1. Introduction

Automatic theorem proving is concerned with designing a computer program, called a *theorem prover*, that can prove, or help prove, mathematical theorems. Automatic theorem proving is of interest because modern computers can perform inferences rapidly and this capability can be applied to many applications such as expert systems, proof checking and program verification.

In this thesis, we are only interested in *refutational theorem provers* for first order logic which accept a set of *clauses* and prove the inconsistency of the set of clauses. Herbrand's Theorem [Gallier 86] is the theoretical basis for this kind of theorem prover. A theorem prover typically consists of an *inference system* and a *search strategy*. The inference system consists of the axioms and the inference rules applicable to clauses. The search strategy organizes and controls the applications of the inference rules. Many inference systems have been developed over the last few decades [Loveland 78, Chang&Lee 73]. Resolution [Robinson 65] is probably the best known example of an inference system. We say an inference system is *complete* if it can detect that a set of clauses is inconsistent whenever it is. An inference system may or may not be complete. We say a search strategy is *complete for an inference system* if the search strategy can obtain all the proofs constructible in the inference system. A search strategy may or may not be complete for an inference system. We say a theorem prover is *complete* if the theorem prover can construct a proof of the inconsistency of a set of clauses whenever it is. It follows that the completeness of a theorem prover requires both a complete inference system and a complete search strategy. Similarly, the efficiency (speed, memory requirements, etc) of a theorem prover also depends both on the inference system and the search strategy. To improve the efficiency of a theorem prover, we can restrict the inference system (for example, an inference rule can only apply if certain conditions are met), or we can improve the search strategy [Kowalski 70].

This thesis is mainly concerned with how to control the searches for proofs in a theorem prover. The problem of search control in automatic theorem proving is important because, although theorem provers can perform thousands of inferences in a matter of seconds, lack of guidance for this power often makes them very inefficient. The search strategy used in the theorem prover is the depth-first iterative deepening search [Korf 85]. Although the general organization of this search strategy is well understood, there are surprisingly many issues that can significantly affect its efficiency. These issues are not only interesting for this particular theorem prover, but also for implementing any other theorem prover that uses depth-first iterative deepening search. We believe that the techniques and heuristics investigated in this thesis are equally applicable to other systems.

1.1. Goal of This Research

To tackle the problem of search control, we refine the search strategy and the inference system of a theorem prover. The search strategy is the *depth-first iterative deepening search* and the inference system is the *modified problem reduction format*. We intend to preserve both the completeness of the search strategy and the completeness of the modified problem reduction format. The resulting search strategy in the theorem prover has so-called *weak search guidance* [Loveland 78]. That is, the techniques and heuristics do not assume knowledge in any particular domain. On the other hand, domain dependent knowledge supplied by the user is usable by the search strategy and by the inference system. The resulting theorem prover can efficiently solve problems of moderate difficulty with little or no user guidance at all. At the same time, the theorem prover can be used as a research tool to solve more difficult problems when various domain dependent knowledge and guidance are supplied by the user. The theorem prover can be also regarded as a limited logic programming system [Lloyd 84] in full first order logic. A theorem prover of this kind is interesting both to the logic programming community and for applications in expert systems, program verification and deductive database systems.

We believe that extensive experiments are essential to evaluate proposed techniques for automatic theorem proving. For each technique we propose, we shall first discuss the motivation behind them, then the techniques themselves. Following discussions of each technique, test data on a large set of problems shall be given. In the following, we will briefly discuss the various techniques that will be covered in greater details later.

Caching. Caching is a technique proposed to avoid repeated work in depth-first iterative deepening search. The idea is to record the solutions found so that they can be used to solve other goals. The effect is that no goal will be solved more than once. We will demonstrate that using depth-first iterative deepening search with caching makes the theorem prover more efficient — obtaining proofs for more problems in less time on the average — than using depth-first iterative deepening search without caching. A particular technique to improve caching is *goal generalization*. The purpose of goal generalization is to solve each goal in as full generality as possible and then cache the most possible general solutions for the goals. We will demonstrate that goal generalization can improve the efficiency of the theorem prover.

Subgoal Reordering. In the modified problem reduction format, a goal may be reduced to several goals, called *subgoals*, and the goal will be solved if all the subgoals are. The order in which these subgoals are solved does not affect the solvability of the subgoals but it can greatly affect the efficiency of the theorem prover. We will investigate different heuristics for determining the order in which these subgoals are solved. We will demonstrate by experiments that it is important to order the subgoals reasonably. The general heuristic we obtain is to solve at each step the subgoal containing the most information, which can be measured in different ways.

