of F, which yields a pure literal and thus a monotone variable. In either case, for some $u \in \{0, 1\}$, we have a u-monotone variable x in S. Setting x to u in F yields $S|_{x=u}$, which is the non-empty solution space of $F|_{x=u}$. Since $F|_{x=u}$ is a satisfiable polytree formula with one less variable than F, by induction on the number of variables appearing in the formula, the solution space of $F|_{x=u}$ is recursively monotone, and therefore S is recursively monotone. Now, for k-recursive monotonicity, notice that for any k variables $I \subseteq \{1, \ldots, n\}, |I| = k$ and any values v_I, $F|_{\vec{x}_I \sim \vec{v}_I}$ is either unsatisfiable (and corresponds to the empty space) or satisfiable with a polytree factor graph. Therefore, S is also k-recursive monotone.

Monotone formulas. The argument for monotone formulas is very similar to that for polytrees, the key things to note being that (1) such formulas are always satisfiable and (2) setting any subset of formula variable to any values or syntactically deleting any variables yields a smaller monotone formula. □

Theorem 4. *Let $S \subseteq \{0,1\}^n$ be a solution space in a binary domain which is a union of t (pairwise) separated 1-recursively monotone spaces S_j. For each variable x_i, let t_i^0, t_i^1, and t_i^* be the number of these t subspaces in which x_i only appears with value 0, 1 in the cluster, or with both, respectively (so that $t = t_i^0 + t_i^1 + t_i^*$). Then*

$$t_i^0 = Z_{(-1)}(y_i = \{0\}) + Z_{(-1)}(y_i = \{0,1\})$$
$$t_i^1 = Z_{(-1)}(y_i = \{1\}) + Z_{(-1)}(y_i = \{0,1\})$$
$$t_i^* = -Z_{(-1)}(y_i = \{0,1\})$$

Proof. We observe, using Corollary 2, that $S|_{x_i=0} = \bigcup_{j=1}^{t} S_j|_{x_i=0}$ with $S_j|_{x_i=0}$ empty or recursively monotone and pairwise separated. We also know that exactly $t_i^0 + t_i^*$ of $S_j|_{x_i=0}$ are nonempty (those subspaces without $x_i = 1$ being its backbone), and the recursive monotonicity assures that $Z_{(-1)}(S_j|_{x_i=0}) = 1$ for each such nonempty subspace. Proposition 4 (additivity) then shows that $Z_{(-1)}(S|_{x_i=0}) = \sum_j Z_{(-1)}(S_j|_{x_i=0}) = t_i^- + t_i^*$. Also, $Z_{(-1)}(S|_{x_i=0}) = Z_{(-1)}(y_i = \{0\})$ by the note above Definition 11, so we have $Z_{(-1)}(y_i = \{0\}) = t_i^0 + t_i^*$. Analogously for the case of cluster backbones $x_i = 1$. Finally, we know that $S_j|_{x_i \sim \{0,1\}}$ is nonempty for exacly the clusters where x_i is not a backbone. This is because x_i is binary, so if it appears both as 0 and 1, there must be two points in S_j which only differ in x_i, since S_j is recursively monotone, and therefore makes one connected cluster. Again by recursive monotonicity we have that $Z_{(-1)}(S_j|_{x_i \sim \{0,1\}}) = 1$, and by syntactic inspection and additivity we see $Z_{(-1)}(y_i = \{0,1\}) = -Z_{(-1)}(S|_{x_i \sim \{0,1\}}) = -\sum_j Z_{(-1)}(S_j|_{x_i \sim \{0,1\}}) = -t_i^*$. □

The construction from Theorem 4 is not necessarily accurate for non-binary domains. Consider the following counter example: a graph coloring problem, where the graph comprises of two nodes connected by an edge, first vertex can take any of two colors $\{c_1, c_2\}$, second any of three colors $\{c_1, c_2, c_3\}$, and they cannot both have the same color simultaneously. The solution space is one connected cluster $\{(c_1, c_2), (c_1, c_3), (c_2, c_3), (c_2, c_1)\}$. It is easy to see that this is a 1-recursively monotone solution space. There are obviously no cluster backbones, as both variables appear with all their possible values in some solution. However, if we try to compute the "number of clusters where x_2 is fixed to c_3" using a formula analogous to the one in Theorem 4, $t_2^{c_3} = Z_{(-1)}(y_2 = \{c_3\}) + Z_{(-1)}(y_2 = \{c_2, c_3\}) + Z_{(-1)}(y_2 = \{c_1, c_3\}) + Z_{(-1)}(y_2 = \{c_1, c_2, c_3\})$, the expression evaluates to -1 as can be readily verified. The problem is not in exactness of $Z_{(-1)}$ (which still correctly reflects number of clusters after x_2 has been fixed), but in the fact that the inclusion-exclusion style argument using number of clusters does not yield number of clusters with a backbone variable.

5.3 Exact Computation of Number of Clusters and $\mathbf{Z}_{(-1)}$

We will use the $Z_{(-1)}$ expression developed in the previous sections as means to count clusters in solutionspaces of both random CSPs and SAT encodings of other problems. First, we develop techniques to count number of solution clusters and evaluate $Z_{(-1)}$ exactly, to assess whether $Z_{(-1)}$ works on these solutionspaces. The exact computation will in general not scale very well, so we also develop a message passing algorithm, analogous to BP, to compute approximations to $Z_{(-1)}$.

Obtaining the exact number of clusters for reasonable size problems is crucial

for evaluating our proposed approach based on $Z_{(-1)}$ and the corresponding BP equations to follow later in the article. A naïve way is to explicitly enumerate all solutions, compute their pairwise Hamming distances, and infer the cluster structure. Not surprisingly, this method does not scale well because the number of solutions typically grows exponentially as the number of variables of the CSP increases. We discuss here a much more scalable approach that uses two advanced techniques to this effect: disjunctive negation normal form (DNNF) and binary decision diagrams (BDDs). Our method scales to graph coloring problems with a few hundred variables (see experimental results) for computing both the exact number of clusters and the exact value of $Z_{(-1)}$.

Both DNNF [Darwiche, 2001] and BDD [Bryant, 1986] are graph based data structures that have proven to be very effective in "knowledge compilation", i.e., in converting a 0-1 function F into a (potentially exponentially long, but often reasonably sized) standard form from which various interesting properties of F can be inferred easily, often in linear time in the size of the DNNF formula or BDD. For our purposes, we use DNNF to succinctly represent all solutions of F and a set of BDDs to represent solution clusters that we create as we traverse the DNNF representation. The only relevant details for us of these two representations are the following: (1) DNNF is represented as an acyclic directed graph with variables and their negations at the leaves and two kinds of internal nodes, "or" and "and"; "or" nodes split the set of solutions such that they differ in the value of the variable labeling the node but otherwise have identical variables; "and" nodes partition the space into disjoint sets of variables; (2) BDDs represent arbitrary sets of solutions and support efficient intersection and projection (onto a subset of variables) operations on these sets.

We use the compiler `c2d` [Darwiche, 2004] to obtain the DNNF form for F.

Since `c2d` works on Boolean formulas and our F often has non-Boolean domains, we first convert F to a Boolean function F' using a unary encoding, i.e., by replacing each variable x_i of F with domain size t with t Boolean variables $x'_{i,j}, 1 \leq j \leq t$, respecting the semantics: $x_i = j$ iff $x_{i,j} = 1$. In order to ensure that F and F' have similar cluster structure of solutions, we relax the usual condition that only one of $x_{i,1}, \ldots, x_{i,t}$ may be 1, thus effectively allowing the original x_i to take multiple values simultaneously. This yields a *generalized function*: the domains of the variables of F' correspond to the power sets of the domains of the respective variables of F. This generalization has the following useful property: if two solutions $\vec{x}^{(1)}$ and $\vec{x}^{(2)}$ are neighbors in the solutionspace of F, then the corresponding solutions $\vec{x}'^{(1)}$ and $\vec{x}'^{(2)}$ are in the same cluster in the solutionspace of F'.

5.3.1 Computing number of clusters

Given F', we run `c2d` on it to obtain an implicit representation of all solutions as a DNNF formula F''. Next, we traverse F'' from the leaf nodes up, creating clusters as we go along. Specifically, with each node U of F'', we associate a set S_U of BDDs, one for each cluster in the sub-formula contained under U. The set of BDDs for the root node of F'' then corresponds precisely to the set of solution clusters of F', and thus of F. These BDDs are computed as follows. If U is a leaf node of F'', it represents a Boolean variable or its negation and S_U consists of the single one-node BDD corresponding to this Boolean literal. If U is an internal node of F'' labeled with the variable x_U and with children L and R, the set of BDDs S_U is computed as follows. If U is an "or" node, then we consider the union $S_L \cup S_R$ of the two sets of BDDs and merge any two of these BDDs if they are adjacent, i.e., have two solutions that are neighbors in the solutionspace (since the DNNF form guarantees that the BDDs in S_L and S_R already must differ in the

value of the variable x_U labeling U, the adjacency check is equivalent to testing whether the two BDDs, with x_U projected out, have a solution in common; this is a straightforward projection and intersection operation for BDDs); in the worst case, this leads to $|S_L| + |S_R|$ cluster BDDs in S_U. Similarly, if U is an "and" node, then S_U is constructed by considering the cross product $\{b_L \text{and } b_R \mid b_L \in S_L, b_R \in S_R\}$ of the two sets of BDDs and merging adjacent resulting BDDs as before; in the worst case, this leads to $|S_L| \cdot |S_R|$ cluster BDDs in S_U.

5.3.2 Evaluating $\mathbf{Z}_{(-1)}$

The exact value of $Z_{(-1)}$ on F' can also be evaluated easily once we have the DNNF representation F''. In fact, as is reflected in our experimental results, evaluation of $Z_{(-1)}$ is a much more scalable process than counting clusters because it requires a simple traversal of F'' without the need for maintaining BDDs. With each node U of F'', we associate a value V_U which equals precisely the difference between the number of solutions below U with an even number of positive literals and those with an odd number of positive literals; $Z_{(-1)}$ then equals $(-1)^N$ times the value thus associated with the root node of F''. These values are computed bottom-up as follows. If U is a leaf node labeled with a positive (or negative) literal, then $V_U = -1$ (or 1, resp.). If U is an "or" node with children L and R, then $V_U = V_L + V_R$. This works because L and R have identical variables. Finally, if U is an "and" node with children L and R, then $V_U = V_L V_R$. This last computation works because L and R are on disjoint sets of variables and because of the following observation. Suppose L has V_L^e solutions with an even number of positive literals and V_L^o solutions with an odd number of positive literals; similarly for R. Then $V_U = (V_L^e V_R^e + V_L^o V_R^o) - (V_L^e V_R^o + V_L^o V_R^e) = (V_L^e - V_L^o)(V_R^e - V_R^o) = V_L V_R$.

5.4 Belief Propagation Inference for Clusters

We present a version of the Belief Propagation algorithm that allows us to deal with the alternating signs of $Z_{(-1)}$ in general factor graphs. The derivation follows closely the one given by Yedidia, Freeman, and Weiss [2005] for a standard BP (and reviewed in Chapter 2), i.e. we will write equations for a stationary point of KL divergence of two sequences (not necessarily probability distributions in our case). Since the $Z_{(-1)}$ expression involves both positive and negative terms, we must appropriately generalize some of the steps.

Given a function $p(\vec{y})$ (the *target* function, with real numbers as its range) on $DomExt^n$ that is known up to a normalization constant but with unknown marginal sums, we seek a function $b(\vec{y})$ (the *trial* function) to approximate $p(\vec{y})$, such that b's marginal sums are known. The target function $p(\vec{y})$ is defined as

$$p(\vec{y}) := \frac{1}{Z_{(-1)}}(-1)^{\#e(\vec{y})} \prod_{\alpha} f_{\alpha}^{ext}(\vec{y}_{\alpha}) \tag{5.5}$$

We adopt previously used notation from Yedidia et al. [2005]: $\vec{y}_{\alpha}$ are values in $\vec{y}$ of variables that appear in factor (i.e. vertex) f_{α}^{ext}; $\vec{y}_{-i}$ are values of all variables in $\vec{y}$ *except* y_i. The marginal sums can be extended in a similar way to allow for any number of variables fixed in $\vec{y}$, specified by the subscript. When convenient, we treat the symbol α as a set of indexes of variables in f_{α}^{ext}, to be able to index them. We begin by listing the assumptions used in the derivation, both the ones that are used in the "standard" BP, and two additional ones needed for the generalization. An assumption on $b(\vec{y})$ is *legitimate* if the corresponding condition holds for $p(\vec{y})$.

5.4.1 Assumptions

The *standard assumptions,* present in the derivation of standard BP [Yedidia et al., 2005], are:

- Marginalization: $b_i(y_i) = \sum_{\vec{y}_{-i}} b(\vec{y})$ and $b_\alpha(\vec{y}_\alpha) = \sum_{\vec{y}_{-\alpha}} b(\vec{y})$. This condition is legitimate, but cannot be enforced. Moreover, it might happen that the solution found by BP does not satisfy it, which is a known problem with BP [Mackay et al., 2001].

- Normalization: $\sum_{y_i} b_i(y_i) = \sum_{\vec{y}_\alpha} b_\alpha(\vec{y}_\alpha) = 1$. It is legitimate, and explicitly enforced.

- Consistency: $\forall \alpha, i \in \alpha, y_i : b_i(y_i) = \sum_{\vec{y}_{\alpha \setminus i}} b_\alpha(\vec{y}_\alpha)$. This is a legitimate assumption and explicitly enforced.

- Tree-like decomposition: says that the weights $b(\vec{y})$ of each configuration can be obtained from the marginal sums as follows (d_i is a degree of the variable node y_i in the factor graph):

$$|b(\vec{y})| = \frac{\prod_\alpha |b_\alpha(\vec{y}_\alpha)|}{\prod_i |b_i(y_i)|^{d_i - 1}} \tag{5.6}$$

  (The standard assumption is without the absolute values.) This assumption is not legitimate, and it is built-in, i.e. it is used in the derivation of the BP equations.

To appropriately handle the signs of $b(\vec{x})$ and $p(\vec{y})$, we have two *additional assumptions*. These are necessary for the BP derivation applicable to $Z_{(-1)}$, but not for standard BP equations.

- Sign-correspondence: $b(\vec{y})$ and $p(\vec{y})$ have the same signs for all configurations $\vec{y}$ (zero, being a singular case, is treated as having a positive sign). This is a built-in assumption and legitimate.

- Sign-alternation: $b_i(y_i)$ is negative iff $|y_i|$ is even, and $b_\alpha(\vec{y}_\alpha)$ is negative iff $\#e(\vec{y}_\alpha)$ is odd. This is also a built-in assumption, but not necessarily legitimate, whether or not it is legitimate depends on the structure of the solutionspace of a particular problem.

The Sign-alternation assumption can be viewed as an application of the inclusion-exclusion principle, and is easy to illustrate on a graph coloring problem with only two colors. In such case if $F^{ext}(\vec{y}) = 1$, then $y_i = \{c_1\}$ means that y_i can have color 1, $y_i = \{c_2\}$ that y_i can have color 2, and $y_i = \{c_1, c_2\}$ that y_i can have *both* colors. The third event is included in the first two, and its probability must thus appear with a negative sign if the sum of probabilities is to be 1. Whether or not the sign-alternation assumption is legitimate depends on the structure of the solutionspace of a particular problem. The following lemma shows a connection to the notion of k-simplicity:

Lemma 8. *Let P be a satisfiable CSP with solutionspace S, and m be the maximum number of variables appearing in any constraint. Then if S is a union of separated m-recursively monotone spaces, the sign-alternation assumption is legitimate for P.*

Proof. We can write an arbitrary marginal sum of $p(\vec{y})$ for some fixed set $I \subseteq V, |I| \leq m$ of variables fixed to $\vec{y}_I \in DomExt^{|I|}$ as:

$$\begin{aligned} p_I(\vec{y}_I) &= \sum_{\vec{y}_{-I} \in DomExt^{|V \setminus I|}} p(\vec{y}) \\ &= \frac{1}{Z_{(-1)}} (-1)^{\#e(\vec{y}_I)} \sum_{\vec{y}_{-I} \in DomExt^{|V \setminus I|}} (-1)^{\#e(\vec{y}_{-I})} S|_{\vec{y}_I}(\vec{y}_{-I}) \\ &= \frac{Z_{(-1)}(S|_{\vec{y}_I})}{Z_{(-1)}(S)} (-1)^{\#e(\vec{y}_I)} \end{aligned}$$

Using Theorem 3 on S along with the assumption that P is satisfiable, we know that $Z_{(-1)}(S) > 0$. In addition, using the assumption of m-simplicity of S and $|I| \leq m$, Corollary 2 shows that $S|_{\vec{y}_I}$ is a collection of recursively monotone spaces for any setting $\vec{y}_I$. This means that $Z_{(-1)}(S|_{\vec{y}_I}) \geq 0$ according to Theorem 3. Thus the sign of $p_I(\vec{y}_I)$ is negative iff the number of even-sized element of $\vec{y}_I$ is odd.

Applying this result to sets $I = \{i\}$ for all variables x_i in P, and $I = \{i \in \alpha\}$ for all constraints α in P yields that $sign(p_i(y_i)) = (-1)^{\delta(|y_i| \text{ is even})}$ and $sign(p_\alpha(\vec{y}_\alpha)) = (-1)^{\#e(\vec{y}_\alpha)}$. This is equivalent to the sign-alternation assumption for P, and thus the assumption is legitimate. □

5.4.2 Kullback-Leibler divergence

The KL-divergence is traditionally defined for probability distributions, for sequences of non-negative terms in particular. We need a more general measure, as our sequences $p(\vec{y})$ and $b(\vec{y})$ have alternating signs. But using the Sign-correspondence assumption, we observe that the usual definition of KL-divergence is still applicable, since the term in the logarithm is non-negative:

$$D(b \parallel p) := \sum_{\vec{y} \in DomExt^n} b(\vec{x}) \log \frac{b(\vec{y})}{p(\vec{y})} = \sum_{\vec{y} \in DomExt^n} b(\vec{x}) \log \frac{|b(\vec{y})|}{|p(\vec{y})|}$$

Moreover, the following Lemma shows that the two properties of KL-divergence that make it suitable for distance-minimization are still valid.

Lemma 9. *Let $b(.)$ and $p(.)$ be (possibly negative) weight functions on the same domain $\mathcal{D}$, with the property that they agree on signs for all states (i.e. $\forall \vec{y} \in \mathcal{D}$: $sign(b(\vec{x})) = sign(p(\vec{y}))$), and that they sum to the same constant (i.e. $\sum_{\vec{y}} b(\vec{y}) = \sum_{\vec{y}} p(\vec{y}) = c$). Then the KL-divergence $D(b \parallel p)$ satisfies $D(b \parallel p) \geq 0$ and $D(b \parallel p) = 0 \Leftrightarrow b \equiv p$.*

The proof is essentially identical to the equivalent statement made about KL-divergence of probability distributions.

5.4.3 Minimizing divergence

We write $p(\vec{x}) = sign(p(\vec{y})) \cdot |p(\vec{y})|$, and analogously for $b(\vec{y})$. This allows us to isolate the signs, and the minimization follows exactly the steps of standard BP

derivation, namely we write a set of equations characterizing stationary points of $D(b \parallel p)$. At the end, using the Sign-alternation assumption, we are able to implant the signs back. The details of the variational derivations follow.

Problem Statement The goal is to minimize the KL divergence $D(b \parallel p)$ between the target function $p(.)$ and the trial function $b(.)$. Substituting expressions (5.5,5.6) into the definition of $D(b \parallel p)$, and using the marginalization assumption we obtain the following:

$$\begin{aligned} D(b \parallel p) = &- \sum_{\alpha} \sum_{\vec{y}_\alpha} b_\alpha(\vec{y}_\alpha) \log f_\alpha^{ext}(\vec{y}_\alpha) + \sum_{\alpha} \sum_{\vec{y}_\alpha} b_\alpha(\vec{y}_\alpha) \log |b_\alpha(\vec{y}_\alpha)| \\ &- \sum_i (d_i - 1) \sum_{y_i} b_i(y_i) \log |b_i(y_i)| + \log Z_{(-1)} \end{aligned}$$

The last term, $\log Z_{(-1)}$, is a constant (albeit unknown) and does not depend on any $b_i(.)$ or $b_\alpha(.)$, the unknowns we now have in the problem. Since the goal is to minimize $D(b \parallel p)$, we can safely leave out the constant, as that will not change the point where the minimum is attained. Let us call the modified expression $\tilde{D}(b \parallel p)$.

We are now ready to state the main problem:

Minimize:

$$\begin{aligned} \tilde{D}(b \parallel p) = &- \sum_{\alpha} \sum_{\vec{y}_\alpha} b_\alpha(\vec{y}_\alpha) \log f_\alpha^{ext}(\vec{y}_\alpha) + \sum_{\alpha} \sum_{\vec{y}_\alpha} b_\alpha(\vec{y}_\alpha) \log |b_\alpha(\vec{y}_\alpha)| \\ &- \sum_i (d_i - 1) \sum_{y_i} b_i(y_i) \log |b_i(y_i)| \end{aligned} \tag{5.7}$$

as an expression in variables $b_i(y_i)$ and $b_\alpha(\vec{y}_\alpha)$ (for every state of $y_i \in DomExt$ and $\vec{y}_\alpha \in DomExt^{|\alpha|}$ a separate variable). They represent the corresponding marginal beliefs, i.e. estimates of marginal sums $p_i(y_i)$ and $p_\alpha(y_\alpha)$. The d_i terms are degrees of variable nodes i.

Subject to:

$$\sum_{y_i \in DomExt} b_i(y_i) = 1 \quad \forall i \tag{5.8}$$

$$\sum_{\vec{y}_\alpha \in DomExt^{|\alpha|}} b_\alpha(\vec{y}_\alpha) = 1 \quad \forall \alpha \tag{5.9}$$

$$\sum_{\vec{y}_{\alpha \setminus i} \in DomExt^{|\alpha|-1}} b_\alpha(\vec{y}_\alpha) = b_i(y_i) \quad \forall \alpha, i \in \alpha, y_i \in DomExt \tag{5.10}$$

The conditions (5.8,5.9) enforce the normalization assumptions, and condition (5.10) enforces consistency.

This problem statement is exactly the same as the original for deriving BP. The difference is that we allow $b(.)$ to be *negative* in a controlled way (there is an implicit assumption that it is non-negative in the derivation of standard BP).

