The Complexity of Automated Reasoning

by

André Vellino

A Thesis submitted in conformity with the requirements for the Degree of Doctor of Philosophy in the University of Toronto

© 1989 André Vellino

Abstract

This thesis explores the relative complexity of proofs produced by the automatic theorem proving procedures of analytic tableaux, linear resolution, the connection method, tree resolution and the Davis-Putnam procedure. It is shown that tree resolution simulates the improved tableau procedure and that SL-resolution and the connection method are equivalent to restrictions of the improved tableau method. The theorem by Tseitin that the Davis-Putnam Procedure cannot be simulated by tree resolution is given an explicit and simplified proof. The hard examples for tree resolution are contradictions constructed from simple Tseitin graphs.

Acknowledgements

I would like to thank Steven Thomason, Marvin Belzer, David Goodman and William Older for their comments on early drafts of my thesis. I am very grateful to John Bell, James Brown, Hector Levesque, and John Slater for serving on my committee and also to William Seager for his equally interesting comments and his continual encouragements. But most of all, I wish to thank Alasdair Urquhart not only for his guidance and inspiring enthusiasm, but also for his patience and generous help. This work would not have been possible without him.

Table of Contents

Acknowledgements	iii				
Table of Contents					
List of Figures					
List of Theorems and Lemmas	viii				
1 Introduction	1				
1-1 Automated Theorem Proving	2				
1-2 Is P = NP?					
1-3 Overview of the thesis					
2 Basic Notions	8				
2-1 Propositional Calculus	8				
2-2 Proof Systems	10				
2-3 Graphs and Trees	12				
2-4 Measures of Complexity	15				
2-4.1 Simulation	16				
2-4.2 Proof Length vs. Search Space	17				
3 Analytic Tableaux and Resolution Trees	19				
3-1 Analytic Tableaux	19				
3-1.1 Clash Restricted Analytic Tableau	22				
3-1.2 Improved Analytic Tableau	26				
3-2 Resolution Proofs and Tree Proofs	28				
3-2.1 Lengths of Resolution Proofs	30				
3-3 Tree resolution and analytic tableaux	34				
4 Refinements and Extensions of Resolution	38				
4-1 The Davis-Putnam Procedure	39				
4-2 Regular Resolution	40				
4-2.1 Splitting Rule	44				
4-3 Extended Resolution	45				
5 Linear Resolution					
5-1 Linear Resolution	47				
5-1.1 Linear Resolution p-simulates Tree Resolution	50				
5-2 SL-Resolution	53				
5-2.1 Linear Resolution and Analytic Tableaux	62				

6	6 The Connection Method		
	6-1 The Connection Method	66	
	6-1.1 Structured Matrices	68	
	6-2 Connection Method and Analytic Tableaux	72	
	6-3 Conclusions	77	
7	79		
	7-1 Graph Clauses	79	
	7-2 Tseitin's Theorem	85	
	7-2.1 Examples	88	
	7-3 Tree Resolution cannot simulate DPP	91	
8 Conclusion		95	
	8-1 Open Problems	97	
9	APPENDIX	99	
1(10 BIBLIOGRAPHY 10		

List of Figures

Figure 1-3.i	6
Figure 2-3.i	13
Figure 2-3.ii	13
Figure 2-3.iii	14
Figure 3-1.i	20
Figure 3-1.ii	21
Figure 3-1.iii	21
Figure 3-1.1.i	23
Figure 3-1.1.ii	23
Figure 3-1.1.iii	24
Figure 3-1.1.iv	25
Figure 3-1.1.v	25
Figure 3-1.2.i	27
Figure 3-2.i	29
Figure 3-2.ii	30
Figure 3-2.1.i (a)	32
Figure 3-2.1.i (b)	32
Figure 3-3.i	35
Figure 3-3.ii	36
Figure 3-3.iii	37
Figure 4-2.i	41
Figure 4-2.ii	42
Figure 4-2.iii	43
Figure 4-2.1.i	45
Figure 5-1.i	49
Figure 5-1.1.i	52
Figure 5-1.1.ii	53
Figure 5-2.i	58
Figure 5-2.ii	61
Figure 5-2.1.ii	64
Figure 5-2.1.iii	64

65
74
76
76
81
83
84
87
92
92
94

List of Theorems and Lemmas

Lemma	3-2.2.	33
Theorem	3-3.1.	34
Lemma	4-2.1.	42
Lemma	4-2.2.	43
Lemma	5-1.1.	49
Lemma	5-1.2.	50
Lemma	5-1.1.1	51
Theorem	5-2.1.	60
Theorem	5-2.1.1.	63
Theorem	6-2.1	72
Lemma	7-1.1.	83
Lemma	7-1.2.	83
Lemma	7-1.3	84
Lemma	7-1.4	84
Theorem	7-2.1	86
Theorem	7-3.1	93

CHAPTER 1

Introduction

One proof method for a system of logic is more powerful than another to the degree that it simplifies the task of producing derivations of theorems. For example, in Mendelson's axiomatic system for the propositional calculus [Mendelson 1964] the proof of even a simple tautology such as $A \rightarrow A$ can be relatively long, yet trivial to prove in a natural deduction system.

The intuition that some proof systems are more powerful than others is a natural one but it is not obvious how precisely this relation should be defined. The view taken in this thesis is that the power of a proof system is inversely proportional to its proof complexity, i.e. to the rate of growth of the length of the shortest possible proofs for theorems in that system relative to the size of these theorems. Although there are other possible measures, such as how difficult proofs are to discover or how long they take to produce mechanically, proof complexity is more general because it sets absolute limits on these other measures.

Until recently, the relative complexity of different logical systems, as measured by the lengths of the proofs they produced, had been given scant attention. The growing interest in the existence of a hierarchy of complexity among theorem proving methods has come from two fields in computer science: automated theorem proving (ATP) and the theoretical question "is P=NP?". This introduction briefly describes these motivations and presents an overview of the main results of the thesis.

1-1 Automated Theorem Proving

Although the earliest computer programs for proving theorems dealt mainly with propositional logic [Newell, Shaw and Simon 1957, Davis and Putnam 1960], the ultimate aim of this research was to develop theorem proving procedures for the full (first order) predicate calculus. The grand goal of theorem proving, at least in the early years, could be viewed as the culmination of the rationalist dream: the mechanization of thought.

The lack of success of early automatic theorem provers was initially thought to be caused by inadequacies in the basic proof searching strategies. The optimistic hope was that better search strategies would increase the efficiency of these programs to the point where they could emulate human thinking. What was not known at the time was that the problem of producing short proofs for theorems in the propositional calculus, let alone the predicate calculus, is intrinsically difficult.

There are basically two kinds of complexity measures of interest in automated theorem proving: how much time it takes for a procedure to find a proof and how much space (computer memory) is required to search for it. One reason that proof length has not been a favoured measure is that there is a practical distinction between the "worst case" and the "average case". Even if a theorem proving procedure is very inefficient at proving some classes of tautologies, it may still be considered acceptable if most theorems can be proved efficiently most of the time. Thus although proof length provides valuable indications of the inherent limitations of a proof procedure, the focus of complexity analysis in the field of ATP has been driven primarily by the goal of reducing the average amount of computing resources (time and space) needed to prove most theorems [Goldberg 1979].

1-2 Is P = NP?

The study of worst-case proof complexity of automatic theorem proving procedures is also motivated by questions regarding the limits of computability. If it can be shown that there exists a decision procedure which can verify whether or not a formula F is a tautology in a number of steps that is a polynomial function of the length of F, then a major open question in computer science, "is P = NP?", would be answered [Cook 1971a].

If one makes a distinction between problems that a computer program can always solve in an amount of time which is a polynomial function of their length—problems in the class P—and another class of problems—the class NP—for which it is possible only to *verify*, in polynomial time, whether or not a proposed solution is indeed a solution to the problem, then the question arises whether P is the same as NP.

Consider the following questions:

- (Q1) Does a formula *F* in the propositional calculus contain an even number of occurrences of each of its propositional symbols?
- (Q2) Does there exists a set of truth-value assignments to the atomic propositions in a formula *G* that makes *G* true?

Since (Q1) is a decision problem that can be solved in polynomial time (a polynomial function of the length of *F*), it belongs to the class *P*. However, the problem (Q2), also known as the SATISFIABILITY problem, belongs to the class *NP*. This is because it is possible to verify whether or not a given set of truth value assignments (a proposed solution) satisfies *G* in an amount of time which is a polynomial function of length of *G*. While the class *NP* clearly contains *P*, it may (or may not) contain problems not contained in *P*. In other words the question of whether or not P=NP is an open one.

The complements of the classes P and NP—*co*-P and co-NP—can be explained analogously. To take the complements of the examples above, the problem (co-Q1) of whether a formula F does *not* contain an even number of occurrences each of its propositional symbols has the same complexity as (Q1), i.e. P = co-P. However, the problem (co-Q2) of whether there is *no* truth value assignment to the propositional symbols in a formula G that satisfies it, is not known to belong to NP. Since this problem is equivalent to deciding whether *all* truth value assignments to the propositional symbols in $\sim G$ satisfy it, the problem is referred to as the TAUTOLOGY problem.

Some problems in the class *NP*—the class of *NP-complete* problems are generic in the sense that if *NP-P* is non-empty then they are characteristic elements of that class. The SATISFIABILITY problem has the distinction of being the first one known to belong to the class *NP-complete* [Cook 1971a].

It follows that the TAUTOLOGY problem is *co-NP-complete* and any complexity result for it applies to all problems in *co-NP*. The importance of the class *co-NP* is that, if $NP \neq co-NP$ then $P \neq NP$. Now, according to a result in

[Cook and Reckhow 1979] NP = co-NP if and only if there exists an efficient (polynomial) proof system for TAUTOLOGY. Thus, one way to show that $P \neq NP$ is to show that there is no polynomial time proof procedure for all tautologies.

1-3 Overview of the thesis

To show that proving tautologies is intractable for any proof procedure, it may be helpful to know that specific proof procedures can be "defeated", i.e. that there are tautologies which are hard for them to prove. For some proof systems, it is possible simply to *construct* such hard-to-prove tautologies, indeed there may be constructive examples that are hard to prove for all proof systems. (Experience with particular examples that defeat specific proof systems seems to indicate, however, that constructive techniques may not be powerful enough to settle the TAUTOLOGY question.)

Rather than showing that a particular tautology or class of tautologies is hard for a proof system A, it is usually simpler to show that another proof system B, already known to be "defeated" by hard examples, *simulates* A. That is, one shows that any proof α in A can be transformed into a proof β in B which proves the same tautology and that the length of β is a polynomial function of the length of α . Conversely, one can also show that proof system A *cannot* simulate another system B by showing that there are tautologies that are hard for A are not hard for B, i.e. B is strictly more powerful than A.

The *simulates* relation (defined in section 2.4.1) imposes a partial ordering among proof systems, and, although many simulation relations are known [Reckhow 1975], there are still a number of gaps in the literature. In

particular, there are several proof procedures used in automated theorem proving for which adequate complexity measures are not known. This thesis attempts to fill these gaps for some specific proof procedures. A schematic diagram of the main simulation results is drawn in figure 1-3.i.



The proof that tree resolution simulates the improved tableau procedure is given in chapter 3. SL-resolution and the connection method are shown to be equivalent to restrictions of the improved tableau method in chapters 5 and 6 respectively.

In chapter 4 we describe the Davis-Putnam procedure (DPP) and some known results about regular resolution. In chapter 7 we present an explicit and simplified proof of Tseitin's theorem that the DPP is strictly more powerful than tree resolution (and hence the other three) by constructing examples of tautologies that are hard for tree resolution, but not hard for the DPP. Chapter 8 concludes with some speculations about the general significance of complexity results for automated reasoning.

CHAPTER 2

Basic Notions

The literature on theorem proving methods explored in this thesis employs a notation that differs somewhat from the conventional terminology in philosophical logic. We begin, therefore, with some definitions for basic notions in the *propositional calculus*. This is followed by a brief introduction to some elementary concepts in *graph theory* that are useful in the analysis of proof complexity as well as in the construction of interesting classes of contradictory propositions. Since later chapters explore the differences in the lengths of derivations for different proof systems we also introduce the notions of *proof length, efficiency* and *simulation*.

2-1 Propositional Calculus

We say that the symbols of the propositional calculus **PC** belong either to the set of *logical constants*, comprising the unary operator \sim and the *binary connectives*:

 $\{ \lor, \&, \equiv, \rightarrow \}$

or to the infinite set of atomic propositional variables:

 $\{a, b, c, ..., a_1, b_1, c_1,\}.$

A *literal* in **PC** is either a propositional variable l or its negation $\sim l$. The *complement* of a literal $\sim l(l)$ is $l(\sim l)$. A *formula* is either a literal or an expression of the form $\alpha \otimes \beta$, where α and β are formulas and \otimes is one of the binary connectives.

A *clause*, denoted by upper case letters { $A, B, C..., A_1, B_1,...$ } is either a finite formula of the form $\alpha \lor \beta$, where α and β are either literals or clauses, or the formula containing no literals, \emptyset , referred to as the *empty clause*. The empty clause is also called the *contradictory clause*. A clause with just a single literal is called a *unit clause*.

Since the literals contained in a clause are all related by the same connective \lor , we will often represent the clause $\alpha \lor \beta$ either in the abbreviated form $\alpha\beta$ or as a set of literals { $a_1, a_2, ..., b_1, b_2...$ }. A set of clauses { A, B, C... } is considered to be a conjunction of clauses, or equivalently, a formula in Conjunctive Normal Form (CNF).

The set of variables S contained in a clause C may be given a truth value assignment (*tva*) by a map $T: S \longrightarrow \{true, false\}$. Let V be a *tva* to the variables S in C. Then the *truth value* of C is a function $\Im(V)$ into the set $\{true, false\}$. The total number of *tva's* to C is 2^n where n is the number of variables in C, |S|. A complete listing of all the possible *tva's* to variables of a clause C and their transformation by the truth-functional connectives of C is the *truth-table* for C.