Refinements to Depth-first Iterative Deepening Search. It is essential for the search strategy of a theorem prover to be able to use priority information to control the search. That is, the search strategy should be able to estimate the importance of the goals and always work on the most important available goals at any time. Depth-first iterative deepening search can not use priority effectively and we think this is a serious problem. To remedy this, we shall study two refinements to the depth-first iterative deepening search. The first refinement is based on a *priority system*. We try to approximate best-first search in a depth-first iterative manner. The second refinement is based on *proof complexity measures*. This refinement views the process of finding a proof as a process of incrementally constructing some instance of the input clauses. We will demonstrate that both refinements can provide substantial improvements to the efficiency of the theorem prover.

Using Semantic Information. Domain dependent knowledge, or semantic information, should be used in automatic theorem proving. Semantic information can be used to detect unachievable goals which can be deleted. Semantic information can also be used to select the inferences rules for a particular theorem. We will develop a semantic version of the modified problem reduction format that will accomplish both.

1.2. Related Work

A large body of knowledge in the field of automatic theorem proving has accumulated over the past few decades. It is beyond the scope of this thesis to give a general review of the field. Rather, we will discuss related works in the appropriate places. For a general introduction to the field, see [Chang&Lee 73, Loveland 78]. In general, however, the work described in this thesis may be of interest to people working in general search strategy, automatic theorem proving, machine learning and heuristic problem solving.

1.3. Organization of the Thesis

This thesis consists of several independent chapters. We will give an overview of each chapter. We make one note first. Throughout the thesis, some experimental results will be given. We use the problems mainly from [Stickel 88] as our standard test problems. There are 95 test problems in all.

In the next section, we will first briefly introduce first order logic. Some standard terminology and some new terminology will be defined. Then we will discuss the inference system of the theorem prover, the *modified problem reduction formation*. We will present the axioms, the inference rules and the soundness and completeness theorems without proofs. We will use some examples to explain the inference system. The material in this section is mainly from [Plaisted 88], to which readers are referred for a complete discussion. Lastly, we will give an overview on various search strategies.

In Chapter 2, we will show how depth-first iterative deepening search is used to implement the modified problem reduction format. We will also discuss how caching is done with depth-first iterative deepening search and discuss how to deal with repeated solutions when caching is performed.

In Chapter 3, we will discuss an interesting technique for making caching more efficient, *goal generalization*. Basically, goal generalization finds proofs for the most general solvable goals rather than the specific goals. We will show that goal generalization is a special case of *Explanation-Based Generalization* in machine learning and discuss how it can be implemented in the theorem prover by augmenting the inference rules.

In Chapter 4, we will discuss how subgoal reordering is performed in the prover. Subgoal reordering considers ordering conjunctive subgoals, that is, subgoals that have to be solved simultaneously. It is based on the observation that the order in which the subgoals of a goal are attempted is unimportant for the solvability of the subgoals and can be determined by some heuristics. We will discuss those heuristics and show how subgoal reordering is performed in the theorem prover.

In Chapter 5 and Chapter 6, we will discuss two refinements to the depth-first iterative deepening search. The first refinement, the *priority system*, incorporates the use of priority into the search strategy. The priority system approximates best-first search using depth-first iterative deepening search guided by the priority information of the priority function. Unlike subgoal reordering, which considers conjunctive subgoals, priority system considers disjunctive subgoals, that is, subgoals of which only one needs to be solved. The second refinement, the *proof complexity measure*, is based on a syntactic viewpoint of proof development in goal-subgoal systems. This viewpoint regards proof development as an incremental process of constructing an instance of the input clauses. This incremental process can be quantified to control depth-first iterative deepening search.

Chapter 7 discusses the use of semantic information in the theorem prover. A semantic variant of the modified problem reduction format is presented in which the inference rules are determined by some arbitrary interpretation. It generalizes Gelernter's system and is a true set-of-support strategy. We will discuss the issue of contrapositives in modified problem reduction format and other inference systems.

In Chapter 8, we will summerize the thesis work and propose some problems for future research.

1.4. Logical Foundations

The materials in the first two subsections are mainly from [Chang&Lee 73]. People who are familiar with first order logic and the concept of resolution theorem proving can skip the first subsection and the first half of the second subsection.

