Two problems in the analysis and design of algorithms concerning the satisfiability of propositional formulas are investigated. With respect to the first problem, there is great experimental evidence for a phenomenon known as the ‘phase transition’ of satisfiability, that is there is a sharp threshold, in the limit, between satisfiable and unsatisfiable formulas as the ratio of clauses to variables increases. To analyze this threshold, we count the number of satisfiable boolean formulas given in conjunctive normal form. The intention is to provide information about the relative frequency of boolean functions with respect to statements of a given size. This in turn will provide information about algorithms attempting to decide problems such as satisfiability and validity. First, we describe a correspondence between the syntax of propositions to the semantics of functions using a system of equations. Then, we show how to solve such a system. This gives a general solution for any set of functions acting as literals; we simplify the specific solution for the variables and their negations as literals. And finally we extract some asymptotics and threshold bounds from this solution. The second problem is the generation of distinct models of propositional formulas, when the variables can be permuted to form the same boolean function. A model is a complete assignment to the variables, but the permutations can render some of these assignments as equivalent. Generation methods are developed for specific permutation groups.
SYMMETERY AND THRESHOLDS IN PROPOSITIONAL FORMULAS

BY
MITCHELL A. HARRIS
A.B., Cornell University, 1986
M.S., University of Illinois at Urbana-Champaign, 1998.

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois
© Copyright by Mitchell A. Harris, 2002
ABSTRACT

Two problems in the analysis and design of algorithms concerning the satisfiability of propositional formulas are investigated. With respect to the first problem, there is great experimental evidence for a phenomenon known as the ‘phase transition’ of satisfiability, that is there is a sharp threshold, in the limit, between satisfiable and unsatisfiable formulas as the ratio of clauses to variables increases. To analyze this threshold, we count the number of satisfiable boolean formulas given in conjunctive normal form. The intention is to provide information about the relative frequency of boolean functions with respect to statements of a given size. This in turn will provide information about algorithms attempting to decide problems such as satisfiability and validity. First, we describe a correspondence between the syntax of propositions to the semantics of functions using a system of equations. Then, we show how to solve such a system. This gives a general solution for any set of functions acting as literals; we simplify the specific solution for the variables and their negations as literals. And finally we extract some asymptotics and threshold bounds from this solution. The second problem is the generation of distinct models of propositional formulas, when the variables can be permuted to form the same boolean function. A model is a complete assignment to the variables, but the permutations can render some of these assignments as equivalent. Generation methods are developed for specific permutation groups.
To my parents
ACKNOWLEDGMENTS

I acknowledge the great debt I owe to all the people who would be hurt if I didn’t mention them:

Profs. Nachum Dershowitz and Ed Reingold, who might both be upset being mentioned in the same breath, but more upset if not at all.

Profs. Jeff Erickson, Lenny Pitt, and Doug West for their use of props in an instructional environment.

Prof. Mehdi Harandi for hiring or not driving off all those people who actually did all the real work in helping me.

Angie Bingaman, Barb Cicone, Molly Flesner, Katie Herrera, Julie Legg, and Kay Tomlin, the ones who did all the real work helping me.

My family, Shohreh, Nolan, and Ian, from all of whom I have learned how to withstand being kicked in the stomach. OK, Shohreh never did that.

And lastly my friends, Tanya Berger-Wolf, John Jozwiak, Shripad Thite, and Afra Zomorodian without whose never failing sympathy and encouragement this would have all been finished in half the time.

Some obscure quote to show my erudition, eruditeness, erudity,... smarts.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Counting and Thresholds in Propositional Logic</td>
<td>6</td>
</tr>
<tr>
<td>3 Symmetry in Model Checking</td>
<td>40</td>
</tr>
<tr>
<td>4 Conclusions and Open Problems</td>
<td>55</td>
</tr>
<tr>
<td>References</td>
<td>57</td>
</tr>
<tr>
<td>VITA</td>
<td>60</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table

Page
LIST OF FIGURES

Figure  
Page
CHAPTER 1

Introduction

The satisfiability problem is one which crosses many field. It is important theoretically as the canonical problem defining computational complexity classes. It is important practically as the model for verification in circuit design. Other problems can be reduced to it, for both theory (many theoretical problems can be reduced to it and from it) and practice (likewise many practical problems are attacked using satisfiability solvers. The satisfiability problem is the problem of deciding whether a propositional formula, consisting of the logical operators “and”, “or”, and “not”, and boolean variables, if that formula has at least one assignment of true and false to the variables so that the entire formula evaluates to true.

It is because of this simple logical formalism that it is applicable in so many different ways. Unfortunately, it is not always easy to solve instances of the problem. All currently known algorithms run in worst case exponential time in the size of the formula. Thirty years of research have only given strong corroborative evidence that this is also a lower bound, that one can do no better. But the problem’s great universality means that we cannot just ignore the problem; instances still need to be solved. So research in algorithms spend much effort in devising better and better heuristics and methods tailored to specific applications.
1.1 Thresholds

The first and still quite popular algorithm for deciding satisfiability is the Davis-Putnam algorithm [DP60, DLL62], which accepts as input a propositional formula in conjunctive normal form (CNF), a conjunction of a sequence of clauses each of which is a disjunction of variables and negated variables (literals). It is essentially a backtracking algorithm which, for each variable in order, tries one value, then, if that fails, another. If both branches fail, the algorithm backs up to the previous variable and tries another branch. If ever the setting of a variable makes the value of the whole formula true, then the algorithm returns with success. The exponential behavior in this is ameliorated somewhat by some heuristics, such as removing unit clauses, or if a variable occurs always either positive or always negative, setting it appropriately. This reduces the base of the exponent considerably by helping to eliminate branching. Further heuristics (see Hirsch [Hir00]) have reduced the running time to \( \Theta(2^{n/p}) \). But it is still exponential.

Experimental evidence has shown that for random instances of input, the Davis-Putnam algorithm is actually quite fast. If the formula has a large number of variables in relation to the total size of the formula, it is very likely that the formula is satisfiable, and Davis-Putnam finds some valuation quickly. At the other end, if the formula has a large number of clauses, corresponding to constraints, the it is not very likely to be satisfiable, and Davis-Putnam does not have to search very deep to show that there are no satisfying valuations. In between these two, it only seems that Davis-Putnam takes a long time on a very small range, a peak in the middle. See Figure 1.1 for an illustration of this phenomenon.

At the same peak in running time, the same experiments also show that it there is a quick jump from satisfiable formulas to unsatisfiable ones, as the number of clauses increases with respect to the variables. This is known as the threshold phenomenon.
**Figure 1.1** The threshold of satisfiability and the time of Davis-Putnam. Experiments performed with \( v = 40 \) and \( k = 3 \). 100 instances were tested for each \( r = c/v \) from 3.5 to 6.
1.2 Symmetries

One of the most important uses of deciding satisfiability is for verifying circuit design or software. Whether designed with the aid of some algorithm, compiler, or purely by hand, these problems usually exhibit a high degree of reuse of components, either with slight modification or not. This results in a high degree of symmetry among the variables in whatever propositional formula that models the problem. What this symmetry allows us is to ignore verification of components that have already been verified. It turns out that both theoretically and practically, finding all the symmetry possible in an arbitrary formula is quite difficult, but in practice, the symmetry is already known (by the design) or is so elementary (pairwise swapping of variables) that this is no problem.

If one knows the symmetries ahead of time, one can immediately restrict oneself to a smaller set in the search space. And if the there is enough symmetry, this reduction can be quite large. The most extreme example is when all variables can be permuted with each other; this results in a reduction of the search space from $2^n$ to $n$.

1.3 Overview

In chapter 2, we will discuss methods for analyzing the threshold phenomenon. Rather than analyzing particular algorithms, like Davis-Putnam, we will look only at the distribution of formulas themselves. Our strategy will be to bypass any probabilistic analysis, and to directly count the formulas that are not satisfiable, thereby yielding the number that are satisfiable. Through combinatorics and linear algebra, the method is general enough to apply to different restrictions on the choice of random formulas. In chapter 3, we develop some algorithms for model generation with respect to permutation groups.
as a heuristic to aid in reducing the search space. And finally in chapter 4, we conclude and mention some problems for further investigation.
CHAPTER 2

Counting and Thresholds in Propositional Logic

2.1 Introduction

There is great experimental evidence for a phenomenon known as the ‘phase transition’ of satisfiability, that is there is a sharp threshold, in the limit, between satisfiable and unsatisfiable formulas as the ratio of clauses to variables increases. Though there had been evidence beforehand, the paper of Kilpatrick and Selman [KS94] popularized the issue and solidified the main concepts. Much work has been done to establish that this phase transition is indeed a sharp threshold (see Friedgut [Fri99]) and to place bounds on that threshold (see Janson et al. [JSV00]). The results of this section could be used to determine the exact threshold analytically.

A very different problem is computing the number of satisfying assignments for a particular formula. This more closely corresponds to the concept of \#P-completeness (see Valiant [Val79], Papadimitriou [Pap94]). Frieze and Suen [FS96], Kamath et al. [KMPS95], and Dubois [Dub91] address $k$-CNF formulas; Creignou and Daude [CD99] consider the XOR-CNF problem (where instead of clauses connected by ‘or’, they are connected by ‘exclusive-or’). These and others all attack the problem algorithmically, that is, knowing that the problem is \#P-complete, they attempt to find efficient heuristics. The input to these algorithms is a single formula in a particular grammar, and
the algorithms attempt to count the number of models of the formula, the number of assignments that make the formula true. The problem addressed here is, for a particular restriction on the formula syntax, how many of all possible formulas in that form have at least one satisfying assignment.

There is also much work on the performance of satisfiability algorithms, for example, Davis-Putnam, with respect to this threshold. It seems that around the same ratio of variables to clauses (4.3 for 3-CNF) where the formulas show a threshold of satisfiability, there is a peak in the average time taken by Davis-Putnam usually in terms of number of backtrack steps in searching for a satisfying assignment. This behavior is also discussed originally in Kilpatrick and Selman [KS94].

2.2 Definitions and Background on Thresholds

**Definition 1.** A literal is a variable or its negation.

A clause is a disjunction of a collection of literals.

A CNF formula is a conjunction of clauses.

A k-CNF formula is a conjunction of clauses, where each clause has exactly k literals.

**Definition 2.** For v variables and c clauses, the number of formula that are equivalent to a particular boolean function $f$ is called $\#\text{FUNC}(f, v, k, c)$.

The total number of k-CNF propositional formula is called $\#\text{CNF - ALL}(v, k, c)$.

The number that are satisfiable is called $\#\text{CNF - SAT}(v, k, c)$.