A clause C (or a set of clauses Σ) is *satisfied* by a *tva* V to the variables in C (Σ) iff $\Im(V)=true$. C (Σ) is *satisfiable* if there is some *tva* that satisfies C(Σ). If all the possible *tva's* for C (Σ) satisfy C (Σ), then C (Σ) is a *tautology*. C(Σ) is *falsified* by a *tva* V to the variables \Im in C (Σ) iff $\Im(V)=false$. If all the *tva's* falsify C (Σ), then C (Σ) is *inconsistent* or *unsatisfiable*. If Σ is an inconsistent set of clauses and $\Im(V)$ is a *tva* to the variables S in Σ then $\Im(V)$ is *critical* for a clause $C \in \Sigma$ if $\Im(V)$ satisfies all the clauses in Σ except C. Σ is *minimally inconsistent* if there exists a critical *tva* for every clause in Σ .

2-2 Proof Systems

Following [Cook and Reckhow 1979] we define a proof system as follows. For a finite alphabet Γ , let Γ^* be the set of finite strings over Γ and a *language* Λ over Γ^* be any subset of Γ^* . If $m \in \Gamma^*$ then we say that |m| is the number of occurrences of symbols in m. Given a function $f:\Gamma_1^* \longrightarrow \Lambda$, $\Gamma^* \supseteq \Gamma_1^*$, we say that f is *polynomial-time* computable if f is computable by a deterministic Turing machine in a number of steps bounded by some polynomial function of |m|. A *proof system* for a language Λ is a polynomialtime function f from Γ_1^* onto Λ .

This general notion of a proof system subsumes all the methods described in this thesis. The proof of a tautology *C* in the language of the propositional calculus **PC**, for example, is a string of characters S that can be mapped into *C* in a polynomial number of steps: i.e. f(S) = C (the last line of the proof.) For instance, the truth-table method for ascertaining that a clause *C* is tautologous (unsatisfiable) simply tabulates all the possible truth value assignments to the variables in *C* to verify that all of the combinations of truth value assignments to the literals in *C* satisfy (falsify) *C*. If the truth-table is taken as the string, then even though the truth-table has a size 2^n (where *n* is the number of literals) the verification that the truth-table is the truth-table for *C* is a polynomial function of its size. Similarly, the *analytic tableau* and

connection method systems described in chapters 3 and 6 can be regarded as proof systems in this general sense.

An *axiomatic* proof system such as the *resolution method* consists of a privileged set of clauses called *axioms* and a set of truth-preserving *rules of inference* that allow the construction of a clause from other clauses and / or axioms. A *proof* for a clause C, the conclusion, is a sequence of clauses $C_1...C_n$ such that

- (1) $C = C_n$, and;
- (2) C_i, for 1 ≤ i ≤ n, is either an axiom or derived from other clauses using a rule of inference.

An inference of *C* from a set of clauses Σ is *truth-preserving* if and only if all the *tva's J* on the variables V_{Σ} in Σ such that $\Im(J) = true$ are such that $\Im(K)=true$, where *K* is the set of truth-value assignments to the variables V_C in the clause *C*.

The special class of proofs whose axioms are an inconsistent set of clauses are called *refutations* and their conclusion is always the contradictory clause \emptyset . Showing that the contradiction follows from a set of clauses using a truth-preserving rule of inference is therefore equivalent to showing that the negation of the formula in CNF is a tautology.

2-3 Graphs and Trees

We define some notions from graph theory [Bondy and Murty 1976] because some interesting classes of tautologies are best represented by graphs (see chapter 7). These same concepts are also useful in connection with the analysis of resolution proof trees and analytic tableaux introduced in the next chapter.

A graph G is a set V of vertices that are connected by a set of edges E: G = (V, E). The number of elements in V (E) is denoted |V| (|E|). If |V| and |E|are finite, the graph is said to be *finite*. An edge e in a graph is associated with two vertices v_i and v_j called its end-points. A graph that has distinct edges with the same end-points is said to have multiple edges. A graph may also contain loop edges whose end-points are the same vertex, however graphs containing loops will not be considered further.

The two end-points v_i and v_j of an edge *e* are said to be *adjacent* and *e* is *incident* with v_i and with v_j . The *degree* $\mathbf{d}(v)$ of a vertex *v*, is the number of edges that are incident with *v*. The degree $\mathbf{d}(G)$ of a graph G = (V, E), is the maximum of $\mathbf{d}(v_i)$ for all $v_i \in V$.

A walk between v_j and v_k in the graph *G* is a sequence $\langle v_j, e_j, v_{j+1}, e_{j+1}, ..., v_{k-1}, e_k, v_k \rangle$ of alternate vertices and edges such that e_i is incident with v_i, v_{i+1} , for $0 \le i \le k$. If the vertices and edges of a walk are distinct then it is a *path*. A *circuit* is a non-empty path to which the first element has been appended (i.e. whose first and last vertices are the same). Two vertices are *connected* if there is a path between them. We say a graph is *connected* if all its vertices are connected. Figure 2-3.i shows a connected graph



Figure 2-3.i A connected graph $G = (\{v_1, v_2, v_3, v_4\}, \{a, b, c, d\})$

A directed graph or digraph is a graph G = (V, A), where V is a set of vertices and A is a set of ordered pairs of elements called *arcs* or directed edges. If a is an arc (v_i, v_j) then v_i is the *tail* of a and v_j is the *head* of a. In a diagram, arcs are drawn with arrows, as in figure 2-3.ii



Figure 2-3.ii A digraph $G = (\{v_1, v_2, v_3\}, \{(v_3, v_2,), (v_2, v_3), (v_2, v_1)\}$

The *indegree* $\mathbf{d}^{-}(v)$ of a vertex *v* is the number of arcs whose head is *v*. The *outdegree* $\mathbf{d}^{+}(v)$ of a *v* is the number of arcs whose tail is *v*. For example v_2 in figure 2-3.ii has indegree 1 and outdegree 2.

A *tree* is a connected graph with no circuits. A node is a vertex on a tree. A *rooted* tree is a tree with a distinguished node, the *root*. The *leaves* of a rooted tree are all the vertices, other than the root, having degree 1. A tree

with a root of degree 2 and whose non-leaf nodes are of degree 3 is a *binary* tree. A path from a node *n* to a leaf *l* is a *branch* from *n* to *l*.

It will often be useful to think of a logical proof in terms of a *directed tree*, i.e. a directed graph with no directed circuits. The root of a directed rooted tree has indegree 0 and its leaves have outdegree 0. The *children* of a node *n* are the heads of the arcs whose tail is *n* and the *parent* of a node *n* is the tail of the arc whose head is *n*. A *descendant* node is either a child node or one of its children whereas an *ancestor* node is either a parent node or one of its ancestor nodes.

We adopt the convention of drawing a tree so that the leaves of the tree are lowermost and each subsequent connected edge is positioned above the previous one. All the nodes immediately below a given node are its children whereas the node above a given node is its parent. Most of the directed trees considered in this thesis will be drawn with the arrows implicit.



Figure 2-3.iii A directed rooted tree

It is also useful to define a specific order for visiting the nodes of a tree. In particular, a *postorder* traversal of a tree traverses the sub-trees of the first (left-most) sub-tree, visits the root and then traverses the remaining sub-trees [Knuth 1968, p.316]. A leaf node is traversed by doing nothing. Thus, the postorder traversal of the tree in figure 2-3.iii is *ebfacgdh*.

2-4 Measures of Complexity

We are interested not only in the absolute *lengths* of proofs but also in the *rate of growth* of proofs as a function of the number of input clauses (axioms).

We say that for the functions f and g, f is of the order of g, $f = \mathbf{O}(g)$ if there is a constant c > 0 and some n_0 such that, for every n, $n > n_0$, $|f(n)| \le c|g(n)|$. Moreover, g grows faster than f, written f < g, if $\lim_{n \to \infty} f(n) / g(n) = 0$. On the other hand, f and g will converge if $\lim_{n \to \infty} f(n) / g(n) = c$ for some c > 0. The function f is said to be superpolynomial if p < f for every polynomial p. For example, if $p(x) = x^2 + 3x$ and $f(x) = 2^x$ then p < f. Note that the relation $f = \mathbf{O}(g)$ is an equivalence relation.

The proof that a set of clauses is inconsistent may *require* a number of steps that is a superpolynomial function of the number of input clauses. Such problems are said to be *intractable*.

Remarks: Although the distinction between polynomial and superpolynomial functions is sharp, it is not evident what importance should be given to it. For any fixed interval of real numbers we can find a polynomial function whose value is greater, in that interval, than any preassigned superpolynomial function. Hence, even though superpolynomial functions always *grow* faster than polynomial ones, they do not always have larger values for any particular problem.

For some practical problems, in computer science for example, it may be better (more efficient) to use an algorithm that suffers from a combinatorial explosion for problems greater than a fixed size rather than use an algorithm whose execution time is a polynomial function of the size of the input, but whose values get very large even for small problem sizes. For instance, the problem of Linear Programming is known to be soluble in polynomial time but the exponential time Simplex algorithm is still used in practical applications because it works well for relatively small problems [Gibbons 1985, Appendix].

2-4.1 Simulation

Given two proof systems, A and B, for a fixed language Λ , and a procedure P which transforms any proof of a clause produced by system A into a proof of the same clause in system B we say that system B simulates system A. Moreover, if P is polynomial-time computable then we say that system B p-simulates system A. In this case, B p-simulates A if P transforms a proof in system A into a proof in system B without increasing the length of the proof by more than a polynomial function of the length of A.

The idea behind the simulation results in this thesis is to show that any proof of length L, obtained in a proof system A, can be translated into another proof system B, without increasing its length by more than a polynomial function of L. But the *p*-simulates relation is not symmetric. Indeed, there are examples of inconsistent sets of clauses whose minimum refutation length is k in system B but whose minimal length in system A is a superpolynomial

function of *k*. In that case, system *B* is, from the complexity point of view, strictly more powerful than system *A*.

2-4.2 **Proof Length vs. Search Space**

It is possible for a theorem proving procedure to produce a short refutation for a set of clauses and yet require an exhaustive search of (almost) all the possible near-refutations in order to find it. In other words, a more powerful theorem proving technique might yield a shorter refutation than any obtainable by a less powerful theorem prover yet may take a long time to find it.

However, if it can be shown that a theorem proving procedure produces refutations whose minimal length is a super-polynomial function of the number of input clauses, then even if the procedure finds the shortest proof in the most direct and efficient way, the intractability of the proof procedure is guaranteed by the necessarily large size of the refutation.

In the chapters ahead we consider proof methods such as restrictions of resolution and analytic tableaux whose purpose is to achieve the efficient automation of the general methods. The set of possible refutations for the refinement of a proof technique is generally different from the set of proofs that are possible without the restriction. Moreover, it is possible that the lengths of minimal proofs deduced from a restricted method is of the same order—and may even coincide with—the minimal proofs obtainable without the restriction. On the other hand, restrictions may also have minimal proofs that cannot p-simulate minimal proofs obtained without the restriction.

Chapter 3

Analytic Tableaux and Resolution Trees

This chapter provides an account of two basic proof methods for the propositional calculus: analytic tableau, and tree resolution. We then define refinements of the analytic tableau method—ancestor and parent restrictions, and the improved tableau—and compare the relative sizes of the proofs that these systems generate. We also show that tree resolution p-simulates the improved analytic tableau.

3-1 Analytic Tableaux

The method of *analytic tableau* may be defined for the general propositional calculus [Smullyan 1968], but here we will be considering only analytic tableaux for *sets of clauses*.

An *analytic tableau* θ for a set of clauses Σ , is a tree such that all the nodes in θ other than the root node are labelled by literals occurring in Σ and, for each interior node *k* in the tree, the set of literals labelling the children of *k* is a clause in Σ . The root node of any analytic tableau for clauses will be designated by the special symbol ϑ .

For example, the clause $ab \sim cd$ is decomposed at node k in an analytic tableau as follows:



Figure 3-1.i Clause *ab~cd* decomposed in an analytic tableau

A branch is *closed* if it contains both a literal and its complement. An analytic tableau is *closed* if all its branches close but *open* if at least one branch is not closed and all the clauses in Σ have been decomposed at some node in that branch.

It is easy to see that if Σ is a consistent set of clauses, an assignment of truth values to the literals which makes Σ true can be read off an open branch of its analytic tableau.

Conversely, all the branches of an analytic tableau for a set Σ of clauses are closed if and only if Σ is inconsistent.

For example, a proof that the clauses $\{ab, \neg ab, \neg b\}$ are unsatisfiable is



Figure 3-1.ii Closed analytic tableau for { ab, $\sim ab$, $\sim b$ }

whereas a consistent assignment of truth values to the clauses $\Sigma = \{ ab, a \sim bc, \sim c \}$ may be read off from any one of its three open branches



Figure 3-1.iii Open analytic tableau for { $ab, a \sim bc, \sim c$ }

Thus one of the tva's that make Σ true can be read off the left-most branch for instance and are T(a)=true, T(c)=false.

3-1.1 Clash Restricted Analytic Tableau

The search for minimal sized tableaux proofs, particularly for sets of non-minimally inconsistent clauses, can be quite inefficient if the tableau method is unrestricted. One restriction of the general tableau method that might reduce the search space is to impose the rule that each clause decomposition closes a branch in the tableau. This is equivalent to the condition that the clause decomposed at each node k in the tableau contain at least one literal that clashes with (i.e. is the complement of) a literal labelling k or an ancestor of k. We will call such a tableau an *ancestor clash restricted* (*ACR*) analytic tableau.