Lagrangian To solve the constrained minimization problem (5.7), we turn it into an unconstrained problem, and will search for stationary points. We obtain the unconstrained version using Lagrangian multipliers λ_i for each condition (5.8), λ_α for each condition (5.9), and $\lambda_{\alpha i}(y_i)$ for each condition (5.10). The associated Lagrangian is then:

$$\begin{aligned}
\Lambda(b_i, b_\alpha, \lambda_i, \lambda_\alpha, \lambda_{\alpha i}) \;=\; & -\textstyle\sum_\alpha \sum_{\vec{y}_\alpha} b_\alpha(\vec{y}_\alpha) \log f_\alpha^{ext}(\vec{y}_\alpha) + \\
& + \textstyle\sum_\alpha \sum_{\vec{y}_\alpha} b_\alpha(\vec{y}_\alpha) \log |b_\alpha(\vec{y}_\alpha)| \\
& - \textstyle\sum_i (d_i - 1) \sum_{y_i} b_i(y_i) \log |b_i(y_i)| \\
& + \textstyle\sum_i \lambda_i \left(\sum_{y_i} b_i(y_i) - 1 \right) \\
& + \textstyle\sum_\alpha \lambda_\alpha \left(\sum_{\vec{y}_\alpha} b_\alpha(\vec{y}_\alpha) - 1 \right) \\
& + \textstyle\sum_\alpha \sum_{i \in \alpha} \sum_{y_i} \lambda_{\alpha i}(y_i) \left(b_i(y_i) - \sum_{\vec{y}_{\alpha \setminus i}} b_\alpha(\vec{y}_\alpha) \right)
\end{aligned} \tag{5.11}$$

Setting the partial derivatives of Λ w.r.t. $b_i(y_i)$ and $b_\alpha(\vec{y}_\alpha)$ to zero yields the following (notice that $\frac{d}{dx} x \log |x| = \log |x| + 1$, so we do not need to worry about

the signs):

$$\frac{\partial \Lambda}{\partial b_i(y_i)}: \qquad -(d_i-1)(\log|b_i(y_i)|+1)+\sum_{\alpha\ni i}\lambda_{\alpha i}(y_i)+\lambda_i=0 \tag{5.12}$$

$$\frac{\partial \Lambda}{\partial b_\alpha(\vec{y}_\alpha)}: \quad -\log f_\alpha^{ext}(\vec{y}_\alpha)+\log|b_\alpha(\vec{y}_\alpha)|+1-\sum_{i\in\alpha}\lambda_{\alpha i}(y_i)+\lambda_\alpha=0 \tag{5.13}$$

which can be rewritten as:

$$\begin{aligned}\log|b_i(y_i)| &= \tfrac{1}{d_i-1}\textstyle\sum_{\alpha\ni i}\lambda_{\alpha i}(y_i)+\tfrac{\lambda_i}{d_i-1}-1\\ \log|b_\alpha(\vec{y}_\alpha)| &= \log f_\alpha^{ext}(\vec{y}_\alpha)+\textstyle\sum_{i\in\alpha}\lambda_{\alpha i}(y_i)-\lambda_\alpha-1\end{aligned}$$

The terms involving λ_i and λ_α plus the constants serve as normalizing constants (since the marginal sums b_i and b_α should sum to one). We will therefore use the "$\propto$" symbol in what follows, dropping these terms. Exponentiating both sides, we arrive at:

$$\begin{aligned}|b_i(y_i)| &\propto \left(\textstyle\prod_{\alpha\ni i} e^{\lambda_{\alpha i}(y_i)}\right)^{\frac{1}{d_i-1}}\\ |b_\alpha(\vec{y}_\alpha)| &\propto f_\alpha^{ext}(\vec{y}_\alpha)\textstyle\prod_{i\in\alpha} e^{\lambda_{\alpha i}(y_i)}\end{aligned} \tag{5.14}$$

Note that we know which of the b_i and b_α terms are non-positive (using sign-alternation assumption), so we can retrieve the sign of the expressions that way.

Building BP$_{(-1)}$ equations We will work with equations (5.14) to get them into a message-update form.

First define $n_{i\to\alpha}(y_i) := e^{\lambda_{\alpha i}(y_i)}$. Then find $\nu_{\beta i}(y_i)$ such that $\lambda_{\alpha i}(y_i) = \sum_{\beta\ni i\setminus\alpha}\nu_{\beta i}(y_i)$. That is always possible since we are solving $A\vec{\nu}_{.i}(y_i) = \vec{\lambda}_{.i}(y_i)$ where $\vec{\lambda}_{.i}(y_i) = (\lambda_{\alpha i}(y_i))_{\alpha\ni i}$ and similarly for $\vec{\nu}_{.i}(y_i)$. The matrix A is all ones with zeros on diagonal. It is clearly invertible, and thus $\vec{\nu}_{.i}(y_i)$ is uniquely defined by $\vec{\lambda}_{.i}(y_i)$. Finally define $m_{\beta\to i}(y_i) := e^{\nu_{\beta i}(y_i)}$, and thus $n_{i\to\alpha}(y_i) = \prod_{\beta\ni i\setminus\alpha} m_{\beta\to i}(y_i)$.

With these definitions, we have

$$
\begin{aligned}
|b_i(y_i)| &\propto \left(\prod_{\alpha \ni i} n_{i\to\alpha}(y_i)\right)^{\frac{1}{d_i-1}} = \left(\prod_{\alpha \ni i} \prod_{\beta \ni i \setminus \alpha} m_{\beta\to i}(y_i)\right)^{\frac{1}{d_i-1}} \\
&= \prod_{\alpha \ni i} m_{\alpha\to i}(y_i) = n_{i\to\alpha}(y_i) m_{\alpha\to i}(y_i) \quad \text{for any } \alpha \ni i \qquad (5.15) \\
|b_\alpha(\vec{y}_\alpha)| &\propto f_\alpha^{ext}(\vec{y}_\alpha) \prod_{i\in\alpha} n_{i\to\alpha}(y_i)
\end{aligned}
$$

Thus, similar to normal BP, we can write $|b_i(y_i)| \propto n_{i\to\alpha}(y_i) m_{\alpha\to i}(y_i)$ for any $\alpha \ni i$, and $|b_\alpha(\vec{y}_\alpha)| \propto f_\alpha^{ext}(\vec{y}_\alpha) \prod_{i\in\alpha} n_{i\to\alpha}(y_i)$, and the difference is only the absolute values.

The consistency condition (5.10) dictates that for any $\alpha, i \in \alpha, y_i \in DomExt$:

$$
\begin{aligned}
b_i(y_i) &= \textstyle\sum_{\vec{y}_{\alpha\setminus i} \in DomExt^{|\alpha|-1}} b_\alpha(\vec{y}_\alpha) \\
&\text{and therefore:} \\
sign(b_i(y_i)) n_{i\to\alpha}(y_i) m_{\alpha\to i}(y_i) &\propto \textstyle\sum_{\vec{y}_{\alpha\setminus i}} sign(b_\alpha(\vec{y}_\alpha)) f_\alpha^{ext}(\vec{y}_\alpha) \prod_{j\in\alpha} n_{j\to\alpha}(y_j) \\
&\text{using the sign-alternation:} \\
(-1)^{\delta(|y_i| \text{ is even})} n_{i\to\alpha}(y_i) m_{\alpha\to i}(y_i) &\propto \textstyle\sum_{\vec{y}_{\alpha\setminus i}} (-1)^{\#e(\vec{y}_\alpha)} f_\alpha^{ext}(\vec{y}_\alpha) \prod_{j\in\alpha} n_{j\to\alpha}(y_j) \\
(-1)^{\delta(|y_i| \text{ is even})} n_{i\to\alpha}(y_i) m_{\alpha\to i}(y_i) &\propto \textstyle\sum_{\vec{y}_{\alpha\setminus i}} f_\alpha^{ext}(\vec{y}_\alpha) \prod_{j\in\alpha} (-1)^{\delta(|y_j| \text{ is even})} n_{j\to\alpha}(y_j) \\
&\text{and if } n_{i\to\alpha}(y_i) \neq 0: \\
m_{\alpha\to i}(y_i) &\propto \textstyle\sum_{\vec{y}_{\alpha\setminus i}} f_\alpha^{ext}(\vec{y}_\alpha) \prod_{j\in\alpha\setminus i} (-1)^{\delta(|y_j| \text{ is even})} n_{j\to\alpha}(y_j)
\end{aligned}
$$

5.4.4 BP equations for $Z_{(-1)}$

The resulting modified BP updates (denoted $\text{BP}_{(-1)}$) are, for $y_i \in DomExt$:

$$
n_{i\to\alpha}(y_i) = \prod_{\beta \ni i \setminus \alpha} m_{\beta\to i}(y_i) \qquad (5.16)
$$

$$
m_{\alpha\to i}(y_i) \propto \sum_{\vec{y}_{\alpha\setminus i} \in DomExt^{|\alpha|-1}} f_\alpha^{ext}(\vec{y}_\alpha) \prod_{j\in\alpha\setminus i} (-1)^{\delta(|y_j| \text{ is even})} n_{j\to\alpha}(y_j) \qquad (5.17)
$$

(Almost equivalent to standard BP, except for the (-1) term.) One would iterate these equations from a suitable starting point[2] to find a fixed point, and then obtain the beliefs $b_i(y_i)$ and $b_\alpha(\vec{y}_\alpha)$ (i.e. estimates of marginal sums) using the Sign-alternation assumption and the standard BP relations:

$$\begin{aligned} b_i(y_i) &\propto (-1)^{\delta(|y_i| \text{ is even})} \prod_{\alpha \ni i} m_{\alpha \to i}(y_i) \\ b_\alpha(\vec{y}_\alpha) &\propto (-1)^{\#e(\vec{y}_\alpha)} f_\alpha^{ext}(\vec{y}_\alpha) \prod_{i \in \alpha} n_{i \to \alpha}(y_i) \end{aligned} \tag{5.18}$$

To approximately count the number of clusters in large problems for which exact cluster count or exact $Z_{(-1)}$ evaluation is infeasible, we employ the generic $\mathrm{BP}_{(-1)}$ scheme derived above. We substitute the extended factor functions f^{ext}, which define the problem, into Equations (5.16, 5.17), iterate from a random initial starting point to find a fixed point, and then use Equations (5.18) to compute the beliefs. The actual estimate of $Z_{(-1)}$ is obtained with the standard BP formula (with signs properly taken care of), where d_i is a degree of the variable node y_i in the factor graph:

$$\begin{aligned} \log Z_{\mathrm{BP}_{(-1)}} = &- \sum_\alpha \sum_{\vec{y}_\alpha} b_\alpha(\vec{y}_\alpha) \log |b_\alpha(\vec{y}_\alpha)| \\ &+ \sum_i (d_i - 1) \sum_{y_i} b_i(y_i) \log |b_i(y_i)| \end{aligned} \tag{5.19}$$

5.5 Comparison with SP

Survey propagation tries to compute statistics about most frequent clusters [Mezard and Montanari, 2009], while $\mathrm{BP}_{(-1)}$ tries to reason about all clusters as captured by $Z_{(-1)}$. In general, these two algorithm are distinct. For SAT, they happen to be syntactically equivalent (as shown in the supplementary material),

[2] Traditionally, the initial values for n messages are chosen randomly in $[0, 1]$, and the algorithm started from there. In this derivation, we need to be careful to respect the sign-alternation condition in the initial point. One way to do this is to assign messages for y_i s.t. $|y_i| = 1$ randomly, and start the rest of the messages with 0.

perhaps due to simplifying assumptions of BP. But for COL they compute different things, as discussed in section 5.6.

5.5.1 Derivation of SP Equations from $\mathrm{BP}_{(-1)}$ for SAT

The $\mathrm{BP}_{(-1)}$ equations can be further simplified for SAT. This will, perhaps somewhat surprisingly, lead exactly to the set of equations used in Survey Propagation [Braunstein et al., 2005]. We will work with Boolean domain $\{0,1\}$. To simplify notation (and bring it syntactically closer to SP notation), let us denote the elements of the extended domain $DomExt = \{\{0\},\{1\},\{0,1\}\}$ by 0,1,$*$, respectively.

We will massage Equations (5.16, 5.17). Let us move the $(-1)^{\delta(|i_i| \text{ is even})}$ term into the definition of n by substituting $\bar{n}_{i\to\alpha}(y_i) := (-1)^{\delta(y_i=*)} n_{i\to\alpha}(y_i)$ (the m expressions then drop the $(-1)^{\delta(|i_i| \text{ is even})}$ term). We can rescale the $\bar{n}_{i\to\alpha}(.)$ values to sum to one, and using the definition of f_α^{ext} for SAT, we have for $y_i \in \{0,1,*\}$:

$$m_{\alpha\to i}(y_i) = \begin{cases} 1 & \text{if } y_i = sign(\alpha,i) \\ 1 - \prod_{j\in\alpha\setminus i}(\bar{n}_{j\to\alpha}(-sign(\alpha,j)) + \bar{n}_{j\to\alpha}(*)) & \text{otherwise} \end{cases}$$

Let us define $\eta_{\alpha\to i} := \prod_{j\in\alpha\setminus i}(\bar{n}_{j\to\alpha}(-sign(\alpha,j)) + \bar{n}_{j\to\alpha}(*))$. And thus we can write:

$$\begin{aligned} \bar{n}_{i\to\alpha}(sign(\alpha,i)) &= \textstyle\prod_{\beta\in C_\alpha^u(i)}(1-\eta_{\beta\to i}) \\ \bar{n}_{i\to\alpha}(-sign(\alpha,i)) &= \textstyle\prod_{\beta\in C_\alpha^s(i)}(1-\eta_{\beta\to i}) \\ \bar{n}_{i\to\alpha}(*) &= -\textstyle\prod_{\beta\in C(i)\setminus\alpha}(1-\eta_{\beta\to i}) \end{aligned} \tag{5.20}$$

with $C_\alpha^u(i), i\in\alpha$ being clauses where i appears with the *opposite* sign than it has in α; $C_\alpha^s(i)$ clauses where i appears with the *same* sign as it has in α (except for α) and $C(i)$ all clauses where i appears.

Finally, let us define

$$\pi^s_{i\to\alpha} := \bar{n}_{i\to\alpha}(sign(\alpha,i)) + \bar{n}_{i\to\alpha}(*)$$

$$\pi^u_{i\to\alpha} := \bar{n}_{i\to\alpha}(-sign(\alpha,i)) + \bar{n}_{i\to\alpha}(*)$$

$$\pi^*_{i\to\alpha} := -\bar{n}_{i\to\alpha}(*)$$

Note that $\pi^s_{i\to\alpha} + \pi^u_{i\to\alpha} + \pi^*_{i\to\alpha} = \bar{n}_{i\to\alpha}(-) + \bar{n}_{i\to\alpha}(+) + \bar{n}_{i\to\alpha}(*) = 1$. And $\eta_{\alpha\to i} = \prod_{j\in\alpha\setminus i} \pi^u_{j\to\alpha}$.

Substituting the definitions of πs into (5.20) yields exactly the SP equations [Braunstein et al., 2005]:

$$\begin{aligned}
\eta_{\alpha\to i} &= \textstyle\prod_{j\in\alpha\setminus i} \frac{\pi^u_{j\to\alpha}}{\pi^u_{j\to\alpha}+\pi^s_{j\to\alpha}+\pi^*_{j\to\alpha}} \\
\pi^u_{i\to\alpha} &= \left[1 - \textstyle\prod_{\beta\in C^u_\alpha(i)}(1-\eta_{\beta\to i})\right] \textstyle\prod_{\beta\in C^s_\alpha(i)}(1-\eta_{\beta\to i}) \\
\pi^s_{i\to\alpha} &= \left[1 - \textstyle\prod_{\beta\in C^s_\alpha(i)}(1-\eta_{\beta\to i})\right] \textstyle\prod_{\beta\in C^u_\alpha(i)}(1-\eta_{\beta\to i}) \\
\pi^*_{i\to\alpha} &= \textstyle\prod_{\beta\in C(i)\setminus\alpha}(1-\eta_{\beta\to i})
\end{aligned} \tag{5.21}$$

5.5.2 Simplifying $\mathrm{BP}_{(-1)}$ for 3-COL

The $\mathrm{BP}_{(-1)}$ equations can also be simplified for the graph three coloring problem. In this case, we will *not* arrive at the same equations as SP derived by Braunstein, Mulet, Pagnani, Weigt, and Zecchina [2003]. The difference in the resulting algorithms is described in Section 5.6.

The domain now is a set of three colors, $Dom = \{1,2,3\}$, so the extended domain has seven elements: $DomExt = \{\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$. For notational convenience, let us abbreviate the *DomExt* elements to 1,2,3,12,13,23,123, respectively. The constraints of a graph coloring problem are associated with the edges of the graph to be colored (with the semantics that the adjacent vertices need to have different colors), so we use notation $f_{(i,j)}(x_i,x_j)$ to denote the constraint that node i and j must have different colors. The extended semantics, that is semantics of factor

functions operating on the extended domain, $f^{ext}_{(i,j)}(y_i, y_j)$, is, by definition, a characteristic function that the two vertices have disjoint set of colors, that is $y_i \cap y_j = \emptyset$.

The update equations of the $\mathrm{BP}_{(-1)}$ are completely symmetric in the colors, so we only list the update messages for *DomExt* values 1,12, and 123.

$$\begin{aligned} m_{(i,j)\to j}(y_j = 1) &\propto n_{i\to(i,j)}(2) + n_{i\to(i,j)}(3) - n_{i\to(i,j)}(23) \\ m_{(i,j)\to j}(y_j = 12) &\propto n_{i\to(i,j)}(3) \\ m_{(i,j)\to j}(y_j = 123) &\propto 0 \end{aligned} \tag{5.22}$$

To simplify the notation further, we can define messages $o_{i\to j} := m_{(i,j)\to j}$ and expand the n messages. The resulting set of update equations is ($N(i)$ is the set of neightbors of vertex i):

$$\begin{aligned} o_{i\to j}(y_j = 1) &\propto \textstyle\prod_{k\in N(i)\setminus\{j\}} o_{k\to i}(2) + \prod_{k\in N(i)\setminus\{j\}} o_{k\to i}(3) - \prod_{k\in N(i)\setminus\{j\}} o_{k\to i}(23) \\ o_{i\to j}(y_j = 12) &\propto \textstyle\prod_{k\in N(i)\setminus\{j\}} o_{k\to i}(3) \end{aligned} \tag{5.23}$$

These are the $\mathrm{BP}_{(-1)}$ update equations we use for empirical validation in the following section.

5.6 Empirical Validation

We empirically evaluate accuracy of our $Z_{(-1)}$ and $Z_{\mathrm{BP}_{(-1)}}$ approximations on an ensemble of random graph 3-coloring, random 3-SAT problems, and selected structured SAT problems. The results are discussed in this section.

5.6.1 $\mathbf{Z}_{(-1)}$ vs. the number of clusters

The left pane of Figure 5.2 compares the number of clusters (on the x-axis, log-scale) with $Z_{(-1)}$ (on the y-axis, log-scale) for 2500 solvable random 3-COL prob-

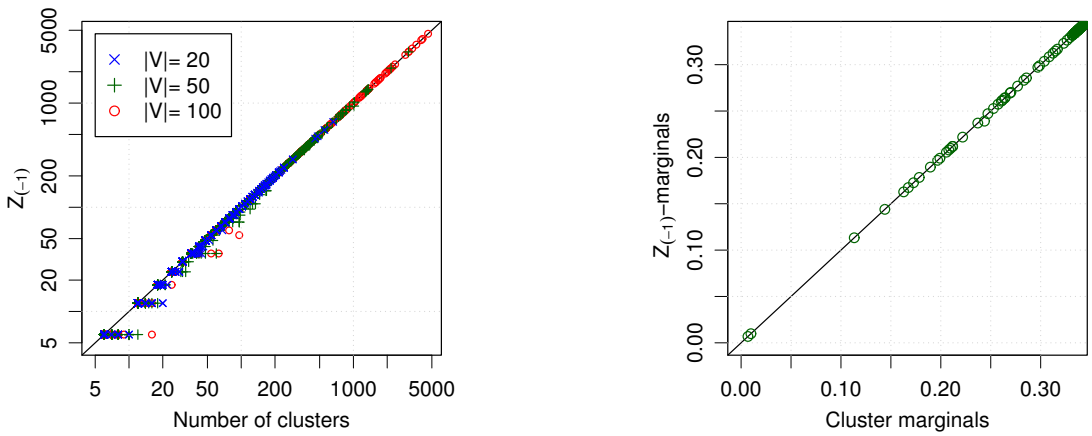


Figure 5.2: Left: $Z_{(-1)}$ vs. number of clusters in random 3-COL problems with 20, 50 and 100 vertices, and average vertex degree between $1.0 - 4.7$. Right: Cluster marginals vs. $Z_{(-1)}$-marginals for one instance of random 3-COL problem with 100 vertices.

lems on graphs with 20, 50, and 100 vertices with average vertex degree ranging between 1.0 and 4.7 (the threshold for 3 colorability). As can be seen, the $Z_{(-1)}$ expression captures the number of clusters almost exactly. The inaccuracies come mostly from low graph density regions; in all instances we tried with density > 3.0 the $Z_{(-1)}$ expression is exact. We remark that although uncolorable instances were not considered in this comparison, $Z_{(-1)} = 0 =$ num-clusters by construction. It is worth noting that for tree graphs (with more than one vertex), the $Z_{(-1)}$ expression gives 0 for any $k \geq 3$ colors although there is exactly one solution cluster. Moreover, given a disconnected graph with at least one such tree partition, the $Z_{(-1)}$ also evaluates to 0 due to the symmetry of colorings in the tree partition. We have thus removed all tree partitions from the generated graphs prior to computing $Z_{(-1)}$ (removing tree partition does not change the number of clusters). For low graph densities, there are still some instances for which $Z_{(-1)}$ evaluates to 0 even after removing tree partitions (these instances are not visible in Figure 5.2 due to the log-log scale). However, if each connected component has a certain sub-structure

(e.g. a triangle), the $Z_{(-1)}$ count is provably exact, as shown by Corollary 1. Such structures become more common as the density increases.