By these definitions it is easy to see that

$$\#\text{CNF - ALL}(v, k, c) = \sum_{f \in \mathcal{B}(v)} \#\text{FUNC}(f, v, k, c)$$

and

$$\#\text{CNF - SAT}(v, k, c) = \#\text{CNF - ALL}(v, k, c) - \#\text{FUNC}(F, v, k, c).$$
There has been much theoretical work on attempting to establishing the exact numeric threshold between satisfiability and unsatisfiability as the ratio of $v$ to $c$ increases (asymptotically for $v$). More formally, the investigation is on the characteristics of the function

$$Pr_{\text{CNF-SAT}}(v,k,r) = \frac{\#\text{CNF-SAT}(v,k,\nu r)}{\#\text{CNF-ALL}(v,k,\nu r)},$$

and

$$Pr_{\text{CNF-SAT}}(k,r) = \lim_{v \to \infty} Pr_{\text{CNF-SAT}}(v,k,r)$$

denoting whether this is a 0-1 or threshold function, and if so what is the value of $r$ for this threshold. Experimentally, by creating large numbers of random CNF formulas and testing them for satisfiability, the threshold has been determined to be approximately between 4.25 and 4.3 For example, see Figure 2.1 for an experimental approximation to $Pr_{\text{CNF-SAT}}(20,3,r)$.

### 2.2.1 Background

Formally, a function $f(x)$ has a threshold at $\gamma$ if $f(x) = 1$ for all $x \leq \gamma - \epsilon$ and $f(x) = 0$ for all $x \geq \gamma + \epsilon$, for any $\epsilon > 0$. The threshold problem is to determine if $Pr_{\text{CNF-SAT}}(k,r)$ is a threshold function for $r$ (for a constant $k$) and if so to find $\gamma$.

The latest work on establishing that this phase transition of $Pr_{\text{CNF-SAT}}(k,r)$ is a threshold is Friedgut [Fri99]. The result there does not definitively establish the fact; it says at best that there is a function $\gamma(v)$, such that the transition happens within an $\epsilon$ neighborhood of $\gamma(v)$, but this function may not converge. The sequence $\gamma(v)$ is widely believed to converge (see Achlioptas [Ach01]) but there has been no further refinement. Friedgut’s method is based on a translation of the threshold properties of another NP-complete problem, $k$-coloring in graphs, that he establishes using the Alon and Spencer [AS92] non-constructive probabilistic method.
Figure 2.1 Ratio of satisfiable formulas to total computed from a sampling of 80 random 3-CNF formulas over 20 variables. The domain is $r$, the ratio of variables to clauses.
Experimentally, there is a perceived threshold for any $k$. For $k = 2$, it has been proven there is a sharp threshold and the threshold value has been computed exactly. Chvátal and Reed [CR92], Goerdt [Goe92, Goe96], Fernandez de la Vega [FdlV01] independently found through different means that the threshold is sharp and equals 1 for $k = 2$. Because the associated decision problem 2-CNFSAT is NL-complete (see Cook [Coo71], Even, Itai, and Shamir [EIS76]), there is a simple (polytime describable) combinatorial characterization of the 2-CNF formulas that are satisfiable. The algorithm for 2-CNFSAT looks for cycles in an implication graph. Because this characterization can be captured so easily, Goerdt was able to exploit this to count exactly those graphs that avoid these paths; Chvátal and Reed [CR92] used non-constructive probabilistic techniques (the second moment method). Unfortunately for both of these methods, they do not seem to apply easily to $k = 3$. The 3-CNFSAT problem is NP-complete, and consistent with all the corroborative evidence against polynomial algorithms for deciding such problems, there is no currently known characterization of satisfiable 3-CNF formulas that is polytime computable. However, if P does not equal NP, this does not imply that counting the satisfiable 3-CNF formulas is also a difficult problem. Also, this problem is not the same as the counting complexity classes like $\#P$ which the NP problems reduce to (where if you know the exact number of solutions, you certainly know if there is one or more).

As to the threshold value itself (assuming that it is a formal threshold), there have been a number of various attacks, all attempts to bound the threshold better. To this date, the best upper bound is 4.506 by Dubois et al. [DBM00]. On the other side, the latest lower bound is 3.145 found by Achlioptas [Ach00]. Both of these researchers, and many others, have contributed to the yearly progress on both upper and lower bounds (see Janson et al. [JSV00], Franco [Fra01], and Achlioptas [Ach01] for summaries of this progress). All of the methods used, for both upper and lower bounds, were either
probabilistic or made some appeal to the action of a heuristic in a backtracking algorithm like Davis-Putnam.

The intent of the present work is to do exact counting and extract probabilities and thresholds afterwards. Given the logical operators $\land$ and $\lor$ and $v$ boolean variables and their negations (the $2v$ literals, we can construct any of the $2^v$ boolean functions over those variables. In fact, because of the associative, commutative, and other properties of the operators, there are often quite a few ways of encoding any particular function. If we could describe in closed form how many ways a boolean function could be created through the syntax of propositional formulas, we would be able to extract information more easily about the behavior in the limit.

### 2.3 Models of counting

The counting problem we address corresponds to the logical problem “$k$-CNF-SAT”. We have a set $V$ of $v$ independent propositional variables, a set $\overline{V}$ of $v$ of their negations. The $2v$ variables and negations are the literals. In $k$-CNF, a clause is a disjunction of $k$ literals, and a formula is a conjunction of $c$ clauses. Most of the literature considers a clause as a non contradictory set, i.e. order does not matter, there are no repeats of literals, and a literal and its negation may not appear together (this would make the clause trivially satisfiable). However, a different model of selection is used for the conjunction of clauses; here the formula is a sequence of clauses selected with replacement. This implies that the same clause may appear in a sequence and that the order of the literals or clauses matter.

There is more than one way to consider a formula “different” from another. The “and” and “or” operators are both associative, commutative, and idempotent. Is $p \lor q$ different syntactically from $q \lor p$? Is $p \lor p \lor q$ syntactically different from $p \lor q$? Consider
a single operator, say in a single clause. Since the operator is associative-commutative, a permutation of the sequence of literals trivially gives the same function because of the other two properties. Likewise, since the operator is idempotent, a repeated literal can be ignored, trivially giving the same function.

In a sequence, the order of the symbols is significant, but not in a set. In selection with replacement, repeats are possible, but not in selection without replacement. With these properties of selection, ordered, unordered, with and without replacement, plus the generally accepted literal selection scheme, there are 20 different models, 5 for selecting the literals in a clause, 4 for the clauses in a formula. To address this choice, we make an analysis of all possibilities.

The choice of model is important for the analysis of the satisfiability threshold. First, there is the precedent that most previous analysis considered only the model of selecting literals in a clause as a set without replacement, but then selecting clauses as a sequence with replacement. The reasons for this precedent are that it is easier in practice to generate a sequence with replacement when the number of clauses increases, but that a set of literals without replacement (and without contradiction) is easy to generate uniformly when $k$ is a constant. And second, a set without replacement is a closer model to CNF formulas in practice; it is hard to imagine a circuit design sloppy enough to create clauses where literals repeat. However, each of these considerations could equally well apply to either literals or clauses. So, we analyze all the possibilities.

### 2.4 Exactly one literal per clause

When there is only one literal per clause, the formula is equivalent to a single clause in DNF form, all the literals are in a single conjunction. The only way to make such a formula unsatisfiable is if there is at least one pair of To prevent any positive/negative
pairs, any literal must consistently appear as either all positive or all negative. This makes counting in some of the models very straightforward. Often it is just a matter of choosing variables with the appropriate elementary combinatorics.

2.4.1 Unordered without replacement

The total number of formulas with \( v \) variables and \( c \) clauses (really \( c \) literals) is

\[
\#\text{CNF} - \text{ALL}(v, 1, c)_{\text{unordered, w/o}} = \binom{2v}{c},
\]

(2.1)

the number of subsets of size \( c \) from the set of literals \( 2v \). A combination has no repeats and no order. Notice that \( c \) is no larger than \( 2v \).

The number that are satisfiable is

\[
\#\text{CNF} - \text{SAT}(v, 1, c)_{\text{unordered, w/o}} = \binom{v}{c} 2^v,
\]

(2.2)

out of \( v \) variables choosing \( c \) of them, then choosing for each variable if it a positive or negative literal. Notice that \( c \) is no larger than \( v \).

2.4.2 Ordered without replacement

This situation is almost identical to the “unordered without replacement” model. Since there are no repeats of variables possible, we only need to permute those positions in the unordered case to distinguish formulas by order. So,

\[
\#\text{CNF} - \text{ALL}(v, 1, c)_{\text{ordered, w/o}} = \binom{2v}{c} c!,
\]

(2.3)

and
\[ \#\text{CNF} - \text{SAT} (v, 1, c)_{\text{ord}, w/o} = \binom{v}{c} 2^n c! \]  \hspace{1cm} (2.4)

2.4.3 Unordered with replacement

An unordered selection of \( c \) items from the \( 2v \) literals, with replacement, is

\[ \#\text{CNF} - \text{ALL} (v, 1, c)_{\text{unord}, w} = \binom{2v - 1 + c}{c}. \]  \hspace{1cm} (2.5)

Since they are unordered, consider the \( c \) literals as sorted and then separated into \( 2v - 1 \) blocks. Notice that, with replacement, \( c \) and \( v \) are independent.

The number that are satisfiable is

\[ \#\text{CNF} - \text{SAT} (v, 1, c)_{\text{unord}, w} = \sum_{t=0}^{v} \binom{v}{t} 2^t \binom{c - 1}{t - 1} \]  \hspace{1cm} (2.6)

Choose \( t \) out of the \( v \) variables, then choose whether each of those literals is positive or negative. Then, to choose \( t \) blocks out of \( c \) items is \( \binom{t - 1 + c}{t - 1} \), but since we must have at least 1 item in each of the blocks, there are \( \binom{c - 1}{t - 1} \) ways.

2.4.4 Ordered with replacement

All formulas, ordered with replacement, are simply the number of functions from \( c \) to the set of literals \( 2v \):

\[ \#\text{CNF} - \text{ALL} (v, 1, c)_{\text{ord}, w} = (2v)^c. \]  \hspace{1cm} (2.7)

On the other hand, the satisfiable formulas are more complicated.

Theorem 2.4.1.

\[ \#\text{CNF} - \text{SAT} (v, 1, c)_{\text{ord}, w} = \sum_{t=0}^{v} \binom{v}{t} 2^t \binom{c}{t} t! \]  \hspace{1cm} (2.8)

\[ = \sum_{t=0}^{v} \binom{v}{t} 2^t t^c (-1)^{v-t} \]  \hspace{1cm} (2.9)
where \( \binom{c}{t} \) is the Stirling number of the second kind or the number of set partitions of \( c \) with exactly \( t \) parts.