A more stringent clash restriction is to specify that some literal in every decomposed clause clash with the literal labelling the *parent* node, i.e. that every non-leaf node k in the tableau is labelled with a literal that clashes with a literal labelling a child of k. We say that such a tableau is *parent clash restricted* (*PCR*). Both *ACR* and *PCR* analytic tableaux methods are complete: a set Σ of clauses is inconsistent if and only if there exists both an *ACR* and a *PCR* analytic tableau for Σ .

Contrast the following analytic tableaux for the refutation of the set $\{ab, \sim ab, \sim b, \sim ac, ef\}$.



Figure 3-1.1.i Tableau with no restrictions for {*ab*, ~*ab*, ~*b*, ~*ac*, *ef* }



Figure 3-1.1.ii Tableau with ancestor clash restriction for {*ab*, ~*ab*, ~*b*, ~*ac*, *ef* }



Figure 3-1.1.iii Tableau with parent clash restriction for{*ab*, ~*ab*, ~*b*, ~*ac*, *ef* }

These show that PCR and ACR tableaux can be shorter than unrestricted analytic tableaux. Thus, as a heuristic for searching, this restriction appears to minimize the decomposition of irrelevant clauses.

Whether or not *PCR* or *ACR* tableaux p-simulate unrestricted analytic tableaux is an open question. However, we know that some minimal unrestricted tableau are smaller than either *PCR* or *ACR* tableaux. The examples that show this are constructed as follows.

Consider a set S of $2(2^n - 1)$ distinct positive literals from which the set C of $2^n - 1$ clauses, each containing exactly two distinct literals, such that each literal in S occurs only once in C. Then construct an open analytic tableau containing only one decomposition of each clause in C. This tableau can be closed by the set of 2^n clauses each containing *n* negative occurrences of literals in S. The resulting tableau T_n has $(n + 2)2^n - 2$ nodes.

For instance, the tableau on figure 3-1.1.iv is minimal and all the *PCR* or *ACR* tableaux for those sets of clauses are larger.



Minimal tableau for {ab, cd, ef, gh, ij, kl, mn, ~a~c~g, ~a~c~h, ~a~d~i, ~a~d~j, ~b~e~k, ~b~e~l, ~b~f~m, ~b~f~n}

Since each clause in these minimally inconsistent sets is decomposed exactly once, by construction, the class of tableaux T_n is minimal for $n \ge 3$. If either the *PCR* or *ACR* restriction is placed on the construction of such a tableau then the same clause must be decomposed more than once because the symmetry of the literal clashes in the tree is broken. This is illustrated by observing the start of an *ACR* tableau for T₃.



Figure 3-1.1.v Partial *ACR* tableaux for T₃

Given the start clause *ab*, the next clause beneath *a* must be one of the four clauses containing $\sim a$, say $\sim a \sim c \sim g$. The branch containing $\sim c$ can be

closed by cd in a parent clash decomposition (right hand side of figure 3-1.1.v) or after some ancestor clash (left hand side of the figure). In either case there are still two more clauses containing $\sim c$ and $\sim d$ which have yet to be decomposed (since the set is minimally inconsistent) and whose branches must be closed by another decomposition of cd.

3-1.2 Improved Analytic Tableau

There is a simple extension to the analytic tableau method that increases its efficiency as a method for proving theorems and produces a considerable reduction in the complexity of its proofs. (A short and elegant implementation of this method in the logic programming language Prolog is described in the appendix.)

We will say that an analytic tableau is an *improved* (I) analytic tableau for a set of clauses Σ if it is *completed* or *checked* which we define simultaneously by induction as follows:

- (i) A sub-tableau is completed if it is closed.
- (ii) If a branch of a sub-tableau ends in a literal *l* and there is an ancestor of this node that has a child also labelled with *l* which is at the top of a completed sub-tableau then the branch ending in *l* is checked.
- (iii) A sub-tableau is completed if all its branches are closed or checked.

For example, a completed *I*-analytic tableau for the clauses $\{ab, \neg ab, \neg b\}$ is given in figure 3-1.2.i. Compare this with the tableau in figure 3-1.ii.



Figure 3-1.2.i Completed *I*-analytic tableau for { *ab*, ~*ab*, ~*b* }

To show the soundness of this method it is sufficient to observe that any completed *I*-analytic tableau can be transformed into a closed tableau by replacing every check mark by a closed sub-tableau containing no check marks (check marks cannot justify each other cyclically). Without loss of generality, we will assume that *I*-analytic tableaux are constructed so that the check marks occur to the left of the nodes by which they are justified. This is always possible since the literals in each clause are not order sensitive.

Remarks: Checking a branch that ends in a literal *l* simply has the effect of reporting (or delaying) the justification for closing that branch to the decomposition of clauses on another branch ending in *l*, provided that both occurrences of the literal have the same ancestor. The checking of a literal always reports its decomposition to a literal belonging to an ancestor clause and effectively merges nodes in the tableau, allowing them to share the closure of a sub-tableau.

3-2 Resolution Proofs and Tree Proofs

The rule of *resolution* was introduced in its present form by J. A. Robinson [Robinson 1965] as a general method for automated theorem proving. Using only this rule, any theorem (tautology) in the full propositional calculus may be proved simply by converting its negation into conjunctive normal form (CNF), i.e. a set of clauses, and deducing a contradiction.

Two clauses *C* and *D* are said to *clash* if there is a literal *l* such that $l \in C$ and $\neg l \in D$. Two clauses that clash may be *resolved* by deleting or *annihilating* the complementary literals and forming a new clause — the resolvent — by disjoining the remaining parts of the clauses. Thus, the resolution of the clashing clauses *abc* with *e*~*ca* is *abea*.

When a clause contains the same literal more than once, that literal may be *merged* : the clause *abea* merges the two occurrences of the literal *a* to form *abe*.

A resolution proof of a clause C from a set of clauses Σ is a sequence of clauses such that each clause is either an input clause (a clause belonging to Σ) or a resolvent of two previous clauses, and C is the last clause in the sequence. A resolution refutation is a resolution proof of the contradictory clause \emptyset (the resolvent of l and ~l for any literal l).

For example, a resolution proof that the clause qr follows from the set of clauses {prs, $\sim pqr$, $pr\sim s$ } is (with justifications in the right margin)

1)	prs	input clause
2)	~pqr	input clause
3)	pr~s	input clause
4)	qrs	1,2
5)	qr~s	2, 3
6)	qr	4, 5

Alternatively this proof may be represented by the *directed acyclic* graph (DAG)



Figure 3-2.i DAG representation of a proof of *qr* from {*prs*, ~*pqr*, *pr~s*}

A resolution proof (refutation) may also be thought of as a binary tree whose conclusion C (contradictory clause \emptyset) is at the root, whose interior nodes contain resolvent clauses and whose leaves are the clauses in Σ . Thus a proof of the clause qr from the set of clauses {prs, $\sim pqr$, $pr\sim s$ } can be represented as the tree:


Figure 3-2.ii Resolution Proof Tree of *qr* from {*prs*, ~*pqr*, *pr~s*}

Note that this tree proof differs from the graph above in that the input clause $\sim pqr$ occurs twice. In general, any clause used more than once in a tree proof must be re-derived whereas in the linear (or graph) representation clauses that have already been proved may be referred to anywhere further down the proof.

Note that if { $A_1, A_2, ..., A_n, P$ } is an unsatisfiable (inconsistent) set of clauses then a resolution refutation is a proof that the clause $\sim A_1 \vee A_2 \vee ... \vee A_n \vee P$ is a tautology. Thus any tautology expressed in DNF may be proved using resolution.

3-2.1 Lengths of Resolution Proofs

We say that the *size* of a binary resolution proof tree T, L(T), is the number of leaf nodes in T. This number is a good gauge of the size of the whole tree because the total number of nodes in T is simply given by 2L(T)-1. We contrast this measure with a measure for the *linear length* of a proof, N(T), which is simply the total number of distinct nodes in the proof tree T, i.e. the number of applications of the resolution rule plus the number of input clauses. The number N(T) can also be thought of as the number of nodes in the DAG representation of T.

The size of a *minimal* resolution proof tree T of clause C from the set of clauses Σ is

$$L_C(\Sigma) = \min \{L(T): T \text{ is a tree proof of } C \text{ from } \Sigma\}$$

Similarly the length of a minimal tree proof of *C* from Σ is

 $N_C(\Sigma) = \min \{ N(T): T \text{ is a tree proof of } C \text{ from } \Sigma \}.$

We say that clause A subsumes clause B if all the literals in A are literals in B; i.e. A logically implies B. Now we prove a lemma about minimal resolution refutations:

Lemma 3-2.1.

No minimal refutation contains two clauses A and B such that (i) A subsumes B and (ii) A is used in deriving B.

proof: By induction on the number of literals in *B*, we show that if (i) and (ii) are true, there is always a shorter refutation obtained by replacing the sub-proof of *B* by the sub-proof of *A*. In the base case where B=l, *A* is either \emptyset or *l*. In either case the tree obtained by substituting the sub-tree whose root is *B* by the sub-tree whose root is *A* is smaller.

For the inductive case (see figure 3-2.1.i), let $B=A \cup X \cup l$. Now, delete each occurrence of l between B and the root (\emptyset) and remove all the sub-trees that resolve on literals contained in clauses containing $\sim l$. This transforms B to $B' = A \cup X$. By induction

hypothesis, there is a smaller proof tree (both in L and N size) with the sub-proof of A substituted for the sub-proof of B.



Corollary 3-2.1.: No minimal refutation contains a tautology.

proof: Suppose a minimal refutation contains a tautologous clause $p \sim pB$. Then there must be a resolution of this clause by some other clause say $\sim pA$ to form $\sim pBA$. But since $\sim pA$ subsumes $\sim pBA$ the original refutation is not minimal.

Now we prove a theorem due to G. S. Tseitin [Tseitin 1968] that relates $L_C(\Sigma)$ and $N_C(\Sigma)$, namely;

Lemma 3-2.2.

(i)
$$\frac{N_C(\Sigma) + 1}{2} \leq L_C(\Sigma)$$

(ii)
$$(3/2)^{N_C(\Sigma)-1} \geq L_C(\Sigma)$$

proof: (i) Since the number of steps in the linear representation of a minimal resolution proof can never exceed the number of nodes in the tree T of that proof, and that the number of nodes in the tree is $2L_C(\Sigma) - 1$, $N_C(\Sigma) \le 2L_C(\Sigma) - 1$.

(ii) We show this by induction on the length of the proof of *C* from Σ , $N_C(\Sigma)$ (henceforth *n* for short). For the base case n = 1, $L_C(\Sigma) = 1 = (3/2)^0$. Assume the theorem is true for $k \le n$ and let C = AB be obtained by resolution on a literal *p* from two previous clauses of the form $A \sim p$ and Bp. (Since there are at least 3 clauses in the proof, $3 \le n$.) By inductive assumption, each of these two clauses have proofs for which the theorem holds. In the worst case, these two clauses immediately precede *C*, i.e.

•

There are two cases to consider. If either $A \sim p$ or Bp is an *axiom* then

$$L_C(\Sigma) \le 1 + (3/2)^{n-1} \le (3/2)^n = (3/2)^{N_C(\Sigma)}$$

If, on the other hand, both $A \sim p$ and Bp have been derived from previous clauses, we know by inductive hypothesis that $N_{Bp}(\Sigma) \leq n-1$. By the corollary to lemma 3-2.1, $A \sim p$ cannot be derived by resolving Bp with any clause which clashes with Bp (otherwise *A* would contain *p* and $A \sim p$ would be a tautology) and it follows that $N_{A \sim p}(\Sigma) \leq n-1$. Thus

$$L_{C}(\Sigma) \leq 2(3/2)^{n-2}$$

< $(3/2)^{2} (3/2)^{n-2}$
= $(3/2)^{n}$
= $(3/2)^{N_{C}(\Sigma)}$ •

3-3 Tree resolution and analytic tableaux

We now prove the theorem that tree resolution p-simulates improved analytic tableaux.

Theorem 3-3.1.

Let θ be a tableau using the set of clauses Σ . Then there is a tree resolution proof T of a sub-clause of C from Σ where C is the disjunction of literals found at the leaves of incomplete branches of θ and $L(T) \leq I(\theta)$, where L(T) is the number of leaf nodes in T and $I(\theta)$ is the number of internal nodes in θ .

proof: The theorem is proved by an induction on the depth of analytic tableau.

case 1 : for the tableau θ of depth 1 we have only one clause decomposition:



Figure 3-3.i

none of whose branches is either checked or closed, and thus the corresponding tree resolution proof is simply the clause

$$a_1 a_2 \dots a_n$$

inductive case: Assume the theorem is true for tableaux of depth less than or equal to *m*. Let the tableau θ of depth *m*+1 be as in figure 3-3.ii and let P_k be the set of literals at the leaves of the open branches of θ_k . We can assume, without loss of generality, that the literals in θ_k which are *checked* occur only in $\{a_{k+1},...,a_n\}$ (see section 3-1.2). The claim is that there exists a tree resolution proof of a sub-clause *C* of $\bigcup \{P_1...P_k...P_n\}$ of the right size.



Now delete the top clause $\{a_1 ... a_k ... a_n\}$, from θ and remove the corresponding checks and crosses from the sub-tableaux $\theta_1 ... \theta_k ... \theta_n$. This produces a new set of sub-tableaux $\theta'_1 ... \theta'_k ... \theta'_n$ containing new open branches. In θ'_k for example, the literals at the newly open branches must be a subset of { $\sim a_k, a_{k+1},...,a_n$ }. Thus the set of literals at the leaves of incomplete branches of θ'_k is a subset, C_k of

$$P_k \cup \{ \sim a_k, a_{k+1}, ..., a_n \}$$

By induction hypothesis, there exists a tree resolution proof T_k of the clause C_k , such that $L(T_k) \leq l(\theta'_k)$. There are *n* such resolution proofs $T_1, T_2,...T_n$ of clauses $C_1, C_2,...C_n$.