Similarly for the SAT domain, Figure 5.7 (left pane) shows that the $Z_{(-1)}$ counts number of clusters in random 3-SAT problems surprisingly well. The red "+" points of the left panel show the number of clusters (x-axis) compared to $Z_{(-1)}$ (y-axis) for 500 randomly generated formulas with $n = 90$ variables and clause-to-variable ratio $\alpha = 4.0$. Almost all the points lie on or near a line $y = x$, indicating a very good match[3]. Indeed, the correlation coefficient between the number of clusters and $Z_{(-1)}$ is 0.994. The blue "x" points show the number of covers (y-axis) for the same set of formulas. Covers (see Section 4.1) were introduced as combinatorial objects that somewhat capture the structure of clusters in random k-SAT formulas [Maneva et al., 2005], but as can be seen, their number does not match number of clusters nearly as well as $Z_{(-1)}$ (the correlation coefficient is 0.852). However, we expect that even covers grow very accurate in approximating clusters as the number of variables increases[4]. The plot shows instances of relatively small size due to hardness of obraining the exact results for covers. Analogous empirical evidence for varying sizes and clause-to-variable ratios support the claim that $Z_{(-1)}$ is very accurate for random SAT; the larger the formula the better the correspondence (the correlation coefficient being 0.999 for random 4-SAT problems with $n = 80$ and $\alpha = 9.0$). Notice that there is *no averaging* taking place in these results, the $Z_{(-1)}$ counts the number of clusters very accurately on individual instances with high probability. In fact, we observed that solution clusters of random formulas are almost always recursively monotone, which explains the high accuracy of $Z_{(-1)}$. More precisely, 95% of clusters in formulas with sizes $n =$

[3]The points are "jittered" by adding uniform random numbers from $[0, 1]$ to both coordinates, which allows to capture the density of each integer result.

[4]This claim is supported by equivalence of BP for $Z_{(-1)}$ and SP in case of SAT as discussed in Section 5.5.1, and a high accuracy of SP for approximating cover marginals for large formulas described in Section 4.4.

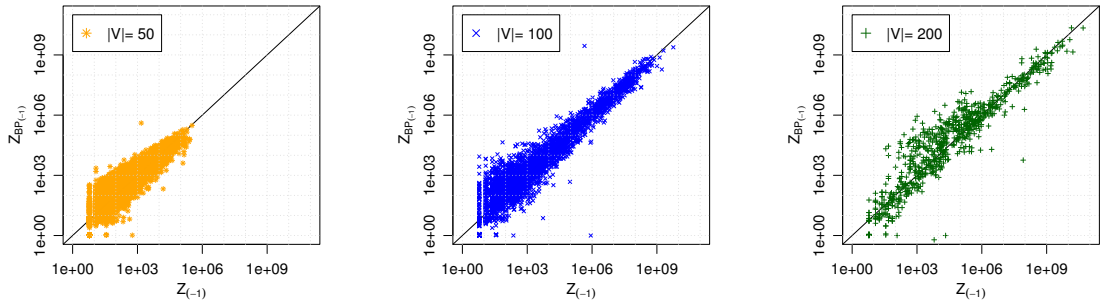


Figure 5.3: $Z_{\mathrm{BP}_{(-1)}}$ compared to $Z_{(-1)}$ for 3-COL problem on random graphs with 50, 100 and 200 vertices and average vertex degree in the range $1.0 - 4.7$.

$60, \ldots, 140$ and $\alpha = 4.0, \ldots, 4.2$ were recursively monotone (the fraction goes down a little to 92% for lower $\alpha = 3.5$). Moreover as we show in the supplementary material, the $Z_{(-1)}$ count is close to exact also in real-world SAT domains.

5.6.2 $\mathrm{BP}_{(-1)}$ Approximation of $\mathbf{Z}_{(-1)}$

Figure 5.3 shows comparison between $Z_{\mathrm{BP}_{(-1)}}$ and $Z_{(-1)}$ for the 3-COL problem on colorable random graphs of various sizes and graph densities. It compares $Z_{(-1)}$ (on the x-axis, log-scale) with $Z_{\mathrm{BP}_{(-1)}}$ (y-axis, log-scale) for 1300 solvable 3-COL problems on random graphs with 50,100, and 200 vertices, with average vertex degree ranging from 1.0 to 4.7. The plots shows that BP is quite accurate in estimating $Z_{(-1)}$ for individual instances, which in turn captures number of clusters. Instances which are not 3-colorable are not shown, and BP in general estimates a non-zero number of clusters.

In the SAT domain, we already know a lot about the properties of $\mathrm{BP}_{(-1)}$ on random formulas due to its equivalence with SP. For instance, we know that $\mathrm{BP}_{(-1)}$ often converges to a "trivial" fixed point, the fixed point that corresponds to all marginals $b_i(x_i = 0) = b_i(x_i = 1) = 0$ and $b_i(x_i = *) = -1$, with an associated estimate $Z_{\mathrm{BP}_{(-1)}} = 1$. This happens when the clause-to-variable ration

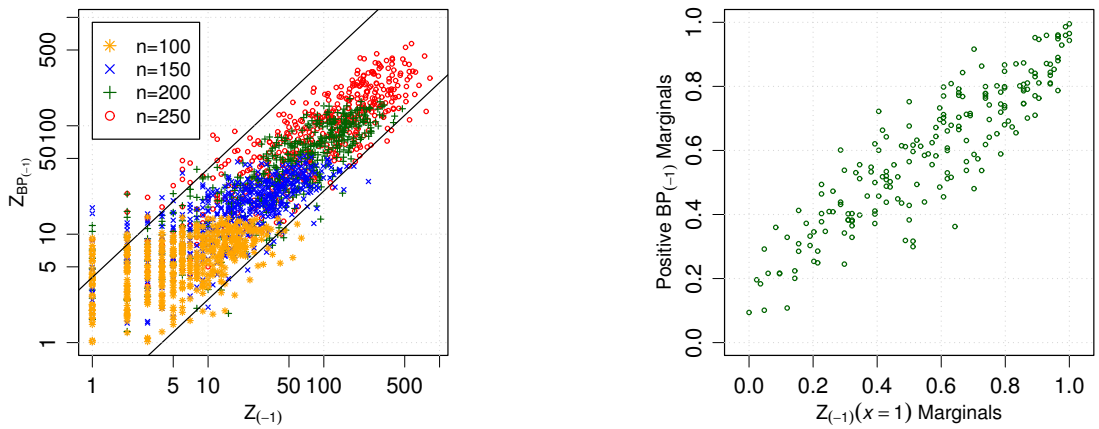


Figure 5.4: Left: $Z_{(-1)}$ vs. $Z_{\mathrm{BP}_{(-1)}}$ for random 3-SAT with varying n and $\alpha = 4.0$. Right: $Z_{(-1)}(x = 1)$ vs. $Z_{\mathrm{BP}_{(-1)}}(x = 1)$ marginals for one instance of random 3-SAT with $n = 200, \alpha = 4.0$.

α is not sufficiently large. It is conjectured that in underconstrained random k-SAT problems, for number of varibles n large enough, there is one cluster that is exponentially larger than the rest [Krzakala et al., 2007]. Nevertheless, there still exist many separate clusters, as e.g. a large number of covers shows [Kroc et al., 2007]. Thus, we only expect $\mathrm{BP}_{(-1)}$ to correctly estimate the number of clusters in a particular region of α.

Figure 5.4 compares $Z_{(-1)}$ (x-axis) with the approximation $Z_{\mathrm{BP}_{(-1)}}$ (y-axis), both axes are in log-scale. Four sizes of random 3-SAT formulas are shown, from $n = 100$ to $n = 250$ variables, all with clause-to-variable ratio $\alpha = 4.0$. There are 500 points for each size, corresponding to individual instances. Unsatisfiable instances or instances for which $\mathrm{BP}_{(-1)}$ converged to the trivial fixed point are excluded. The black parallel diagonal lines are functions $y = 4x$ and $y = \frac{1}{4}x$, bounding the region where $Z_{\mathrm{BP}_{(-1)}}$ is within a factor of 4 from $Z_{(-1)}$. As can be seen, vast majority of the approximations are within this region, and the accuracy improves with increasing size n.

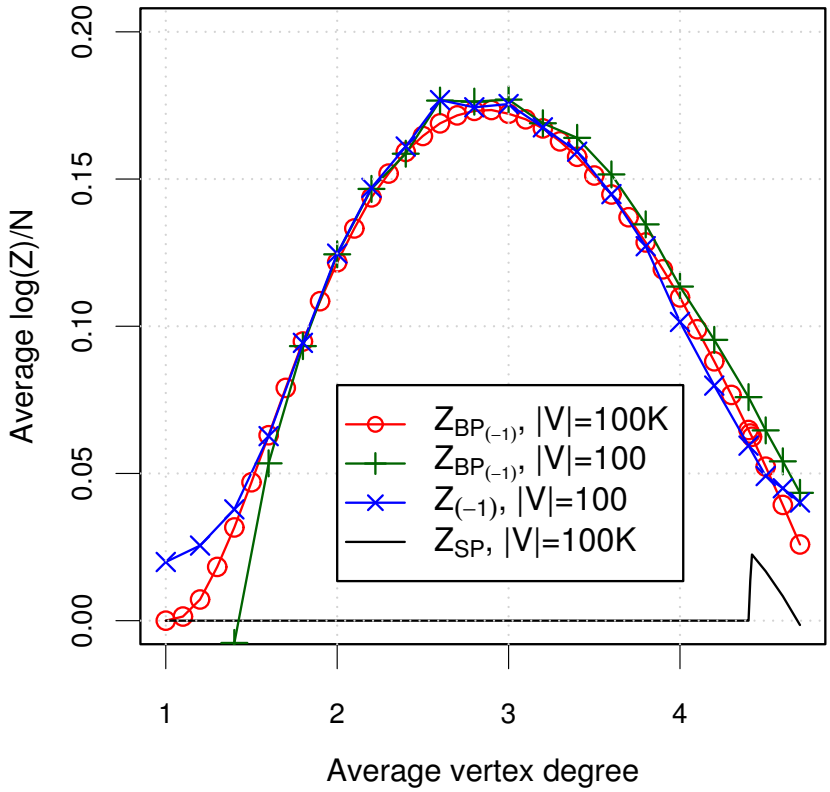


Figure 5.5: Average $Z_{\mathrm{BP}_{(-1)}}$ and $Z_{(-1)}$ for 3-COL vs. average vertex degrees for small and large random graphs.

5.6.3 Estimates on very large problems and for various constraint densities

Figure 5.5 shows data for random 3-COL for a wide range of graph densities, and graph sizes up to $100,000$ vertices. Plotted is a rescaled average estimate of number of clusters (y-axis) for average vertex degrees 1.0 to 4.7 (x-axis). The average is taken across different colorable instances of a given size, and the rescaling assumes that the number of clusters $= \exp(|V| \cdot \Sigma)$ where Σ is independent of number of vertices [Braunstein et al., 2003]. The three color curves show, respectively, BP's estimate for graphs with $100,000$ vertices, BP's estimate for graphs with 100

vertices, and $Z_{(-1)}$ for same graphs of size 100. The averages are done across 3000 instances of the small graphs, and only 10 instances of the large ones, where the instance-to-instance variability is practically nonexistent. The fact that the curves nicely overlay shows that $\mathrm{BP}_{(-1)}$ computes $Z_{(-1)}$ very accurately on average for colorable instances (where we can compare it with exact values), and that the estimate remains accurate for large problems. Note that the Survey Propagation algorithm developed by Braunstein et al. [2003] also aims at computing number of certain clusters in the solutionspace. However, that algorithm counts only number of clusters with a "typical size", and would show non-zero values in Figure 5.5 only for average vertex degrees between 4.42 and 4.7. The corresponding curve is shown in black in the figure. Our algorithm counts clusters of all sizes, and is very accurate in the entire range of graph densities.

Finally, Figure 5.6 shows similar data for random 3-SAT problems.

5.6.4 $\mathbf{Z}_{(-1)}$ vs. the number of clusters for structured instances

Next to trying the $Z_{(-1)}$ on random SAT problems, we also ran some experiments on formulas that belong to the "industrial" category, as specified by the SAT competition [Le Berre et al., 2007]. These are translations of problems that often occur in reality, which makes their structure very different from the random formulas. The biggest difficulty with these is the relative size: most of such formulas are too large to compute $Z_{(-1)}$ and count the number of clusters exactly. Nevertheless, we were able to identify some formulas from the SAT'05 competition belonging to a couple of different domains of an appropriate size. These formulas come from a broad category of SAT encoded planning problems. The results are summarized in the following table:

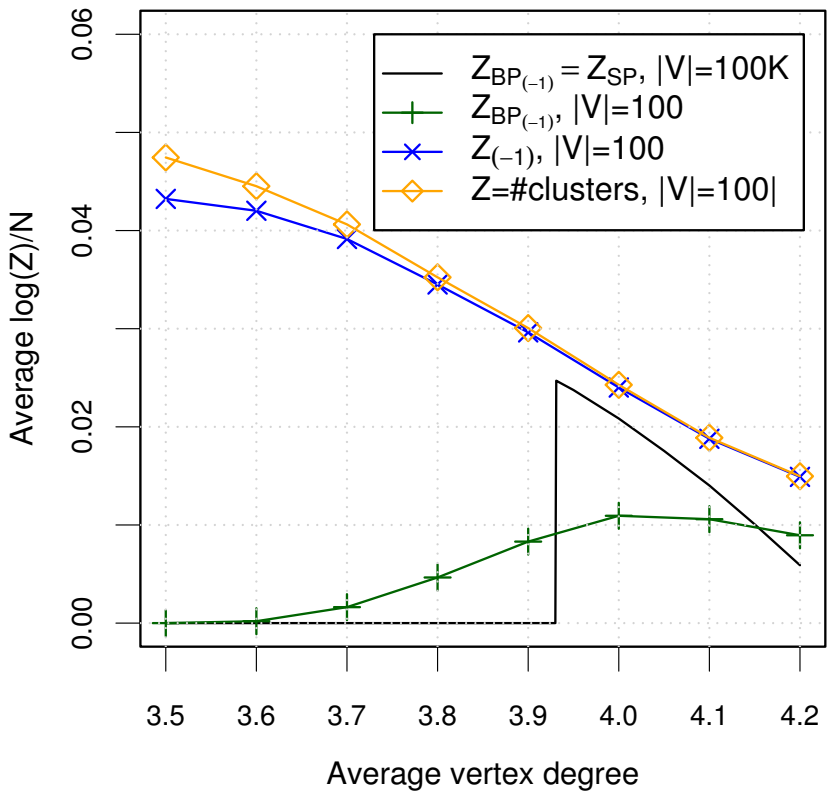


Figure 5.6: Average $Z_{\mathrm{BP}_{(-1)}} = Z_{SP}$ and $Z_{(-1)}$ for 3-SAT vs. clause-to-variable ratio for small and large random formulas.

Instance Name	# solutions	# clusters	$Z_{(-1)}$
v32r250p1	52081218	6	6
v32r500p5	1543304664	6	6
driverlog1_ks99i	856152	338100	338100
rovers1_ks99i	17850294	15	15
rovers1_v01a	83200608	46	46
rovers1_v01i	266000	15	15
rovers2_ks99i	531360	8	8
rovers2_v01a	52107696	316	308
rovers2_v01i	21504	8	8
rovers4_ks99i	13794198600	11	11
rovers4_v01a	2592794880	22	22
rovers4_v01i	28447200	11	11

There are also many problems in the suite for which all clusters consists of individual solutions. For those solutionspaces the $Z_{(-1)}$ counts correctly the number of clusters (same as number of solutions), but such examples are not interesting.

In addition, we computed $Z_{(-1)}$ and compared it to number of clusters in formulas originating from a CircuitSynthesis problem generator, which encodes problems of creating logical circuits that implements a given Boolean function [Kamath et al., 1993]. The parameters of this process are: number of input bits, number of outputs bits, number of logical gates to use, and number of input-output pairs defining the desired circuit (the pairs are generated at random as part of the input). The number of logical gates that are to be used is the one parameter that controls whether the problem has a solution or not. We observed an interesting phenomenon with respect to this parameter: if it is set exactly to the least number of gates possible for the given input, the solutionspace breaks into multiple clusters and the $Z_{(-1)}$

expression exactly counts them (as shown in the table below). If this parameter is larger than minimum, the solutionspace becomes connected (only one cluster), but the cluster is such that $Z_{(-1)}$ does not count it properly. Results for three sample circuit synthesis problems with varying sizes are summarized in the following table (the numbers in the instance name correspond to the four parameters):

Instance Name	# solutions	# clusters	$Z_{(-1)}$
rand-3-2-3-8-renum	288	6	6
rand-3-3-6-8-renum	52560	720	720
rand-4-2-8-16-renum	4193280	80640	80640

We see that the number of clusters in these problems in not a trivial property, being larger than one but orders of magnitude smaller than the number of solutions. And in almost all cases, the $Z_{(-1)}$ exactly corresponds to this count (the one exception being the rovers2_v01a instance). On the industrial instances, we again observed that most clusters were in fact recursively monotone (all but the 8 clusters in rovers2_v01a instance causing $Z_{(-1)}$ to be inexact)

5.6.5 $\mathbf{Z_{(-1)}}$ marginals vs. clusters marginals

COL The right pane of Figure 5.2 shows results of a comparison between $Z_{(-1)}$-marginals and cluster marginals on one random 3-COL problem with 100 vertices. The axis show cluster marginals and $Z_{(-1)}$-marginals for one color, points correspond to individual variables. The $Z_{(-1)}$-marginals are close to perfect. This is a typical situation, although it is important to mention that $Z_{(-1)}$-marginals are not necessarily correct, or even non-negative. They are merely an estimate of the true cluster marginals, and how well they work depends on the solutionspace structure at hand. They are exact if the solutionspace decomposes into separated hypercubes and, as the figure shows, remarkably accurate also for random coloring instances.

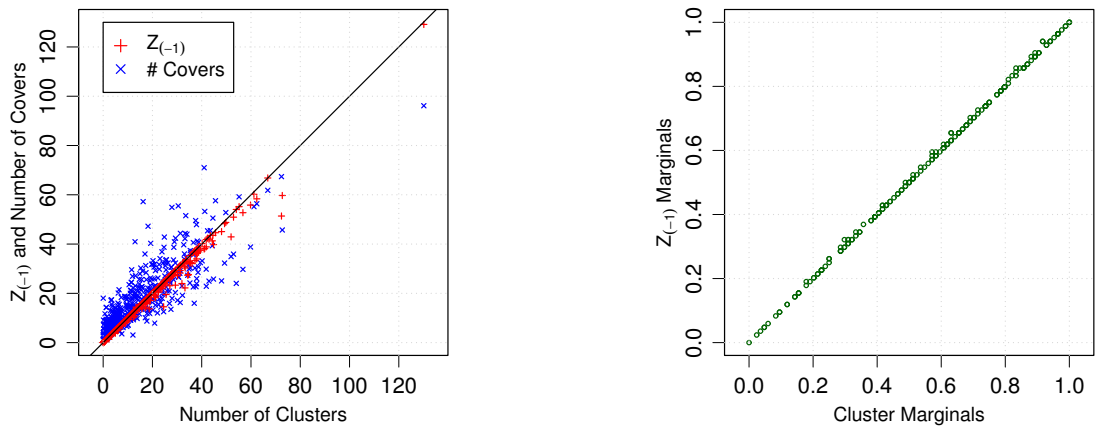


Figure 5.7: Left: number of clusters vs. $Z_{(-1)}$ and number of covers in random 3-SAT for $n = 90, \alpha = 4.0$. Right: positive cluster marginals vs. $Z_{(-1)}(x = 1)$ marginals for one instance of random 3-SAT with $n = 200, \alpha = 4.0$.

SAT The right pane of Figure 5.7 is a more detailed look into the cluster structure of one selected formula, with $n = 200$ variables, $\alpha = 4.0$, with 84 clusters and $Z_{(-1)} = 94$. For each variable (represented by a point), the plot compares its positive *cluster marginal* (i.e. fraction of clusters in which the variable can appear positively) on the x-axis and $Z_{(-1)}(x = 1)$-marginal on the y-axis. The points lie close to a diagonal line, indicating that the fit is almost perfect. This suggests that clusters of random 3-SAT problems are perhaps 1-recursively monotone, on top of being recursively monotone, and Theorem 4 is applicable to random SAT problems.

5.7 Discussion

We discuss a purely combinatorial construction to estimate number of solution clusters in constraint satisfaction problems with very high accuracy. The technique is based on a inclusion-exclusion argument coupled with solution counting, and lends itself to an application of a modified belief propagation algorithm. This

way, number of clusters in huge random combinatorial problems can be accurately and efficiently estimated, as we show for the case of SAT and graph 3-coloring. Moreover, due to the nature of the construction, combinatorial arguments can be used to argue that the counts are exact on certain kinds of solution spaces. We also observe that many problems coming from industrial domains indeed have the property that the cluster counts are exact. These insights might eventually lead to new techniques for solving hard CSPs, and bounding solvability transitions in random ensembles.

Chapter 6

Leveraging Belief Propagation, Backtrack Search, and Statistics for Model Counting

The model counting problem for Boolean satisfiability or SAT is the problem of computing the number of solutions or satisfying assignments for a given Boolean formula. Often written as #SAT, this problem is #P-complete [Valiant, 1979] and is widely believed to be significantly harder than the NP-complete SAT problem, which seeks an answer to whether or not the formula in satisfiable. With the amazing advances in the effectiveness of SAT solvers since the early 90's, these solvers have come to be commonly used in combinatorial application areas like hardware and software verification, planning, and design automation. Efficient algorithms for #SAT will further open the doors to a whole new range of applications, most notably those involving probabilistic inference [Bacchus et al., 2003, Darwiche, 2005, Littman et al., 2001, Park, 2002, Roth, 1996, Sang et al., 2005].

A number of different techniques for model counting have been proposed over the last few years. For example, `Relsat` [Bayardo Jr. and Pehoushek, 2000] extends systematic SAT solvers for model counting and uses component analysis for effi-

ciency, `Cachet` [Sang et al., 2004] adds caching schemes to this approach, `c2d` [Darwiche, 2004] converts formulas to the d-DNNF form which yields the model count as a by-product, `ApproxCount` [Wei and Selman, 2005] and `SampleCount` [Gomes et al., 2007] exploit sampling techniques for estimating the count, `MBound` [Gomes et al., 2006] relies on the properties of random parity or XOR constraints to produce estimates with correctness guarantees, and the recently introduced `SampleMinisat` [Gogate and Dechter, 2007] uses sampling of the backtrack-free search space of systematic SAT solvers. While all of these approaches have their own advantages and strengths, there is still much room for improvement in the overall scalability and effectiveness of model counters.