**Proof:** First, a straightforward combinatorial proof. In the first equality, each factor can be interpreted directly is proven combinatorially as follows. Let there be \( t \) variables appearing in the formula. First choose a subset of \( t \) of the \( v \) variables, for \( \binom{v}{t} \) ways. Then each of these variables is either positive or negative, for \( 2^t \) ways. Then, as a sequence, the correspondence between the linear position in the sequence and the literal that goes there is a surjective function from the literals to the positions; each position must be assigned a literal), and there are \( \binom{c}{t} t! \) ways to do that since a set partition of \( c \) with \( t \) parts separates the \( c \) positions into \( t \) equivalence classes, and there are \( t! \) ways to assign the literals to the classes in all possible ways. Since, for each \( t \), these descriptions are mutually exclusive and cover all possibilities, the summation counts all the possible forms.

In the second equality, the first two factors are explained similarly. The third factor is the number of unrestricted functions functions from the positions to the literals, for \( t^c \) ways. The last factor takes into account inclusion/exclusion to remove those assignments that are not onto functions. For \( t = v \), there are \( v^c \) functions; but we have over counted those that do not include the whole range, so we subtract those that include one less of the domain, and so on. \( \square \)

A technical algebraic proof involves the generating function for \( \# \text{CNF-SAT} (v, 1, c)_{\text{ord}, w}. \)

**Theorem 2.4.2.**

\[
\# \text{CNF-SAT} (v, 1, c)_{\text{ord}, w} = c! \left[ z^c \right] (2e^z - 1)^v
\]  

(2.10)
Proof: The second equality 2.9 is short:

\[ c! \left[ z^c \right] (2e^z - 1)^v = c! \left[ z^c \right] \sum_{t=0}^{v} \binom{v}{t} (-1)^{v-t} (2e^z)^t \]

\[ = c! \left[ z^c \right] \sum_{t=0}^{v} \binom{v}{t} 2^t (-1)^{v-t} e^{zt} \]

\[ = c! \left[ z^c \right] \sum_{t=0}^{v} \binom{v}{t} 2^t (-1)^{v-t} \sum_{k=0}^{\infty} \frac{z^k}{k!} \]

\[ = c! \left[ z^c \right] \sum_{k=0}^{\infty} \frac{z^k}{k!} \sum_{t=0}^{v} \binom{v}{t} 2^t \frac{z^k}{k!} (-1)^{v-t} \]

\[ = \sum_{t=0}^{v} \binom{v}{t} 2^t \frac{z^t}{t!} (-1)^{v-t} \]

\[ \square \]

A purely algebraic manipulation proves the equality of (2.8) and (2.9). The identity

\[ \begin{bmatrix} n \\ m \end{bmatrix} = \sum_{k=0}^{m} \binom{m}{k} (-1)^{m-k} k^n \frac{1}{m!} \]

comes in useful here.

Theorem 2.4.3.

\[ \sum_{t=0}^{v} \binom{v}{t} 2^t \left\{ \binom{c}{t} \right\} t! = \sum_{t=0}^{v} \binom{v}{t} 2^t c^t (-1)^{v-t} \]
Proof:

\[
\sum_{t=0}^{v} \binom{v}{t} 2^t \binom{c}{t} t! = \sum_{t=0}^{v} \binom{v}{t} 2^t t! \sum_{k=0}^{t} \binom{t}{k} (-1)^{t-k} k^c \frac{1}{t!} \quad \text{identity 2.11}
\]

\[
= \sum_{t=0}^{v} \sum_{k=0}^{t} \binom{v}{t} 2^t \binom{t}{k} (-1)^{t-k} k^c
\]

\[
= \sum_{t=0}^{v} \sum_{k=0}^{v-t} \binom{v}{k+t} 2^{k+t} \binom{k+t}{t} (-1)^k t^c \quad \text{swap vars and shift}
\]

\[
= \sum_{t=0}^{v} 2^t t^c \sum_{k=0}^{v-t} \binom{v}{k+t} \binom{k+t}{t} (-2)^k
\]

\[
= \sum_{t=0}^{v} 2^t t^c \sum_{k=0}^{v-t} \binom{v}{v-k-t} \binom{v-k-t}{v-t} (-2)^k
\]

\[
= \sum_{t=0}^{v} \binom{v}{t} 2^t t^c (-1)^{v-t} \quad (-1)^{v-t} = (1 - 2)^{v-t}
\]

\[
\square
\]

There is little hope for further simplification of the summations in equations 2.6 or 2.9 because they are not solutions to linear recurrence relations with polynomial coefficients (as is proved by Gosper’s algorithm for computing such recurrences when they exist). Otherwise, the convenient machinery of the WZ-method and hypergeometric summations for solving and simplifying binomial summations would apply.

2.4.5 Summary of results

A summary of these results for \(k = 1\) can be found in table 2.4.5.
### Table 2.1
The number of functions, number of satisfiable functions, and corresponding generating functions 1-CNF formulas under the 4 different models.

#### 2.5 Exactly one variable

When \( v = 1 \), and therefore only 4 boolean functions \( T, P, \overline{P}, \) and \( F \), many of the models degenerate nicely, when the clauses are ordered with replacement. For the “without replacement” or non-contradictory models, \( k \) is either 1 or 2. If \( k = 1 \) then a formula is satisfiable only if all the literals are the same, either all positive or all negative, and there are only two ways to do that. If \( k = 2 \), every clause is “\( p \lor \neg p \)” (or the reverse if ordered), so all \( 2^c \) are satisfiable. Unordered with replacement (\( k \) is unlimited), a formula is satisfiable if one clause is all \( p \) but no other all \( \neg p \) (and the reverse). There are \( k + 1 \) different clauses, so there are \( 2^{k_c} \) satisfiable ones.

The only non-trivial model is ordered with replacement. Because of the restricted syntax, the set of \( k \)-CNF formulas is straightforward to specify as a regular language: \(((V + \overline{V})^k)\). From this one can quickly derive the number of satisfiable formulas for degenerate and small values of \( k, c, \) and \( v \), as shown in Table 2.2.

---

1The rest uses the notation of lower case plain type for a formulas, and upper case bold type for the corresponding boolean function, notation uppercase bold for the boolean function, e.g. “\( P \)”.

The smallest formula will be used for the label of a boolean function.
<table>
<thead>
<tr>
<th>clause size, $k$</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>$k$</th>
<th>$k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>clauses, $c$</td>
<td>1</td>
<td>$c$</td>
<td>1</td>
<td>$c$</td>
<td>1</td>
</tr>
<tr>
<td>vars, $v$</td>
<td>1</td>
<td>1</td>
<td>$v$</td>
<td>$v$</td>
<td>$v$</td>
</tr>
<tr>
<td>total # formulas</td>
<td>$2^k$</td>
<td>$2^c$</td>
<td>$2v$</td>
<td>$(2v)^c$</td>
<td>$(2v)^k$</td>
</tr>
<tr>
<td># satisfiable</td>
<td>$2^k$</td>
<td>2</td>
<td>$2v$</td>
<td>$\sum_{i=1}^{v} \left(\begin{array}{c}v \ t\end{array}\right) 2^j {c\choose j} j!$</td>
<td>$2(2^k - 1)^c - (2^k - 2)^c$</td>
</tr>
</tbody>
</table>

Table 2.2 Counting satisfiable formulas for small values of $v$, $k$, and $c$

There are two nontrivial entries. For $k = 1$, to be satisfiable, a literal and its negation cannot appear. So, for the number of variables appearing, $t \geq 1$, we choose first the variables, $\left(\begin{array}{c}v \\ t\end{array}\right)$, then their sign, $2^t$, then their locations, $\left\{\begin{array}{c}c \\ t\end{array}\right\} t!$. This was derived in theorem 2.4.1.

For $v = 1$, we consider the set of 4 boolean functions and how they are produced syntactically.\(^2\) First, there are $2^k$ possible clauses and none of them can be $\textbf{F}$. There is exactly one way to produce $\textbf{P}$ (likewise $\overline{\textbf{P}}$) in a single clause (by having all the literals identical) and so there are $2^k - 2$ ways to produce $\textbf{T}$. In a sequence of $c$ of these clauses, to get $\textbf{T}$ all the clauses must be $\textbf{T}$, so there are $(2^k - 2)^c$ ways for $\textbf{T}$. One can only get $\textbf{P}$ from a sequence of $\textbf{P}$ and $\textbf{T}$ clauses with at least one $\textbf{P}$ clause. So from $(2^k - 1)^c$ we subtract the completely true sequences, for a total $(2^k - 1)^c - (2^k - 2)^c$. The number for $\overline{\textbf{P}}$ is the same, so the total of satisfiable formulas is:

$$\#\text{CNF-SAT}(1,k,c)_{\text{ord.with}} = 2((2^k - 1)^c - (2^k - 2)^c) + (2^k - 2)^c \quad (2.12)$$

$$= 2(2^k - 1)^c - (2^k - 2)^c. \quad (2.13)$$

\(^2\)We use bold capital notation for both the boolean function itself and the number of formulas representing that function.
2.6 More than one variable

A similar analysis would be feasible for \( v = 2 \), but a bit tedious, given that one would have to consider a lot more inclusion-exclusion involving all \( 2^{2^v} = 16 \) functions, though some of the difficulty could be reduced by appealing to symmetry. For \( v = 3 \), following this method by hand would be quite tedious, involving 256 functions. Instead, we will set up a system of equations involving a counting generating function for all the \( 2^{2^v} \) functions, and then we will solve the system in general. The system takes as base case the number of clauses that are equivalent to each boolean function; the solution of the system is then a generating function for each boolean function, by number of clauses in a disjunction. The model of selection of clauses is ordered with replacement, but any model can be used for the literals. Dershowitz and Lindenstrauss [DL89] use generating function techniques for counting with boolean formulas; that method is generalized here for use in a system of equations.

For arbitrary (positive integral) values of all three parameters and for every one of the \( 2^{2^v} \) boolean functions, we will count the number of formulas (by number of literals and clauses) that evaluates to each function. For \( v = 1 \), there are 4 functions, \( \mathbf{F}, \mathbf{F'}, \mathbf{P}, \) and \( \mathbf{T} \). For an arbitrary number of variables, we notate a function by the correspondence between their truth tables and the integer corresponding to the truth table as a binary representation.

Figures 2.6 and 2.6 show the boolean functions over 1 and 2 variables respectively. The set of functions form a lattice with the greatest lower bound and least upper bound are determined by the “and” and “or” operators, respectively.