Now we construct the resolution proof T of C from $C_1, C_2,...C_n$ and the clause $C_0 = \{a_1...a_k,...a_n\}$ as follows. If C_1 contains $\sim a_1$ then resolve C_1 against C_0 to obtain a sub-clause of $\{a_2...a_k,...a_n\} \cup P_1$. If C_1 does not contain $\sim a_1$ then C_1 is already a sub-clause of $\{a_2,...a_k,...a_n\} \cup P_1$.

Suppose then, that we obtain a resolution proof of *D*, a sub-clause of $\{a_{k+1},...,a_n\} \cup \{P_1, P_2,...P_k\}$ in this manner and we are trying to obtain a resolution proof of *E*, a sub-clause of $\{a_{k+2},...a_n\} \cup \{P_1, P_2,...P_k, P_{k+1}\}$. If C_{k+1} does not contain $\sim a_{k+1}$ then the tree for *E* is T_{k+1} . If *D* does not contain a_{k+1} then *E* is *D*. Otherwise, *E* is obtained by resolution on *D* and C_{k+1} . When $C_{k+1} = C_n$, *E* becomes a sub-clause of $\cup \{P_1, P_2,...P_n\}$



We can therefore obtain the tree T in at most *n* resolution steps containing *n* sub-trees T_k plus the clause C_0 . Thus $L(T) = 1 + \sum_{i=1}^n L(T_k)$

and

$$l(\theta) = 1 + \sum_{i=1}^{n} l(\theta_k)$$
. Thus $L(T) \le l(\theta)$.

Corollary

Tree resolution p-simulates the analytic tableau method. •

This corollary was proved in [Cook and Reckhow 1974].

Chapter 4

Refinements and Extensions of Resolution

Although the rule of resolution is sufficient to refute any contradictory set of clauses, it is not, by itself, a practical theorem proving technique. Without some *strategy* that guides the resolution theorem prover to avoid certain paths, it may take an inordinately large number of steps to *find* a proof, even if the length of the proof itself is small.

Researchers in ATP have therefore attempted to supplement resolution with proof strategies that produce both the shortest proofs and the most efficient strategy for finding them while also retaining the completeness property of the general resolution method. One way to incorporate a proof strategy for how a proof should unfold is to apply a *refinement* or a *restriction* to the resolution method. Typically, a restriction on the resolution proof procedure reduces the number of possible resolution proofs (and blindalley non-proofs) by restricting the possible sequences of literal annihilations. What this means is that the space of all possible resolution proofs is pruned by the restriction. But the pruning may also eliminate the shortest proofs from the space of all possible proofs.

From the point of view of ATP, it would be ideal to find a restriction that guarantees the elimination of necessarily inefficient proofs but does not, at the same time, prune away the shortest possible ones. The search for such an ideal refinement of resolution has lead to a plethora of proof strategies.

Although our principal concern in later chapters will be the SL refinement of linear resolution, there are a variety of other methods and some results associated with them are introduced here for the sake of comparison and completeness.

4-1 The Davis-Putnam Procedure

The first effective ATP method for producing resolution refutations was the Davis-Putnam procedure (DPP). Although the original paper describing this procedure [Davis and Putnam 1960] antedates the discovery of resolution, it is usually thought of and presented as a resolution procedure.

One way to express the DPP, found in the secondary literature [Galil 1977] is this:

1) Pick a variable *l* from the set S of variables in Σ and delete literal *l* from S.

2) Resolve all the clauses containing *l* with the clauses containing $\sim l$ and put the resolvent clauses in Σ' .

3) If a resolution between two unit clauses of the form $\sim l$ and l takes place, the contradiction \emptyset been reached and Σ is inconsistent. Otherwise, form the set $\Sigma \cup \Sigma'$ and delete from it all the clauses containing either l or $\sim l$ and repeat the procedure with this new set. It follows from lemma 3-2.1 that there is no point in inferring a clause that is already implied by a preceding clause (either an axiom or another resolvent). Therefore, it increases the computational efficiency of this method to add a rule of *subsumption* for each of the set formation steps. The rule of subsumption allows DPP to

4) delete all the clauses in a set of clauses that are subsumed by any other clause in that set.

Refutations obtained from the DPP are characterized by the order in which variables are resolved. Moreover, the order of variable elimination from some sets of inconsistent clauses has been shown to affect drastically the lengths of some refutations. If literals are eliminated in a specific order, the resulting proofs are short, but if they are eliminated in a different order, the proofs are long [Galil 1977]. It is also known that the DPP method without the subsumption rule cannot simulate the DPP with subsumption [Cook 1971b].

4-2 Regular Resolution

The restriction of *regularity* was first discussed by Tseitin [Tseitin 1968]. The restriction applies to the sequence of resolvents in each of the branches in a resolution proof tree. A proof tree T is said to be *regular* if no branch of T has both a clause that contains a literal *l* and a clause containing its complement. If the derivation is a refutation then this amounts to the constraint that no literal be resolved on more than once in a given branch. Thus all the resolution trees produced by the DPP are regular.

For example, the proof tree in figure 3-4.i is regular whereas the tree in figure 4-2.i is not because one of its branches contains both pqr and $\sim pqr$.



Figure 4-2.i Irregular resolution proof tree of *qr* from {*prs*, ~*pqr*, *pr~s*}

The restriction of regularity is important because the tree size of regular proofs is known to be *minimal*. That is, for any contradictory set Σ of clauses there is a regular resolution refutation of minimal tree size [Tseitin 1968]. Intuitively, the reason for this is that it is always possible to prune away irregularities in an irregular proof, without increasing its size.

On the other hand, it has been shown that the linear proof *length* N(T) of a regular resolution proof tree T is not always shorter than the linear length of an irregular proof, i.e. the DAGs representing some irregular proof trees are sometimes smaller than the graph representing the minimal regular resolution proof tree [Wenqi and Xiangdong 1985].

We need two definitions to prove two useful lemmas about regular resolution trees. These lemmas are due to G.S. Tseitin [Tseitin 1968].

If *A* is a clause then *A*-*p* is the result of deleting the literal *p* from *A*.

If Σ is a set of clauses then Σ / p is the result of both deleting p from clauses in Σ containing p and deleting the clauses in Σ containing $\sim p$.

Lemma 4-2.1.

Given a regular resolution proof tree T of a clause A (the root of T) from the set of clauses Σ , and a literal $x \in A$, where $A \in \Sigma$, then there is a regular resolution proof of A-x from Σ / x .

proof: We induce on the sub-trees of T. For the base case (tree of length 0) $A \in \Sigma$. Then for any $x \in A$, $A - x \in \Sigma / x$. For the inductive case, let $A = B \cup C$ be derived by resolution on q:



Supposing that $x \in A$, then by induction hypothesis, there are regular resolution derivations of $(B \cup q)$ -x and $(C \cup \neg q)$ -x from Σ / x . Now, q cannot be x since $x \in A$ and the sub-trees T_B and T_C are regular. So we can resolve $(B \cup q)$ -x and $(C \cup \neg q)$ -x to obtain $(B \cup C)$ -x. • To illustrate this lemma, consider the derivation of the clause qr from $\Sigma = \{prs, \sim pqr, pr \sim s\}$, (see figure 3-2.ii). Then, by definition, $\Sigma / r = \{ps, \sim pq, pr \sim s\}$ and the proof tree of q from Σ / r is:



Figure 4-2.iii

Another useful lemma about regular resolution proofs is

Lemma 4-2.2.

If $\Sigma / l \models \emptyset$, then either (i) $\Sigma \models l$ and $L_l(\Sigma) \le L(\Sigma / l)$; or (ii) $\Sigma \models \emptyset$ and $L(\Sigma) \le L(\Sigma / l)$

proof: (i) If the derivation of \emptyset from Σ / l uses any input clause from Σ / l which resulted from the deletion of l from any clause in Σ , then by adding l to all the clauses in Σ from which it was removed and also to all the resolvents derived from such axioms, we obtain a derivation of l from Σ of the same L-complexity as the original derivation.

(ii) If the derivation of \emptyset from Σ / l does not involve any input clause which resulted from the deletion of *l* then it is already a derivation of \emptyset and $L(\Sigma) \leq L(\Sigma / l)$.

4-2.1 Splitting Rule

The first computer implementation of the Davis-Putnam procedure reported in [Davis, Logemann and Loveland 1962] actually describes a slightly different procedure that we may call the *splitting rule*. The splitting rule is defined as follows:

1) Pick a variable *l* from the set S of variables in Σ and delete *l* from S.

2) Create the sets of clauses $\Sigma' = \Sigma / \sim l$ and $\Sigma'' = \Sigma / l$ as defined for lemma 4-2.1.

3) For each set of clauses generated by step 1 repeat step 1 for Σ' and Σ'' so long as neither is the set containing the contradictory clause \emptyset .

The subsumption rule introduced for the DPP in the previous section may also be used for the splitting rule.

For example, if $\Sigma = \{a \sim b, \neg ab, b \sim c, \neg bc, ac, \neg a \sim c\}$, then a splitting rule refutation is



Figure 4-2.1.i

In this case, the symmetry of the enumeration tree merely reflects the relationship of the clauses to one another, not any intrinsic characteristic of the method.

The splitting rule produces what has been called [Galil 1975, Reckhow 1975] *enumeration trees*. It is known [Galil 1975] that the splitting rule, semantic trees and regular tree resolution have the same worst case complexity.

4-3 Extended Resolution

Proof strategies may refine the structure of a resolution proof but there are also ways of making it more powerful. One such method is *extended resolution* which permits definitions that abbreviate terms as steps in a proof. Since definitions may be iterated, this may result in substantially shorter proofs in some cases.

If *a* and *b* are variables in the set of clauses Σ whereas *c* is a variable not in Σ , then we can extend the set of clauses Σ with the set of clauses E generated by the conjunctive normal form for the formula $c \equiv (a \otimes b)$ where \otimes is any one of the 16 possible truth-functional binary connectives. For the case where \otimes is &, the set of clauses E is { $\sim ca, \sim cb, \sim a \sim bc$ }.

An extended resolution *refutation* of a set of clauses Σ is a resolution refutation of $\Sigma' = \Sigma \cup E$, where E is an extension of Σ .

Some tautologies may thus be efficiently proved with extended resolution, whereas they could not with resolution alone. For example the class of clauses that encodes the proposition that *n* objects cannot fit into *n*-1 holes, called *pigeonhole clauses* have been shown to be intractable for resolution [Haken 1985] yet polynomial in length for extended resolution [Cook 1976]. This class of clauses has also been shown to be tractable for Frege systems in [Buss 1987].

Chapter 5

Linear Resolution

In this chapter we examine a refinement of resolution that has been studied extensively in the field of automatic theorem proving: *linear resolution*. We give a detailed proof of the result in [Kowalski and Kuehner 1971] that linear resolution with subsumption, s-linear resolution, p-simulates regular tree resolution. We also examine a restriction of linear resolution, *SL-resolution*, interesting in part because it forms the basis of the programming language Prolog [Lloyd 1984]. We show that the SL refinement of resolution p-simulates and is p-simulated by the ancestor clash-restricted improved analytic tableau.

5-1 Linear Resolution

A *linear resolution* proof is a resolution proof where each resolvent is one of the parents of the next resolvent and the other parent is either an input clause or a previous resolvent. Resolvents neither of whose parents are input clauses are said to follow by *reduction*.

In the following we consider linear resolution proofs that obey the subsumption restriction: s-linear proofs. The subsumption restriction is placed on the reduction steps as follows. We say that an s-linear resolution proof of clause *C*, from a set of clauses Σ is a sequence $\Psi = C_1...C_n$ of clauses such that

(i) $C_n = C$.

(ii) C_1 is an input clause.

(iii) C_k , $1 < k \le n$, is derived by resolution on C_{k-1} and C_j , where C_j is not a tautology, and either

- a) $C_j \in \Sigma$ (input resolution) or
- b) $C_j \in \Psi$ and $1 < j \le k-1$ (reduction step), provided that the resolvent C_k subsumes C_j .

A linear resolution proof may be represented as a special kind of graph called a vine [Andrews 1968]. To each resolvent there corresponds an interior node on the vine and each instance of an input clause used in the proof corresponds to a leaf node. Each interior node has two parents one of which is the immediately preceding interior node and the other either a leaf node or another interior node. These arcs differentiate vines from rooted binary trees. For example for $\Sigma = \{a \sim b, \neg ab, b \sim c, \neg bc, ac, \neg a \sim c\}$ a linear resolution vine is:



Figure 5-1.i

The subsumption condition in the definition of s-linear resolution ensures that a resolvent obtained by reduction does not introduce variables that have already been eliminated by previous resolutions. For reasons analogous to the ones given for corollary 3-2.1, disallowing tautologous resolvents never lengthens a linear resolution proof.

Some general properties of linear resolution proofs should be mentioned here.

Lemma 5-1.1.

There is a linear resolution proof of *P* from Σ such that every resolvent has an ancestor clause which is a unit literal (unit resolution) if and only if there is a linear resolution proof of *P* from Σ

with no reduction steps (i.e. every clause in definition 2 (b) is a member of Σ)

proof: [Chang & Lee 1973] p.134.

Lemma 5-1.2.

If Σ is a set of clauses, none of which is a unit clause, then a linear refutation of Σ must include at least one reduction step, namely the last one.

proof: The last step is a resolution of clauses of the form p and $\sim p$ (for some literal p) and since no input clause is a unit clause both p and $\sim p$ must have been derived from Σ . Hence the reduction step. •

Note that the reduction step is what makes a linear resolution proof irregular. Hence, by the lemma, any linear refutation of a minimally inconsistent set of clauses not containing a unit clause is irregular.