In this chapter, we propose two new techniques for model counting that leverage the strength of message passing and systematic algorithms for SAT. The first of these yields probabilistic lower bounds on the model count, and for the second we introduce a statistical framework for obtaining upper bounds.

The first method, which we call `BPCount`, builds upon a successful approach for model counting using local search, called `ApproxCount`. The idea is to efficiently obtain a rough estimate of the "marginals" of each variable: what fraction of solutions have variable x set to TRUE and what fraction have x set to FALSE? If this information is computed accurately enough, it is sufficient to recursively count the number of solutions of only *one* of $F|_x$ and $F|_{\neg x}$, and scale the count up appropriately. This technique is extended in `SampleCount`, which adds randomization to this process and provides lower bounds on the model count with high probability correctness guarantees. For both `ApproxCount` and `SampleCount`, true variable marginals are estimated by obtaining several solution samples using local search techniques such as `SampleSat` [Wei et al., 2004] and computing marginals from the samples. In many cases, however, obtaining many near-uniform solution

samples can be costly, and one naturally asks whether there are more efficient ways of estimating variable marginals.

Interestingly, the problem of computing variable marginals can be formulated as a key question in Bayesian inference, and the Belief Propagation or BP algorithm [Pearl, 1988], at least in principle, provides us with exactly the tool we need. The BP method for SAT involves representing the problem as a factor graph and passing "messages" back-and-forth between variable and factor nodes until a fixed point is reached. This process is cast as a set of mutually recursive equations which are solved iteratively. From the fixed point, one can easily compute, in particular, variable marginals.

While this sounds encouraging, there are two immediate challenges in applying the BP framework to model counting: (1) quite often the iterative process for solving the BP equations does not converge to a fixed point, and (2) while BP provably computes exact variable marginals on formulas whose constraint graph has a tree-like structure (formally defined later), its marginals can sometimes be substantially off on formulas with a richer interaction structure. To address the first issue, we use a "message damping" form of BP which has better convergence properties (inspired by a damped version of BP due to Pretti [2005]). For the second issue, we add "safety checks" to prevent the algorithm from running into a contradiction by accidentally eliminating all assignments. Somewhat surprisingly, avoiding these rare but fatal mistakes turns out to be sufficient for obtaining very close estimates and lower bounds for solution counts, suggesting that BP does provide useful information even on highly structured loopy formulas. To exploit this information even further, we extend the framework borrowed from `SampleCount` with the use of biased coins during randomized value selection.

The model count can, in fact, also be estimated directly from just one fixed

point run of the BP equations, by computing the value of so-called partition function [Yedidia et al., 2005]. In particular, this approach computes the exact model count on tree-like formulas, and appeared to work fairly well on random formulas. However, the count estimated this way is often highly inaccurate on structured loopy formulas. `BPCount`, as we will see, makes a much more robust use of the information provided by BP.

The second method, which we call `MiniCount`, exploits the power of modern DPLL [Davis et al., 1962, Davis and Putnam, 1960] based SAT solvers, which are extremely good at finding single solutions to Boolean formulas through backtrack search. (Gogate and Dechter [2007] have independently proposed the use of DPLL solvers for model counting.) The problem of computing upper bounds on the model count has so far eluded solution because of an asymmetry which manifests itself in at least two inter-related forms: the set of solutions of interesting N variable formulas typically forms a minuscule fraction of the full space of 2^N variable assignments, and the application of Markov's inequality as in `SampleCount` does not yield interesting upper bounds. Note that systematic model counters like `Relsat` and `Cachet` can also be easily extended to provide an upper bound when they time out (2^N minus the number of non-solutions encountered), but these bounds are uninteresting because of the above asymmetry. To address this issue, we develop a statistical framework which lets us compute upper bounds under certain statistical assumptions, which are independently validated. To the best of our knowledge, this is the first effective and scalable method for obtaining good upper bounds on the model counts of formulas that are beyond the reach of exact model counters.

More specifically, we describe how the DPLL-based solver `MiniSat` [Eén and Sörensson, 2005], with two minor modifications, can be used to estimate the total number of solutions. The number d of branching decisions (not counting unit

propagations and failed branches) made by `MiniSat` before reaching a solution, is the main quantity of interest: when the choice between setting a variable to TRUE or to FALSE is randomized,[1] the number d is provably not any lower, in expectation, than $\log_2$(model count). This provides a strategy for obtaining upper bounds on the model count, only if one could efficiently estimate the expected value, $\mathbb{E}[d]$, of the number of such branching decisions. A natural way to estimate $\mathbb{E}[d]$ is to perform multiple runs of the randomized solver, and compute the average of d over these runs. However, if the formula has many "easy" solutions (found with a low value of d) and many "hard" solutions, the limited number of runs one can perform in a reasonable amount of time may be insufficient to hit many of the "hard" solutions, yielding too low of an estimate for $\mathbb{E}[d]$ and thus an incorrect upper bound on the model count.

Interestingly, we show that for many families of formulas, d has a distribution that is very close to the normal distribution. Under the assumption that d is normally distributed, when sampling various values of d through multiple runs of the solver, we need not necessarily encounter high values of d in order to correctly estimate $\mathbb{E}[d]$ for an upper bound. Instead, we can rely on statistical tests and conservative computations [Thode, 2002, Zhou and Sujuan, 1997] to obtain a statistical upper bound on $\mathbb{E}[d]$ within any specified confidence interval.

We evaluated our two approaches on challenging formulas from several domains. Our experiments with `BPCount` demonstrate a clear gain in efficiency, while providing much higher lower bound counts than exact counters (which often run out of time or memory) and competitive lower bound quality compared to `SampleCount`. For example, the runtime on several difficult instances from the FPGA routing family with over 10^{100} solutions is reduced from hours for both exact counters and

[1] `MiniSat` by default always sets variables to FALSE.

`SampleCount` to just a few minutes with `BPCount`. Similarly, for random 3CNF instances with around 10^{20} solutions, we see a reduction in computation time from hours and minutes to seconds. With `MiniCount`, we are able to provide good upper bounds on the solution counts, often within seconds and fairly close to the true counts (if known) or lower bounds. These experimental results attest to the effectiveness of the two proposed approaches in significantly extending the reach of solution counters for hard combinatorial problems.

6.1 Lower Bounds Using BP Marginal Estimates: BPCount

In this section, we develop a method for obtaining a lower bound on the solution count of a given formula, using the framework recently used in the SAT model counter `SampleCount` [Gomes et al., 2007]. The key difference between our approach and `SampleCount` is that instead of relying on solution samples, we use a variant of belief propagation to obtain estimates of the fraction of solutions in which a variable appears positively. We call this algorithm `BPCount`. After describing the basic method, we will discuss two techniques that improve the tightness of `BPCount` bounds in practice, namely, *biased variable assignments* and *safety checks*.

We begin by recapitulating the framework of `SampleCount` for obtaining lower bound model counts with probabilistic correctness guarantees. A variable u will be called *balanced* if it occurs equally often positively and negatively in all solutions of the given formula. In general, the *marginal probability* of u being TRUE in the set of satisfying assignments of a formula is the fraction of such assignments where $u =$ TRUE. Note that computing the marginals of each variable, and in particular identifying balanced or near-balanced variables, is quite non-trivial. The model

counting approaches we describe attempt to estimate such marginals using indirect techniques such as solution sampling or iterative message passing.

Given a formula F and parameters $t, z \in \mathbb{Z}^+, \alpha > 0$, `SampleCount` performs t iterations, keeping track of the minimum count obtained over these iterations. In each iteration, it samples z solutions of (potentially simplified) F, identifies the most balanced variable u, uniformly randomly sets u to TRUE or FALSE, simplifies F by performing any possible unit propagations, and repeats the process. The repetition ends when F is reduced to a size small enough to be feasible for exact model counters like `Cachet`. At this point, let s denote the number of variables randomly set in this iteration before handing the formula to `Cachet`, and let M' be the model count of the residual formula returned by `Cachet`. The count for this iteration is computed to be $2^{s-\alpha} \times M'$ (where α is a "slack" factor pertaining to our probabilistic confidence in the bound). Here 2^s can be seen as scaling up the residual count by a factor of 2 for every uniform random decision we made when fixing variables. After the t iterations are over, the minimum of the counts over all iterations is reported as the lower bound for the model count of F, and the correctness confidence attached to this lower bound is $1 - 2^{-\alpha t}$. This means that the reported count is a correct lower bound with probability $1 - 2^{-\alpha t}$.

The performance of `SampleCount` is enhanced by also considering balanced variable pairs (v, w), where the balance is measured as the difference in the fractions of all solutions in which v and w appear with the same sign vs. with different signs. When a pair is more balanced than any single literal, the pair is used instead for simplifying the formula. In this case, we replace w with v or $\neg v$ uniformly at random. For ease of illustration, we will focus here only on identifying and randomly setting balanced or near-balanced variables.

The key observation in `SampleCount` is that when the formula is simplified

by repeatedly assigning a positive or negative polarity to variables, the expected value of the count in each iteration, $2^s \times M'$ (ignoring the slack factor α), is exactly the true model count of F, from which lower bound guarantees follow. We refer the reader to Gomes et al. [2007] for details. Informally, we can think of what happens when the first such balanced variable, say u, is set uniformly at random. Let $p \in [0, 1]$. Suppose F has M solutions, $F|_u$ has pM solutions, and $F|_{\neg u}$ has $(1-p)M$ solutions. Of course, when setting u uniformly at random, we don't know the actual value of p. Nonetheless, with probability a half, we will recursively count the search space with pM solutions and scale it up by a factor of 2, giving a net count of $pM.2$. Similarly, with probability a half, we will recursively get a net count of $(1-p)M.2$ solutions. On average, this gives $\frac{1}{2}.pM.2 + \frac{1}{2}.(1-p)M.2 = M$ solutions.

Interestingly, the correctness guarantee of this process holds irrespective of how good or bad the samples are. However, when balanced variables are correctly identified, we have $p \approx \frac{1}{2}$ in the informal analysis above, so that for both coin flip outcomes we recursively search a space with roughly $M/2$ solutions. This reduces the variance tremendously, which is crucial to making the process effective in practice. Note that with high variance, the minimum count over t iterations is likely to be much smaller than the true count; thus high variance leads to poor quality lower bounds.

The idea of ***BPCount*** *is to "plug-in" belief propagation methods in place of solution sampling in the* ***SampleCount*** *framework,* in order to estimate "p" in the intuitive analysis above and, in particular, to help identify balanced variables. As it turns out, a solution to the BP equations [Pearl, 1988] provides exactly what we need: an estimate of the marginals of each variable. This is an alternative to using sampling for this purpose, and is often orders of magnitude faster.

One bottleneck, however, is that the basic belief propagation process is iterative and does not even converge on most SAT formulas of interest. In order to use BP for estimating marginal probabilities and identifying balanced variables, one must either cut off the iterative computation or use a modification that does converge. Unfortunately, some of the known improvements of the belief propagation technique that allow it to converge more often or be used on a wider set of problems, such as Generalized Belief Propagation [Yedidia et al., 2000], Loop Corrected Belief Propagation [Mooij et al., 2007], or Expectation Maximization Belief Propagation [Hsu and McIlraith, 2006], are not scalable enough for our purposes. The problem of very slow convergence on hard instances seems to plague also approaches based on other methods for solving the BP equations than the simple iteration scheme, such as the convex-concave procedure introduced by Yuille [2002]. Finally, in our context, the speed requirement is accentuated by the need to use marginal estimation repeatedly during the decimation process.

We use a parameterized variant of BP, BP^κ, introduced in Section 3.1.2, which is guaranteed to converge when this parameter κ is small enough, and which imposes no additional computational cost over the traditional BP. The damped equations are analogous to standard BP for SAT, differing only in the added κ exponent in the iterative update equations. When $\kappa = 1$, BP^κ is identical to regular belief propagation. On the other hand, when $\kappa = 0$, the equations surely converge in one step to a unique fixed point and the marginal estimates obtained from this fixed point have a clear probabilistic interpretation in terms of a local property of the variables. The κ parameter thus allows one to interpolate between regimes where the equations correspond to the original BP providing global information if the iterations converge ($\kappa = 1$), and the regime where the iterations surely converge but provide only local information (for $\kappa = 0$). We use the output of BP^κ as an

estimate of the marginals of the variables in `BPCount`.

Given this process of obtaining marginal estimates from BP, `BPCount` works almost exactly like `SampleCount` and *provides the same lower bound guarantees.* The only difference between the two algorithms is the manner in which marginal probabilities of variables is estimated. Formally,

Theorem 5 (Adapted from [Gomes et al., 2007]). *Let s denote the number of variables randomly set by an iteration of* `BPCount`*, M' denote the number of solutions in the final residual formula given to an exact model counter, and $\alpha > 0$ be the slack parameter used. If* `BPCount` *is run with $t \geq 1$ iterations on a formula F, then its output—the minimum of $2^{s-\alpha} \times M'$ over the t iterations—is a correct lower bound on $\#F$ with probability at least $1 - 2^{-\alpha t}$.*

As we see, the correctness confidence can be easily boosted by increasing the number of iterations, t, (thereby incurring a higher runtime), or by increasing the slack parameter, α, (thereby reporting a somewhat smaller lower bound), or by a combination of both. In our experiments, we will aim for a correctness confidence of over 99%, by using values of t and α satisfying $\alpha t \geq 7$. Specifically, most runs will involve 7 iterations and $\alpha = 1$, while some will involve fewer iterations with a slightly higher value of α.

Using Biased Coins. We can improve the performance of `BPCount` (and also of `SampleCount`) by using biased variable assignments. The idea here is that when fixing variables repeatedly in each iteration, the values need not be chosen uniformly. The correctness guarantees still hold even if we use a biased coin and set the chosen variable u to TRUE with probability q and to FALSE with probability $1-q$, for any $q \in (0, 1)$. Using earlier notation, this leads us to a solution space of size pM with probability q and to a solution space of size $(1-p)M$ with probability $1-q$. Now, instead of scaling up with a factor of 2 in both cases, we scale up based

on the bias of the coin used. Specifically, with probability q, we go to one part of the solution space and scale it up by $1/q$, and similarly for $1-q$. The net result is that in expectation, we still get $q.pM/q + (1-q).(1-p)M/(1-q) = M$ solutions. Further, the variance is minimized when q is set to equal p; in `BPCount`, q is set to equal the estimate of p obtained using the BP equations. To see that the resulting variance is minimized this way, note that with probability q, we get a net count of pM/q, and with probability $(1-q)$, we get a net count of $(1-p)M/(1-q)$; these balance out to exactly M in either case when $q = p$. Hence, when we have confidence in the correctness of the estimates of variable marginals (i.e., p here), it provably reduces variance to use a biased coin that matches the marginal estimates of the variable to be fixed.

Safety Checks. One issue that arises when using BP techniques to estimate marginals is that the estimates, in some case, may be far off from the true marginals. In the worst case, a variable u identified by BP as the most balanced may in fact be a backbone variable for F, i.e., may only occur, say, positively in all solutions to F. Setting u to FALSE based on the outcome of the corresponding coin flip thus leads one to a part of the search space with no solutions at all, so that the count for this iteration is zero, making the minimum over t iterations zero as well. To remedy this situation, we use safety checks using an off-the-shelf SAT solver (`Minisat` or `Walksat` in our implementation) before fixing the value of any variable. The idea is to simply check that u can be set both ways *before* flipping the random coin and fixing u to TRUE or FALSE. If `Minisat` finds, e.g., that forcing u to be TRUE makes the formula unsatisfiable, we can immediately deduce $u =$ FALSE, simplify the formula, and look for a different balanced variable. This safety check prevents `BPCount` from reaching the undesirable state where there are no remaining solutions at all.

In fact, with the addition of safety checks, we found that the lower bounds on model counts obtained for some formulas were surprisingly good even when the marginal estimates were generated purely at random, i.e., without actually running BP. This can perhaps be explained by the errors introduced at each step somehow canceling out when several variables are fixed. With the use of BP, the quality of the lower bounds was significantly improved, showing that BP does provide useful information about marginals even for loopy formulas. Lastly, we note that with `SampleCount`, the external safety check can be conservatively replaced by simply avoiding those variables that appear to be backbone variables from the obtained samples.

6.2 Upper Bound Estimation Using Backtrack Search: MiniCount

We now describe an approach for estimating an upper bound on the solution count. We use the reasoning discussed for `BPCount`, and apply it to a DPLL style search procedure. There is an important distinction between the nature of the bound guarantees presented here and earlier: here we will derive *statistical* (as opposed to probabilistic) guarantees, and their quality may depend on the particular family of formulas in question. The applicability of the method will also be determined by a statistical test, which succeeded in most of our experiments.

For `BPCount`, we used a backtrack-less branching search process with a random outcome that, in expectation, gives the exact number of solutions. The ability to randomly assign values to selected variables was crucial in this process. Here we extend the same line of reasoning to a search process *with* backtracking, and argue that the expected value of the outcome is an upper bound on the true count.

We extend the `MiniSat` SAT solver [Eén and Sörensson, 2005] to compute the information needed for upper bound estimation. `MiniSat` is a very efficient SAT solver employing conflict clause learning and other state-of-the-art techniques, and has one important feature helpful for our purposes: whenever it chooses a variable to branch on, it is left unspecified which value should the variable assume first. One possibility is to assign values TRUE or FALSE randomly with equal probability. Since `MiniSat` does not use any information about the variable to determine the most promising polarity, this random assignment in principle does not lower `MiniSat`'s power. Note that there are other SAT solvers with this feature, e.g. `Rsat` [Pipatsrisawat and Darwiche, 2006], and similar results can be obtained for such solvers as well.

Algorithm MiniCount: Given a formula F, run `MiniSat`, choosing the truth value assignment for the selected variable at each choice point uniformly at random (option `-polarity-mode=rnd`). When a solution is found, output 2^d, where d is the "perceived depth", i.e., the number of choice points on the path to the solution (the final decision level), not counting those choice points where the other branch failed to find a solution (a backtrack point). We rely on the fact that the default implementation of `MiniSat` never restarts unless it has backtracked at least once.[2]

We note that we are implicitly using the fact that `MiniSat`, and most SAT solvers available today, assign truth values to *all* variables of the formula when they declare that a solution has been found. In case the underlying SAT solver is designed to detect the fact that all clauses have been satisfied and to then declare that a solution has been found even with, say, u variables remaining unset, the

[2]In a preliminary version of this work [Kroc et al., 2008b], we did not allow restarts at all. The reasoning given here extends the earlier argument and permits restarts as long as they happen after at least one backtrack.

definition of d should be modified to include these u variables; i.e., d should be u plus the number of choice points on the path minus the number of backtrack points on that path.

Note also that for an n variable formula, d can be alternatively defined as n minus the number of unit propagations on the path to the solution found minus the number of backtrack points on that path. This makes it clear that d is after all tightly related to n, in the sense that if we add a few "don't care" variables to the formula, the value of d will increase appropriately.

We now prove that we can use `MiniCount` to obtain an upper bound on the true model count of F. Since `MiniCount` is a probabilistic algorithm, its output, 2^d, on a given formula F is a random variable. We denote this random variable by $\#F_{\text{MiniCount}}$, and use $\#F$ to denote the true number of solutions of F. The following theorem forms the basis of our upper bound estimation. We note that the theorem provides an essential building block but by itself does not fully justify the statistical estimation techniques we will introduce later; they rely on arguments discussed after the theorem.

Theorem 6. *For any CNF formula F, $\mathbb{E}\left[\#F_{\text{MiniCount}}\right] \geq \#F$.*

Proof. The expected value is taken across all possible choices made by the `MiniCount` algorithm when run on F, i.e., all its possible computation histories on F. The proof uses the fact that the claimed inequality holds even if all computation histories that incurred at least one backtrack were modified to output 0 instead of 2^d once a solution was found. In other words, we will write the desired expected value, by definition, as a sum over all computation histories h and then simply discard a subset of the computation histories—those that involve at least one backtrack—from the sum to obtain a smaller quantity, which will eventually be shown to equal #F exactly.

Once we restrict ourselves to only those computation histories h that do not involve any backtracking, these histories correspond one-to-one to the paths p in the search tree underlying `MiniCount` that lead to a solution. Note that there are precisely as many such paths p as there are satisfying assignments for F. Further, since value choices of `MiniCount` at various choice points are made independently at random, the probability of a computation history following path p is precisely $1/2^{d_p}$, where d_p is the "perceived depth" of the solution at the leaf of p, i.e., the number of choice points till the solution is found (recall that there are no backtracks on this path; of course, there might be unit propagations along p, due to which d_p may be smaller than the total number of variables). The value output by `MiniCount` on this path is 2^{d_p}.

Mathematically, the above reasoning can be formalized as follows:

$$
\begin{aligned}
\mathbb{E}\left[\#F_{\text{MiniCount}}\right] &= \sum_{\substack{\text{computation histories } h \\ \text{of } \texttt{MiniCount} \text{ on } F}} \Pr[h] \cdot \text{output on } h \\
&\geq \sum_{\substack{\text{computation histories } h \\ \text{not involving any backtrack}}} \Pr[h] \cdot \text{output on } h \\
&= \sum_{\substack{\text{search paths } p \\ \text{that lead to a solution}}} \Pr[p] \cdot \text{output on } p \\
&= \sum_{\substack{\text{search paths } p \\ \text{that lead to a solution}}} \frac{1}{2^{d_p}} \cdot 2^{d_p} \\
&= \text{number of search paths } p \text{ that lead to a solution} \\
&= \#F
\end{aligned}
$$

This concludes the proof. □ □

The reason *restarts without at least one backtrack are not allowed* in `MiniCount` is hidden in the proof of Theorem 6. With such early restarts, only solutions

reachable within the current setting of the restart threshold can be found. For restarts shorter than the number of variables, only "easier" solutions which require very few decisions are ever found. `MiniCount` with early restarts could therefore always undercount and not provide an upper bound—even in expectation. On the other hand, if restarts happen only after at least one backtrack point, then the proof of the above theorem shows that it is safe to even output 0 on such runs and still obtain a correct upper bound in expectation; restarting and reporting a non-zero number only helps the upper bound.

With enough random samples of the output, $\#F_{\text{MiniCount}}$, obtained from `MiniCount`, their average value will eventually converge to $\mathbb{E}[\#F_{\text{MiniCount}}]$ by the Law of Large Numbers, thereby providing an upper bound on $\#F$ because of Theorem 6. Unfortunately, providing a useful correctness guarantee on such an upper bound in a manner similar to the lower bounds seen earlier turns out to be impractical, because the resulting guarantees, obtained using a reverse variant of the standard Markov's inequality, are too weak. Further, relying on the simple average of the obtained output samples might also be misleading, since the distribution of $\#F_{\text{MiniCount}}$ often has significant mass in fairly high values and it might take very many samples for the sample mean to become as large as the true average of the distribution.