1.4.1. First Order Logic

Definition. The *alphabet* of a first order language consists of the following pairwise disjoint sets of symbols:

Logical connectives: $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\supset$ (implication), $\equiv$ (equivalence), universal quantifier $\forall$ (for all), existential quantifier $\exists$ (there exists).

Variables: a countably infinite set V = $\{v_0, v_1, \ldots,\}$.

Function Symbols: A (countable, possibly empty) set F of symbols. Each function symbol takes a specified number of arguments. If a function symbol f takes n arguments, we say that f is *n-place function symbol*, or f has *arity* n.

Constants: A (countable, possibly empty) set C of symbols.

Predicate Symbols: A (countable, possibly empty) set P of symbols. Each predicate symbol takes a specified number (possibly zero) of arguments. If a predicate symbol p takes n arguments, we say that p is *n-place predicate symbol*, or p has *arity* n.

Definition. A *term* is defined as follows:

1. A constant is a term.
2. A variable is a term.

3. If f is a n-place function symbol and $t_1, t_2, \ldots, t_n$ are terms, then $f(t_1, t_2, \ldots, t_n)$ is a term.

4. All terms are generated by applying rules 1-3 above.

Definition. An *atom* is of the form $p(t_1, t_2, \ldots, t_n)$ where p is a n-place predicate symbol and $t_1, t_2, \ldots, t_n$ are terms.

Definition. The *scope* of a quantifier occurring in a formula is the part of the formula to which the quantifier applies. An occurrence of a variable in a formula is *bound* if and only if the occurrence is within the scope of a quantifier employing the variable, or is the occurrence in that quantifier. An occurrence of a variable in a formula is *free* if and only if this occurrence of the variable is not bound. A variable is *free* in a formula if at least one occurrence of it is free in the formula. A variable is bound in a formula if at least one occurrence of it is bound.

Definition. A *formula* is defined as follows:

1. An atom is a formula.

2. If F and G are formulas, so are $\neg$ F, $(F \vee G)$, $(F \wedge G)$, $(F \supset G)$, and $(F \equiv G)$.

3. If F is formula and x is a variable, then $(\forall x)F$ and $(\exists x)F$ are formulas. We say, free occurrences of the variable x in F, or sometimes simply variable x, is *universally quantified* in $(\forall x)$ F and *existentially quantified* in $(\exists x)$ F.

4. Formulas are generated only by a finite number of applications of rules 1-3 above.

Definition. An *interpretation* of a formula consists of a *domain* D (nonempty set) and an assignment of "values" to each constant, function symbols, and predicate symbols occurring in the formula as follows:

1. To each constant, we assign an element in D.

2. To each n-place function symbol G, we assign a mapping G_I: $D^n \rightarrow D$.

3. To each n-place predicate symbol P, we assign a mapping P_I: $D^n \rightarrow \{T, F\}$, where *T* represents *true* and *F* represents *false*.

An interpretation I is *finite* if the domain D is finite. The *truth value* of a formula can be evaluated in an interpretation with domain D to be *T* or *F* according to the following rules:

1. If the truth values formula F and G are evaluated, then truth values of the formulas $\neg$ F, $(F \vee G)$, $(F \wedge G)$, $(F \supset G)$, and $(F \equiv G)$ can be evaluated using *truth tables*.

2. $(\forall x)$ F is evaluated to *T* if the truth value of F is evaluated to *T* for every element of D; otherwise it is evaluated to *F*.

3. $(\exists x)$ F is evaluated to *T* if the truth value of F is evaluated to *T* for at least one element of D; otherwise it is evaluated to *F*.

Definition. Two formulas F and G are said to be *equivalent* iff the truth values of F and G are the same under every interpretation of F and G.

Definition. If a formula F is evaluated to *T* in an interpretation I, we say I is a *model* of F, or I *satisfies* F. We denote the fact that I is a model of F using $I \mid = F$. We say a formula F is *consistent* (*satisfiable*) if F has a model. A formula F is *valid* iff every interpretation satisfies F. We denote that F is valid using $\mid = F$. A formula F is *inconsistent* (*unsatisfiable*) iff it has no model.

Definition. A formula F is a *logical consequence* of formulas $F_1, F_2, \ldots, F_n$ iff, for every interpretation I, if $F_1 \wedge \cdots \wedge F_n$ is true in I, F is also true in I. Given a set of formulas S and a formula G, we use $S \mid == G$ to denote the fact that G is a logical consequence of the formulas in S.