For any given boolean operator (here we restrict ourselves to ‘\( \lor \)’, and ‘\( \land \)’ but the method can be applied to any operator, of any arity), the function produced depends only on the functions represented by the operands. For example, ‘\( \lor \)’ has the behavior
**Figure 2.2** The boolean functions over 1 variable.

**Figure 2.3** The boolean functions over 2 variables.
Table 2.3 The binary operator ‘\( \lor \)’ for one variable boolean functions.

<table>
<thead>
<tr>
<th>( \lor )</th>
<th>( F )</th>
<th>( P \land Q )</th>
<th>( P \lor Q )</th>
<th>( P )</th>
<th>( P \land \neg Q )</th>
<th>( P \lor \neg Q )</th>
<th>( \neg Q )</th>
<th>( P \lor \neg Q )</th>
<th>( \neg )</th>
<th>( \neg P \lor Q )</th>
<th>( \neg P \land Q )</th>
<th>( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( P \land Q )</td>
<td>( P \land Q )</td>
<td>( P \land Q )</td>
<td>( P \land Q )</td>
<td>( P \land Q )</td>
<td>( P \land Q )</td>
<td>( P \land Q )</td>
<td>( P \land Q )</td>
<td>( P \land Q )</td>
<td>( P \land Q )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( P \lor Q )</td>
<td>( P \lor Q )</td>
<td>( P \lor Q )</td>
<td>( P \lor Q )</td>
<td>( P \lor Q )</td>
<td>( P \lor Q )</td>
<td>( P \lor Q )</td>
<td>( P \lor Q )</td>
<td>( P \lor Q )</td>
<td>( P \lor Q )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( \neg Q )</td>
<td>( \neg Q )</td>
<td>( \neg Q )</td>
<td>( \neg Q )</td>
<td>( \neg Q )</td>
<td>( \neg Q )</td>
<td>( \neg Q )</td>
<td>( \neg Q )</td>
<td>( \neg Q )</td>
<td>( \neg Q )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( \neg P \lor Q )</td>
<td>( \neg P \lor Q )</td>
<td>( \neg P \lor Q )</td>
<td>( \neg P \lor Q )</td>
<td>( \neg P \lor Q )</td>
<td>( \neg P \lor Q )</td>
<td>( \neg P \lor Q )</td>
<td>( \neg P \lor Q )</td>
<td>( \neg P \lor Q )</td>
<td>( \neg P \lor Q )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( \neg P \land Q )</td>
<td>( \neg P \land Q )</td>
<td>( \neg P \land Q )</td>
<td>( \neg P \land Q )</td>
<td>( \neg P \land Q )</td>
<td>( \neg P \land Q )</td>
<td>( \neg P \land Q )</td>
<td>( \neg P \land Q )</td>
<td>( \neg P \land Q )</td>
<td>( \neg P \land Q )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
</tbody>
</table>

Table 2.4 The binary operator ‘\( \lor \)’ for two variable boolean functions.

shown in Table 2.3. This is also the table for the least upper bound in the lattice. The binary function ‘\( \land \)’ has the corresponding dual behavior.

2.7 A system of equations

From the table of a logical operator acting on boolean functions, one can construct a system of equations whose solution is the number of formulas for each function. For the
\[
\begin{array}{|c|c|c|c|c|}
\hline
\lor & F_1 & P_1 & P_1 & T_1 \\
\hline
F_{k-1} & F_k & P_k & P_k & T_k \\
\hline
\overline{F}_{k-1} & \overline{F}_k & \overline{P}_k & T_k & T_k \\
\hline
P_{k-1} & P_k & T_k & P_k & T_k \\
\hline
T_{k-1} & T_k & T_k & T_k & T_k \\
\hline
\end{array}
\]

Table 2.5 The recurrence for the binary operator ‘\lor’ for one variable.

moment, let us consider the functions that can be represented with a single clause over one variable.

Let \( f_k \) be the number of formulas for the boolean function \( f \) using \( k \) literals, ‘\lor’, and 1 variable. Given a clause of length \( k - 1 \), we can compute \( f_k \) by constructing a recurrence from Table 2.5:

\[
\begin{align*}
F_k &= F_1 F_{k-1} \\
\overline{F}_k &= F_1 \overline{P}_{k-1} + \overline{P}_1 (F_{k-1} + \overline{P}_{k-1}) \\
P_k &= F_1 P_{k-1} + P_1 (F_{k-1} + P_{k-1}) \\
T_k &= F_1 T_{k-1} + \overline{F}_1 (\overline{F}_{k-1} + T_{k-1}) + \\
&\quad P_1 (P_{k-1} + T_{k-1}) + T_1 (F_{k-1} + \overline{F}_{k-1} + P_{k-1} + T_{k-1}) \\
\end{align*}
\]

with the base cases

\[
\begin{align*}
F_1 &= 0 \\
\overline{F}_1 &= 1 \\
P_1 &= 1 \\
T_1 &= 0 \\
\end{align*}
\]

reflecting that the only possible clauses of length \( k = 1 \) are the single literals. Note that a solution for this system is a solution for a system based on ‘\land’ but with functions ordered in reverse (the complement of the bitwise representation).
The linear system can be represented as a matrix of coefficients for the recurrence:

\[
\begin{pmatrix}
F_k \\
P_k \\
P_k \\
T_k
\end{pmatrix} = \begin{pmatrix}
f_{00} & f_{01} & f_{10} & f_{11} \\
0 & f_{00} + f_{01} & 0 & f_{10} + f_{11} \\
0 & 0 & f_{00} + f_{10} & f_{01} + f_{11} \\
0 & 0 & 0 & f_{00} + f_{01} + f_{10} + f_{11}
\end{pmatrix}^k \begin{pmatrix}
F_1 \\
P_1 \\
P_1 \\
T_1
\end{pmatrix}.
\]

If \( f_k \) is the vector for the set of all boolean functions formed by a clause of length \( k \), and then

\[ f_k = \text{OR}(1)^k \cdot f_1 \]

This can be solved by simply multiplying out the matrix for fixed \( k \) or by Gaussian elimination symbolically. But we would like to solve the system symbolically for an arbitrary number of variables. That the system is linear is a direct consequence of the grammar for \( k \)-CNF being regular or, equivalently, the selection model being ordered selection with replacement.

The linear system for an arbitrary number of variables can now be described recursively.

**Definition 3.** Let \( \text{or}(n) \) be defined recursively as

\[
\text{or}(n) = \begin{pmatrix}
0 \cdot \text{or}(n - 1) & 1 \cdot \text{or}(n - 1) \\
0 & 0 \cdot \text{or}(n - 1) + 1 \cdot \text{or}(n - 1)
\end{pmatrix}
\]

\[ i \cdot \text{or}(0) = f_i \]

where \( x \cdot \text{or}(n) \) adds the digits \( x \) to the front of all digits referred to in the matrix \( \text{or}(n) \) and \('+' is matrix addition.
Since we are only concerned with $n = 2^v$, we end up with a system of $2^{2^v}$ equations.

**Definition 4.** Let $\text{OR}(v)$ be defined as

$$\text{OR}(v) = or(2^v)$$

All we need now is to justify that the recursively constructed system counts the functions as we expect:

**Theorem 2.7.1.** If $f$ is the vector that counts the boolean functions on $v$ variables, and $f^n_i$ is a disjunction of $n$ such functions, then

$$f^n = \text{OR}(v)^{n-1} \cdot f$$

**Proof:** We’ll work with $or(2^n)$ first, and the result follows immediately for $\text{OR}(n)$. The matrix $or(n)$, and so also $or(n)^k$, must be upper triangular, since for $i \leq j$, there is no $x$ such that $f_i$ will contribute to $f_x \lor f_j$. The $[i,j]$ entry of $or(v)$ is the sum of all the $f_x$ such that $f_x \lor f_i = f_j$. This leads to 3 entries in $or(n + 1)$: in the upper left quadrant, a zero bit has been added to the truth table representation, so nothing has changed; in the upper right, the additional bit is one and so only those functions with 1 in the most significant place are added; and in the lower right, the extra bit can be either 0 or 1 and so both sets of functions are added. \(\square\)

The system to solve for $\text{AND}(v)$ is dual. Because of this duality, we can use the same matrix as for $\text{OR}$, but reversing the indices for the entries.

### 2.8 Solving the system

The partial order $\preceq$ be the $\mathcal{B}_n$ is the Boolean lattice with $n$ generators where $a \preceq b$ is defined if $a \lor b = b$, the symbol $\lor$ conveniently overloaded for both the lattice’s least
upper bound and the bitwise-or operation on the binary representation of integers from 0 to $2^v - 1$. For $v$ variables, we are concerned with $B_{2^v}$.

**Theorem 2.8.1.** The number of formulas of length $n$ equivalent to $f_i$ generated as a conjunction of atomic function symbols $f_i$ is given by

$$
\text{OR}(v)^n_i = \sum_{s \leq i} \left( (-1)^{2^r - d(s)} \left( \sum_{t \leq s} f^1_t \right)^n \right)
$$

and as a conjunction is given by

$$
\text{AND}(v)^n_i = \sum_{s \geq i} \left( (-1)^{2^r - d(s)} \left( \sum_{t \geq s} f^1_t \right)^n \right)
$$

where $d(s)$ is the number of that are true, and $x \preceq y$ is with respect to the boolean lattice: $x \preceq y \Leftrightarrow x \land y = x$.

For example, $\text{OR}(1)^2_T = (F + P + \bar{P} + T)^k - (P + T)^k - (\bar{P} + T)^k + T^k$. If the base cases are as above, then $\text{OR}(1)^2_T = 2^k - 2$, This counts the number of tautologies in a conjunction of $p$ and $\neg p$ ordered selection with replacement. Note that to reduce the symbolic complexity of equation 2.14, the base case $f^1_m$ is implied.

**Proof:** We prove equation 2.14, since the proof for equation 2.15 is dual.

By induction on $n$. If $n = 1$, then by inclusion-exclusion, all that is left is $f^1_i$. From the linear system in 2.14, $\text{OR}(v)^n = \text{OR}(v) \cdot \text{OR}(v)^{n-1}$. Any entry in $\text{OR}(v)^n$ is the dot product of constants, the coefficients from $\text{OR}(v)$ being the complement of the those in $\text{OR}(v)^{n-1}$. Since these are of opposite sign from what is left over for a term is

$$
\left( \sum_{t \leq s} f^1_t \right) \left( \sum_{t \geq s} f^1_t \right)^{n-1}\text{ which is } \left( \sum_{t \leq s} f^1_t \right)^n.
$$

We will use, with great abuse of notation, both an integer $n$ and its binary representation interchangeably.