The way we proved Theorem 6, in fact, suggests that we could simply report 0 and start over every time we need to backtrack, which would actually result in a random variable that is in expectation *exact*, not only an upper bound. This approach is of course impractical, as we would almost always see zeros and see a very high non-zero output with exponentially small probability. Although the expected value of these numbers is, in principle, the true model count of F, estimating the expected value of the underlying extreme zero-heavy 'bimodal' distribution

through a few random samples is infeasible in practice. We therefore choose to trade off tightness of the reported bound for the ability to obtain values that can be argued about, as shown next.

An important observation that justifies the use of the statistical estimation techniques discussed next is that when `MiniCount` does backtrack, we do *not* report 0; rather we continue to explore the other side of the "choice" point under consideration and eventually report a non-zero value. Since our strategy will be to fit a statistical distribution on the output of several samples from `MiniCount`, and because except for rare occasions all of these samples come after at least one backtrack, it is crucial that the non-zero value output by `MiniCount` when a solution is found after a backtrack does have information about the number of solutions of F. Fortunately, this is indeed the case—the value 2^d that `MiniCount` outputs even after at least one backtrack does contain valuable information about the number of solutions of F. To see this, consider a stage in the algorithm which is perceived as a choice point but is in fact not a true choice point. Specifically, suppose at this stage, the formula has M solutions when $x = \text{TRUE}$ and no solutions when $x = \text{FALSE}$. With probability $1/2$, `MiniCount` will set x to TRUE and in fact estimate an upper bound on $2M$ from the resulting sub-formula, because it did not discover that it wasn't really at a "choice" point. This will, of course, still be a legal upper bound on M. More importantly, with probability $1/2$, `MiniCount` will set x to FALSE, discover that there are no solutions in this sub-tree, backtrack, set x to TRUE, realize that this is not actually a "choice" point, and recursively estimate an upper bound on M. Thus, even with backtracks, the output of `MiniCount` is very closely related to the actual number of solutions in the sub-tree at the current stage (unlike in the proof of Theorem 6, where it is thought of as being 0), and it is justifiable to deduce an upper bound on #F by fitting sample outputs of `MiniCount`

to a statistical distribution. We also note that the number of solutions reported after a restart is just like taking another sample of the process with backtracks, and thus is also closely related to #F.

6.2.1 Estimating the Upper Bound Using Statistical Methods

In this section, we develop an approach based on statistical analysis of the sample outputs that allows one to estimate the expected value of $\#F_{\text{MiniCount}}$, and thus an upper bound with statistical guarantees, using relatively few samples.

Assuming the distribution of $\#F_{\text{MiniCount}}$ is known, the samples can be used to provide an unbiased estimate of the mean, along with confidence intervals on this estimate. This distribution is of course not known and will vary from formula to formula, but it can again be inferred from the samples. We observed that for many formulas, the distribution of $\#F_{\text{MiniCount}}$ is well approximated by a log-normal distribution. Thus we develop the method under the assumption of log-normality, and include techniques to independently test this assumption. The method has three steps:

1. Generate n independent samples from $\#F_{\text{MiniCount}}$ by running `MiniCount` n times on the same formula.
2. Test whether the samples come from a log-normal distribution (or a distribution sufficiently similar).
3. Estimate the true expected value of $\#F_{\text{MiniCount}}$ from the samples, and calculate the $(1-\alpha)\%$ confidence interval for it, using the assumption that the underlying distribution is log-normal. We set the confidence level α to 0.01, and denote the upper bound of the resulting confidence interval by $c_{\max}$.

This process, some of whose details will be discussed shortly, yields an upper bound $c_{\max}$ along with a statistical guarantee that $c_{\max} \geq \mathbb{E}\left[\#F_{\text{MiniCount}}\right]$ and thus $c_{\max} \geq \#F$:

$$\Pr\left[c_{\max} \geq \#F\right] \geq 1 - \alpha \tag{6.1}$$

The caveat in this statement (and, in fact, the main difference from the similar statement for the lower bounds for `BPCount` given earlier) is that it is true only if our assumption of log-normality holds.

Testing for Log-Normality. By definition, a random variable X has a log-normal distribution if the random variable $Y = \log X$ has a normal distribution. Thus a test whether Y is normally distributed can be used, and we use the Shapiro-Wilk test [cf. Thode, 2002] for this purpose. In our case, $Y = \log(\#F_{\text{MiniCount}})$ and if the computed p-value of the test is below the confidence level $\alpha = 0.05$, we conclude that our samples do *not* come from a log-normal distribution; otherwise we assume that they do. If the test fails, then there is sufficient evidence that the underlying distribution is not log-normal, and the confidence interval analysis to be described shortly will not provide any statistical guarantees. Note that non-failure of the test does not mean that the samples *are* actually log-normally distributed, but inspecting the Quantile-Quantile plots (QQ-plots) often supports the hypothesis that they are. QQ-plots compare sampled quantiles with theoretical quantiles of the desired distribution: the more the sample points align on a line, the more likely it is that the data comes from the distribution.

We found that a surprising number of formulas had $\log_2(\#F_{\text{MiniCount}})$ very close to being normally distributed. Figure 6.1 shows normalized QQ-plots for $d_{\text{MiniCount}} = \log_2(\#F_{\text{MiniCount}})$ obtained from 100 to 1000 runs of `MiniCount` on various families of formulas (discussed in the experimental section). The top-left QQ-plot shows the best fit of normalized $d_{\text{MiniCount}}$ (obtained by subtracting

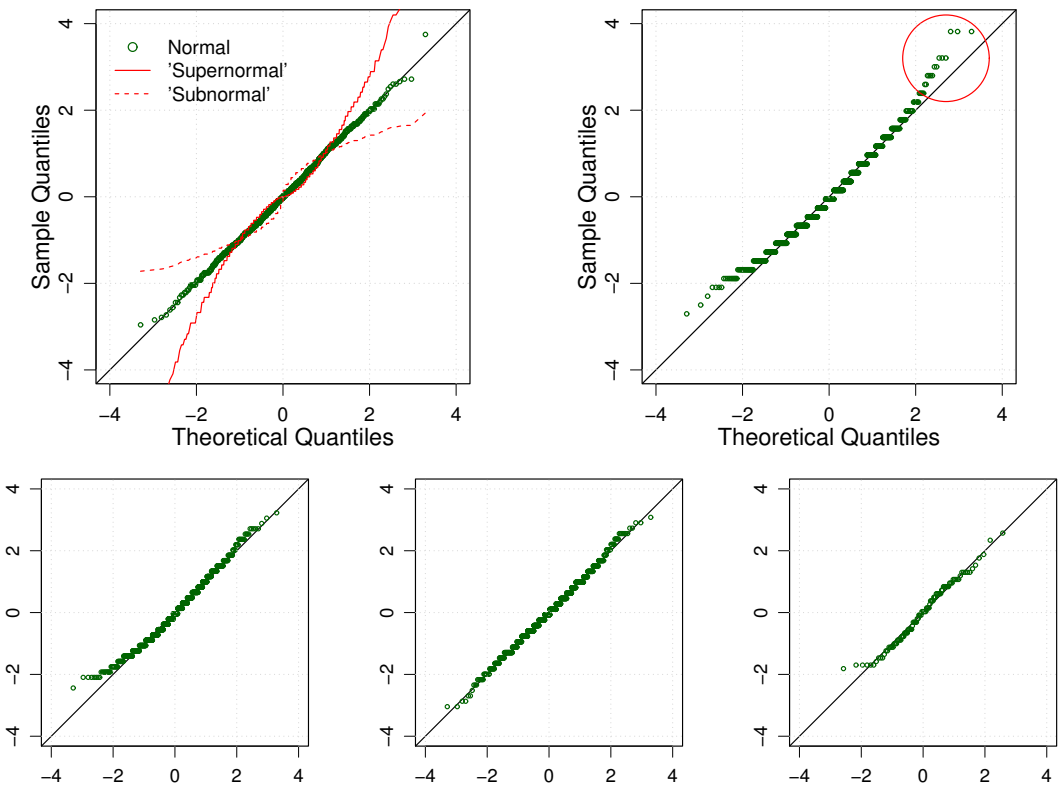


Figure 6.1: Sampled and theoretical quantiles for formulas described in the experimental section (top: `alu2_gr_rcs_w8`, `lang19`; bottom: `2bitmax_6`, `wff-3-150-525`, `ls11-norm`).

the average and dividing by the standard deviation) to the normal distribution: (normalized $d_{\text{MiniCount}} = d$) $\sim \frac{1}{\sqrt{2\pi}} e^{-d^2/2}$. The 'supernormal' and 'subnormal' lines show that the fit is much worse when the exponent of d is, for example, 1.5 or 2.5. The top-right plot shows that the corresponding domain (Langford problems) is somewhat on the border of being log-normally distributed, which is reflected in our experimental results to be described later.

Note that the nature of statistical tests is such that if the distribution of $\mathbb{E}\left[\#F_{\text{MiniCount}}\right]$ is not *exactly* log-normal, obtaining more and more samples will eventually lead to rejecting the log-normality hypothesis. For most practical purposes, being "close" to log-normally distributed suffices.

Confidence Interval Bound. Assuming the output samples from `MiniCount` $\{o_1, \ldots, o_n\}$ come from a log-normal distribution, we use them to compute the upper bound $c_{\max}$ of the confidence interval for the mean of $\#F_{\text{MiniCount}}$. The exact method for computing $c_{\max}$ for a log-normal distribution is complicated, and seldom used in practice. We use a conservative bound computation [Zhou and Sujuan, 1997]: let $y_i = \log(o_i)$, $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ denote the sample mean, and $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(y_i - \bar{y})^2$ the sample variance. Then the conservative upper bound is constructed as

$$\tilde{c}_{\max} = \exp\left(\bar{y} + \frac{s^2}{2} + \left(\frac{n-1}{\chi^2_\alpha(n-1)} - 1\right)\sqrt{\frac{s^2}{2}\left(1 + \frac{s^2}{2}\right)}\ \right) \tag{6.2}$$

where $\chi^2_\alpha(n-1)$ is the α-percentile of the chi-square distribution with $n-1$ degrees of freedom. Since $\tilde{c}_{\max} \geq c_{\max}$ we still have $\Pr\left[\tilde{c}_{\max} \geq \mathbb{E}\left[\#F_{\text{MiniCount}}\right]\right] \geq 1 - \alpha$.

The main assumption of the method described in this section is that the distribution of $\#F_{\text{MiniCount}}$ can be well approximated by a log-normal. This, of course, depends on the nature of the search space of `MiniCount` on a particular formula. As noted before, the assumption may sometimes be incorrect. In particular, one can construct a pathological search space where the reported upper bound will be lower than the actual number of solutions. Consider a problem P that consists of two non-interacting subproblems P_1 and P_2, where it is sufficient to solve either one of them to solve P. Suppose P_1 is very easy to solve (e.g., requires few choice points that are easy to find) compared to P_2, and P_1 has very few solutions compared to P_2. In such a case, `MiniCount` will almost always solve P_1 (and thus estimate the number of solutions of P_1), which would leave an arbitrarily large number of solutions of P_2 unaccounted for. This situation violates the assumption that $\#F_{\text{MiniCount}}$ is log-normally distributed, but it may be left unnoticed. This possibility of a false upper bound is a consequence of the inability to prove from samples that a random variable is log-normally distributed (one may only disprove

this assertion). Fortunately, as our experiments suggest, this situation is rare and does not arise in many real-world problems.

6.3 Experimental Results

We conducted experiments with `BPCount` as well as `MiniCount`, with the primary focus on comparing the results to exact counters and the recent `SampleCount` algorithm providing probabilistically guaranteed lower bounds. We used a cluster of 3.8 GHz Intel Xeon computers running Linux 2.6.9-22.ELsmp. The time limit was set to 12 hours and the memory limit to 2 GB.

We consider problems from five different domains, many of which have previously been used as benchmarks for evaluating model counting techniques: circuit synthesis, random k-CNF, Latin square construction, Langford problems, and FPGA routing instances from the SAT 2002 competition. The results are summarized in Tables 6.1 and 6.2. The columns show the performance of `BPCount` and `MiniCount`, compared against the exact solution counters `Relsat`, `Cachet`, and `c2d` (we report the best of the three for each instance; for all but the first instance, `c2d` exceeded the memory limit) and `SampleCount`. The table shows the reported bounds on the model counts and the corresponding runtime in seconds.

For `BPCount`, the damping parameter setting (i.e., the κ value) we use for the damped BP marginal estimator is 0.8, 0.9, 0.9, 0.5, and either 0.1 or 0.2 for the five domains, respectively. This parameter is chosen (with a quick manual search) as high as possible so that BP converges in a few seconds or less. The exact counter `Cachet` is called when the formula is sufficiently simplified, which is when 50 to 500 variables remain, depending on the domain. The lower bounds on the model count are reported with 99% confidence. We see that a significant improve-

Table 6.1: Performance of `BPCount` and `MiniCount`. [R] and [C] indicate partial counts obtained from `Cachet` and `Relsat`, respectively. `c2d` was slower for the first instance and exceeded the memory limit of 2 GB for the rest. Runtime is in seconds. Numbers in **bold** indicate the dominating techniques, if any, for that instance, i.e., that the best bound (lower and upper separately) obtained in the least amount of time.

			`Cachet` / `Relsat` / `c2d` (exact counters)		`SampleCount` (99% confidence)		`BPCount` (99% confidence)		`MiniCount` (99% confidence)			
Instance	num. of vars	True Count (if known)	Models	Time	LWR-bound	Time	LWR-bound	Time	S-W Test	Average	UPR-bound	Time
COMBINATORIAL PROBS.												
Ramsey-20-4-5	190	—	$\geq 9.0 \times 10^{11}$	12 hrs[C]	$\geq 3.3 \times 10^{35}$	3.5 min	$\geq 5.2 \times 10^{30}$	1.7 min	√	1.9×10^{37}	$\leq \mathbf{9.1 \times 10^{42}}$	**2.7 sec**
Ramsey-23-4-5	253	—	$\geq 8.4 \times 10^{6}$	12 hrs[C]	$\geq 1.4 \times 10^{31}$	53 min	$\geq 2.1 \times 10^{24}$	12 min	√	2.3×10^{37}	$\leq \mathbf{5.5 \times 10^{43}}$	**11 min**
Schur-5-100	500	—	$\geq 1.0 \times 10^{14}$	12 hrs[C]	$\geq \mathbf{1.3 \times 10^{17}}$	**20 min**	—	12 hrs	√	1.7×10^{21}	$\leq \mathbf{1.0 \times 10^{27}}$	**52 sec**
Schur-5-140	700	—	—	12 hrs[C]	—	12 hrs	—	12 hrs		—	—	12 hrs
fclqcolor-18-14-11	603	—	$\geq 2.4 \times 10^{33}$	12 hrs[C]	$\geq \mathbf{3.9 \times 10^{50}}$	**3.5 min**	—	12 hrs	√	1.3×10^{47}	$\leq \mathbf{1.2 \times 10^{51}}$	**1.5 sec**
fclqcolor-20-15-12	730	—	$\geq 8.6 \times 10^{38}$	12 hrs[C]	$\geq \mathbf{3.1 \times 10^{57}}$	**6 min**	—	12 hrs	√	2.1×10^{60}	$\leq \mathbf{5.1 \times 10^{66}}$	**2 sec**
CIRCUIT SYNTH.												
2bitmax_6	252	2.1×10^{29}	$\mathbf{2.1 \times 10^{29}}$	**2 sec**[C]	$\geq 2.4 \times 10^{28}$	29 sec	$\geq 2.8 \times 10^{28}$	5 sec	√	2.4×10^{29}	$\leq 8.9 \times 10^{30}$	2 sec
3bitadd_32	8704	—	—	12 hrs[C]	$\geq \mathbf{5.9 \times 10^{1339}}$	**32 min**	—	12 hrs		—	—	12 hrs
RANDOM k-CNF												
wff-3-3.5	150	1.4×10^{14}	1.4×10^{14}	7 min[C]	$\geq 1.6 \times 10^{13}$	4 min	$\geq 1.6 \times 10^{11}$	3 sec	√	9.8×10^{14}	$\leq 1.7 \times 10^{17}$	0.6 sec
wff-3-1.5	100	1.8×10^{21}	1.8×10^{21}	3 hrs[C]	$\geq 1.6 \times 10^{20}$	4 min	$\geq 1.0 \times 10^{20}$	1 sec	√	5.9×10^{20}	$\leq 2.7 \times 10^{22}$	0.5 sec
wff-4-5.0	100	—	$\geq 1.0 \times 10^{14}$	12 hrs[C]	$\geq 8.0 \times 10^{15}$	2 min	$\geq \mathbf{1.0 \times 10^{16}}$	**2 sec**	√	1.0×10^{17}	$\leq \mathbf{1.1 \times 10^{19}}$	**0.6 sec**
FPGA routing (SAT2002)												
apex7_*_w5	1983	—	$\geq 4.5 \times 10^{47}$	12 hrs[R]	$\geq 8.8 \times 10^{85}$	20 min	$\geq \mathbf{2.9 \times 10^{94}}$	**3 min**	√	2.9×10^{94}	$\leq \mathbf{2.2 \times 10^{109}}$	**2 min**
9symml_*_w6	2604	—	$\geq 5.0 \times 10^{30}$	12 hrs[R]	$\geq 2.6 \times 10^{47}$	6 hrs	$\geq 1.8 \times 10^{46}$	6 min	√	6.5×10^{55}	$\leq \mathbf{4.8 \times 10^{61}}$	**52 sec**
c880_*_w7	4592	—	$\geq 1.4 \times 10^{43}$	12 hrs[R]	$\geq 2.3 \times 10^{273}$	5 hrs	$\geq 7.9 \times 10^{253}$	18 min	√	2.6×10^{260}	$\leq \mathbf{1.6 \times 10^{315}}$	**6 sec**
alu2_*_w8	4080	—	$\geq 1.8 \times 10^{56}$	12 hrs[R]	$\geq 2.4 \times 10^{220}$	143 min	$\geq 2.0 \times 10^{205}$	16 min	√	2.3×10^{209}	$\leq \mathbf{7.7 \times 10^{248}}$	**7 sec**
vda_*_w9	6498	—	$\geq 1.4 \times 10^{88}$	12 hrs[R]	$\geq 1.4 \times 10^{326}$	11 hrs	$\geq 3.8 \times 10^{289}$	56 min	√	4.1×10^{302}	$\leq \mathbf{4.4 \times 10^{410}}$	**13 sec**

Table 6.2: (Continued from Table 6.1.) Performance of `BPCount` and `MiniCount`.

			`Cachet` / `Relsat` / `c2d` (exact counters)		`SampleCount` (99% confidence)		`BPCount` (99% confidence)		`MiniCount` S-W (99% confidence)			
Instance	num. of vars	True Count (if known)	Models	Time	LWR-bound	Time	LWR-bound	Time	Test	Average	UPR-bound	Time
LATIN SQUARE												
ls8-norm	301	5.4×10^{11}	$\geq 1.7 \times 10^{8}$	12 hrs[R]	$\geq 3.1 \times 10^{10}$	19 min	$\geq \mathbf{1.9 \times 10^{10}}$	**12 sec**	$\times$	7.3×10^{14}	$\leq 5.6 \times 10^{14}$	1 sec
ls9-norm	456	3.8×10^{17}	$\geq 7.0 \times 10^{7}$	12 hrs[R]	$\geq 1.4 \times 10^{15}$	32 min	$\geq \mathbf{1.0 \times 10^{16}}$	**11 sec**	$\surd$	4.9×10^{19}	$\leq \mathbf{2.2 \times 10^{21}}$	**2 sec**
ls10-norm	657	7.6×10^{24}	$\geq 6.1 \times 10^{7}$	12 hrs[R]	$\geq 2.7 \times 10^{21}$	49 min	$\geq \mathbf{1.0 \times 10^{23}}$	**22 sec**	$\surd$	7.2×10^{25}	$\leq \mathbf{1.4 \times 10^{30}}$	**10 sec**
ls11-norm	910	5.4×10^{33}	$\geq 4.7 \times 10^{7}$	12 hrs[R]	$\geq 1.2 \times 10^{30}$	69 min	$\geq \mathbf{6.4 \times 10^{30}}$	**1 min**	$\surd$	3.5×10^{33}	$\leq \mathbf{1.3 \times 10^{41}}$	**100 sec**
ls12-norm	1221	—	$\geq 4.6 \times 10^{7}$	12 hrs[R]	$\geq 6.9 \times 10^{37}$	50 min	$\geq \mathbf{2.0 \times 10^{41}}$	**70 sec**	$\times$	3.3×10^{43}	$\leq 3.8 \times 10^{54}$	10 min
ls13-norm	1596	—	$\geq 2.1 \times 10^{7}$	12 hrs[R]	$\geq 3.0 \times 10^{49}$	67 min	$\geq \mathbf{4.0 \times 10^{54}}$	**6 min**	$\times$	1.2×10^{51}	$\leq 3.5 \times 10^{62}$	1.5 hrs
ls14-norm	2041	—	$\geq 2.6 \times 10^{7}$	12 hrs[R]	$\geq 9.0 \times 10^{60}$	44 min	$\geq \mathbf{1.0 \times 10^{67}}$	**4 min**	$\surd$	3.0×10^{65}	$\leq \mathbf{3.7 \times 10^{88}}$	**12 hrs**
ls15-norm	2041	—	$\geq 9.1 \times 10^{6}$	12 hrs[R]	$\geq 1.1 \times 10^{73}$	56 min	$\geq 5.8 \times 10^{84}$	12 hrs		—	—	12 hrs
ls16-norm	2041	—	$\geq 1.0 \times 10^{7}$	12 hrs[R]	$\geq \mathbf{6.0 \times 10^{85}}$	**68 min**	—	12 hrs		—	—	12 hrs
LANGFORD PROBS.												
lang-2-12	576	1.0×10^{5}	1.0×10^{5}	15 min[R]	$\geq 4.3 \times 10^{3}$	32 min	$\geq 2.3 \times 10^{3}$	50 sec	$\times$	2.5×10^{6}	$\leq 8.8 \times 10^{6}$	1 sec
lang-2-15	1024	3.0×10^{7}	$\geq 1.8 \times 10^{5}$	12 hrs[R]	$\geq 1.0 \times 10^{6}$	60 min	$\geq 5.5 \times 10^{5}$	1 min	$\times$	8.1×10^{8}	$\leq 4.1 \times 10^{9}$	2.5 sec
lang-2-16	1024	3.2×10^{8}	$\geq 1.8 \times 10^{5}$	12 hrs[R]	$\geq 1.0 \times 10^{6}$	65 min	$\geq 3.2 \times 10^{5}$	1 min	$\surd$	3.4×10^{7}	$\leq \mathbf{5.9 \times 10^{8}}$	**2 sec**
lang-2-19	1444	2.1×10^{11}	$\geq 2.4 \times 10^{5}$	12 hrs[R]	$\geq 3.3 \times 10^{9}$	62 min	$\geq 4.7 \times 10^{7}$	26 min	$\times$	9.2×10^{9}	$\leq 2.5 \times 10^{11}$	3.7 sec
lang-2-20	1600	2.6×10^{12}	$\geq 1.5 \times 10^{5}$	12 hrs[R]	$\geq 5.8 \times 10^{9}$	54 min	$\geq 7.1 \times 10^{4}$	22 min	$\times$	1.1×10^{13}	$\leq 6.7 \times 10^{13}$	4 sec
lang-2-23	2116	3.7×10^{15}	$\geq 1.2 \times 10^{5}$	12 hrs[R]	$\geq 1.6 \times 10^{11}$	85 min	$\geq 1.5 \times 10^{5}$	15 min	$\surd$	1.5×10^{13}	$\leq \mathbf{7.9 \times 10^{15}}$	**6 sec**
lang-2-24	2304	—	$\geq 4.1 \times 10^{5}$	12 hrs[R]	$\geq 4.1 \times 10^{13}$	80 min	$\geq 8.9 \times 10^{7}$	18 min	$\times$	3.0×10^{13}	$\leq 1.5 \times 10^{16}$	7 sec
lang-2-27	2916	—	$\geq 1.1 \times 10^{4}$	12 hrs[R]	$\geq 5.2 \times 10^{14}$	111 min	$\geq 1.3 \times 10^{6}$	80 min	$\surd$	3.3×10^{15}	$\leq \mathbf{3.4 \times 10^{16}}$	**7 sec**
lang-2-28	3136	—	$\geq 1.1 \times 10^{4}$	12 hrs[R]	$\geq 4.0 \times 10^{14}$	117 min	$\geq 9.0 \times 10^{2}$	70 min	$\surd$	1.2×10^{14}	$\leq \mathbf{3.3 \times 10^{16}}$	**11 sec**

Table 6.3: Performance of `MiniCount` without restarts.