Theorem. Given formulas $F_1, F_2, \ldots, F_n$ and F, the following statements are equivalent:

1. $\{F_1, F_2, \ldots, F_n\} \mid == F$;
2. $(F_1 \wedge \cdots \wedge F_n) \supset F$ is valid; and
3. $F_1 \wedge \cdots \wedge F_n \wedge \neg F$ is inconsistent.

1.4.2. Clause Form and Herbrand Theorem

Definition. A formula F in first order logic is in *prenex normal form* iff the formula F is in the form of $(Q_1 v_1) \cdots (Q_n v_n)$ M where every $(Q_i v_i)$ $(i = 1, \ldots, n)$ is either $\forall v_i$ or $\exists v_i$, and M is a formula containing no quantifiers. $(Q_1 v_1) \cdots (Q_n v_n)$ is called the *prefix* and M is called the *matrix* of the formula F.

Definition. Let A be an atom. A and $\neg$A are called *literals*, and A is called a *positive literal* and $\neg$A is called a *negative literal*.

Definition. A formula F is in *conjunctive normal form* iff it is of the form $F_1 \wedge \cdots \wedge F_n$ where each formula F_i $(i = 1, \ldots, n)$ is a disjunction of literals. Every quantifier-free formula can be transformed into an equivalent conjunctive normal form.

Definition. A *clause* is a disjunction of literals. That is, a clause is of the form $L_1 \vee \cdots \vee L_n$ where $L_1, \ldots, L_n$ are literals. Variables in a clause are considered universally quantified. We call a clause a *unit clause* if it has only one literal. We call a clause a *Horn clause* if it contains at most one positive literal. We call a clause a *non-Horn clause* if it contains more than one positive literal. We call a clause a *negative clause* if it only contains negative literals. A set of clauses is regarded as a conjunction of all clauses in S. We use □ to denote the clause containing no literal, called the *empty clause*. The truth value of □ is always *F*.

Theorem. Each formula F can be represented by a set of clauses S such that F is inconsistent iff S is inconsistent.

Notes. We sketch a procedure to transform a formula F into a set of clauses S such that F is inconsistent iff S is.

1. Transform F into prenex normal form PF.
2. Transform the matrix of PF into a conjunctive normal form CPF.
3. Eliminate the existential quantifiers in the prefix of CPF by using Skolem functions.
4. Drop the remaining universal quantifiers. The result will be a set of clauses.

Definition. Given a set of clauses S, let H_0 be the set of constants appearing in S. If no constant appears in S, then H_0 is to consist of a single arbitrary constant. For i = 0, 1, 2 , . . . , let H_{i+1} be the union of H_i and the set of all terms of the form $f(t_1, \ldots, t_n)$ for all n-place function symbols f occurring in S, each where t_j (j = 1 , . . . , n) are members of the set H_i. H_∞ is called the *Herbrand universe* of S. The terms in H_∞ are called *ground terms*.

Definition. Given a set of clauses S, a *substitution* is a finite set of the form $\{t_1/v_1, \ldots, t_n/v_n\}$ where every v_i is a variable, every t_i is a term different from v_i, and no two elements in the set have the same variable after the stroke symbol. When all $t_1, \ldots, t_n$ are ground term, the substitution is called a *ground substitution*.

Definition. Given a set of clauses S, let $\theta = (t_1/v_1, \ldots, t_n/v_n)$ be a substitution and E be an expression. $E\theta$ is an expression obtained from E by replacing simultaneously each occurrence of the variable v_i in E by the term t_i. $E\theta$ is called an *instance* of E. $E\theta$ is called a *ground instance* of E if θ is a ground substitution. $E\theta$ is called a *variant* of E if all t_i (i = 1, 2 , . . . , n) are variables such that $t_m \neq t_k$ if $m \neq k$.

Herbrand Theorem. A set S of clauses is inconsistent if and only if there is a finite inconsistent set S' of ground instances of clauses of S.

Definition. A substitution θ is called a *unifier* for a set $\{E_1, \ldots, E_n\}$ if and only if $E_1\theta = \cdots = E_n\theta$. The set $\{E_1, \ldots, E_n\}$ is said to be *unifiable* if there is a unifier for it. A unifier θ for a set $\{E_1, \cdots, E_n\}$ is a *most general unifier* if and only if for each unifier β for the set there is a substitution λ such that $\beta = \theta\lambda$, that is, for each E_i (i = 1, 2 , . . . , n), $(E_i\theta)\lambda = E_i\beta$.