26
Corollary 2.8.2. The number of formulas as a disjunction of \( k \) literals evaluating to \( i \) is

\[
\text{OR}(v)^k_i = \begin{cases} 
(2v)^k - \sum_{j=0}^{\nu} \binom{\nu}{j} 2^j (-1)^{\nu-j} j^k, & \text{if } i = 2^2 - 1 \\
\sum_{j=0}^{m} \binom{m}{j} (-1)^{m-j} j^k, & \text{if } d(i) = 2^v - 2^{v-m}, 1 \leq m \leq v, \\
0 & \text{otherwise}
\end{cases}
\] (2.16)

the second case, for only \( \binom{v}{m} 2^{v-m} \) different functions having \( 2^v - 2^{v-m} \) nonzero bits.

Proof: The value \( \text{OR}(v)^k_i \) can only be nonzero if it is equal to or above a nonzero value of \( \text{OR}(v)^1_i \). In equation 2.14, every term in the sum is a sum raised to the \( k \)th power. If \( i \) is above both a function corresponding to a literal and its negation, then \( i \) must be \( 2^2 - 1 \), and the first term of equation 2.14 is \( (2v)^k \). For the rest of the terms for \( i = 2^2 - 1 \), we can choose either a literal or its negation but not both. If we choose \( j \) of these, there are \( 2^j \binom{\nu}{j} \) ways, since for each \( j \) we are over-counting the level before, we use the \( (-1)^{\nu-j} \) for inclusion-exclusion. For \( i \) from a set of \( m \) literals, \( 1 \leq m \leq v \), we sum in the same way over \( j \), choosing \( j \) of these literals for the subset to exclude. For these however we do not need the \( 2^j \) factor, since for the terms to exclude below, we have already chosen the positive or negative literal. The choices made for the \( 2^j \) factor appear as the number of functions with \( 2^{v-m} \) bits having nonzero solution. \( \square \)

Note that for \( T \), this corresponds to the solution given above in theorem 2.4.1 for \( k = 1 \).

For small values of \( v \) we can compute exact formulas from this theorem. For example, since \( \text{OR}(1)^k_p = \text{OR}(1)^k_p = 1 \) and \( \text{OR}(1)^k_f = 0 \), we get

\[
\text{AND}(1)^c_f = (0 + 1 + 1 + 2^k - 2)^c - 2(1 + 2^k - 2)^c + (2^k - 2)^c \\
= (2^k)^c - 2(2^k - 1)^c + (2^k - 2)^c
\]

27
and so

$$\# \text{CNF} - \text{SAT} (1, k, c)_{\text{ord, with}} = 2(2^k - 1)^c + (2^k - 2)^c$$

which is equivalent to that found in equation 2.12.

Similarly, we can compute for \( v = 2 \) that

$$\# \text{CNF} - \text{SAT} (2, k, c)_{\text{ord, with}} = 4(4^k - 2^k)^c - 2(4^k - 2 \cdot 2^k)^c - 4(4^k - 2 \cdot 2^k + 1)^c + 4(1 - 2^k + 2)^c - (4^k - 4 \cdot 2^k + 4)^c$$

$$= 4(2^k (2^k - 1)^c - 2(2^k (2^k - 2))^c - (2^k - 1)^2)^c + 4((2^k - 1)^c - ((2^k - 2)^2)^c$$

and its ratio to the total number of formulas, \( 4^{2c} \) when \( k = 2 \) is shown in figure 2.8. In that selection model, the symbolic complexity of the clauses, as given in 2.8.2, is large enough to make computation for formulas with \( v \geq 3 \) infeasible.

For the traditional non-complementary model for selection of literals in a clause, no matter the size of \( k \), there is exactly one way to get a particular function in one clause. This makes computation for \( v \leq 4 \) in this model feasible. For \( v = 4 \) and \( k = 3 \), the number of satisfiable formulas is

$$\# \text{CNF} - \text{SAT} (4, 3, c)_{\text{trad}} = 16 \cdot 28^c - 32 \cdot 25^c - 88 \cdot 24^c + 96 \cdot 22^c + 256 \cdot 21^c$$

$$+ 184 \cdot 20^c - 256 \cdot 19^c - 736 \cdot 18^c - 384 \cdot 17^c + 460 \cdot 16^c$$

$$+ 1568 \cdot 15^c - 1312 \cdot 13^c - 2184 \cdot 12^c + 352 \cdot 11^c + 2154 \cdot 10^c$$

$$+ 1024 \cdot 9^c - 2396 \cdot 8^c - 928 \cdot 7^c + 960 \cdot 6^c + 928 \cdot 5^c$$

$$- 976 \cdot 4^c + 128 \cdot 3^c - 64 \cdot 2^c + 32$$

and its ratio to the total number of formulas, \( 2^3 \left( \frac{4}{3} \right) = 32 \) is shown in figure 2.8.

Theorem 2.8.1 gives a closed form formula for the number of propositional formula as a sequence of clauses in disjunction, regardless of the makeup of the clauses. If some other
Figure 2.4 Ratio of satisfiable to total formulas for $v = 2$, $k = 2$, ordered selection with replacement model. The $x$-axis is $c$. 


Figure 2.5 Ratio of satisfiable to total formulas for \( v = 4, \ k = 3, \) traditional model. The \( x \)-axis is \( c \).
method were used to count clauses, or even some other syntax were used to construct
text items in place of clauses, the above theorem would allow us to enumerate the disjunctions
resulting in any particular boolean function. There is a difficulty in purely computational
terms; the summations range over all the items in the boolean algebra, namely all \(2^{2^v}\)
functions. For \(v > 4\), this is quite infeasible.

Some symbolic simplification with appropriate parameters can be made though. In
two different ways we proved that, for \(k = 1\),

\[
\#\text{CNF} - \text{SAT} (v, 1, c)_{\text{trad}} = \sum_{t=0}^{v} \binom{v}{t} 2^t \cdot t^c \cdot (-1)^{v-t}.
\]

First, this is shown through Theorem 2.4.1, because, with the restriction to one literal
per clause, the traditional selection model is equivalent to ordered selection with replacement. The second way is through Theorem 2.8.2, where it equals \((2v)^c - \text{OR}_T(v)_c\), since

\[
\text{OR}_T(v)_c = \text{AND}_F(v)_c.
\]

For \(k = v\), there are \(2^v\) possible clauses, conveniently uniform, so

\[
\#\text{CNF} - \text{SAT} (v, v, c)_{\text{trad}} = 2^{vc} - \sum_{t=1}^{2v} \binom{2v}{t} (-1)^{t+1} \cdot t^c
\]

\[
= 2^{vc} - \binom{v}{c} \cdot (2^v)!,
\]

where the first line is from noticing that the contribution to every boolean function having
t \(1\)'s is equal to \(\binom{2^v}{t}\), and the second line using the generating function for \((e^z - 1)^v\).

### 2.9 Approximations, Asymptotics and Thresholds

The number of satisfiable \(c\)-clause \(k\)-CNF formulas over \(v\) variables is the total number
of formulas less the unsatisfiable ones, namely \((2v)^{kc} - \text{AND}_0^k\) where the base case is

\[
\text{AND}_1^i = \text{OR}_2(v)^{2v-i}.
\]

As a corollary to Theorem 2.8.1, we first determine \(\text{OR}_1^1\) for
all \(i\) and then \(\text{OR}_i^k\), then use that to compute \(\text{AND}_0^k\).
Under the different models of random selection, there are different ways of specifying \( \text{OR}(v)_i \). Theorem 2.8.1 works for a disjunction (or dually a conjunction) as a sequence, it does not matter how \( f \), the clauses, are computed. We can use the model of selection for how a clause is generated to simplify. The traditional method for generating random \( k \)-CNF formulas is by the literals in the clause as a set of variables without replacement, then a sign for each variable, this gives

\[
2^k \binom{v}{k}
\]

possible clauses, and these clauses all produce distinct functions.

Since there is only one of each of these functions, we can apply this to Theorem 2.8.1, and simplify. Under this model, in the equation 2.8.1, there are at most \( 2^k \binom{v}{k} \) possible distinct terms. In computing the number of unsatisfiable formulas, \( (2^k \binom{v}{k})^c \) is the largest term, and is removed by subtracting to get the satisfiable formulas. The second largest term is then the largest term for satisfiable formulas and is

\[
\left( 2^v(2^k - 1) \binom{v}{k} \right)^c.
\]

This is an upper bound on the number of satisfiable formulas. The ratio of satisfiable to total is then

\[
\frac{(2^v(2^k - 1) \binom{v}{k})^c}{(2^k \binom{v}{k})^c} = \left( 2^v \frac{2^k - 1}{2^k} \right)^c = \left( 2 \left( \frac{2^k - 1}{2^k} \right)^r \right)^v
\]

when \( c = vr \), a linear ratio of \( v \). The limit of this as \( v \) goes to infinity is a step function, where the step is at the solution for \( r \) in

\[
\left( \frac{2^k - 1}{2^k} \right)^r = 1/2
\]
or

\[ r = \frac{\log(2)}{\log \left( \frac{2^k}{n+1} \right)} \]

which, for \( k = 3 \), gives \( r = \frac{\log 2}{\log n} \approx 5.191 \), a known upper bound for the satisfiability threshold for \( k = 3 \). This is not a terrible excess beyond the experimentally established threshold around 4.25. For \( k = 1 \) the threshold is 0 but bound gives 1, and for \( k = 2 \) the threshold is 1 and the bound gives \( \log 2/\log(4/3) \approx 2.41 \).

2.10 Other Syntax

The syntax for the \( k \)-CNF formulas is very simple and flat. We now address the counting problem for more complicated formulas. Consider propositional formulas with “and,” “or,” and positive and negative literals. Here the syntax encompasses all propositional formulas. Combinatorially, this corresponds to all full binary trees, with “and” and “or” as labels for internal nodes, and the 2\( v \) literals as leaves.

The total number of formulas having \( n + 1 \) literals and \( n \) operators is then

\[ \#ALL(n, v) = \frac{1}{n+1} \binom{2n}{n} 2^n (2v)^{n+1} \]

because there are \( \binom{2n}{n} \frac{1}{n+1} \) trees with \( n + 1 \) leaves (the Catalan numbers), 2 choices (“and” and “or”) for each of \( n \) internal nodes, and \( 2v \) choices for each leaf (note that in the CNF syntax, we considered the operators as “flat” having unlimited arity, but here the arity is 2).}

---

\(^3\)This upper bound can be found by more elementary methods as follows. Considering the same model for literal selection of a clause, for any one of the \( 2^n \) assignments to the variables, only \( \binom{2^n}{n} \) clauses yield the function \( F \), so \( (2^k - 1) \binom{n}{k} \) are satisfiable. The probability that a sequence of \( c \) clauses is satisfiable is then \( \left( \frac{2^c - 1}{2^c} \right)^c \). The expected number of satisfying truth assignments then is \( 2^n \left( \frac{2^c - 1}{2^c} \right)^c \). The upper bound follows from the derivation above. Historically, this value was computed early (see Franco [FP83]) as a value for which asymptotically the ratio goes to zero, not with respect to a threshold.
Just as with \( k \)-CNF formulas, we can set up a system of equations to describe how each boolean function is generated in this situation.