Instance	num. of vars	True Count (if known)	`MiniCount` **without restarts** S-W Test	Average	(99% confidence) UPR-bound	Time
CIRCUIT SYNTH.						
2bitmax_6	252	2.1×10^{29}	$\surd$	3.5×10^{30}	$\leq 4.3 \times 10^{32}$	2 sec
RANDOM k-CNF						
wff-3-3.5	150	1.4×10^{14}	$\surd$	4.3×10^{14}	$\leq 6.7 \times 10^{15}$	2 sec
wff-3-1.5	100	1.8×10^{21}	$\surd$	1.2×10^{21}	$\leq 4.8 \times 10^{22}$	2 sec
wff-4-5.0	100	—	$\surd$	2.8×10^{16}	$\leq 5.7 \times 10^{28}$	2 sec
LATIN SQUARE						
ls8-norm	301	5.4×10^{11}	$\surd$	6.4×10^{12}	$\leq 1.8 \times 10^{14}$	2 sec
ls9-norm	456	3.8×10^{17}	$\surd$	6.9×10^{18}	$\leq 2.1 \times 10^{21}$	3 sec
ls10-norm	657	7.6×10^{24}	$\surd$	4.3×10^{26}	$\leq 7.0 \times 10^{30}$	7 sec
ls11-norm	910	5.4×10^{33}	$\surd$	1.7×10^{34}	$\leq 5.6 \times 10^{40}$	35 sec
ls12-norm	1221	—	$\surd$	9.1×10^{44}	$\leq 3.6 \times 10^{52}$	4 min
ls13-norm	1596	—	$\surd$	1.0×10^{54}	$\leq 8.6 \times 10^{69}$	42 min
ls14-norm	2041	—	$\surd$	3.2×10^{63}	$\leq 1.3 \times 10^{86}$	7.5 hrs
LANGFORD PROBS.						
lang-2-12	576	1.0×10^{5}	$\times$	5.2×10^{6}	$\leq 1.0 \times 10^{7}$	2.5 sec
lang-2-15	1024	3.0×10^{7}	$\surd$	1.0×10^{8}	$\leq 9.0 \times 10^{8}$	8 sec
lang-2-16	1024	3.2×10^{8}	$\times$	1.1×10^{10}	$\leq 1.1 \times 10^{10}$	7.3 sec
lang-2-19	1444	2.1×10^{11}	$\times$	1.4×10^{10}	$\leq 6.7 \times 10^{12}$	37 sec
lang-2-20	1600	2.6×10^{12}	$\surd$	1.4×10^{12}	$\leq 9.4 \times 10^{12}$	3 min
lang-2-23	2116	3.7×10^{15}	$\times$	3.5×10^{12}	$\leq 1.4 \times 10^{13}$	23 min
lang-2-24	2304	—	$\times$	2.7×10^{13}	$\leq 1.9 \times 10^{16}$	25 min
FPGA routing (SAT2002)						
apex7_*_w5	1983	—	$\surd$	7.3×10^{95}	$\leq 5.9 \times 10^{105}$	2 min
9symml_*_w6	2604	—	$\surd$	3.3×10^{58}	$\leq 5.8 \times 10^{64}$	24 sec
c880_*_w7	4592	—	$\surd$	1.0×10^{264}	$\leq 6.3 \times 10^{326}$	26 sec
alu2_*_w8	4080	—	$\surd$	1.4×10^{220}	$\leq 7.2 \times 10^{258}$	16 sec
vda_*_w9	6498	—	$\surd$	1.6×10^{305}	$\leq 2.5 \times 10^{399}$	42 sec

ment in efficiency is achieved when the BP marginal estimation is used through `BPCount`, compared to solution sampling as in `SampleCount` (also run with 99% correctness confidence). For the smaller formulas considered, the lower bounds reported by `BPCount` border the true model counts. For the larger ones that could only be counted partially by exact counters in 12 hours, `BPCount` gave lower bound counts that are very competitive with those reported by `SampleCount`, while the running time of `BPCount` is, in general, an order of magnitude lower than that of `SampleCount`, often just a few seconds.

For `MiniCount`, we obtain $n = 100$ samples of the estimated count for each formula, and use these to estimate the upper bound statistically using the steps described earlier. The test for log-normality of the sample counts is done with a rejection level 0.05, that is, if the Shapiro-Wilk test reports p-value below 0.05, we conclude the samples do *not* come from a log-normal distribution, in which case no upper bound guarantees are provided (`MiniCount` is "unsuccessful"). When the test passed, the upper bound itself was computed with a confidence level of 99% using the computation of Zhou and Sujuan [1997]. The results are summarized in the last set of columns in Tables 6.1 and 6.2. We report whether the log-normality test passed, the average of the counts obtained over the 100 runs, the value of the statistical upper bound $c_{\max}$, and the total time for the 100 runs. We see that the upper bounds are often obtained within seconds or minutes, and are correct for all instances where the estimation method was successful (i.e., the log-normality test passed) and true counts or lower bounds are known. In fact, the upper bounds for these formulas (except `lang-2-23`) are correct w.r.t. the best known lower bounds and true counts even for those instances where the log-normality test failed and a statistical guarantee cannot be provided. The Langford problem family seems to be at the boundary of applicability of the `MiniCount` approach, as indicated

by the alternating successes and failures of the test in this case. The approach is particularly successful on industrial problems (circuit synthesis, FPGA routing), where upper bounds are computed within seconds. Our results also demonstrate that a simple average of the 100 runs provides a very good approximation to the number of solutions. However, simple averaging can sometimes lead to an incorrect upper bound, as seen in `wff-3-1.5`, `ls13-norm`, `alu2_gr_rcs_w8`, and `vda_gr_rcs_w9`, where the simple average is below the true count or a lower bound obtained independently. This justifies our statistical framework, which as we see provides more robust upper bounds.

As stated earlier, we allow restarts in `MiniCount` after at least one backtrack has occurred. In the preliminary version of this work [Kroc et al., 2008b], we reported results without restarts, and although the results in the two scenarios are sometimes fairly close, we believe allowing restarts will be effective and even indispensable on harder problem instances. We thus report here numbers only with restarts. For completeness, the number for `MiniCount` *without restarts* are given in Table 6.3. In most cases, this is somewhat slower than running `MiniCount` with restarts. However, there are cases this is a bit faster or where the resulting numbers appear to fit a log-normal distribution better than with restarts.

6.4 Discussion

This work brings together techniques from message passing, DPLL-based SAT solvers, and statistical estimation in an attempt to solve the challenging model counting problem. We show how (a damped form of) BP can help significantly boost solution counters that produce lower bounds with probabilistic correctness guarantees. `BPCount` is able to provide good quality bounds in a fraction of the time

compared to previous, sample-based methods. We also describe the first effective approach for obtaining good upper bounds on the solution count. Our framework is general and enables one to turn any state-of-the-art complete SAT/CSP solver into an upper bound counter, with very minimal modifications to the code. Our `MiniCount` algorithm provably converges to an upper bound, and is remarkably fast at providing good results in practice.

Chapter 7

Integrating Systematic and Local Search Paradigms: A New Strategy for MaxSAT

Boolean Satisfiability (SAT) solvers have seen tremendous progress in recent years. Several of the current best open source SAT solvers scale up to instances with over a million variables and several million clauses. These advances have led to an ever growing range of applications, such as in hardware and software verification, and planning (cf. Handbook of SAT, Biere et al. [2009]). In fact, the technology has matured from being a largely academic endeavor to an area of research with strong academic and industrial participation. The current best SAT solvers for handling "structured" instances are based on Davis-Putnam-Logemann-Loveland or DPLL [Davis and Putnam, 1960, Davis et al., 1962] style complete, systematic search. The competing search paradigm for SAT solving is based on local search (cf. Hoos and Stützle [2004]), which performs well in certain problem domains but, in general, is not as effective on highly structured problem domains.

Determining whether a Boolean formula is satisfiable or not is a special case of the maximum satisfiability (MaxSAT) problem, where the goal is to find an

assignment that satisfies as many clauses, or constraints, as possible. Even though MaxSAT is a natural generalization of SAT, and thus closely related, progress has been much slower on efficient solution strategies for the MaxSAT problem. There is a good explanation as to why this is the case. Two key components behind the rapid progress for DPLL based SAT solvers are: highly effective unit propagation, and clause learning. (Other factors include randomization and restart strategies, and effective data structures.) Both techniques in a sense focus on avoiding local inconsistencies: when a unit clause occurs in a formula, one should immediately assign a truth value to the variable so that it satisfies the clause, and when a branch reaches a contradiction, a no-good clause can be derived which captures the cause of the local inconsistency.

In a MaxSAT setting, these strategies, at least in the context of branch-and-bound techniques, can be quite counter-productive and in fact lead to incorrect results. For example, for an unsatisfiable instance, the optimal assignment, i.e., one satisfying the most clauses, may be the one that violates several unit clauses. Also, when a contradiction is reached, the best solution may be to violate one of the clauses that led to the contradiction rather than adding a no-good which effectively steers the search away from the contradiction. Hence, neither unit propagation nor clause learning appear directly suitable for a MaxSAT solver. Unfortunately, taking such mechanisms out of the DPLL search strategy dramatically reduces the effectiveness of the search. This is confirmed when one considers the performance of exact solvers for MaxSAT that, in effect, employ a branch-and-bound search but do not have unit propagation or clauses learning incorporated. Although progress has been made in the area of exact MaxSAT solvers, the instances that can be solved in practice are generally much smaller than instances that can be handled by SAT solvers. Just as an example, a bounded model checking instance considered in our

experiments, cmu-bmc-barrel6.cnf, is solved by the state-of-the-art exact MaxSAT solvers such as `maxsatz` [Li et al., 2007a] and `msuf` [Marques-Silva and Manquinho, 2008] in 20-30 minutes while within only a couple of seconds by an exact SAT solver like `MiniSat` [Eén and Sörensson, 2005].

A more natural fit for the MaxSAT problem is to use local search. Such methods are incomplete but they can find approximate solutions for large problem instances. The advantage of a local search strategy is that it in a sense operates in a more global manner, generally using the current number of unsatisfied clauses as a gradient to guide a further descent. In this process local search doesn't hesitate to violate unit clauses if that appears beneficial. Some of the earliest local search results for MaxSAT were based on the `Walksat` procedure [Selman et al., 1996]. From the perspective of local search, the basic SAT and MaxSAT strategies are quite similar. Researchers have recently showed that more sophisticated local search strategies can significantly improve upon the `Walksat` performance. For example, two state-of-the-art local search solvers, which also work very well for MaxSAT, are `saps` [Hutter et al., 2002, Tompkins and Hoos, 2003] and `adaptg2wsat+p` [Li et al., 2007b].

The performance of the recent local search methods on large hard problem instances (unsatisfiable instances at the edge of feasibility for current SAT solvers) appears impressive. For example, on the industrial instance babic-dspam-vc973.cnf from the SAT Race-2008 [Sinz (Organizer), 2008], a typical unsatisfiable benchmark instance with 900,000+ clauses, `Walksat` can find an assignment that leaves around 35,000 clauses unsatisfied but `saps` and `adaptg2wsat` can find solutions with around 1,500 clauses unsatisfied (see Tables 7.1 and 7.2). A systematic SAT solver, `MiniSat` [Eén and Sörensson, 2005], can prove the instance to be unsatisfiable in around 3 hours. The question remains, how close the obtained MaxSAT solution

is to the optimal solution? The fact that different local search MaxSAT methods converge to roughly the same number of unsatisfied clauses and running them for many more hours does not further improve the solution may lead one to conjecture that 1,000 might be close to optimal. However, one reason to be less confident of the quality of the solution is that local search methods have been shown to have trouble dealing with long chains of dependencies in structured problem instances, even though special encodings and the addition of inferred clauses can help alleviate some of these problems (e.g., Prestwich [2007], Hirsch and Kojevnikov [2005]).

The issue of problem structure, on which DPLL methods excel but which challenges local search style methods, leads us naturally to the two main questions addressed in this chapter: (1) *How good are the current best solutions on structured problems?* (2) *Can state-of-the-art SAT solvers help on the MaxSAT problem?* As we will demonstrate, the current best solutions on structured problems are often surprisingly far from optimal. We show this by finding solutions with a single unsatisfied clause (and therefore optimal) for unsatisfiable problem instances, where the best previously known solutions had hundreds or thousands of unsatisfied clauses. For example, the optimal solution for the instance mentioned above does not have around 1,000 unsatisfied clauses but actually just one (out of 900,000+ clauses total). We demonstrate this using a new solver that came forth out of our study of the second question—can we use DPLL to boost a local search solver? We introduce a **hybrid solution strategy**, where information from a DPLL SAT solver provides continued guidance for a local search style MaxSAT solver. We use `MiniSat` as our DPLL solver and `Walksat` as our MaxSAT solver. Our hybrid solver is called **MiniWalk**. The integration of the solvers is surprisingly clean, requiring only a few lines of code.

The incremental, systematic search approach behind backtrack search (as in

MiniSat) and the stochastic local search approach (as in Walksat) represent the two main combinatorial search paradigms. It appears natural to integrate these approaches to leverage each other's strengths. In fact, there have been various attempts at such integration, for example, using local search to find good branching variables for DPLL or to identify minimal unsatisfiable subsets (e.g., Mazure et al. [1998], Grégoire et al. [2007a]). However, these attempts, especially those targeted at traditional SAT solving, have not been as effective as one would have hoped. One particular issue that hampers integration of solvers in general is that time spent in the less effective solver is often more costly than the time saved by the faster solver when using the information obtained with the slower one. More concretely, although Walksat may be able to provide better branching information leading to a smaller DPLL tree, the time saved through the reduction in tree size is often less than the time spent on running Walksat. Hence, the issue of how much time to spend in each solver becomes a careful balancing act, and is often problem instance dependent.

Fortunately, the compute paradigm based on multi-core processors eliminates much of these difficulties. In particular, in our hybrid solver, we run both MiniSat and Walksat at full speed in parallel on two different cores of a standard dual-core processor. During the run, MiniSat writes its current branching information into a shared memory. More specifically, this memory contains the values of all variables that are set in the current branch of the DPLL search. Running on the other processor, before flipping the value of a variable, Walksat "peeks" at the shared memory and only makes the flip if the variable is not set on the DPLL branch or if the flip will set the variable on the branch to its current value. (Stated differently, Walksat does not flip any variable to a setting conflicting with that of the current DPLL branch.) In this setting, information from DPLL continuously steers the

`Walksat` search strategy. And, as we will see, the DPLL search frequently steers `Walksat` to extremely promising regions of the search space, where near-satisfying assignments exist. Moreover, without the guidance, `Walksat` or other local search methods do not appear to reach such promising areas of the search space. We will discuss the search behavior of our hybrid strategy in more detail in the text. We will see that our hybrid search progresses in a manner not observed in any non-hybrid search strategy. We again stress that the dual-core mechanism is a key factor behind the success of our approach, because it eliminates the need for intricate time allocations for the two types of search.

In summary, our results show that DPLL can provide highly effective guidance for a local search style solver for the MaxSAT problem, leading to the optimal solution on many structured instances. From the SAT Race-2008 benchmark set, we find a provably optimal solution (one unsatisfied clause) on 37 of the 52 unsatisfiable instances, while the current best alternative approaches suggest hundreds, if not thousands, of unsatisfied clauses in the best solutions to these very instances. Inspired by this success, we designed an alternative way of combining systematic and local search: systematic search is ran first and tolerates a certain number of contradictions, and local search is performed at the end to "fine tune" the assignment. This modified approach, which we call `RelaxedMinisat`, was able to show that 51 out of the 52 (!) instances have assignments with only one unsatisfied clause. This work therefore provides a step towards closing the performance gap between SAT solvers and MaxSAT solvers. The results also demonstrate that there is a real potential in using multi-core processors in combinatorial search, where shared memory is used to provide a low-cost communication channel between the processes.

We note that the focus of our hybrid approach is naturally on instances that

are non-trivial for both DPLL and local search solvers. Most of the MaxSAT benchmarks currently available, such as the ones used in MaxSAT Evaluation 2007 [Argelich et al., 2007], are targeted towards the scalability region of exact MaxSAT solvers, and are thus too easy for DPLL-based SAT solvers such as `Minisat`. On such instances, the DPLL part of our hybrid solver, `MiniWalk`, often terminates within a second, providing little guidance to the local search part. The hybrid method therefore essentially falls back to pure local search, performing no better (but also no worse) than alternative approaches on these instances. To illustrate the strength and promise of our approach, we perform an evaluation on *all* unsatisfiable instances used in SAT Race-2008, which are challenging not only as MaxSAT instances but also as satisfiability instances. We hope our positive results will encourage the development of MaxSAT benchmarks that are non-trivial to prove unsatisfiable.

Given the performance of other MaxSAT solvers on the SAT Race-2008 instances, it is quite surprising that all but one of these instances have only a *single* unsatisfied clause in the optimal MaxSAT solution. This clause can be thought of as a "bottleneck constraint" for the instance. In fact, by running our solver multiple times with different random seeds, we can identify several different bottleneck constraints, relaxing any one of which will turn the instance into a satisfiable one. In this sense, bottleneck constraints provide a form of explanation of unsatisfiability, complementing the information provided by other concepts being explored in the literature such as minimal unsatisfiable cores, minimal sets of unsatisfiable tuples, etc. (cf. Marques-Silva and Manquinho [2008], Grégoire et al. [2007a,b]).

The kind of information provided by the single violated constraint obviously depends on the problem encoding. To investigate this further, we conducted experiments on AI planning instances for which we have full knowledge of the

variable and clause semantics. Specifically, we considered the TPP (Traveling Purchase Problem) domain from the IPC-5 Competition [Gerevini et al., 2006], giving the planner one fewer time step than what it needs to solve the instance. We translated instances of this infeasible problem into unsatisfiable SAT formulas following the SatPlan framework [Kautz and Selman, 1998]. Our solver, `RelaxedMinisat`, was able to identify a single violated clause in instances thus generated. With different random seeds, we obtained a number of different "explanations" of the unsatisfiability of each instance, in the form of a bottleneck constraint. As an example, in an instance involving 10 goods to be purchased in various quantities and transported, 1 storage depot, 3 markets, and 3 trucks, a violated bottleneck constraint had the following precondition-style semantics: "in order to drive truck-3 from depot-1 to market-2 at time-step-7, truck-3 must be at depot-1 at the end of time-step-6." One thing this bottleneck constraint suggests is that there is a plan that "almost" works if the post condition achieved by the action under consideration is somehow made to hold, i.e., if we could somehow make a truck available at market-2 at the end of time-step-7. Indeed, if we add a fourth truck as a new resource at market-2 as part of the initial conditions, the instance becomes satisfiable in the given time steps.

7.1 Preliminaries

When a CNF formula F is unsatisfiable, i.e., there is no truth assignment to the variables in V for which all clauses of F are satisfied, one is often interested in solving the problem as much as possible. Formally, *Maximum Satisfiability* or MaxSAT is the optimization problem of finding a truth assignment that satisfies

as many clauses of an input formula F as possible. We will refer to such truth assignments as *optimal MaxSAT solutions*. One natural quantity of interest when performing a search for an optimal MaxSAT solution is the *number of unsatisfied clauses* found at the end of a search procedure. As we will see in Section 7.4, the optimal MaxSAT solutions for many interesting industrial problem instances happen to have only one unsatisfied clause, and the proposed hybrid method is often able to find such solutions very efficiently.

Most of the successful search methods for SAT can be classified into two categories: systematic complete search and heuristic local search. For SAT, systematic complete search takes the form of the Davis-Putnam-Logemann-Loveland or DPLL procedure [Davis and Putnam, 1960, Davis et al., 1962]. The idea is to do a standard branch-and-backtrack search in the space of all partial truth assignments. Heuristics are used to set variables to promising values until either a solution is found or a contradiction is reached; in the latter case, the solver backtracks, flips the value of a variable higher up in the search tree, and systematically continues the search for a solution—now in a previously unexplored part of the search space. Modern SAT solvers based on DPLL employ additional techniques such as clause learning, restarts, highly efficient data structures, etc. While the systematic solver, `Minisat`, that forms one half of our hybrid approach does implement these advanced techniques, the details of these techniques are not crucial for understanding the rest of this chapter.