5-1.1 Linear Resolution p-simulates Tree Resolution

Now we prove a version of a lemma by Kowalski and Kuehner from which it follows that linear resolution p-simulates tree resolution. It is worth noting that, in the original formulation of this lemma, the term "minimal proof" refers to what is called in this thesis a "regular proof".

Lemma 5-1.1.1

For any regular resolution tree refutation T of Σ with some leaf clause $C, C \in \Sigma$, there is a s-linear refutation R of Σ , with start clause

C, of size $L(R) \le L(T) + K$, where K is some constant times the number of interior nodes in T.

proof: (i) The base step is trivial. (ii) For the induction step: Assume the lemma is true for proofs of length < L(T). In particular let T' be the sub-tree of T with literal *a* as the root node. T' is a proof of *a* from Σ and L(T') < L(T). Using lemma 4-2.2 let T*' be the regular resolution refutation of Σ / a . By induction hypothesis there is a linear resolution refutation of Σ / a , with start node *C-a*, R*' (note that *C* cannot contain $\sim a$ since, by assumption, the proof tree T' is regular). Let R' be the linear resolution refutation that results from the restitution of the literal *a* into the clauses in R*' where it was deleted. Then R' is the linear resolution proof of *a* from Σ . By hypothesis, R' is of length L(T') + K' where K' is some constant times the number of interior nodes in T'.

By inductive hypothesis, the lemma also holds for T", the sub-tree of T that constitutes a tree resolution proof of $\sim a$ from Σ . Let R" be the proof of \emptyset from $\Sigma / \sim a$ with start clause $B \sim a$. By hypothesis, R" is of length L(T") + K". To R" perform the following surgery: Add *a* to the start clause and resolve it with clause *B* to form $B \sim a$. Then, before every resolvent C_i " whose input parent was S_i "- $\sim a$ insert the clause C_i " U { $\sim a$ } and count the inference from C_i " U { $\sim a$ } to C_i " as a reduction step. Since the number of reduction steps is at most equal to N, the number of interior nodes in T, the union of the proof R' and R" is the linear resolution proof R such that

$L(R) \leq L(R') + L(R'') + N = L(T') + K' + L(T'') + K'' + N$ = L(T) + K.

where K = K' + K'' + N

Example: Consider the regular resolution refutation tree for the set of clauses $\Sigma = \{ a \sim b, \neg ab, b \sim c, \neg bc, ac, \neg a \sim c \}$



Figure 5-1.1.i Regular resolution refutation tree for { $a \sim b, \sim ab, b \sim c, \sim bc, ac, \sim a \sim c$ }

A reconstruction of the linear refutation for S is as follows. Perform the operation described in the proof on the maximal interior head node, in this case the contradictory clause \emptyset . This breaks up the proof into two sub-trees R' and R", which, as it happens, are already linear proofs.



Figure 5-1.1.ii

5-2 SL-Resolution

Using only the definition above, linear resolution proofs do not provide any specific search strategy (e.g. which clauses to use for the first resolution, when to perform reduction steps, which literal to resolve on at any given point, etc.) to optimize the search for short resolution proofs. Without some further restrictions simple linear resolution may produce repetitious subderivations. The main results of this section will focus on the SL refinement of linear resolution—linear resolution with selection function—first introduced by Kowalski and Kuehner [1971]. SL-resolution is also a general form of OL (ordered linear) resolution [Chang and Lee 1973, p. 144-159]. The strategy underlying SL-resolution is derived from considerations common to *semantic resolution* and *set of support resolution* (see [Chang and Lee 1973, Chapter 6]). The idea with both of these restrictions is to force a choice of resolutions that will yield a contradiction quickly by imposing a strict order on the elimination of the literals. Since a partial *tva* that satisfies a group of clauses also satisfies all its resolvents it follows that a good strategy is to separate a (minimally) inconsistent set of clauses into two classes of self-consistent but mutually inconsistent clauses and resolve the clauses from each set against one another.

Applied to linear resolution, this insight means that the *tva* assigned to the literals of the set of input clauses S, must be *critical* for the start clause of the linear proof: that is, make the start clause false and each of the other input clauses true. The choice of start clause in a linear proof therefore imposes an order on the resolutions of clauses with one another. A further restriction on the progression of resolutions is to specify the order in which literals in a resolvent should be annihilated. In SL-resolution the literal scheduled for the next resolution on the vine is the right-most literal of the current clause.

To avoid redundant sub-proofs, it is useful to keep track of the previous resolutions representing a clash of tva's. This means that a certain amount of syntactic overhead must be grafted onto each clause in a linear resolution in order to maintain a record of both the ordering of the literals in the clauses and the previously resolved literals, the proof then verifies that no *tva* can be consistently assigned to all the clauses in S - {start clause} U {start clause}. We ignore the method of choosing appropriate tva's and set of support. The results below are independent of such heuristic mechanisms.

To keep track of each literal which is resolved on from an input clause and to ensure that the next resolvent inherits the semantic information about the previously resolved literal until the information is no longer necessary, every resolved literal in an SL-resolution proof is kept track of by *framing* the resolved literal in the position in which it occurs in the previous resolvent. ([Kowalski and Kuehner 1971] distinguish between A-literals, here called *framed* literals and B-literals, here called *unframed* literals, following [Chang and Lee 1973].)

A sequence of sets of literals is called a *chain*. Each literal belonging to a chain is either framed or unframed depending on whether the literal has undergone a previous resolution. Input clauses then, are chains of unframed literals and SL resolvents, in general, are chains containing framed literals.

We call each sequence of contiguous unframed literals in a chain a *cell*. Thus a chain is a sequence of cells. Note that while the order of the elements in a cell is immaterial, the order of cells in the chain is significant since it partially determines the order in which literals are resolved.

Now, if any resolvent contains both a literal framed and its complement unframed, the *reduction* operation is trivial: it consists in simply deleting the unframed literal. On the other hand, a framed literal indicates that its resolution has already been performed. Therefore a resolution on a framed literal is equivalent to using the resolvent immediately prior to the framing of that literal, thus forming an arc in the vine. If the right-most cell is empty, it may be discarded (retention of the information that previous resolutions on these literals has been performed is no longer necessary). The choice of literal for resolution depends on a *selection function* which picks out a literal from the right-most cell containing a non-framed literal. The question of which literal is to be selected on for resolution does not affect the results below.

We can now define an SL-resolution [Kowalski and Kuehner 1971] proof of *C* from a set of chains (clauses) Σ as a sequence of $C_1...C_n$ of chains such that:

- (1) $C_n = C$.
- (2) C_1 is an input clause
- (3) Ck, 1 < k ≤ n is obtained from Ck-1 by one of extension (with an input chain) or reduction (with a previous resolvent).
- (4) No two literals occurring at distinct positions in C_k have the same variable unless C_{k+1} is a reduction step (admissibility restriction).

A resolvent C_i is obtained by *extension* with an input chain B iff (a)-(c):

- (a) The right-most literal in C_{i-1} is an unframed literal.
- (b) The selected literal l in C_{i-1} (picked out by the selection function) is the complement of the left-most literal in B.
- (c) C_i is the chain obtained by concatenating C_{i-1} / l, <l> and B / ~l, in that order. The literal l in C_{i-1} is framed in C_i and every other literal has the same status as it had in its ancestor.

A resolvent C_i is obtained by *reduction* iff (a)-(c):

(a) The right-most literal in C_{i-1} is an unframed literal.

- (b) The literal l occurs both framed and unframed in C_{i-1} .
- (c) C_i is the chain obtained by deleting the unframed occurrence of l in C_{i-1} .

C_i is obtained by *truncation* iff (a)-(b):

- (a) The right-most literal l in C_{i-1} is a framed literal.
- (b) C_i is obtained by deleting every framed literal to the right of the right-most unframed literal in C_{i-1} .

C_i is obtained by *merging* iff :

- (a) C_{i-1} contains two or more occurrences of an unframed literal *l*.
- (b) C_i is obtained by deleting all the occurrences of l subsequent to the first.

For instance, an SL-resolution refutation for the set of clauses $\Sigma = \{a \sim b, ab, b \sim c, \neg bc, ac, \neg a \sim c\}$ is:



Figure 5-2.i

Contrast this figure with figure 5-1.i.

It should be noted from the definition of SL-resolution that, from the point of view of a worst case complexity measure, neither merging nor truncation nor reduction are crucial. The number of truncations will be a maximum in a refutation with no reductions. If the number of resolution steps is K, there will be at most K more truncation steps.

As for merging, it is evident that if L is the number of distinct literals in Σ then the merging procedure need be performed at most L*K times, where K

is the number of extension steps in the proof — only a linear increase. Similarly, the definition of reduction makes it clear that reduction steps can be made just by keeping track of previous resolutions (by framing literals). But since reduction steps add no new literals to the resolvent, performing the reduction step is computationally equivalent to merging literals. One need only scan the resolvent chain to check for the presence of a framed (unframed) literal and its unframed (framed) complement. Moreover, the number of reduction steps is also bounded by the number of extension steps. The worst case is when the number of merges in the proof is smallest, i.e. 0. Then each input resolution will add a maximum of N-1 literals to the chain, where N is the length of the longest input clause. Assuming that all the remaining literals are eliminated by reduction, the greatest number of reduction steps is bounded by K*(N-1) where K is the number of input clauses used in the proof times N-1.

Having observed that the merging, reduction and truncation steps can be ignored when analyzing the complexity of SL-resolution refutations, we now show that SL-resolution refutations are at least as complex as minimal regular resolution refutations. This means that the size of the tree proof, measured as the number of nodes on the tree, is of the same order as the number of resolvent chains produced by linear resolution. It follows from this result that if the tree size of a minimal regular refutation of a set of clauses Σ increases superpolynomially with the size of Σ [Tseitin 68] then so does the size of the SL proof from Σ .

Theorem 5-2.1.

Tree resolution refutations p-simulates SL-resolution refutations.

proof: Consider an SL-resolution refutation of Σ . The base case for the induction is where $\Sigma = \{p, \neg p\}$. Then the SL-resolution refutation is identical to the tree refutation. For $|\Sigma| > 2$, there are two cases.

Case (i). The last resolution is an input resolution (resolution with an input clause). This can happen just in case there is a unit clause $\sim p$ in Σ and an SL-resolution proof of the chain p from Σ . Let N be the size of the SL sub-proof of p from Σ then the size of the SL refutation of Σ is N+1. To the sub-proof of p from Σ delete every occurrence of p and omit all the chains containing $\sim p$. Then we have an SL proof of \emptyset from Σ / p of size less than or equal to N. By induction hypothesis there is a minimal regular resolution tree proof T of \emptyset from Σ / p of size **O**(N). Thus, substituting p in the input clauses of T where p was deleted, yields a regular proof of \emptyset from Σ , constructed by resolving p with the input clause $\sim p$, is **O**(N)+1, since $\sim p \in \Sigma$.

Case (ii). Let L be an SL-resolution refutation of length M such that the last resolution is a reduction step. Let $\sim p$ be the ancestor chain used in the reduction step, i.e. the proof has the form:



Figure 5-2.ii

If the first half (Part 1 in figure 5-2.ii) of the SL proof of $\sim p$ is of length K, then the same argument as in case 1 applies and there is a tree resolution proof of $\sim p$ from Σ of O(K). Let L' be the sub-proof of p in L, of length N', sandwiched in between the proof of $\sim p$ and \emptyset (Part 2 in figure 5-2.ii). We need to show that L' has a corresponding tree resolution proof of the same size. Now L' contains no unframed occurrence of $\sim p$ since by the admissibility restriction no two literals occurring at distinct positions can have the same variable unless a reduction step is made. Therefore all the occurrences of p in the spine (the non leaf nodes) of the sub-proof must be eliminated by reduction steps (since reduction is mandatory for SL). Deletion of all the occurrences of p in L' thus produces a proof of \emptyset from Σ / p of length N'. By induction hypothesis there is

60

a tree proof of \emptyset from Σ / p of size O(N'). Thus there is a tree proof of *p* from Σ of the same size O(N'). Thus there is a regular tree proof of \emptyset from Σ of size K + O(N') + 1 = O(M).•

This a more precise version of lemma 6 in [Kowalski and Kuehner 1971].

5-2.1 Linear Resolution and Analytic Tableaux

In this section we show that SL-resolution and the improved parent clash restricted (*IPCR*) analytic tableau method exactly simulate each other.

Let θ be an *IPCR* analytic tableau for the set of clauses Σ . Since the decomposition of the literals in a clause can be performed in any order, we can assume that the tableau is constructed in such a way that, at each decomposition stage, the literals are ordered so that the parent clashed literal is always *right-most* in the extension step. However, the tableaux are constructed depth-first from left to right.

Now let *C'* be the clause formed by traversing θ in reverse postorder (see section 2.3) gathering all and only the *unchecked* literals at the leaves of θ up to and including the literal on the left-most open branch. Let *C''* be the result of replacing every literal *p* in *C'* that corresponds to a literal whose branch is closed by a parent literal, by the framed literal $\langle p \rangle$. Then let *C* be the result of deleting from *C''* all the literals *q* whose branch is checked or closed by a complementary *ancestor* in θ .

For example, in the tableau shown in figure 5-2.1.i, $C' = b \sim ac \sim ae$, $C'' = b < a > < \sim c > \sim ae$ and $C = b < a > < \sim c > e$.



Figure 5-2.1.i

Theorem 5-2.1.1.

SL-resolution and *IPCR* analytic tableau exactly simulate each other.

proof: We must show that a sequence of tableaux $\theta_1 \ \theta_2 \dots \ \theta_n$ constructed in the stages described in section 3-1, corresponds to a sequence of chains in an SL-resolution proof. For the case of the singleton clause, $\Sigma = \{ C \}$, the theorem is obvious. There are three cases to consider:

Case 1: Extension

If *p* labels a node in θ_n and $C_n = a_1 a_2 \dots a_j p$ is the chain obtained from θ_n by the construction above, then θ_{n+1} is obtained by extending the tableau θ_n with a clause $C = \sim pq_1q_2 \dots q_k$.