Let \( v = 1 \), a very restricted situation. There are 4 boolean functions, \( T, P, \overline{P}, F \) based on the operators in Table 2.3.

From that table, we get the system of equations of generating functions

\[
\begin{align*}
T &= (2T(T + P + \overline{P} + F) + 2P\overline{P})z \\
P &= 2P(F + P + T)z + 1 \\
\overline{P} &= 2\overline{P}(F + \overline{P} + T)z + 1 \\
F &= (2F(T + P + \overline{P} + F) + 2P\overline{P})z
\end{align*}
\]

where each internal node has weight 1 and each literal has weight 0. To count the number of satisfiable formulas of weight \( n \) again we need to solve this system for the generating function for \( F \) and subtract.

Systems of equations arising from context free grammars, such as the one above, are inherently non-linear, so the linear elimination method used for \( k \)-CNF will not work here. However, the system is still amenable to ad hoc elimination. Since the system is over a multivariate polynomial ring, we can instead use Groebner basis completion as a systematic method to eliminate all but one variable. This results in the following single equation in \( T \):

\[
T = 32T^1z^3 - 32T^3z^2 - 16T^2z^2 + 10T^2z + 8Tz + 2z
\]

The monomial ordering that allows such variable elimination is the lexicographic ordering, i.e. the function symbols (as variables) are given an ordering, and then two monomials are compared in order of these symbols, along with the degree. The generating function for \( T \) is one of the roots of this equation (the root chosen is that which gives nonnegative
integer coefficients), namely

\[ T(z) = \frac{1 - \sqrt{1/2(1 + 8z + \sqrt{1 - 16z})}}{4z} \]

From this we need to extract a closed form for the coefficients. The generating function for the inner square root \( \sqrt{1 - 16z} \) can be found either in analogy with that for some binomials (like the Catalan numbers), or by solving the recurrence implied by successive derivatives from the Taylor expansion:

\[ (\sqrt{1 - 16z})^{(n)} = (\sqrt{1 - 16z})^{(n-1)} \frac{8(2n - 3)}{1 - 16z} \]

Along with the other terms inside the outermost square root, this gives:

\[ 1 + 8z + \sqrt{1 - 16z} = 1 - \sum_{k \geq 2} 2^{2k} \binom{2k - 2}{k - 1} \frac{z^k}{k}. \]

So,

\[ 4zT(z) = 1 - \sqrt{1 - \sum_{k \geq 2} 2^{2k} \binom{2k - 2}{k - 1} \frac{z^k}{k}} \]

\[ = 1 - \sum_{n \geq 0} \binom{1/2}{n} (-1)^n \left( \sum_{k \geq 2} 2^{2k} \binom{2k - 2}{k - 1} \frac{z^k}{k} \right)^t \text{ binomial theorem} \]

\[ = 1 - \sum_{n \geq 0} 2^{-2n} \binom{2n}{n} \frac{1}{1 - 2n} \left( \sum_{k \geq 2} 2^{2k} \binom{2k - 2}{k - 1} \frac{z^k}{k} \right)^t \text{ substitute for } \binom{1/2}{n} \]

The inner Catalan summation raised to an arbitrary power, has a convenient identity removing the exponent:

\[ \left( \sum_{k \geq 2} 2^{2k} \binom{2k - 2}{k - 1} \frac{z^k}{k} \right)^n = \sum_{k \geq 0} 2^{2k+1} \binom{2k + 2n - 1}{k} \frac{z^k}{k + 2n} \quad (2.18) \]
Then the derivation follows:

\[
4z^2 T(z) = 1 + \sum_{n \geq 0} 2^{-2n} \binom{2n}{n} \frac{1}{2n-1} \sum_{k \geq 0} 2^{2k+1} \frac{n}{k+2n} \binom{2k+2n-1}{k} z^k \\
\text{substitute 2.18}
\]

\[
= 1 + \sum_{n \geq 0} \sum_{k \geq 0} 2^{-2n} \binom{2n}{n} \frac{1}{2n-1} 2^{2k+1} \frac{n}{k+2n} \binom{2k+2n-1}{k} z^k \\
\text{move summation}
\]

\[
= 1 + \sum_{n \geq 1} z^n \sum_{k=0}^{\left\lfloor \frac{n}{2} \right\rfloor} \binom{2k}{k} \frac{1}{2k-1} 2^{2n-2k-1} \frac{k}{n} \binom{2n-2k-1}{n-2k} z^k \\
\text{move summation}
\]

\[
T(z) = \sum_{n \geq 1} z^n \sum_{k=0}^{\left\lfloor \frac{n}{2} \right\rfloor} \binom{2k}{k} \frac{1}{2k-1} 2^{2n-2k-1} \frac{k}{n} \binom{2n-2k-1}{n-2k} z^k
\]

So \( T(n) \) is the coefficient of \( z^n \) in this sum. From this, we would like to extract an asymptotic approximation. Or rather, we pursue the asymptotics of the ratio of satisfiable formulas to all formulas for a given size \( n \). As \( n \) approaches infinity, many factors drop out. All that’s left of the second binomial is and we are left with

\[
\lim_{n \to \infty} T(n) = \frac{1}{4} \sum_{k \geq 0} \binom{2k}{k} 2^{-4k}.
\]

The ratio of successive summands \( k + 1 \) to \( k \) is \( \frac{1}{4} \frac{k+1}{k+1} \) from which we can read directly the hypergeometric formula

\[
\lim_{n \to \infty} T(n) = \frac{1}{4} \sum_{k \geq 0} \frac{\binom{2k}{k}}{k!} \left( \frac{1}{4} \right)^k = \frac{1}{4} F\left( \frac{1/2}{1/4} \right)
\]

A basic hypergeometric (and generating function) identity says that

\[
F\left( \frac{a}{1-z} \right) = \sum_{k \geq 0} \frac{a^k}{k!} z^k
\]

\[
= \sum_{k \geq 0} \binom{a+k-1}{k} z^k
\]

\[
= \frac{1}{(1-z)^a}
\]

36
which gives

\[
\lim_{n \to \infty} T(n) = \frac{1}{4} \frac{1}{\sqrt{1 - \frac{1}{4}}}
= \frac{1}{2\sqrt{3}} \approx 0.288675
\]

Since \( F(n) \) is the same, the ratio of satisfiable to all formulas as \( n \) goes to infinity converges to a constant \( 1 - \frac{1}{2\sqrt{3}} \approx 0.711325 \).

### 2.10.1 More variables

For \( v = 1 \), this method worked out fine, but for larger \( v \) it is computationally much more difficult. There will be \( 2^{2^v} \) equations (though only one needs to be solved for). Some reduction can be made by noticing that, because of boolean duality, the number of formulas equaling one function, is the same as that for its negation (syntactically formed by changing every literal to its negative, and swapping and’s and or’s. However, even if from such a system a single equation for \( T \) is found, it will be a high degree polynomial. Computing exact roots symbolically is possible, using hypergeometric functions, but the time and space complexity of this procedure is terrible (it has not been pinpointed exactly). And even if a symbolic root is found, extracting a formula for the coefficient will not be so easy either. There are algorithms to automate such extraction, but even those were unable to solve the nested square root above.

### 2.11 The complexity of counting formulas

Given a propositional formula, the problem of counting the number of different variable assignments that make the formula true, i.e. the number of satisfying assignments, is \#P-complete \([V al79]\). This is quite a different problem from that here, which is, given
a syntax of formulas with given parameters, counting the number that are satisfiable. However, it is still reasonable to ask the question how difficult it is to compute this.

Both equations in theorem 2.8.1 involve doubly nested summation over the boolean lattice generated by \( v \) variables, for a total of \( 2^{v+1} \) operations. This is of course terribly inefficient, when given as input just \( v \). On the other hand, if for any constant \( v \), the input is the set \( f \) of counts for every boolean function over \( v \), then the input size is also doubly exponential in \( v \) and so the algorithm could be considered to run in quadratic time in the input size. And this is actually an improvement over the cubic time (or really time equivalent to matrix multiplication) needed for solving an arbitrary linear system. Therefore we can say that the generalized problem of counting sequences evaluating to a particular function is in NC\(_2\) but NL hard.

This is a bit unreasonable in the sense that our input, the base cases for the functions, is hardly ever arbitrary. It is specified quite succinctly by the selection model for the clauses. In fact, this could be considered so succinct as to be a constant, the only variables being \( v \), \( k \), and \( c \).

## 2.12 Conclusions

Using elementary counting arguments, we were able to count the number of \( k\)-CNF formulas for every boolean function, and specifically the number of satisfiable formulas for a given number of variables, clauses, and literals per clause. The method of creating a system of equations to count the functions can be applied to any formula syntax using any set of operators. In this case, CNF syntax and the dual boolean operators simplify the analysis considerably.

Simply counting combinatorial objects is interesting enough of itself but the method and the result can be used for other things. The general method can be applied to
other families of formulas, especially those with a simple syntax and those with other logical operators. For non-linear grammars, that is non-regular grammars, the results will involve the Catalan numbers and its analogues.

Once the result is refined to give the exact function for the number of satisfiable $k$-CNF formulas, asymptotic analysis will give better bounds on the SAT/UNSAT threshold phenomenon.
CHAPTER 3

Symmetry in Model Checking

3.1 Symmetry in Boolean Formulas

Since most programs are not random, but purposefully designed, they tend to follow certain patterns, certain recurring structure, not only in the text of the code but also in the logic of their operation. This is especially true with respect to parallel programs because often, except for a single parameter like a label, processes may be identical. Instead of enumerating all possible assignments, we may know that some assignments are virtually identical; a heuristic would be to enumerate only those assignments that are non-isomorphic on permutation of the variables. For example, consider a simple boolean formula:

\[ f(a, b, c, d) = (a \lor c \lor d) \land (b \lor c \lor d) \]

We can exchange just \( a \) for \( b \), or just \( c \) for \( d \), or we can do both exchanges at the same time, and still get the same boolean function. Table 3.1 shows the truth table for the function defined by the formula. By inspection of the table, we can see that there are only three distinct possibilities for \( a \) and \( b \) together: \((0, 0), (1,0), \) and \( (1,1) \) and likewise for \( c \) and \( d \). Combined, these account for all the essentially distinct valuations for \( f \).
Table 3.1 The truth table for \( f(a, b, c, d) = (a \lor c \lor d) \land (b \lor c \lor d) \). The checked unique entries are arbitrarily chosen representatives of the set of those to which it can be permuted.