Local search SAT solvers, also referred to as stochastic local search or SLS solvers, work with complete truth assignments which, of course, violate some number of clauses before a solution is found. The idea here is to do local modifications to the current complete truth assignment, guided essentially by the currently unsatisfied clauses. The local modifications often take the form of heuristically

selecting one variable to *flip*, based often on how many of the currently unsatisfied clauses will become satisfied and how many of the currently satisfied clauses will become unsatisfied. Refined local search solvers employ techniques such as selecting variables mostly only from currently unsatisfied clauses, clause re-weighting and stochastic noise to escape local minima, adaptively adjusting the noise level, etc.

7.2 Hybrid Solver: Using DPLL to Guide Local Search

Our hybrid MaxSAT solver, `MiniWalk`, has two parts that are very independent except for sharing a small amount of memory for information exchange: a DPLL solver and a local search solver. The main idea is the following: *`MiniSat` informs the local search which part of the search space it is currently searching in, and `Walksat` loosely restrains itself to the same part of the search space by not flipping a literal against `MiniSat`.*

Both DPLL and local search are performed *simultaneously*, so there are literally two processes (solvers) running at the same time. This does not slow down the performance of either, given the multi-core architecture that is becoming a standard in the computer industry and the very low communication overhead involved. The details will follow shortly, but let us remark already that any DPLL heuristic and any local search heuristic can be instrumented to create a hybrid solver in this fashion, and the actual implementation requires only a few additional lines of code.

A DPLL solver proceeds by successively selecting variables and their polarities (truth values), fixing those variables accordingly, and simplifying the instance. A

lot of effort has been invested in designing heuristics that guide the search into parts of the search space where solutions are likely to be found. A good DPLL heuristic often guides the search in the direction where as many clauses as possible are satisfied. We use this search bias even on formulas that are not satisfiable, assuming that good near-solutions will lie in the regions that look attractive to a DPLL heuristic. The information about the region of the search space that the DPLL solver is currently exploring is communicated using a *shared memory array*. In this array shared between the two solver processes, each variable of the problem instance has either the value "unassigned" or the polarity (0 or 1) that is currently assigned to it by the DPLL search. The only change to the code of the DPLL solver is thus a line that writes the correct polarity every time a variable is fixed (branched on or set, e.g., by unit propagation). The variable value in the shared array is reverted back to "unassigned" upon backtracking.

On the local search side, the modification is also only very slight. The standard local search procedure has two steps that keep repeating: pick a variable to flip, and flip the selected variable. The only modification we make is in the second step: we flip the truth value of a variable *only if* the new value does not violate the setting the DPLL search has for that variable in the shared array. In other words, we flip a variable from, say, TRUE to FALSE only if it is either unassigned by the DPLL search or is assigned FALSE. Otherwise we simply do nothing and `Walksat` selects a different variable to flip in the next step. This is depicted formally as Algorithm 1. The two steps marked with "***" are literally the only change that needs to be made to pure `Walksat`.

We use the Unix IPC shared memory framework in both systematic and local search solvers as follows (a C code snippet):

```
#include <sys/shm.h>
int *sharedMem = NULL;
```

```
long shmKey = 0x11112222;   //any unique id
int shmId = shmget( shmKey, \
   sizeof(int)*(nVars+1), IPC_CREAT|0600 );
sharedMem = (int*)shmat( shmId, NULL, 0 );
```

This memory can now be written to (within DPLL) and read from (within local search) by `sharedMem[verId]`. At the end of each program, `shmctl(shmid, IPC_RMID, 0)` frees up the shared memory.

Algorithm 1: Hybrid-Walksat part of `Miniwalk`

```
begin
    *** Initialize shared memory array M
    σ ← a randomly generated truth assignment for F
    for j ← 1 to MAX-FLIPS do
        if σ satisfies F then return σ
        Select a variable v using a heuristic
        *** if M[v] ≠ value(v) then
            Flip the value of v in σ
end
```

This way, the information flows only in one direction: from DPLL to the local search. The local search is responsible for reporting the best-so-far achieved assignment, which is in turn used as an estimated solution to the MaxSAT instance. If the DPLL search finishes very quickly (i.e., easily determines unsatisfiability within a few seconds), then the local search has no time to take advantage of the guidance provided by the shared array, and the hybrid method does not improve upon plain local search. If, on the other hand, we have a sufficiently hard instance at hand, we found that this strategy is remarkably successful at finding very good solutions.

There is, of course, the question of which DPLL and local search solvers to select. The `Minisat` and `Walksat` combination turned out to perform the best. We also considered `Rsat` DPLL solver, which adds the concept of "restart memory" to the search. While successful for SAT, we found that `Rsat` did not perform as well as

`Minisat` for our purposes, perhaps because the memory constrained the search to too local a region and also because `Rsat` generally terminates quicker than `Minisat`, providing less guidance to `Walksat`. On the local search side, we tried using more powerful algorithms, namely `adaptg2wsat+p` and `saps`, which performed best on our problem suite as stand-alone local search solvers (see Section 7.4). But we found that neither performed as well as `Walksat` when coupled with a DPLL solver, presumably because their decisions/flips were much more focused than `Walksat`'s, and it was harder for the solvers to follow the DPLL guidance.

7.3 Relaxed Solver: Tolerating Contradictions

The encouraging results we see from the hybrid approach call for investigation as to where is most of the power coming from: the local search, or the systematic search? By designing a two-stage search process, which we call `RelaxedMinisat`, we show that relaxing the systematic search to tolerate a few contradictions is already a very powerful approach for MaxSAT. Local search is then used to "fine tune" the result. We thus show that traditional SAT solver techniques like unit propagation and clauses learning can be directly used as effective heuristics for an incomplete MaxSAT algorithm.

The idea behind `RelaxedMinisat` is relatively simple: use a state-of-the-art DPLL solver such as `Minisat` [Eén and Sörensson, 2005] but relax it to "ignore" a fixed number ℓ of conflicts on each search branch, and quit once a truth assignment violating at most ℓ clauses is found. If necessary, run a low-noise local search initiated at the partial assignment found by the DPLL solver.

We chose `Minisat` [Eén and Sörensson, 2005] as the DPLL solver to build on. To implement the one extra feature we need—allowing up to ℓ conflicts on a search

branch—we slightly modify the routine performing unit propagation. When a conflict is detected on a search branch b and it is amongst the first ℓ conflicts along b, the clause causing the conflict is silently ignored until the solver later backtracks the closest branching decision made before getting to this point. All other functionality of the solver is left intact, including clause learning. If ℓ conflicts are reached on the branch b, conflict directed backtracking is performed as usual. It is not hard to see that the conflict clause C learned at this point has the following property: *if any truth assignment σ satisfies all clauses except the ℓ conflict generating clauses seen on branch b, then σ also satisfies C.* Adding C as a learned clause therefore preserves the soundness of the technique. (But C could, in principle, rule out other potential solutions that violate a different set of ℓ or fewer clauses; adding C therefore does not preserve completeness.) If a solution is found before reaching ℓ conflicts, this solution is reported as a candidate MaxSAT solution; this provides an upper bound for the optimal MaxSAT solution. Alternatively, `RelaxedMinisat` can return the "UNSAT" status, in which case we increase the parameter ℓ to attempt to find some other truth assignment. (The "UNSAT" status of `RelaxedMinisat` does *not* mean that there is no assignment violating at most ℓ clauses.) Using binary search, one can find the smallest value of ℓ for which `RelaxedMinisat` does report an assignment; for nearly satisfiable instances such as the ones with very few bottleneck constraints that we focus on, this is very quick as ℓ is small. Experiments suggest that *rapid restarting* improves the performance of `RelaxedMinisat`. We restart after every 100 conflicts.

For some of the harder formulas, the candidate solution found by `RelaxedMinisat` was not clearly optimal (i.e., had more than one violated clause). In this case, we ran the local search solver `Walksat` [Selman et al., 1996] (without any modification) with the search initiated at this candidate truth assignment,

looking for a better solution. Empirically, we observed that rapid restarts are again beneficial, along with very low noise to make the local search extremely greedy and focused for a thorough search in the vicinity of the candidate solution found by the DPLL part.

7.4 Experimental Results

We conducted experiments on all 52 unsatisfiable formulas from the SAT Race-2008 suite [Sinz (Organizer), 2008]. The reason for choosing these instances rather than the Max-SAT Evaluation 2007 [Argelich et al., 2007] instances is that the latter are all too easy for `MiniSat`, thus limiting the DPLL guidance provided to the hybrid method to just a few seconds, turning it into essentially local search. The SAT Race instances well illustrate the strengths and promises of the approach, and by using all unsatisfiable ones, we did not bias our selection to only "good" instances. Although not traditional in the MaxSAT domain, we believe that useful information can be obtained from the near-solutions which our technique finds. The usefulness of such information, in the sense of identifying bottleneck constraints, is relatively limited if the minimum number of unsatisfied clauses is large. That is why we do not discuss in depth the performance of our algorithms on instances with no "near solutions," i.e., where the optimal solution has a large number of unsatisfied clauses.

The solvers used in the comparison were from four families: exact MaxSAT solvers `maxsatz` [Li et al., 2007a] and `msuf` [Marques-Silva and Manquinho, 2008]; local search SAT solvers `saps`, `adaptg2wsat+p`, and `walksat`; our hybrid solver `MiniWalk`; and our relaxed DPLL solver `RelaxedMinisat`. We used a cluster of 3.8 GHz Intel Xeon computers running Linux 2.6.9-22.ELsmp. The time limit for the

Table 7.1: Comparison of MaxSAT results for exact, local search, and currently proposed hybrid, and relaxed DPLL methods. Timelimit: 1 hour (except for two instance which took two hours). If a sure optimum was achieved (i.e., one unsatisfied clause), the time is reported in parenthesis. The `RelaxedMinisat` column shows the final number of unsatisfied clauses reached, and in parenthesis the number of allowed conflicts ℓ, time for the DPLL part, and, if applicable, time for local search.

			Exact	Local Search		**Hybrid**		**Relaxed DPLL**	
			#unsat	best #unsat		best #uns		best #unsat	
Instance	#vars	#cls	`maxsatz` or `msuf`	`Adapt-g2wsat+p`	`SAPS`	`MiniWalk`		`RelaxedMinisat`	
babic-dspam-vc1080	118K	375K	—	728	306	20		**1**	**(1,35s,–)**
babic-dspam-vc973	274K	908K	—	2112	1412	267		**1**	**(4,100s,44s)**
ibm-2002-22r-k60	209K	851K	—	198	409	10		**1**	**(3,115m,1s)**
ibm-2002-24r3-k100	148K	550K	—	205	221	2		**1**	**(1,7m,–)**
manol-pipe-f7nidw	310K	923K	—	810	797	7		**1**	**(1,3m,–)**
manol-pipe-f9b	183K	547K	—	756	600	177		**1**	**(1,8s,–)**
manol-pipe-g10nid	218K	646K	—	585	727	27		**1**	**(1,12s,–)**
manol-pipe-g8nidw	121K	358K	—	356	336	7		**1**	**(1,6s,–)**
post-c32s-col400-16	286K	840K	—	88	111	698		**1**	**(50,1m,8s)**
post-c32s-gcdm16-23	136K	404K	—	25	225	127		**1**	**(3,100m,1m)**
post-cbmc-aes-ele	277K	1601K	—	864	781	2008		**1**	**(1,14s,–)**
simon-s03-fifo8-400	260K	708K	—	89	289	13		**1**	**(1,11m,–)**
babic-dspam-vc949	113K	360K	1 (315s)	797	216	250		**1**	**(2,10s,0s)**
cmu-bmc-barrel6	2306	8931	1 (19m)	**1 (0s)**	**1 (0s)**	**1**	**(0s)**	**1**	**(1,0s,–)**
cmu-bmc-longmult13	6565	20K	1 (171s)	5	12	1	(1s)	**1**	**(1,0s,–)**
cmu-bmc-longmult15	7807	24K	1 (137s)	6	4	1	(5s)	**1**	**(1,0s,–)**
een-pico-prop00-75	94K	324K	1 (4m)	23	108	276		**1**	**(2,2m,1s)**
goldb-heqc-alu4mul	4736	30K	1 (14m)	1 (105s)	1 (47m)	1	(1s)	**1**	**(1,0s,–)**
jarvi-eq-atree-9	892	3006	1 (158s)	**1 (0s)**	**1 (0s)**	**1**	**(0s)**	**1**	**(1,0s,–)**
marijn-philips	3641	4456	1 (336)	**1 (0s)**	**1 (0s)**	**1**	**(0s)**	**1**	**(1,0s,–)**
post-cbmc-aes-d-r1	41K	252K	1 (177s)	7	10	**1**	**(1s)**	**1**	**(1,1s,–)**
velev-engi-uns-1.0-4nd	7000	68K	1 (76s)	1 (3s)	2	**1**	**(0s)**	**1**	**(1,0s,–)**

Table 7.2: (Continued from Table 7.1.) Comparison of MaxSAT results.

Instance	#vars	#cls	Exact	Local Search		**Hybrid**		**Relaxed DPLL**	
			#unsat	best #unsat		best #uns		best #unsat	
			`maxsatz` or `msuf`	`Adapt-g2wsat+p`	`SAPS`	`MiniWalk`		`RelaxedMinisat`	
aloul-chnl11-13	286	1742	—	**4**	**4**		**4**	**4**	**(4,3s,×)**
anbul-dated-5-15-u	152K	687K	—	12	22	1	(15m)	**1**	**(1,8s,–)**
een-pico-prop05-75	77K	248K	—	2	47	**1**	**(4s)**	1	(1,2m,–)
fuhs-aprove-15	21K	74K	—	35	31	**1**	**(0s)**	**1**	**(1,0s,–)**
fuhs-aprove-16	52K	182K	—	437	246	1	(1s)	**1**	**(1,0s,–)**
goldb-heqc-dalumul	9426	60K	—	11	10	**1**	**(0s)**	1	(1,1s,–)
goldb-heqc-frg1mul	3230	21K	—	**1 (0s)**	**1 (0s)**	**1**	**(0s)**	**1**	**(1,0s,–)**
goldb-heqc-x1mul	8760	56K	—	**1 (0s)**	**1 (0s)**	**1**	**(0s)**	**1**	**(1,0s,–)**
hoons-vbmc-lucky7	8503	25K	—	**1 (0s)**	3		9	1	(7,27s,36s)
ibm-2002-25r-k10	61K	302K	—	111	95	1	(9s)	**1**	**(1,2s,–)**
ibm-2002-31_1r3-k30	44K	194K	—	78	101	**1**	**(2s)**	1	(1,6s,–)
ibm-2004-29-k25	17K	78K	—	14	12	1	(6m)	**1**	**(1,5s,–)**
manol-pipe-c10nid_i	253K	751K	—	678	695	1	(20m)	**1**	**(1,13s,–)**
manol-pipe-c10nidw	434K	1292K	—	1013	1363	**1**	**(16s)**	1	(1,37m,–)
manol-pipe-c6bidw_i	96K	284K	—	239	274	1	(24s)	**1**	**(1,3s,–)**
manol-pipe-c8nidw	269K	800K	—	697	742	**1**	**(7s)**	1	(1,6m,–)
manol-pipe-c9n_i	35K	104K	—	214	66	1	(3s)	**1**	**(1,0s,–)**
manol-pipe-g10bid_i	266K	792K	—	723	822	1	(103s)	**1**	**(1,18s,–)**
post-c32s-ss-8	54K	148K	—	1 (2s)	1 (8s)	**1**	**(0s)**	**1**	**(1,0s,–)**
post-cbmc-aes-d-r2	278K	1608K	—	834	734	1	(69s)	**1**	**(1,8s,–)**
post-cbmc-aes-ee-r2	268K	1576K	—	839	760	1	(37s)	**1**	**(1,12s,–)**
post-cbmc-aes-ee-r3	501K	2928K	—	1817	1822	1	(37m)	**1**	**(1,47s,–)**
schup-l2s-abp4-1-k31	15K	48K	—	7	16	**1**	**(0s)**	1	(1,2s,–)
schup-l2s-bc56s-1-k391	561K	1779K	—	5153	26312	**1**	**(168s)**	1	(1,11m,–)
simon-s02-f2clk-50	35K	101K	—	1 (110s)	32	**1**	**(12s)**	1	(1,17s,–)
velev-vliw-uns-2.0-iq1	25K	261K	—	1 (40m)	4	**1**	**(0s)**	**1**	**(1,0s,–)**
velev-vliw-uns-2.0-iq2	44K	542K	—	2	2	**1**	**(1s)**	**1**	**(1,1s,–)**
velev-vliw-uns-2.0-uq5	152K	2466K	—	40	11	1	(18s)	**1**	**(1,4s,–)**
velev-vliw-uns-4.0-9-i1	96K	1814K	—	12	10	1	(23s)	**1**	**(1,4s,–)**
velev-vliw-uns-4.0-9	154K	3231K	—	2	3	1	(10s)	**1**	**(1,3s,–)**

main experiments was set to 1 hour and the memory limit to 2 GB. The main findings are reported in Table 7.1 (continued in Table 7.2).

The two local search algorithms `saps` and `adaptg2wsat+p` were selected as the best performing ones on our suite from a wide pool of choices offered by the `UBCSAT` solver [Tompkins and Hoos, 2004]. Pure `Walksat` was added to contrast performance of an unguided local search with the guided version introduced in this chapter. Three runs for each problem and algorithm were performed with default parameters (or those used in the accompanying papers for the solvers, e.g., $\alpha = 1.05$ for `saps`), and the best run is reported.

The exact MaxSAT solvers selected were those that performed exceptionally well in Max-SAT Evaluation 2007, on industrial instances in particular. Nevertheless, only 10 instances in our suite were small enough to be solved by these solvers, and are reported as the bottom two sets of instances in the table. While 8 of these are still solved by `MiniWalk`, such instances are often too easy for `MiniSat` to provide more than a couple of seconds of useful guidance in the hybrid strategy. Out of the 42 remaining harder instances, `MiniWalk` found surely optimal solutions (i.e., with 1 unsatisfied clause) in as many as 20 instances, out of which 13 instances were solved by it in under a minute. Note that the previously best known MaxSAT solutions for, e.g., schup-12s-bc56s-1-k391 and post-cbme-aes-ee-r3 had over 5,000 and 1,000 unsatisfied clauses, resp.

For `RelaxedMinisat`, we first run the modified DPLL part allowing at most one conflict ($\ell = 1$), and increase this relaxation parameter when necessary. We report in Tables 7.1 and 7.2 the final number of unsatisfied clauses reached, followed by, in parentheses, the relaxation parameter, the time taken by the DPLL part, and the additional time taken to run the local search part afterwards (if applicable). In fact, in only 8 instances the DPLL part alone did not find the sure optimum

solution; for these instances we increase ℓ (not necessarily by one) to the number reported in the table so that a candidate truth assignment is found. The table reports the time for the last run of `RelaxedMinisat`, with the appropriate value of ℓ (often 1). On only two instances did `RelaxedMinisat` require more than one hour (ibm-2002-22r-k60 and post-c32s-gcdm16-23). For local search, we use the default quick restarts of `Walksat` every 100,000 flips. (However, for post-c32s-col400-16 and post-c32s-gcdm16-23 instances, this was increased to 1 million.) In all cases the noise parameter for `Walksat` was set to a low value of 5%. Tables 7.1 and 7.2 shows that `RelaxedMinisat` is the only solver able to solve 51 of out 52 instances to optimality. (For aloul-chnl11-13, we know that the optimum is not 1.) Moreover, 39 of these instances, i.e. 76%, were solved in under one minute. In contrast, the best local search approaches often could not find a solution violating less than a few hundred or thousand clauses in one hour of computation time.

These results demonstrate that both our hybrid and relaxed DPLL solvers are able to efficiently identify bottleneck constraints in challenging unsatisfiable problem instances.

7.4.1 Hybrid Search Pattern

We now explore a little deeper into the search behavior of `MiniWalk` and contrast it with the local search heuristics (`adaptg2wsat+p` and `saps` heuristics). Figure 7.1 shows a comparison of the behavior for the babic-dspam-vc973 instance from our suite, with x-axis showing the time elapsed since the solver started and the y-axis the number of unsatisfied clauses at a given time (log-scale). The instance was chosen because it highlights some of the key features of the search methods. It is solved to optimality (one unsatisfied clause) by `MiniWalk` within a few hours, although the data shown in the plot has it come down to two unsatisfied clauses.

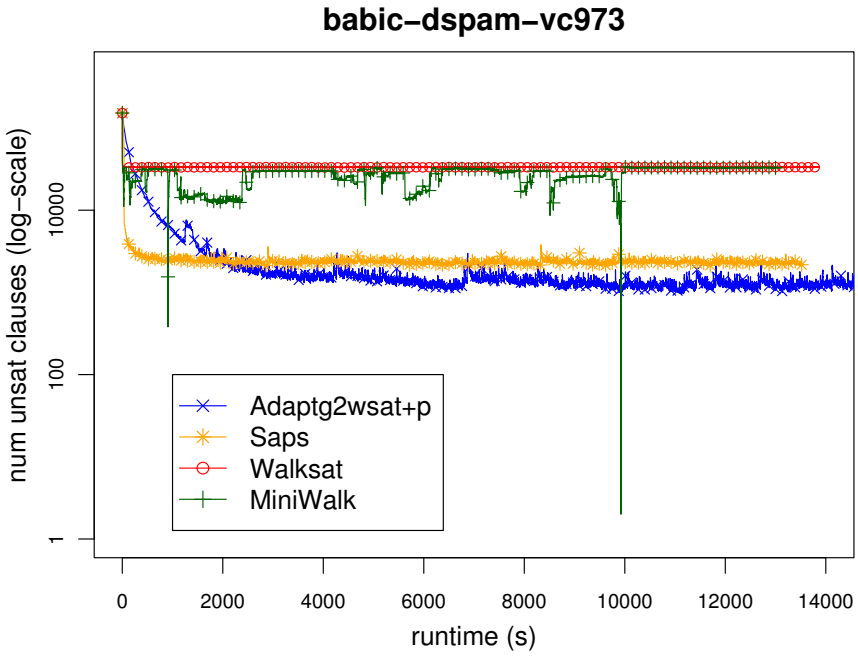


Figure 7.1: Qualitative search behavior in terms of the number of unsatisfied clauses (y-axis, log scale) as runtime progresses (x-axis). Both state-of-the-art pure local search methods, unguided `Walksat` and `MiniWalk` are shown. Note the deep drops of `MiniWalk`, which distinguish it from the other techniques. Instance: babic-dspam-vc973.cnf.