Figure 5-2.1.ii

The chain C_{n+1} obtained by construction on θ_{n+1} is then $a_1a_2...a_j q_1q_2...q_k$ which corresponds exactly to an input resolution (extension) in the corresponding SL proof as shown in figure 5-2.1.iii.



Figure 5-2.1.iii

Case 2: Checking

Checking a literal in a tableau θ corresponds exactly to the merging step in an SL proof. If there are two open branches containing the literal p in θ then the SL clause obtained by construction has the form $a_1a_2...pq_1q_2..p...$ After checking the left-most open branch the clause has the literal p merged to the left.



Figure 5-2.1.iv

Case 3: Crossing

Closing a branch in θ whose endnode is labelled by a literal that is not complementary to a parent ancestor is the equivalent of the *reduction* operation in SL-resolution. •

Thus, an SL-resolution refutation can be interpreted as instructions for the construction of a *IPCR* analytic tableau and visa versa. This result is quite interesting and somewhat unexpected insofar as analytic tableau methods and resolution methods appear to be quite different.
CHAPTER 6

The Connection Method

The subject of this chapter is a proof technique advanced in [Bibel 1983] called the *connection method* (in this chapter unaccompanied page references will be references to [Bibel 1983]). The connection method has been proposed as an alternative to the resolution methods discussed so far, but we shall show that while its formalism appears to be quite different, this method is substantially the same as *IACR* analytic tableau.

6-1 The Connection Method

Whereas the resolution proof systems we have considered in previous chapters prove that statements in CNF are contradictory, Bibel's connection method proves statements in DNF are tautologies. But it is easy to see that a contradictory formula in CNF becomes a tautology in DNF when "v" and "&" are interchanged. So for the sake of continuity we shall consider the connection method as a technique for proving that sets of clauses are contradictory instead.

We can think of this conjunction of disjuncts as a matrix of literals, each column of which is a clause. For example, if we let Σ be the set of clauses { $a \sim b, \sim ab, \sim bc, b \sim c, ac, \sim a \sim c$ } we obtain the following matrix:

а	~ a	~b	b	а	~ a
~b	b	С	~ C	С	~ C

Any matrix of literals of this kind represents a contradictory conjunction of clauses provided there is no 'path' through the matrix such that each column of the matrix has one literal lying on the path and such that no two literals along the path clash. If such a path can be found then the original disjunction is not a contradiction. Thus a proof that no such path exists is a proof that the set of clauses is contradictory.

Bibel's *connection method* is an algorithm for proving that no path exists through the literal matrix that does not contain a pair of clashing literals. The following is a general sketch of the procedure. A more formal account is given in the next section.

With the matrix of literals M,

(1) Start at column 1. Initialize the set of literals in the "trial" path, (called the *active* path by Bibel) to the null set. Choose a literal in column 1 for membership to a new (tentative) active path through the matrix and mark the literal as having been chosen. The remaining literals are marked as possible alternatives.

(2) If the active path is non-empty go to the first column that contains both a literal complementary to some literal in the active path and a literal that can extend the path (i.e. one not marked as either impossible or already chosen). Then chose a literal in this

66

column whose propositional variable is *not* already in the active path (i.e. such that neither it nor its complement are in the active path) and mark the literals in the current column that clash with any of the literals in the current active path as impossible continuations of the current path. The remaining literals are then marked as possible alternatives to the chosen literal.

(3) If no column can be chosen to continue the active path, then go to the earliest marked alternative (choice point) and reset the active path to the value it would have had if this new literal had been chosen.

(4) If the active path is empty, and no literal is left in the matrix as marked as a possible alternative, then the matrix is contradictory. Otherwise, if a path through the matrix exists then the clauses are satisfiable by the set of truth values to the variables that makes the literals in the active path jointly true.

6-1.1 Structured Matrices

The formal account of the connection method offered below differs only in notation from Bibel's. First we define a *structure* on a matrix and then define the connection procedure in terms of transformations on that structure.

A structured matrix M for a set of clauses Σ is defined by three constructions:

(1) A (possibly empty) *push-down stack* S of clauses selected from the set Σ . The current clause is at the top of the stack S. Bibel

represents this stack by an integer-valued function α . So, for example, if $\Sigma = \{a \sim b, a, b, b, b, c, a, a, a, c\}$ and the stack S is

$$\mathbf{S} \qquad \qquad \mathbf{S} \qquad \mathbf{S} \qquad \qquad \mathbf{S} \qquad \mathbf{S} \qquad \mathbf{S} \qquad \qquad \mathbf{S} \qquad \mathbf$$

the α function yields

$$\alpha(\sim bc) = 3$$
$$\alpha(b\sim c) = 2$$
$$\alpha(ac) = 1$$

(2) A *boolean function* β that assigns the value 0 (signaling the impossible path or membership in the current active path) or 1 (possible alternative path) to each occurrence of a literal in each clause of the stack.

(3) A *choice function* γ defined on a subset of the clauses in the stack below the top clause. γ selects a literal occurrence *l* from each clause on the stack so that $\beta(l)=0$. The set of literals selected by γ is the *active path*.

In Bibel's graphic representation for β and γ , the active path is represented by a shaded line and literals in a clause that clash with literals in the active path are marked with a period, indicating that their β value is 0. Thus a connection graph where the stack is three clauses deep might look like this:



whereas in the stack based notation



and

$$\alpha(\sim bc) = 3 \quad \beta(\sim b) = 0 \qquad \beta(c) = 0$$

$$\alpha(b\sim c) = 2 \quad \beta(b) = 0 \qquad \beta(\sim c) = 0$$

$$\alpha(ac) = 1 \qquad \beta(a) = 1 \qquad \beta(c) = 0$$

The domain of γ is $\{ac, b \sim c\}$ and $\gamma(ac) = c, \gamma(b \sim c) = b$, so that the active path is $\{c, b\}$.

The *rules* for the connection method are then as follows. Given a structured matrix M constructed from the set of clauses Σ , proceed to obtain the matrix M' as follows:

Begin the stack with a single clause C with all the β-values for the literals in C set to 1.

(2) If there is no way to extend the active path of Σ to the top clause D in Σ so that there is a literal l in the remaining clauses which clashes with a literal in the extended active path, then

(a) Add to the top of the stack any new clause E from Σ not already in the stack; and

(b) Set the β -values of all literals in the new clause *E* to 1 and all the literals in the rest of the stack to 0. The domain of γ is now empty and we say that we have obtained a new matrix from M by *separation*.

Otherwise;

(3) The matrix M' can be obtained from M by *extension*. If D is the top clause in M then

(a) Extend the active path to D by choosing a literal *l* in D such that $\beta(l)=1$.

(b) Add a new clause E to the top of the stack such that E contains at least one literal k whose complement is in the active path determined by γ .

(c) The β -values for the clauses in the stack remain the same except that for the literal *l* in *D* just added to the active path, $\beta(l) = 0$. For each literal *m* in *E*, $\beta(m) = 0$ if the complement of *m* is in the active path.

(4) An extension may be followed immediately by *truncation* when the top clause in the stack has all its literals set to 0 by β. In that case the top clause and all the clauses immediately below it whose literals all have 0 β -values, should be popped off the stack. The new β -values and γ -values are the appropriate restrictions of the previous values. If all the clauses are popped, i.e. the stack is empty, there is no possible path and the clauses are contradictory.

(5) An extension step is *factorized* by setting $\beta(l)=0$ for every literal l in the top clause whose occurrence in another clause in the stack has its β -value equal to 1 and does not occur in the active path.

Remarks: The separation step is only ever used in the connection method if the original set of clauses is not minimally inconsistent. Without loss of generality all further discussion of the method will focus on minimally inconsistent sets of clauses.

6-2 Connection Method and Analytic Tableaux

This section makes explicit the relationship between Bibel's connection method and the analytic tableau method.

Theorem 6-2.1

The connection method p-simulates and is p-simulated by *IACR* analytic tableaux.

proof: Consider an *IACR* analytic tableau θ for a set of minimally inconsistent clauses Σ (see section 3-1.2). This tableau corresponds to a matrix in a connection proof that Σ is inconsistent and every step in the connection proof corresponds to a step in the construction of another analytic tableau.

For the base case there are only two possible tableaux (modulo clause decomposition order):



and the corresponding steps in the connection proof are just.

 $p \sim p \qquad p \sim p$

Consider the induction hypothesis in which the tableaux θ correspond to the stack of clauses S = [$C_k ... C_i ... C_1$], where C_i is the clause expanded at node *i* in θ .



Figure 6-2.i

For $l \in C_j$, let $\beta(l) = 0$ iff *l* labels an interior node in θ or *l* is checked. Let all the interior nodes of the longest open branch be the set G. To every literal $l \in G$ set $\beta(l)=0$, and to every literal *k* in an expanded clause that either clashes with an ancestor or is checked, assign $\beta(l) = 0$. By the definition of the function γ in section 6.1.1, $G = \{ \gamma(D) : D \in S \}$. Now set $\alpha(D)$ to be the depth of expansion of clause *D* in the literal tree, i.e. the number of literals in a branch from a literal in *D* to the root of θ .

It is easy to see that each step in a connection method proof corresponds to a step in the construction of an *IACR* analytic tableau.

Since Σ is minimally inconsistent, there is no need to consider the *separation* rule. There are only three possible connection method steps: *extension*, extension followed by *truncation*, and *factorization*.

In the stack model of the connection method, *extension* adds a clause to the top of the stack such that at least one literal in the added clause is complementary to the adjacent clause or to an element of the active path. The former corresponds to a tableau *crossing* step and the latter to a tableau *extension* step.

An extension step followed by *truncation* corresponds to the closure of a sub-tableau of θ . In other words all the literals in *D* are crossed or checked and the next clause is decomposed below a literal whose distance from the root (i.e. depth) is smaller than the depth of the literals in *D*.

The rule of *factorization* is simulated by the checking of literals. Setting $\beta(l)=0$ for every literal *l* in the top clause whose occurrence in another clause in the stack has its β -value equal to 1 can be viewed simply as an alternative notation for the *checking* operation in a tableau.

Remarks: This proof shows that the *IACR* tableau method and the connection method are notational variants. If the connection method rules are further restricted so that the literal (in the active path) which is chosen for extension have the largest α -index (i.e. belong to the most recently introduced clause), then a similar proof shows that this connection method is equivalent to the *IPCR* tableau method.

To illustrate the theorem consider the connection matrices that refute $\{klm, \sim kl, \sim l, \sim m\}$:



Figure 6-2.ii

These connection matrices correspond exactly to the following sequence of analytic tableaux:



Figure 6-2.iii

6-3 Conclusions

Bibel has suggested that the connection method is superior in generality, elegance, efficiency of computer implementation, and computational complexity to all refinements of resolution, including linear resolution [p.142-143]. Although the connection method might be preferable to linear resolution for some reasons, it follows from theorem 6-2.1 that worst case computational complexity is not one of them.

We have shown that the connection method exactly simulates the improved clash restricted (*IPCR*) analytic tableau and we know from results in chapter 3 that tree resolution p-simulates analytic tableaux. It follows, by results proved in chapter 7, that the connection method does require a superpolynomial number of proof steps for some tautologies. This fact contradicts Bibel's opinion that

"...it is hard to believe that there are propositional formulas for which the connection method requires an exponential number of proof steps..." [p.170].

Moreover, since these tautologies admit of short proofs using other refinements of resolution, namely the Davis-Putnam procedure, the connection method is not the computationally least complex theorem proving method.

Other theorem proving techniques such as Kowalski's connection graph resolution [Kowalski 1975, Shostak 1976] and Prawitz's matrix reduction procedure [Prawitz 1970] have been compared to the connection method [Bibel 1982]. These comparisons suggest that simulation results similar to the one proved in this chapter can also be proved for these methods.

CHAPTER 7

Hard Examples For Tree Resolution

There is a class of inconsistent sets of clauses discovered by Tseitin [Tseitin 1968] which may be used to show that the length of any regular resolution refutation of an element of that class is a super-polynomial function of the number of input clauses. This is the result proved in Tseitin's paper. Similar results have recently been proved for the length of all unrestricted resolution proofs [Haken 1985, Urquhart 1987, Chvátal and Szemerédi 1988] although some of these demonstrations use non-constructive (probabilistic) arguments.

In this chapter we reconstruct and attempt to clarify Tseitin's somewhat sketchy proof that, for some classes of Tseitin clauses, regular tree refutations are not bounded by a polynomial function of the number of input clauses even though they have short (linear length) proofs using the Davis-Putnam procedure. This therefore puts a complexity bound on the proof procedures considered in previous chapters.

7-1 Graph Clauses

The aim of the proof is to find a class of inconsistent clauses for which any regular tree refutation grows as a superpolynomial function of the number of input clauses. To do this, we find a canonical, graphical representation for contradictory clauses — so-called Tseitin clauses — and show that regular tree resolution refutations for them grow exponentially with respect to the number of nodes in the graph. A less direct proof of the same result may be found in [Galil 1975].

In the following we use the notation adopted in [Urquhart 1987]. Tseitin clauses are constructed by labelling each edge in a finite undirected graph G with a distinct literal to form a labelled graph Γ . Each vertex $v \in \Gamma$ has associated with it a *charge*, **charge**(v)=1 or **charge**(v)=0, and a set, **literals**(v), consisting of the literals associated with each edge incident with v. For a labelled graph Γ , we define its total charge to be:

$$\mathbf{charge}(\Gamma) = \sum_{\mathcal{V}} (\text{mod } 2) \ \mathbf{charge}(\mathcal{V}).$$

If $charge(\Gamma) = 1$ ($charge(\Gamma) = 0$) we say the labelling of Γ is odd (even).