Unification Theorem. There is an algorithm that, for a finite nonempty set of expressions W, always terminates and finds a most general unifier for W, if one exists.

Definition. A *simplified first-order formula* is a quantifier-free first order formula containing no logical symbols other than $\neg$, $\wedge$ and $\vee$, and each negation symbol $\neg$ occurring in the formula is applied to an atom. A clause is a simplified first-order formula by definition.

Definition. Given an interpretation with domain D, an *interpretation instance* of a term or a simplified first-order formula W, denoted by W', is obtained from W by substituting elements in D for all variables occurring in W. For a simplied first-order formula W, we use $I \mid =_E W$ to denote the fact that there is an interpretation instance W' of W which is interpreted to be *T* by I; We use $I \mid =_U W$ to denote the fact that the interpretation I interprets all the interpretation instances W' of W to be *T*; We use $I \not\models_E W$ to denote $I \mid =_U \neg W$ and use $I \not\models_U W$ to denote $I \mid =_E \neg W$. We call an interpretation I a *model* for a simplified first-order formula W if $I \mid =_U W$. An interpretation I is a model for a set of simplified first-order formulae if it is a model for each formula in S.

Definition. A *Horn-like clause* is an expression $L :- L_1, L_2, \ldots, L_n$, which represents a clause $L \vee \neg L_1 \vee \neg L_2 \cdots \neg L_n$, where L is called the *head literal* and the *clause body* is the set of L_i's (i = 1 , . . . , n). A clause **C** is converted into a Horn-like clause **HC** as follows. One of the literals or the reserved literal FALSE is chosen as the head literal of **HC** and all other literals in **C** are negated and put in the clause body of **HC**. A clause containing n literals corresponds to n+1 Horn-like clauses. These n+1 Horn-like clauses are called *contrapositives* of each other. If two Horn-like clauses are contrapositives of one another, they are equivalent if we always evaluate FALSE to *F* in any interpretation.

1.4.3. Modified Problem Reduction Format

The modified problem reduction format is a refutational inference system for first order logic in clause form. To prove that a formula TH is valid (a theorem), we transform the negation of the formula ($\neg$TH) into a set of clauses TS. If TS is proven to be inconsistent, the formula TH will be proven to be valid. The modified problem reduction format accepts a set of clauses as its input and tries to prove the inconsistency of the set of clauses. The input clauses need to be given as Horn-like clauses. For each clause C, only one Horn-like clause HC will be needed. Specifically, the clause heads of the input clauses must be a positive literal (for clauses containing positive literals) or FALSE (for all-negative clauses). An input clause whose head is FALSE is called a *goal clauses*.

Definition. A *goal* or *subgoal* is an expression $\Gamma \rightarrow L$, where L is a literal and Γ is a list of literals. Γ is the *assumption list* of the goal $\Gamma \rightarrow L$. A subgoal $\Gamma \rightarrow L$ denotes the formula $L_1 \wedge \cdots \wedge L_n \supset L$ if $\Gamma = [L_1, \ldots, L_n]$. For simplicity, we say a subgoal $\Gamma \rightarrow L$ denotes the formula $\Gamma \supset L$. We call the goal $[\,] \rightarrow$ FALSE the *top-level goal* ($[\,]$ denotes the empty list). We abbreviate the subgoal $[\,] \rightarrow L$ as $\rightarrow L$. A *goal transformation* is an expression $\Gamma_1 \rightarrow L \Rightarrow \Gamma_2 \rightarrow L$, where Γ_2 is Γ_1, possibly with some literals appended at the front. For any list of literals Γ, $|\Gamma|$ denotes the length of Γ.

The inference rules for the modified problem reduction format consist of the *clause rules*, which are obtained from the input clauses, the *assumption axioms* and the *case analysis rule*. For each Horn-like clause $L :- L_1, L_2, \ldots, L_n$ in a set of clauses S, we have exactly one clause rule. We show the inference rules of the modified problem reduction format in Figure 1.1. The reader can refer to [Plaisted 88] for a complete discussion of the inference system. We give the soundness theorem and the completeness theorem without proofs.