The three curves that level-off represent, in descending order, `Walksat`, `saps`, and `adaptg2wsat+p`. The remaining curve with steep drops depicts `MiniWalk`. While the local search algorithms initially descent rapidly and then stabilize at around 1,000 unsatisfied clauses with some natural noise, the hybrid method stays relatively high during the entire search (as high as the unguided `Walksat`, nearly 35,000 unsatisfied clauses), with occasional but extremely steep drops into promising regions. These regions are exactly where the best solutions are found, thanks to the DPLL guidance. While the local search often gets stuck in a plateau, the hybrid method keeps trying new promising regions as the DPLL search continues its systematic exploration. Even DPLL does not make very informed choices at the beginning of its search, but due to restarts, which are an integral part of the state-of-the-art DPLL solvers, these decisions are revised and a promising region

is found relatively quickly.

Figure 7.2 shows the comparison between local search and `MiniWalk` (the bottom-most curve, finishing early), this time for the ibm-2002-31_1r3-k30 instance. This instance is one where `MiniWalk` is clearly the best approach and very quickly discovers an assignment with only one unsatisfied clause (and thus an optimal MaxSAT solution for this unsatisfiable instance). In fact, this is the more common mode of operation of the hybrid solver: it is able to very quickly find assignments that are optimal or close to optimal.

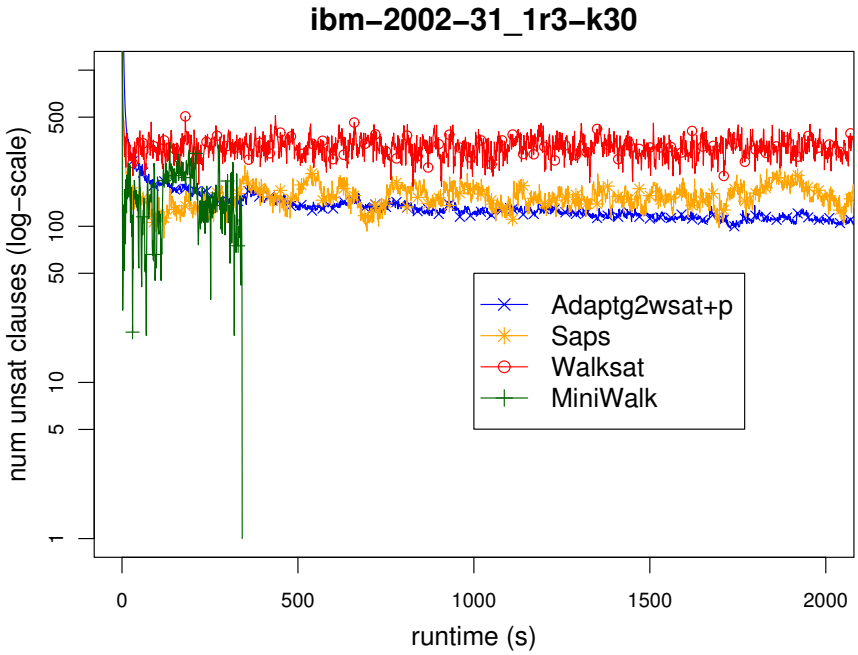


Figure 7.2: Qualitative search behavior in terms of the number of unsatisfied clauses (y-axis, log scale) vs. runtime (x-axis). Both state-of-the-art pure local search methods, unguided `Walksat` and `MiniWalk` are shown. Note the unique steep drops of `MiniWalk`. Instance: ibm-2002-31_1r3-k30.cnf.

Figure 7.3 shows a more detailed look at the internals of the hybrid solver. The y-axis shows a comparison of the depth of the DPLL search (number of choice points, scaled down by a constant factor) and the quality of the current assignment found by the solver (number of unsatisfied clauses). The x-axis is again time and

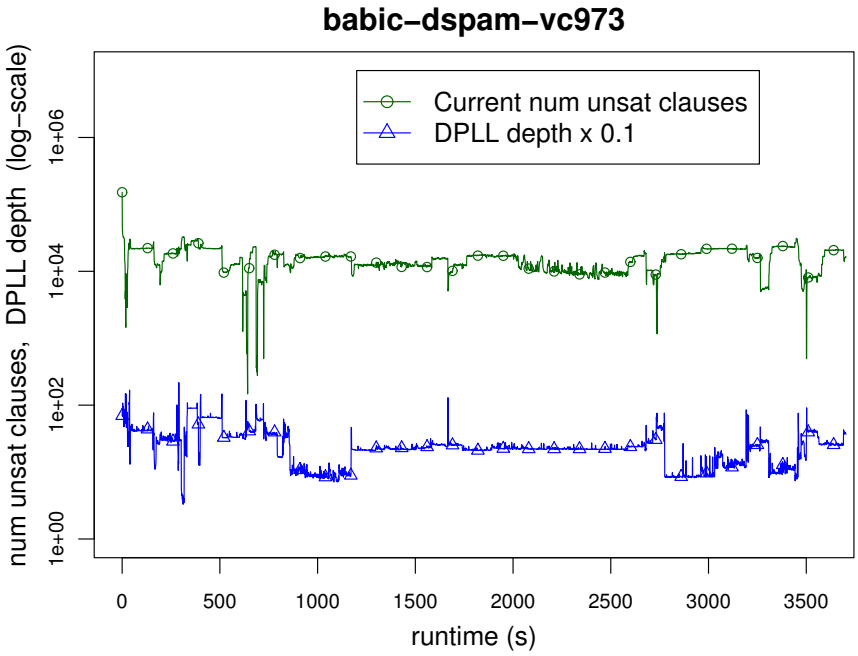


Figure 7.3: Number of unsatisfied clauses in `MiniWalk` and DPLL search depth vs. time in babic-dspam-vc973.cnf.

the instance is, as before, babic-dspam-vc973 (although the data is from a different, shorter run than in Figure 7.1). The curves show some amount of correlation between the DPLL depth and the quality of solution, suggesting that indeed when a brand new region is explored by the DPLL search, a good quality solution can be discovered.

Relative speed of the two solvers plays an important role in the process. The slower the DPLL search, the more time the local search has to explore given regions, but on the other hand, the whole search space is traversed more slowly. We experimented with this effect by artificially slowing down `MiniSat`, and running the resulting solver on the instances. There appeared to be no obvious setting that is clearly better than all others, and we used the default speed of `Minisat` in the results presented here. It is the case, however, that some instances were solved better when `MiniSat` was slowed down. Other parameters of the DPLL solver (the

restart frequency in particular) have also mixed effect on the results. We left these parameters to `MiniSat`'s default setting in our experiments.

7.5 Discussion

This chapter presents a novel approach to solving MaxSAT instances that combines strengths of both DPLL and local search SAT solvers. The proposed hybrid solver can be easily constructed from any pair of such solvers, and we found that `Minisat` and `Walksat` work exceptionally well, solving many hard problems to optimality that were not solved by any other state-of-the-art technique. The ideas presented here can be explored also for other related problems, such as weighted and partial MaxSAT. There is a clear potential of using the DPLL search to satisfy all hard constraints (or those with large weight) and leaving the "fine tuning" of the lower-weight constraints to the local search. Finally, further exploration of information flow in the other direction—from local search to DPLL—beyond what is known in the context of SAT (e.g., Mazure et al. [1998]) is left as future work.

Chapter 8

Conclusions and Future Research Directions

The main objective of my dissertation work was to explore ways to better understand and supplement state-of-the art solution techniques for various problems related to constraint satisfaction. The work contained in this dissertation contributes to the field in several ways.

It contains new observation about the survey propagation algorithm. This algorithm has stirred the CS community in 2002 by being able to quickly solve random problems that were previously thought extremely difficult. Being devised in the statistical physics community, it remained hard to understand within CS. This dissertation contains material on comparison between SP and other decimation heuristics for solving SAT problems which reveals some surprising evidence for where is its power coming from. It also includes a novel framework for reasoning about clusters, which are sets of solutions similar to each other. Clusters play a prominent role in the derivation of SP, and contained here are derivations based on two complementary approaches to reason about clusters. The ultimate goal, to devise a technique as efficient as SP for non-random problems, in particular, for problems coming from hard industrial tasks like hardware verification, remains a direction for further research exploration. The cluster-counting framework intro-

duced in this dissertation allows one to accurately reason about clusters in many structured problem domains, but devising message passing algorithms (e.g. based on belief propagation) that would allow for efficient evaluation of the required quantities is still an open problem in the case of non-random instances.

The dissertation also describes an incorporation of belief propagation into a framework for lower-bounding solution counts. This improves the efficiency of the technique, due to BP's ability to quickly estimate value biases of each variable in the set of all solutions. To do so, we introduced a damped version of the BP equations, with a quality-efficiency tradeoff parameter, which enables BP to work for a wide range of problems. While we do include a technique for upper-bounding the solution count to complement the lower bounds, the technique does not provide any (probabilistic) guarantees of accuracy. An assumption about a distribution of certain run-time statistics is necessary, which might not be satisfied. Devising a scheme to efficiently compute upper-bounds with strong (e.g. probabilistic) guarantees is an open problem, which appears to be a lot harder than computing lower-bounds.

Finally, the most recent work contained in the dissertation combines strengths of complementary solution techniques, systematic and local search, to construct a solver for maximum satisfiability problems. This hybrid approach works very well on certain classes of problems (in particular, on problems that are hard to show unsatisfiable), which shows this to be a promising research direction. However, it remains to be seen whether this approach can also improve efficiency for instances that are easy to show unsatisfiable, which are presently far more common.

In conclusion, the dissertation shows that combining techniques from various fields (e.g. logic, probability, statistics, and statistical mechanics) results in improved approaches to reasoning about constraint satisfaction problems.

BIBLIOGRAPHY

Dimitris Achlioptas, Assaf Naor, and Yuval Peres. Rigorous location of phase transitions in hard optimization problems. *Nature*, 435:759–764, 2005.

Dimitris Achlioptas and Federico Ricci-Tersenghi. On the solution-space geometry of random constraint satisfaction problems. In *38th STOC*, pages 130–139, Seattle, WA, May 2006.

Dante Alighieri and Mark Musa. *The Divine Comedy, Volume 1: Inferno*. Penguin Classics, 2002. ISBN 0142437220. Canto XXVI.

John Ardelius, Erik Aurell, and Supriya Krishnamurthy. Clustering of solutions in hard satisfiability problems. *J. Stat. Mech.*, P10012, 2007.

Josep Argelich, Chu Min Li, Felip Manyà, and Jordi Planes. Max-SAT Evaluation 2007, May 2007. URL `http://www.maxsat07.udl.es`.

Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *44th FOCS*, pages 340–351, Cambridge, MA, October 2003.

Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. Counting models using connected components. In *17th AAAI*, pages 157–162, Austin, TX, July 2000.

Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009. ISBN 978-1-58603-929-5.

A. Braunstein, M. Mezard, and R. Zecchina. Survey propagation: an algorithm for satisfiability. *Random Structures and Algorithms*, 27:201, 2005.

A. Braunstein, R. Mulet, A. Pagnani, M. Weigt, and R. Zecchina. Polynomial iterative algorithms for coloring and analyzing random graphs. *Physical Review E*, 68:036702, 2003.

Alfredo Braunstein and Riccardo Zecchina. Survey propagation as local equilibrium equations. *J. Stat. Mech.*, P06007, 2004. URL `http://lanl.arXiv.org/cond-mat/0312483`.

Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

Adnan Darwiche. Decomposable negation normal form. *J. Assoc. Comput. Mach.*, 48(4):608–647, 2001.

Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In *16th ECAI*, pages 328–332, Valencia, Spain, August 2004.

Adnan Darwiche. The quest for efficient probabilistic inference, July 2005. Invited Talk, IJCAI-05.

M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *CACM*, 5:394–397, 1962.

M. Davis and H. Putnam. A computing procedure for quantification theory. *CACM*, 7:201–215, 1960.

Niklas Eén and Niklas Sörensson. MiniSat: A SAT solver with conflict-clause minimization. In *8th SAT*, St. Andrews, U.K., June 2005.

Alfonso Gerevini, Yannis Dimopoulos, Patrik Haslum, and Alessandro Saetti (Organizers). IPC-5 international planning competition, June 2006. URL `http://zeus.ing.unibs.it/ipc-5`.

Vibhav Gogate and Rina Dechter. Approximate counting by sampling the backtrack-free search space. In *22nd AAAI*, pages 198–203, Vancouver, BC, July 2007.

Carla P. Gomes, Joerg Hoffmann, Ashish Sabharwal, and Bart Selman. From sampling to model counting. In *20th IJCAI*, pages 2293–2299, Hyderabad, India, January 2007.

Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *21st AAAI*, pages 54–61, Boston, MA, July 2006.

Éric Grégoire, Bertrand Mazure, and Cédric Piette. Boosting a complete technique to find MSS and MUS thanks to a local search oracle. In *20th IJCAI*, pages 2300–2305, Hyderabad, India, January 2007a.

Éric Grégoire, Bertrand Mazure, and Cédric Piette. MUST: Provide a finer-grained explanation of unsatisfiability. In *13th CP*, volume 4741 of *LNCS*, pages 317–331, Providence, RI, September 2007b.

MohammadTaghi Hajiaghayi and Gregory B. Sorkin. The satisability threshold of random 3-SAT is at least 3.52. Technical Report RC22942, IBM Research Report, 2003. http://arxiv.org/pdf/math.CO/0310193.

A.K. Hartmann, A. Mann, and W. Radenback. Clusters and solution landscapes for vertex-cover and SAT problems. In *Workshop on Physics of Distributed Systems*, Stockholm, Sweden, May 2008.

Edward A. Hirsch and Arist Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Annals Math. and AI*, 43(1): 91–111, 2005.

Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, San Francisco, CA, 2004.

Eric I. Hsu and Sheila A. McIlraith. Characterizing propagation methods for boolean satisfiabilty. In *9th SAT*, volume 4121 of *LNCS*, pages 325–338, Seattle, WA, August 2006.

Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *8th CP*, volume 2470 of *LNCS*, pages 233–248, Ithaca, NY, September 2002.

A. P. Kamath, N.K. Karmarkar, K. G. Ramakrishnan, and M. G. C. Resende. An interior point approach to boolean vector function synthesis. In *In Proceedings of the 36th MSCAS*, pages 185–189, 1993.

Alexis C. Kaporis, Lefteris M. Kirousis, and Efthimios G. Lalas. The probabilistic analysis of a greedy satisfiability algorithm. *Random Structures and Algorithms*, 28(4):444–480, 2006.

Henry A. Kautz and Bart Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Working notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98*, Pittsburgh, PA, 1998.

Lukas Kroc, Ashish Sabharwal, Carla P. Gomes, and Bart Selman. Integrating systematic and local search paradigms: A new strategy for MaxSAT. In *21st IJCAI*, pages 544–551, Pasadena, CA, July 2009a.

Lukas Kroc, Ashish Sabharwal, and Bart Selman. Survey propagation revisited. In *23rd UAI*, pages 217–226, Vancouver, BC, July 2007.

Lukas Kroc, Ashish Sabharwal, and Bart Selman. Counting solution clusters in graph coloring problems using belief propagation. In *22nd NIPS*, pages 873–880, Vancouver, BC, December 2008a.

Lukas Kroc, Ashish Sabharwal, and Bart Selman. Leveraging belief propagation, backtrack search, and statistics for model counting. In *5th CPAIOR*, volume 5015 of *LNCS*, pages 127–141, Paris, France, May 2008b.

Lukas Kroc, Ashish Sabharwal, and Bart Selman. Message-passing and local heuristics as decimation strategies for satisfiability. In *24th Annual ACM Symposium on Applied Computing*, pages 1408–1414, Honolulu, HI, March 2009b.

Lukas Kroc, Ashish Sabharwal, and Bart Selman. Relaxed DPLL search for MaxSAT. In *12th SAT*, volume 5584 of *LNCS*, pages 447–452, Swansea, Wales, U.K., June 2009c.

Florent Krzakala, Andrea Montanari, Federico Ricci-Tersenghi, Guilhem Semerjian, and Lenka Zdeborova. Gibbs states and the set of solutions of random constraint satisfaction problems. *PNAS*, 104(25):10318–10323, June 2007.

F. R. Kschischang, B. J. Frey, and H. A. Loeliger. Factor graphs and the sum-product algorithm. *Information Theory, IEEE Transactions on*, 47(2):498–519, 2001.

Daniel Le Berre, Olivier Roussel, and Laurent Simon (Organizers). SAT 2007 competition, May 2007. URL `http://www.satcompetition.org/2007`.

Chu Min Li, Felip Manyà, and Jordi Planes. New inference rules for Max-SAT. *JAIR*, 30:321–359, 2007a.

Chu Min Li, Wanxia Wei, and Harry Zhang. Combining adaptive noise and look-ahead in local search for SAT. In *10th SAT*, volume 4501 of *LNCS*, pages 121–133, Lisbon, Portugal, May 2007b.

Michael L. Littman, Stephen M. Majercik, and Toniann Pitassi. Stochastic Boolean satisfiability. *J. Auto. Reas.*, 27(3):251–296, 2001.

D. Mackay, J. Yedidia, W. Freeman, and Y. Weiss. A conversation about the bethe free energy and sum-product, 2001. URL `citeseer.ist.psu.edu/mackay01conversation.html`.

Elitza Maneva, Elchanan Mossel, and Martin J. Wainwright. A new look at survey propagation and its generalizations. In *16th SODA*, pages 1089–1098, Vancouver, BC, January 2005.

Elitza Nikolaeva Maneva. *Belief propagation algorithms for constraint satisfaction problems*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2006. Adviser-Sinclair, Alistair.

João P. Marques-Silva and Vasco M. Manquinho. Towards more effective unsatisfiability-based maximum satisfiability algorithms. In *11th SAT*, volume 4996 of *LNCS*, pages 225–230, Guangzhou, China, May 2008.

Bertrand Mazure, Lakhdar Sais, and Éric Grégoire. Boosting complete techniques thanks to local search methods. *Annals Math. and AI*, 22(3-4):319–331, 1998.

Stephan Mertens, Marc Mézard, and Riccardo Zecchina. Threshold values of random K-SAT from the cavity method. *Random Struct. and Alg.*, 28(3):340–373, 2006.

Marc Mezard and Andrea Montanari. *Information, Physics, and Computation*. Oxford University Press, USA, 2009. ISBN 9780198570837.

Marc Mézard, Giorgio Parisi, and Riccardo Zecchina. Analytic and algorithmic solution of random satisfiability problems. *Science*, 297(5582):812–815, 2002.

Joris M. Mooij, Bastian Wemmenhove, Hilbert J. Kappen, and Tommaso Rizzo. Loop corrected belief propagation. In *11th AISTATS*, San Juan, Puerto Rico, March 2007.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *38th DAC*, pages 530–535, Las Vegas, NV, June 2001.

Elchanan Mossel. Personal communication, April 2007.

Kevin Murphy, Yair Weiss, and Michael Jordan. Loopy belief propagation for approximate inference: An empirical study. In *15th UAI*, pages 467–475, Sweden, July 1999.

J. Mycielski. Sur le coloriage des graphs. *Colloquium Mathematicum*, 3:161–162, 1955.

Richard E. Neapolitan. *Learning Bayesian Networks*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004. ISBN 0130125342.

James D. Park. MAP complexity results and approximation methods. In *18th UAI*, pages 388–396, Edmonton, Canada, August 2002.

Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

Knot Pipatsrisawat and Adnan Darwiche. RSat 1.03: SAT solver description. Technical Report D–152, Automated Reasoning Group, Computer Science Department, UCLA, 2006.

S. D. Prestwich. Variable dependency in local search: Prevention is better than cure. In *10th SAT*, volume 4501 of *LNCS*, pages 107–120, Lisbon, Portugal, May 2007.

Marco Pretti. A message-passing algorithm with damping. *J. Stat. Mech.*, P11008, 2005.

R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2005. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

Dan Roth. On the hardness of approximate reasoning. *AI J.*, 82(1-2):273–302, 1996.

Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *7th SAT*, Vancouver, BC, May 2004.

Tian Sang, Paul Beame, and Henry A. Kautz. Performing Bayesian inference by weighted model counting. In *20th AAAI*, pages 475–482, Pittsburgh, PA, July 2005.

Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring and Satisfiability: the Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in DMTCS*, pages 521–532. Amer. Math. Soc., 1996.

Carsten Sinz (Organizer). SAT-race 2008, May 2008. URL `http://www-sr.informatik.uni-tuebingen.de/sat-race-2008`.

Henry C. Thode. *Testing for Normality*. CRC, 1st edition, 2002.

Dave A. D. Tompkins and Holger H. Hoos. Scaling and probabilistic smoothing: Dynamic local search for unweighted MAX-SAT. In *16th Canadian Conf. on AI*, volume 2671 of *LNCS*, pages 145–159, Halifax, Canada, June 2003.

Dave A. D. Tompkins and Holger H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In *7th SAT*, Vancouver, BC, May 2004. Solver description.

Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Comput. Sci.*, 8:189–201, 1979.

Wei Wei, Jordan Erenrich, and Bart Selman. Towards efficient sampling: Exploiting random walk strategies. In *19th AAAI*, pages 670–676, San Jose, CA, July 2004.

Wei Wei and Bart Selman. A new approach to model counting. In *8th SAT*, volume 3569 of *LNCS*, pages 324–339, St. Andrews, U.K., June 2005.

Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Generalized belief propagation. In *14th NIPS*, pages 689–695, Denver, CO, November 2000.

Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Understanding belief propagation and its generalizations. In *Exploring artificial intelligence in the new millennium*, pages 239–269, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc. ISBN 1-55860-811-7.

Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Constructing free-energy approximations and generalized belief propagation algorithms. *IEEE Trans. Inf. Theory*, 51(7):2282–2312, 2005.

Alan L. Yuille. CCCP algorithms to minimize the Bethe and Kikuchi free energies:

Convergent alternatives to belief propagation. *Neural Comput.*, 14(7):1691–1722, 2002.

Xiao-Hua Zhou and Gao Sujuan. Confidence intervals for the log-normal mean. *Statistics In Medicine*, 16:783–790, 1997.