Associated with every vertex v is also a set of clauses clauses(v). clauses(v) is the set of all clauses containing all the literals(v) such that the number of complemented literals in each clause is odd if charge(v)=0, and even if charge(v)=1. The set of clauses for a Tseitin graph $\Sigma(\Gamma) = U^{\circ} clauses(v)$, for all $v \in \Gamma$.

Example: Consider the connected graph $\Gamma = (\{X, Y\}, \{a, b, c\})$ such that **charge**(X)=1, **charge**(Y)=0. The charge on a vertex is shown diagrammatically in figure 7-1.i by its colouring (black or white).



Figure 7-1.i

 $\Sigma(\Gamma)$, then, is given by **clauses**(*X*) = { $\sim a \sim bc$, $a \sim b \sim c, \sim ab \sim c, abc$ } **clauses**(*Y*) = { $\sim abc$, $ab \sim c$, $a \sim bc$, $\sim a \sim b \sim c$ }. Note that $\Sigma(\Gamma)$ is inconsistent because the set of clauses at vertex *X* with charge 1 is equivalent to the formula

(1)
$$a \equiv (b \equiv c)$$

whereas the set of clauses at vertex Y with charge 0 is equivalent to the negation of that formula

(2) $\sim (a \equiv (b \equiv c))$

Alternatively, we can express these formulas as the modulo 2 additions

- (1') $a \oplus b \oplus c = 1$
- (2') $a \oplus b \oplus c = 0$

which clearly constitutes a contradiction.

In general, for any connected graph Γ , **clauses**(*v*), for each $v \in \Gamma$ can be represented by the conjunctive normal form of the modulo 2 equation E(*v*)

$$a_1 \oplus a_2 \oplus \dots \oplus a_n = \mathbf{charge}(v)$$

for $a_i \in \mathbf{literals}(v)$. Summing the left hand side for all such equations for $v \in \Gamma$ will always produce two occurrences of each literal in Γ , if Γ is connected and each edge has a distinct labelling. But $a_i \oplus a_i = 0$, and if the right hand side, **charge**(Γ), has the value 1 then the set of clauses $\Sigma(\Gamma)$ is contradictory. So if $\Sigma(\Gamma)$ is satisfiable, **charge**(Γ) = 0.

Note that we can rewrite each equation E(v) by replacing one of the literals a_i by its complement on the left hand side and inverting the charge on the right hand side. For example, the equations (1') and (2') are equivalent to

$$(1") \quad a \oplus \sim b \oplus c = 0$$

(2") $a \oplus \sim b \oplus c = 1$

Since the substitution of a literal by its complement in a labelled graph always affects the charge on two vertices we say that the charge has been *transferred*.

We now show that if $charge(\Gamma) = 0$, then there is a satisfying assignment of truth values to the literals in $\Sigma(\Gamma)$. This can be seen by observing that any vertext with charge 1 can be made to change to a vertex with charge 0 by repeated charge transfer (since Γ is connected). Since $charge(\Gamma) = 0$ there are an even number of vertices with charge 1, all the vertices v may be converted to having charge(v) = 0 and the satisfying *tva* is obtained by setting the literals on each edge to 0 (*false*). This proves the following lemma:

Lemma 7-1.1.

 $\Sigma(\Gamma)$ is satisfiable iff **charge**(Γ) = 0.

Lemma 7-1.2.

If Γ is an unconnected labelled graph and D is a derivation of a clause *C* from $\Sigma(\Gamma)$, then there is a connected component Ψ of Γ such that D is a derivation of *C* from $\Sigma(\Psi)$.

proof: We prove by induction on the sub-trees of D that all the input clauses belong to a single connected component. If $C \in \Sigma(\Gamma)$ then the connected component is simply a single vertex. For the inductive case, consider the derivation of C = AB as a tree T and induce on the sub-trees. Let *C* be the resolution of Ap and $B\sim p$:



Figure 7-1.ii

Now, we assume that the lemma is true for the sub-trees T_A and T_B , i.e. that ψ_1 and ψ_2 are the associated connected components of Γ . Since *p* is in *Ap* and $\sim p$ is in $B \sim p$, each sub-tree T_A and T_B must contain an input clause attached to a vertex adjacent to the edge in Γ labelled with *p*. So the tree T has all its input clauses attached to vertices in the component containing both Ψ_1 and Ψ_2 .

If Γ is a labelled graph and p is a literal then Γ / p is obtained by deleting the edge labelled by either p or its complement. If the edge is labelled by p then the charges attached to the vertices in Γ / p are unchanged. If the edge is labelled by the complement of p then the charge is transferred from one vertex to the adjacent one. Lemma 7-1.3 follows from this definition.

Lemma 7-1.3

 $charge(\Psi) = charge(\Psi / l).$

Lemma 7-1.4

Given a labelled graph Ψ , $\Sigma(\Psi / l) = \Sigma(\Psi) / l$, for *l* any literal *p* or ~*p*.

proof: We need only consider the edge in Ψ labelled by l = p or $l = \sim p$.



Figure 7-1.iii

Suppose *B* is a clause attached to the vertex *X* and $B \in \Sigma(\Psi / l)$. If l = p, then Bp is a clause in $\Sigma(\Psi)$ since **charge**(*X*) remains unchanged by the addition of the edge *p*. If $l = \sim p$ then $B \sim p$ is a clause attached

to X because charge(X) with edge $\sim p$ is 1-charge(X) without the edge $\sim p$. In either case $B \in \Sigma(\Psi) / l$.

Conversely, suppose $B \in \Sigma(\Psi) / l$. If l = p and B is attached to vertex X, then B can be obtained by deleting p from a clause attached to X in $\Sigma(\Psi)$, i.e. Bp. Therefore $B \in \Sigma(\Psi / p)$ since deleting the edge p does not change the charge. Similarly if $l = \sim p$ then Bcan be obtained by deleting $\sim p$ from a clause $B \sim p$ attached to X in $\Sigma(\Psi)$, provided the charge at X is inverted. Hence $B \in \Sigma(\Psi / \sim p)$.

7-2 Tseitin's Theorem

We say that a connected graph is *ruptured* by the deletion of an edge if it splits the graph into two connected components.

If G is an unlabelled connected graph and e is one of its edges, then G_e is the result of deleting e from G if the deletion does not rupture G. If the deletion of e ruptures G then the two connected components are G_e' and G_e'' .

Let G be an unlabelled graph and Γ be any odd labelling of G. Then the size of the tree refutation of $\Sigma(\Gamma)$, $L(\Sigma(\Gamma))$ is abbreviated to L(G). This definition is permitted because the refutation trees of clauses from any odd labelling of some connected graph G are isomorphic after relettering the literals.

Now Tseitin's theorem is

Theorem 7-2.1

If G is an unlabelled graph, then

$$L(G) = \begin{cases} 1 \text{ if } G \text{ is a single vertex} \\ \min_{a \in G} L^{a}(G) \text{ otherwise} \end{cases}$$

where $L^{a}(G)$ is given by

$$L^{a}(G) = \begin{cases} L(G_{a}') + L(G_{a}'') & \text{if } a \text{ ruptures } G \\ 2L(G_{a}) & \text{otherwise} \end{cases}$$

proof: The proof is by induction on the number of edges in the connected graph G. For the base case, if G consists of a single vertex (no edges) then $\Sigma(G) = \{ \emptyset \}$, so L(G) = 1.

Now, let G have one or more edges and let T be a minimal size refutation tree for $\Sigma(G)$. Let Γ be the labelled graph that defines the set of clauses $\Sigma(G)$. Since G is connected, $\Sigma(G)$ does not contain \emptyset , so the last step in the proof must be by resolution:



Let *a* (in the statement of the theorem) be the edge which is labelled with the literal *p*. There are two cases: deleting the edge *a* either (1) ruptures G or (2) it doesn't.

Case 1: By assumption **charge**(Γ) = 1, so by lemma 7-1.3 **charge**(Γ / p) = 1. Thus, exactly one of the two components of Γ / p , call it G_a ' has an odd labelling. The other component, G_a " must have an odd labelling in $\Gamma / \sim p$. Let T_1 ' and T_2 ' be the result of deleting pand $\sim p$ from T_1 and T_2 respectively. By lemma 4-2.1, T_1 ' and T_2 ' are the regular resolution refutations from $\Sigma(\Gamma) / p$ and $\Sigma(\Gamma) / \sim p$. By lemma 7-1.4 above, T_1 ' and T_2 ' are refutations of the clauses $\Sigma(\Gamma / p)$ and $\Sigma(\Gamma / \sim p)$. From lemma 7-1.2 each of these derivations are derivations from connected components of $\Sigma(\Gamma / p)$ and $\Sigma(\Gamma / \sim p)$ respectively. Since each of these components must have an odd labelling (by lemma 7-1.1), it follows that T_1 ' is a proof of \emptyset from an odd labelling of G_a ' and T_2 ' is a proof of \emptyset from an odd labelling of G_a ". Thus, the complexity of T is $L(G_a') + L(G_a'') = L^a(G)$.

Case 2: By the same argument as in case 1, T_1' is a derivation of \emptyset from $\Sigma(\Gamma / p)$ and T_2' is a derivation of \emptyset from $\Sigma(\Gamma / p)$. But in this

case, both Γ / p and $\Gamma / \sim p$ are odd labellings of the connected graph Γ_a . Hence, the complexity of both T_1' and T_2' is $L(\Gamma_a)$, so the complexity of T is $2L(\Gamma_a)$.

Now since the tree complexity of T is minimized by minimizing the sum of the tree complexities of the two sub-trees T_1 and T_2 , we assume that the edge a is chosen to minimize $L^a(G)$. Hence the theorem. •

Remarks: Note that while tree complexity may be minimized by minimizing the complexity of the sub-trees, this may not minimize the linear length of a proof because of the possible overlap between clauses in T_1 and T_2 .

7-2.1 Examples

To illustrate theorem 7-2.1, observe that the graph with just one node has L = 1, that a graph with two nodes and one edge has L = 2 and that one with three nodes and two edges



has length 3(2+1).

Now consider the graph:



that represents the set of clauses

{*ade, a~d~e, ~a~de, ~ad~e, a~b, ~ab, bc~e, b~ce, ~bce, ~b~c~e, c~d~cd*}

The deletion of literals <a, b, c, d, e> produces the following sequence < Γ , Γ 1, Γ 2, Γ 3, Γ 4, Γ 5> of sub-graphs:



And the complexity of $\boldsymbol{\Gamma}$ is given by

$$L(\Gamma) = 2*L(\Gamma 1)$$

= 2* (1 + L(\Gamma 2))
= 2* (1 + (2 * L(\Gamma 3)))
= 2* (1 + (2 * (1 + L(\Gamma 4)))
= 2* (1 + (2 * (1 + 1 + L(\Gamma 5)))
= 2* (1 + (2 * (1 + 1 + 1)))
= 14

Theorem 7-2.1 may be reformulated as an algorithm for calculating the size of a tree proof for inconsistent Tseitin clauses obtained by undertaking a sequence of edge deletions on a connected graph G.

- (1) Set the count on each vertex to 0.
- (2) At the *n*th step, (a) delete an edge *a* (b) if the deletion does not rupture the graph then add 1 to all the vertices of the component to which *a* belongs, otherwise leave all associated numbers alone.
- (3) The sequence ends when all edges of G have been deleted. The complexity of a deletion sequence D for G is $L(D) = \sum_{v \in G} 2^{n(v)}$ where n(v) is the number associated with the vertex v when all the edges have been deleted.

So theorem 7-2.1 may also be stated as

 $L(G) = \min \{ L(D) \mid D \text{ is a deletion sequence for } G \}$

For example, consider the application of this algorithm to the deletion sequence $S = \langle \Gamma', \Gamma'1, \Gamma'2, \Gamma'3, \Gamma'4, \Gamma'5 \rangle$:



The complexity of this deletion sequence is $L(\Gamma') = \sum_{v \in G} 2^{n(v)} = 14.$

7-3 Tree Resolution cannot simulate DPP

In this section we show that there exist a class of Tseitin graphs that have large minimal tree sizes but short minimal linear length proofs using the DPP.

Let G_n be the set of graphs consisting of a chain of 2^n vertices, with each pair of adjacent vertices joined by a group of *n* edges. For example, the graph for G_3 is:

89



Figure 7-3.i

For such chain graphs $\Sigma(G_n)$ contains $2(2^{n-1})=2^n$ clauses of length *n* (attached to the end nodes), and $(2^{n-2})2^{2n-1} = (2^{n-1}-1)2^{2n}$ clauses of length 2n attached to interior nodes. For example, G_2 with **charge**(G_2)=1 is



Figure 7-3.ii

and $\Sigma(G_2)$ is the set

{*ab*,~*a*~*b*, ~*abcd*, *a*~*bcd*, *ab*~*cd*, *abc*~*d*, *a*~*b*~*c*~*d*, ~*a*~*bc*~*d*, ~*a*~*b*~*cd*, ~*cdef*, *c*~*def*, *cd*~*ef*, *cd*~*ef*, *c*~*d*~*ef*, ~*c*~*de*~*f*, ~*c*~*d*~*ef*, *e*~*f*, ~*ef* }

The *large component* in a graph from the deletion sequence is as follows.

- (1) G_n is the large component of G_n ;
- (2) If step *n*+1 does not rupture the large component then the large component at step *n*+1 is the same as the large component at step *n* (with the exception of a deleted edge).

(3) If step n+1 does rupture the large component into sub-graphs Γ₁ and Γ₂, then the new large component is Γ₁ or Γ₂ depending on which has the larger number of vertices.