If we want to check the values of \( f \), we need not examine all \( 2^4 = 16 \) valuations, only 
\[ 3 \cdot 3 = 9 \] of them.

So there are two separate goals here: one, to find the structure of the symmetries inherent in the function given the formula, and two, to be able to generate those unique valuations in order to show equality of functions or validity.

A finite set of permutations generates a group. So we can use the tools of finite permutation group theory and associated algorithms. For the example above, the function \( f \) above exhibits the symmetries for the group on 4 elements \( V_4 = C_2^2 \) of transpositions of 2 disjoint pairs (see Figure 3.1).

It is interesting to note the following phenomenon, since it is relevant to correct generation of distinct valuations. In Figure 3.2, we give a different set of permutations
\[ x : (ab) \]
\[ y : (cd) \]
\[ x^2 = y^2 = (x \cdot y)^2 = I \]
\[ \{I, (a b), (c d), (a b)(c d)\} \cong C_2^2 \]
\[ Z_{\{x,y\}}(z) = \frac{1}{4}(z_1^4 + 2z_1^2 z_2^2 + z_2^4) \]

\[ \text{Figure 3.1} \quad V_4 = C_2^2, \text{ a presentation, and its Cayley graph. 'I' is the identity permutation.} \]

\[ x : (ab)(cd) \]
\[ y : (ac)(bd) \]
\[ x^2 = y^2 = (x \cdot y)^2 = I \]
\[ \{I, (a b)(c d), (a c)(b d), (a d)(b c)\} \cong C_2^2 \]
\[ Z_{\{x,y\}}(z) = \frac{1}{4}(z_1^4 + 3z_2^2) \]

\[ \text{Figure 3.2} \quad V_4 = C_2^2, \text{ a presentation, and its Cayley graph. 'I' is the identity permutation.} \]

acting on the same set, generating the same abstract group, but having a different cycle index.

A boolean function that has this permutation group is given in Table 3.2 with the distinct representatives marked. Notice that this function has seven distinct valuations\(^1\) in comparison to the nine for the function in Figure 3.1.

The group operations permute the positions of the variables. Of the \(2^n\) possible arrangements for \(n\) variables, the permutation group induces a partition based on the values. Knowing the group, we seek a single representative from each block of the partition. For example, if all permutations yield the same function, we have a classic totally symmetric boolean function. The only thing that distinguishes any valuation from another is the number of similar elements in the valuation, i.e. the number of 1’s. For

\(^1\)This does not imply that the model symmetry group of the second function is the one of maximum order. There is no boolean function on four variables whose full symmetry group is given by that permutation presentation.
Table 3.2 The truth table for \( f(a, b, c, d) = (a \vee b \vee c \vee d) \land (\neg a \vee b \vee c \vee \neg a) \land (a \vee \neg b \vee \neg c \vee d) \).

example, the formula \( a \vee b \vee c \vee d \) is totally symmetric (the group is \( S_n \)); to compare with another function, all that is needed are \( n+1 = 5 \) different valuations: \((0,0,0,0), (1,0,0,0), (1,1,0,0), (1,1,1,0), (1,1,1,1)\). On the other hand when no permutations hold, then this yields the trivial group \( E_n \) and all \( 2^n \) assignments are unique.

3.2 Computational Complexity of Symmetry

3.2.1 Finite Model Checking

Since we equate here finite model checking with satisfiability or validity/tautology of a propositional formula with variables, the two problems are NP-complete and coNP-complete respectively. Deciding whether two formulas are equal for all valuations is
equivalent to validity (one direction uses biconditional, the other checks equivalence with $T$).

### 3.2.2 Symmetry

The decision problem $BI$, boolean isomorphism, is that, given a set of variables and a two boolean formulas over those variables, do the two boolean function defined by the formulas have an isomorphism of variable, preserving all values of the function? More formally: instance: given a set $V$ of variables, $G, H$ boolean formulas, is there a permutation $\sigma$ of the variables which gives the same value of the two functions for all valuations: $\exists \sigma \forall x : f(x) = g(\sigma(x))$.

**Theorem 3.2.1**. $BI$ is in $\Sigma_2^P$ and is coNP-Hard.

First, let’s attack this naively by trying to come up with a reasonable certificate for some complexity class. It is most likely not in NP, because just the certificate of a permutation (an isomorphism) of variables, one would still need to check all $2^n$ valuations. Likewise it is most likely not in coNP because even a certificate of a permutation - and all valuations would not prevent some other permutation from working. This leads us up higher in the hierarchy: a certificate of a permutation and an NP oracle to check all $2^n$ valuations would certainly work, so the problem is in NP$^{NP}$ which is $\Sigma_2^P$. This is now seen to be the case by inspection of the formal statement of the problem, because there is an existential quantifier on the permutation (a function which is therefore a second order logical variable), and the universal quantifier is over the set of all valuations of the variables, another second order variable.

To be a little more firm, there is a trivial reduction of coNP to BI ISO: let $G$ be the formula given to coNP, and $H$ be the formula $T$. There is an isomorphism between the $G$ and $H$, iff $G$ is equivalent to $H$ or $G$ is a tautology, which is coNP-complete. There is
also an obvious poly-time reduction from graph isomorphism, GI, to BI: from a graph, construct a formula such that each vertex corresponds to a variable, each edge corresponds to a clause with the two vertex variables that are incident to the edge, and the formula is the conjunction of all the clauses. This also gives an exact characterization of a subset of BI to GI: GI is poly-time equivalent to monotone 2-CNF boolean isomorphism. Since there are the analogous poly-time equivalences among variants of BI as there are among CIRCUIT-SAT, SAT and 3-CNF-SAT, a further correspondence can be made: BI is poly time equivalent to edge labeled hypergraph isomorphism where one label (the negation operator) forms a perfect matching. □

Borchert et al. [BRS98] and Agrawal and Therauf [AT96] discuss the computational complexity of detecting isomorphism of boolean functions. They derive some theorems that show that BI is a counterpart to GI but higher in the hierarchy. GI is in NP but is not known to be NP-complete, in P, or even P-hard, and contrary to alternative classes built upon SAT, many of the similar alternatives to GI are poly-time equivalent to GI, e.g. #GI (the number of isomorphisms) and GA (nontrivial graph automorphism). The corresponding problems, #BI and BOOL AUTO are both poly-time equivalent to BI. The most interesting result ([AT96]) is that if BI ISO were \( \Sigma^p_2 \)-complete the poly-time hierarchy would collapse to \( \Sigma^p_3 \). So BI is most likely not \( \Sigma^p_2 \)-complete.

### 3.3 Algorithms for symmetry

#### 3.3.1 Finding the symmetry group of a boolean formula

First, let’s make some characterizations of formulas with respect to groups.

**Theorem 3.3.1.** All finite abstract groups are realizable by some boolean formula.
**Figure 3.3** Known inclusions for complexity classes. There are no strict inclusions known between P and PSPACE.

**Figure 3.4** Complete problems for complexity classes and runtimes of associated algorithms.
**Proof:** This comes via the reduction from GI. Frucht’s theorem states that every group is realized in some graph: consider a presentation (by generators and relations) of the group. Construct the Cayley graph (nodes are elements of the group, directed edges are labeled by generators that take one element to another). Note that this graph is vertex-transitive. We convert this to an undirected graph by replacing each edge labeled by $g_i$ by a unique gadget $g'_i$ that has no isomorphisms with the other gadgets. Form the boolean formula (as in the reduction to monotone 2-CNF) corresponding to that graph. \(\square\)

So all groups are possible, but not necessarily for any given number of variables. For example, a function that has a cyclic permutation of order 4 is not possible with 4 variables though it is possible for 12 variables.

**Theorem 3.3.2.** *Almost all boolean functions have the trivial symmetry group.*

**Proof:** Almost all graphs have the trivial symmetry group. So, by the realizability theorem above, the distribution of groups among boolean functions is even more skewed towards the trivial group. \(\square\)

Also by analogy with graphs, most small formulas have a nontrivial group. The number of functions grows doubly exponentially with the variables, giving much more room for asymmetry.

Given that we cannot reduce our general search at all to a subset of finite groups, and the complexity is intractable, we propose simply a general strategy of backtracking (attempting permutations), dynamic programming (finding symmetries on subsets of the variables), and finding chains of quotient groups (when a permutation is found, divide it out, remove it).

A heuristic here would be to search for very specific groups that might be found in practice. The symmetric group is not likely to be found by itself; the identity (trivial)
group is very likely but knowing it doesn’t get us anything. But both of those groups will be subgroups of a group for larger formulas, so they will be useful to recognize. Cyclic groups are very unlikely, unless by design (for example in a set of processes passing messages in a ring). The most likely groups will be direct products of those above (symmetric and identity groups) or one whose quotient is symmetric (having many isomorphic pieces, which is common by design).

3.4 Generating Specific Groups

Before addressing the generation of unique representatives in the general case, we will discuss generation of the groups just mentioned. An important strategy in generating these valuations is that they be done in sequence, with little overhead in passing from one to the next. So as much as possible, we’ll use Gray codes for computing individual valuations with least cost.

- A symmetric group on \( n \) elements, \( S_n \), acts on the assignments such that only the number of distinct values of true and false distinguishes any two valuations. So there are only \( n + 1 \) distinct assignments. These are generated in numerical order, 0 through \( n \).

- If a group is the direct product of subgroups, \( G = G_1 \times G_2 \) the set of distinct valuations is the direct product of the distinct valuations of the subgroups, \( V_G = V_{G_1} \times V_{G_2} \). There are a number of ways of generating a direct product in Gray code fashion. One is a ‘boustrophedon’ pattern, going from the first to the last coordinate in one entry, then in the next entry alternating first to last, then last to first. Another way is a hybrid of a binary reflected Gray code and a Hilbert
space filling curve. A convenient property of this hybrid is that the sequence never ‘strays’ (locally) too far from itself.

- The identity group on \( n \) elements, \( E_n \), is just the direct product of \( n \) copies of \( S_1 \), which gives the expected \( 2^n \). Because there are only two values per coordinate, the bouystrophedon method generates the same sequence as the binary reflected Gray code.

- If the variables can be partitioned into \( k \) parts such that each part is identical with the others, we call this symmetry \( S_k(I_{n/k}) \) (there are groups of size \( n/k \) with no symmetries within a part but any part is isomorphic to any other part). This situation is common with multiple communicating identical processes. To sequence through these valuations, one is essentially giving a Gray code for multisets (see Harris and Dershowitz [HD99] for details).