Theorem 7-3.1

$$L(\mathbf{G}_n) > 2^{n(n-1)}.$$

proof: Suppose D is an optimal deletion sequence for G_n . At the end of the deletion sequence, the large component consists of a single vertex. Since this vertex v belonged to the large component at all earlier stages, this means that the component to which v belongs must have been ruptured at least n times because every time the large component is ruptured, the new large component contains at least half the number of the vertices in the previous large component. But to rupture a component of G_n requires the deletion of n edges, so that the vertex v must, at the end of the deletion sequence, bear the number n(n-1). Since $L(G_n) = L(D) = \sum_{v \in G} 2^{n(v)} L(G_n) > 2^{n(n-1)} \bullet$

Now we compute the complexity of $N(G_n)$ by using the Davis-Putnam procedure. We delete the edges of G_n successively from left to right. The procedure divides into 2^{n-1} -1 stages, each stage consisting in the deletion of the edges attached to adjacent vertices *X* and *Y*.



Figure 7-3.iii

The clauses produced (and the input clauses used) at this stage involve only the variables attached to *X* and *Y*, of which there are 2n (recall that all the edges to the left of *X* are already deleted). There are at most 3^{2n} such clauses, so the entire refutation produced by DPP has length $(2^n - 1)3^{2n}$, i.e. the entire refutation contains at most

$$(2^{n}-1)2^{(3,2)n} < 2^{n}(2^{3,2n}) = 2^{4,2n}.$$

Thus the length of the minimal DPP refutation is bounded by a quadratic (polynomial) function of the size of the input. •

Corollary: Tree resolution cannot p-simulate the DPP.

proof: $2^{n(n-1)}$ eventually dominates 2^{kn} for any fixed *k*. This follows from the fact that n(n-1) grows asymptotically faster than any linear function. •

CHAPTER 8

Conclusion

This thesis aims to fill some lacunae in the landscape of the relative complexity of automated theorem proving methods by showing how different techniques (connection method, restrictions of the improved analytic tableau and SL-resolution) compare with respect to the minimum lengths of proofs in the worst-case. It was shown that SL-resolution and the connection method are equivalent to refinements of the improved analytic tableau method and that all of these automatic theorem proving methods can be p-simulated by tree resolution. These results are significant in light of the theorem by Tseitin that tree resolution cannot p-simulate the Davis-Putnam procedure, for which we offer a more detailed proof than Tseitin's own.

The hierarchy of relative complexities is determined by the notion of polynomial (p-) simulation which defines an equivalence relation among proof methods. However, the significance of this notion is unclear. Since the value of a polynomial function (of degree k, for some constant k) may always exceed the value of a super-polynomial function (of degree n) for any given n, there may be no distinction between the difficulty of proving (i.e. the length of a proof for) a particular *instance* of a class of tautologies using a polynomial proof method and the difficulty of proving the same tautology with a super-polynomial proof method. The essential difference between one method and

the other is the *rate of growth* of proof lengths as the length of the tautology (n) increases. The question then arises whether the rate of growth of worst-case proofs for a given method says something meaningful about the power of the method.

Our answer to this question is affirmative. If a proof system A that properly p-simulates another system B (i.e. A p-simulates B but B does not psimulate A) then A has a rule of inference which permits significantly shorter minimal length proofs, for some classes of tautologies, than B. If there exists a short proof for a tautology in one proof system but not in another, this says something about the extent to which our knowledge can be certified using this proof technique, i.e. it sets limits—practical limits—to what it is that a logical system can allow us to infer. (There is perhaps an analogy here with another meta-logical property—decidability—that reflects the strength of a logical system. Just as an undecidable system of logic is more *expressive* than a decidable one, a proof system which properly p-simulates another has a more *powerful* proof mechanism.)

What is it, then, about one inference mechanism that makes it more powerful than another? One characteristic seems to be that more powerful inference rules have a greater capacity for pattern-recognition. For example, the improved analytic tableau method (i.e. tableaux with the checking rule), has the advantage over analytic tableaux that identical sub-tableaux can be collapsed into one (see figures 3-1.ii & 3-1.2.i). Similarly, if we take a look at the minimal DAGs proofs generated by the Davis-Putnam procedure (DDP) and compare them to the corresponding minimal tree proofs we see that the DPP proofs eliminate the multiple occurrences of identical sub-proofs present in the tree proof (see figures 3.2-i & 3.2-ii).

An even more powerful technique is found in those methods that permit some degree of term-rewriting. Comparing the (relatively short) extended resolution proofs of the pigeon hole clauses to the minimal resolution proofs shows that the greater power of the extension rule is due to the fact that the substitution of one literal for a sub-expression permits the computation of a simple induction problem in polynomial time. Systems that use axiom schemas allow greater degrees of freedom for term-rewriting and it seems natural that they should be higher still in the hierarchy of proof systems. Although quantum leaps in proof length reduction can be obtained by such extensions, there is little evidence to suggest that there exists an automatic theorem proving method that cannot be defeated by some class of tautologies.

8-1 Open Problems

There are several unanswered questions raised in this thesis whose answers would help complete the details of figure 1-3.i. To summarize, they are: (i) can *IPCR* analytic tableau simulate *IACR* analytic tableau? (ii) can *ACRI* analytic tableau simulate unrestricted improved tableau? (iii) can analytic tableau simulate the improved analytic tableau? (iv) can SL-resolution simulate s-linear resolution?

Further research could be directed at answering similar questions concerning the relative complexity of other restrictions of resolution such as Lock resolution [Boyer 1971] and other kinds of theorem proving methods such as connection graph resolution [Shostak 1976]. It would undoubtedly be useful, both for deepening our understanding of these systems and for the practical requirements of automated theorem proving, if it were possible to characterize the Achilles heel that causes all these methods to suffer combinatorial explosions. Examples that are hard for these systems such as those produced by Tseitin graphs or pigeonhole clauses appear to have a structure or "connectedness" that makes them hard for these simple systems. A better understanding of this attribute could help to characterize the limitations of these proof methods and to suggest techniques that might circumvent them in a practical setting.

APPENDIX

In this section we briefly describe a particularly simple and elegant Prolog program which implements the improved analytic tableau method in five lines of computer-executable code. It has been shown empirically (Vellino 1989) that this program is also superior (in performance or search time) to resolution theorem provers, against some examples that are hard-toprove for resolution such as the pigeon-hole clauses.

The program operates on a list (conjunction) of lists (disjunctions) of literals which have the form

```
lit(NegOrPos,Variable)
```

where NegOrPos indicates the presence or absence of a negation sign in front of the Variable. For example the clause $((a \vee b) \& (b \vee c))$ is represented as the list of lists

```
[[lit(true,A),lit(false,B)],[lit(true,B),lit(false,C)]].
```

The complete program that implements the improved tableau method is this:

```
satisfiable([]).
satisfiable([C|Clauses]) :- satisfied(C), satisfiable(Clauses).
satisfied([lit(T,T)|_]).
satisfied([lit(true,false) | Literals]):- satisfied(Literals).
satisfied([lit(false,true) | Literals]):- satisfied(Literals).
```

The propositional variables (A, B, C...) in the clauses processed by the program are first-order variables in predicate logic to which the program attempts to assign satisfying truth-values by making at least one literal true in

each clause. This is done by instantiating these variables to values with which they are prefixed. The propagation of these tva's is guaranteed by the instantiation mechanism (unification) in Prolog, which obeys the rule of first-order universal instantiation.

The first two lines in the program express (recursively) the idea that a conjunction of formulas is satisfiable if each formula can be satisfied. The third line either instantiates a free variable in the first literal of a clause to a satisfying *tva* or confirms that the clause is satisfied if that variable has already been instantiated. If a satisfying assignment for a literal closes all the branches of the tree, Prolog backtracks to either the second or the third rule (exclusively) for satisfied/1 which reverses the original *tva* (checking rule) and recursively tries to assign a satisfying *tva* to the remaining literals. If a tva was merely verified by the call to the first rule for satisfied/1, then neither the second nor the third rule can apply since the variable acquired its truth-value from a clause higher up the tree. Thus clauses containing the same literal as some ancestor in the tree are not considered (since they are already true).

Of course, this program does not implement any parent clash or ancestor clash clause selection strategy. It merely considers the clauses in the order in which they occur in the list.

BIBLIOGRAPHY

- Andrews, P. B. (1968). "Resolution with Merging" J. ACM, Vol. 15, No. 3, (Reprinted in Siekmann and Wrightson).
- Bibel, W. (1982). Automated Theorem Proving. Vieweg Verlag, Braunschweig; Wiesbaden: Vieweg.
- Bibel, W. (1983). "A Comparative Study of Several Proof Procedures" Artificial Intelligence 18, p. 269-293.
- Bondy, J. A. and Murty, U. S. R. (1976). *Graph Theory with Applications*, American Elsevier, New York, and Macmillan, London.
- Boyer, R. S. (1971) "Locking: a Restriction of Resolution" Ph.D. Thesis, University of Texas at Austin.
- Buss, S. A. (1987). "Polynomial Size Proofs of the Propositional Pigeonhole Principle" *Journal of Symbolic Logic*, **52**, pp. 916-927.
- Chang, C. L., and Lee, R. T. C. (1973). Symbolic Logic and Mechanical *Theorem Proving*, New York.
- Chvátal, V., and Szemerédi, E. (1988). "Many Hard Examples for Resolution" *J. ACM.* **35** No.4, pp. 759-768.
- Cook, S. A. (1971a). "The Complexity of Theorem Proving Procedures" *Proc. 3rd ACM STOC* pp.151-158.
- Cook, S. A. (1971b). "Examples for the Davis-Putnam Procedure", unpublished manuscript, referenced in [Cook 1971a]
- Cook, S. A. (1976). "A Short Proof of the Pigeon-Hole Principle Using Extended Resolution", *ACM SIGACT News* **8** p. 28-32.

- Cook, S. A., and Reckhow, R. A. (1974). "On the Length of Proofs in the Propositional Calculus" *Proc. 6th ACM STOC*, pp.135-148.
- Cook, S. A., and Reckhow, R. A. (1979). "The Relative Efficiency of Propositional Proof Systems" *Journal of Symbolic Logic*, **44**, pp.36-50.
- Davis, M., Logemann, G., and Loveland D. (1962). "A Machine Program for Theorem Proving" *Communications of the ACM* **5**, No. 7, July (Reprinted in Siekmann and Wrightson.)
- Davis, M., and Putnam, H. (1960). "A Computing Procedure for Quantification Theory," *J. ACM.* **7** pp. 201-215.
- Galil, Z. (1975). "The Complexity of Resolution Procedures for Theorem Proving in the Propositional Calculus" Ph.D Thesis, TR 75-239 Cornell University.
- Galil, Z. (1977). "On the Complexity of Regular Resolution and the Davis-Putnam Procedure" *Theoretical Computer Science* **4** pp.23-46.
- Garey, M. R., and Johnson, S. D. (1979). Computers and Intractability, A Guide to the Theory of NP-Completeness, W.H. Freeman & Co.: New York.
- Gibbons, A. M. (1985). *Algorithmic Graph Theory*, Cambridge University Press.
- Goldberg, A. T. (1979). "On the Complexity of the Satisfiability Problem", Courant Institute for Computer Science report #16, New York University.
- Haken, A. (1985). "The Intractability of Resolution" *Theoretical Computer Science* **39** pp.297-308.
- Kirkpatric, D. G. (1974). "Topics in the Complexity of Combinatorial Algorithms" Ph.D. Thesis, University of Toronto.
- Knuth, D. E. (1968). *The Art of Computer Programming* Vol.1, Addison-Wesley.
- Kowalski, R., and Kuehner, D. (1971). "Linear Resolution with Selection Function" *Artificial Intelligence* 2 p. 227-260, Reprinted in Siekmann and Wrightson (1983) Vol. 2.
- Kowalski, R., (1975). "A proof procedure using connection graphs", *J. ACM*. **22**, pp. 572-595.
- Lewis, R. L. and Papadimitriou, C. H. (1981). *Elements of the Theory of Computation*, Prentice-Hall: New Jersey (1981).
- Lloyd, J. W., (1987). *Foundations of Logic Programming* (second edition) Springer-Verlag, New York.
- Loveland, D. W. (1970). "A Linear Format for Resolution", Symposium on Automatic Demonstration, Lecture Notes in Mathematics, 125, Springer-Verlag, New York, p. 147-163, Reprinted in Siekmann and Wrightson (1983) Vol 2.
- Mendelson, E. (1964). Introduction to Mathematical Logic (van Nostrand).
- Prawitz, D. (1970). "A proof procedure with matrix reduction", Symposium on Automatic Demonstration, Lecture Notes in Mathematics, 125, Springer-Verlag, New York, p. 207-213.
- Reckhow, R. A. (1975). "On the Lengths of Proofs in the Propositional Calculus" Ph.D. Thesis University of Toronto.
- Robinson, J. A. (1965). "A Machine Oriented Logic Based on the Resolution Principle" J. ACM 12, pp 23-41. Reprinted in Siekmann and Wrightson (1983) Vol 1.
- Shostak, R. E. (1976). "Refutation Graphs" Artificial Intelligence 7, pp.51-64.

Smullyan, R. (1968). First Order Logic Springer-Verlag, New York.

- Tseitin, G. S. (1968). "On The Complexity of Derivation in The Propositional Calculus" *Studies in Constructive Mathematics and Mathematical Logic* part 2, (1968) p. 115-125. Reprinted in Siekmann and Wrightson (1983) Vol 2.
- Urquhart, A. I. F. (1987). "Hard Examples for Resolution" J. ACM. 34 No.1, pp. 209-219.
- Vellino, A. (1989). "A Prolog Implementation of an Analytic Tableau Theorem Prover for the Propositional Calculus " BNR Computing Research Lab Report 89024.
- Wenqi, H., and Xiangdong, Y., (1985). "A DNF Without Regular Shortest Consensus Path" (unpublished manuscript).