For example, if all variables pair up, then the group is \( S_2^{n/2} \). Each pair has only 3 distinct valuations and they form a direct product of \( n/2 \) of these for a total \( 3^{n/2} = 1.73^n \) valuations. If the variables form symmetric factors of size \( k \), for a group \( S_k^{n/k} \), this extends to \( (1 + k)^{n/k} \) valuations. See Figure 3.3 for a comparison of the number of distinct valuations. Notice how great the reduction is the greater the symmetry.

### 3.5 The general case for generation

Often, when generating unique objects according to some isomorphism criterion, a sieve method is used: generate a new item, check against all old items, throw out if it matches any previous one. This is a naive algorithm that always works but it is inherently wasteful of time and space, especially when other methods are available (See Goldberg [Gol93] for a description of sieve methods). In the case of model checking through
<table>
<thead>
<tr>
<th>group</th>
<th># distinct</th>
<th>generated by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>identity (no symmetry)</td>
<td>$E_n$</td>
<td>$2^n$</td>
</tr>
<tr>
<td>list of pairs</td>
<td>$S_2^{n/2}$</td>
<td>$3^{n/2} \times 1.73^n$</td>
</tr>
<tr>
<td>$k$ parts symmetric within</td>
<td>$S_{n/k}^k$</td>
<td>$(k + 1)^{n/k}$</td>
</tr>
<tr>
<td>$k$ parts symmetric among</td>
<td>$S_k(I_{n/k})$</td>
<td>$O(n^k)$</td>
</tr>
<tr>
<td>symmetric (total symmetry)</td>
<td>$S_n$</td>
<td>$n + 1$</td>
</tr>
</tbody>
</table>

**Table 3.3** Comparison of symmetries and number of distinct valuations.

symmetry, one might as well ignore symmetry altogether and just try all valuations since the storage and checking would take even longer than the evaluation of every possible valuation.

Instead we will pursue a general algorithm through extensions of older techniques. A precursor to generation is enumeration and one can consider the counting problem as solved with respect to permutation groups: Pólya enumeration and the calculation of cycle index polynomials. Given a finite permutation group one can calculate a generating function from a listing of the permutations that are elements of the groups, and this generating function (the cycle index) counts the number of distinct orbits of the set being acted upon by the permutations. These orbits correspond to the unique labelings/valuations.

The cycle index of a permutation group is the sum over all the permutations in the group considered in cycle form. A permutation in cycle form $(abc)(de \cdots)(yz)$, has a number of cycles of different length. A term in the sum is a monomial that captures this number and length: if $\sigma = (12)(345)(678)(9)$ then the corresponding term is $x_1x_2x_3^2$, 50
because there is 1 cycle of length 1, 1 of length 2 and 2 of length 3. The subscript on the variable $x$ helps differentiate the cycle types for the more general case.

For example, consider the group $G = C_2^2$ (and its presentation above). Its elements are

$\{(a)(b)(c)(d), (a b)(c)(d), (a)(b)(c d), (a b)(c d)\}$ corresponding to a cycle index polynomial $Z(G : x) = \frac{1}{4}(x_1^4 + 2x_2^2 + x_3^2)$. Since there are two possible values at any position, we let $x = 2$, so, $Z(G : 2) = (16 + 16 + 4)/4 = 9$ (which we found by inspection above).

Here is a larger example, a presentation of a finite group isomorphic to $A_4$, the alternating group on 4 elements. Both permutations and relations are given. Note that this is not possible from any boolean formula on 4 variables.

$x : (a \ b \ c)(4)$

$y : (a \ b)(c \ d)$

$x^3 = y^2 = (xy)^3 = 1$

The corresponding Cayley graph for these relations is given in Figure 3.5. The cycle index for $A_4$ is $1/12(x_1^4 + 8x_1^2 x_3 + 3x_2^3)$, giving the minimum 5 unique valuations, the same as those for $S_4$. 

Figure 3.5 The Cayley graph for $A_4$ acting on 4 elements. A hamiltonian cycle is in bold. Every edge has a reverse between the same two nodes for the inverse of the generator.
The main obstacle is that the construction of the cycle index polynomial is a mapping from the cycle form of the elements as permutations to terms of polynomial. The transformation from permutation to monomial removes much of the combinatorial action of moving items in cycles, leaving only a record of the size and number of cycles. For example, the set of permutations \( G' = \{I, (ab)(cd), (a c)(b d), (a d)(b c)\} \) is also isomorphic to \( V_4 \) but give a different cycle index: \( Z(G' : x) = x_1^4 + 3x_2^3 \). So, \( Z(G : 2) = (16 + 3 \cdot 4)/4 = 7 \). The unique valuations are then: 0000, 0001, 0011, 0101, 0110, 0111, 1111. This is not so terrible as long as we have available the permutations; we can expect to extract the unique valuations given those permutations, whatever the abstract nature of the group is.

It is more troublesome, however, that two nonisomorphic permutation groups can have the same cycle index polynomial. Because counting polynomials are commutative and the structures being counted are not so necessarily, there is not a perfect correspondence. For example, Pólya [PR87] (also Harary [HP73]) gives the example that there are two groups of order \( p^m \), \( p \) odd, \( p, m \geq 3 \) one which is abelian, the other not, but they both have the same cycle index.

Many families of combinatorial objects can be enumerated and generated using functional equations with special operations acting on multivariate generating functions (see Harris and Dershowitz [HD99]). The cycle index polynomial is a generating function that has not yet been given such a treatment mostly because of the difficulty in interpreting the operations that create it. All the above special case algorithms can be analyzed through the cycle index polynomial; the generation of unique valuations is still ad hoc though. An analysis of non-commutative Pólya enumeration should provide a unified general algorithm for the construction problem.
### 3.6 Other applications in automated deduction

Since model checking necessitates model generation, we can use the same mechanism to speed up the search through large sets of models. This is often done explicitly in any kind of experimental mathematics, where conjectures or refutations are sought in small instances, for example the computer proof that there are no projective planes of order 10 (see Lam [LTS89]). Another application has been to model generation for finite algebras, for example those satisfying Tarski’s High School identities (see Burris and Lee [BL93]). Symmetry cutting, another term group invariant testing, has been used in a constraint satisfiability system to help make conjectures in large search spaces, e.g. Ramsey theory (see Hsiang and Huang [HH98]). A benefit of the generating function and cycle index polynomial method for model generation is that it is not tailored specifically to binary valuations. The method is easily generalizable to models for arbitrary algebras over sets larger than two.

In program verification or automated deduction, one often comes across the situation where multiple lemmas or cases are needed with virtually identical structure, except that some variables are permuted. To save effort in producing a practically identical proof, a human prover is able to invoke the concept “without loss of generality”, which lets a reader know that an assumption has been made which really makes no difference. Such an assumption, especially those for proofs of program correctness, involve only a permutation of variables. For example, in algorithms for manipulating complex data structures like red-black trees, there are many cases and sub cases which refer to left and right branches in a symmetrical manner. This situation can use a rule of inference called “by symmetry”. A lemma about the symmetry of the variables is made, and then lemmata for each unique instance is made:

\[
\forall \sigma \in G_x : P(\sigma(x)) \cong P(x) \quad \forall \text{ unique } x : P(x) \\
\forall x : P(x) \quad \text{Sym}(G_x)
\]
where \( x \) is a tuple of variables, \( \sigma \) is a permutation of \( x \), and \( G_x \) is a group acting on \( x \). This is mostly a convenience for a human using an interactive proof editor, where they can more easily see a strategy where taking advantage of symmetry would reduce the reproduction of isomorphic cases.

### 3.7 Other comments

Taking advantage of symmetry in model checking has been addressed before (Clarke et al. [CFJ93], Clarke and Jha [CJ95] and Emerson and Sistla [ES93]) all in OBDDs, but none of these papers gives any indication as to how the models are generated. The general problem of symmetry in propositional logic is addressed in Benhamou and Sais [BS94] and Peltier [Pel98] but with no mention of groups, only experimental evidence that it helps, and again no algorithms for model generation are presented.

### 3.8 Summary

It is very difficult in general to determine the full symmetry group of a boolean formula; the complexity is most likely somewhere strictly between \( \text{coNP} \) and \( \Sigma^p_2 \). But specific groups can be very easy to find and verify, using graph isomorphism techniques. Generating specific groups is very fast, usually an \( O(1) \) ‘next’ operation, through some known Gray code technique.
CHAPTER 4

Conclusions and Open Problems

4.1 Conclusions

The first part of the thesis applies combinatorial techniques to solve the probability of satisfiability. It gives a closed form formula for the number of satisfiable \(k\)-CNF formula parameterized by the number of variables and the number of clauses. It also gives some asymptotics under certain initial conditions. The second part develops algorithms for generating distinct models of propositional formulas when the variables can be permuted.

4.2 Open problems

simplify the counting Theorem 2.8.1 does give an exact count, but in doubly exponential time. It is expected that these formulas can be simplified significantly, at least to a singly exponential number of terms.

better thresholds In Equation 2.17 we found the dominating term of the summation, and this gave us an easy upper bound. Each successive term will give a better and better approximation to the function and therefore better bounds on the threshold.
other models Though the study of the selection models for literals helped in analyzing selection of clauses, we only addressed the larger coaccounting problem of the whole formula using the traditional literal selection model. The other models would produce more complex summations, but progress in the previous two areas would help in the analysis here.

other syntax The result of section 2.10, is surprising. Can we generalize this result like that for k-CNFSAT? Is there a general solution to the non-linear systems on an arbitrary number of variables $v$? As mentioned, the approach there is highly intractable even with the aid of computer algebra algorithms. A promising approach that may not involve so much computation is multivariate Lagrange Inversion, which can extract coefficients directly from such systems of equations without having to solve the system or eliminate variables first using Groebner bases.

P-complete problems CNFSAT is not the only NP-complete problem that exhibits a sharp threshold. The parsimonious reductions that establish hardness for ‘counting’ complexity classes could possibly be applied to the dual problem here to provide counting and threshold information for these other problems.

symmetry Pólya enumeration is so successful at enumeration with respect to symmetries, but the unique representatives of the classes which are counted are not constructed. A whole theory of Pólya enumeration giving constructions needs to be developed as opposed to ad hoc methods for each presentation of a group.
References


VITA

Mitch Harris was born in 1964. He attended Cornell University from 1982 to 1986. He received the degree of Bachelor of Arts in Computer Science there in 1986. He worked as a programmer at the Illinois Criminal Justice Information Authority on software for crime analysis and spatial statistics and at SPSS, Inc. on statistical software. He received the degree of Master of Science in 1998 from the University of Illinois at Urbana-Champaign.