Definition. Given a set of clauses S, we say the goal transformation $GT = \Delta_1 \rightarrow L \Rightarrow \Delta_2 \rightarrow L$ is *valid according to S*, denoted by $\vdash_S$ GT, if and only if

---

Clause Rule for Horn-like Clause $L :- L_1, L_2, \ldots, L_n$

$$\frac{[\Gamma_0 \rightarrow L_1 \Rightarrow \Gamma_1 \rightarrow L_1],\ [\Gamma_1 \rightarrow L_2 \Rightarrow \Gamma_2 \rightarrow L_2],\ \ldots,\ [\Gamma_{n-1} \rightarrow L_n \Rightarrow \Gamma_n \rightarrow L_n]}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_n \rightarrow L}$$

Assumption Axioms

$\Gamma \rightarrow L \Rightarrow \Gamma \rightarrow L$ if $L \in \Gamma$ L is a literal.

$\Gamma \rightarrow \neg L \Rightarrow \Gamma, \neg L \rightarrow \neg L$ L is a positive literal.

Case Analysis Rule

$$\frac{[\Gamma_0 \rightarrow L \Rightarrow \Gamma_1, \neg M \rightarrow L],\ \ [\Gamma_1, M \rightarrow L \Rightarrow \Gamma_1, M \rightarrow L] \quad |\Gamma_0| \le |\Gamma_1|}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_1 \rightarrow L}$$

Figure 1.1: Inference Rules for Modified Problem Reduction Format

1. $\Delta_1 = \Delta_2$ and $L \in \Delta_1$; or
2. L is a negative literal and $\Delta_2 = \Delta_1, L$; or
3. $\Delta_1 = \Delta_2$ and L is an instance of a unit clause in S; or
4. There is an instance of a Horn-like clause in S, $L :- L_1, L_2, \ldots, L_n$, and a sequence of goal transformations $GT_1, GT_2, \ldots, GT_n$, such that for each $GT_i = \Gamma_{i-1} \rightarrow L_i \Rightarrow \Gamma_i \rightarrow L_i$ $(i = 1, 2, \ldots, n)$, GT_i is valid according to S ($\vdash_S GT_i$), and $\Gamma_0 = \Delta_1$ and $\Gamma_n = \Delta_2$; or
5. There is a positive literal M such that two goal transformations, $GT_1 = \Delta_1 \rightarrow L \Rightarrow \Delta_2, \neg M \rightarrow L$ and $GT_2 = \Delta_2, M \rightarrow L \Rightarrow \Delta_2, M \rightarrow L$, such that $\vdash_S GT_1$ and $\vdash_S GT_2$.

Theorem 1.1 (Soundness Theorem. [Plaisted 88]): If $\vdash_S \Gamma_1 \rightarrow L \Rightarrow \Gamma_2 \rightarrow L$, then list Γ_1 is a prefix of list Γ_2 and $S \models \Gamma_2 \supset L$.

Theorem 1.2 (Completeness Theorem. [Plaisted 88]): If a set of clauses S is unsatisfiable, then $\vdash_S \rightarrow \text{FALSE} \Rightarrow \rightarrow \text{FALSE}$.

The idea of the modified problem reduction format is as follows. Suppose $\Gamma \rightarrow M$ is a goal. We attempt to prove this goal, detecting when subgoals of the form $\neg L$ (negative subgoals) occur. When a negative subgoal $\neg L$ is encountered, we assume temporarily that L is false, so $\neg L$ is true and the subgoal $\neg L$ succeeds. We indicate that we assumed $\neg L$ is true by adding $\neg L$ to the assumption list Γ (see the second assumption axiom in Figure 1.1). Later, we need also to consider the case in which L is true, by adding L to the assumption list Γ and using the case analysis rule.

We will give a simple example to illustrate what a proof would look like and how the proof would proceed. Consider the inconsistent clause set

$$\{\neg P \vee \neg Q, \neg P \vee Q, \neg Q \vee P, P \vee Q\}.$$

They are converted into the Horn-like clauses in Figure 1.2. All the inference rules for this problem are shown in Figure 1.3. One proof is shown in Figure 1.4. The labels in the parenthesis in Figure 1.4 indicate which rules are used to carry out the valid goal transformation. For example,

$$(A2) \rightarrow \neg Q \Rightarrow \neg Q \rightarrow \neg Q$$

means that the assumption axiom A2 is used to solve $\rightarrow \neg Q$ to obtain the solution $\neg Q \rightarrow \neg Q$.

In the proof of Figure 1.4, the subgoals of a goal appears immediately on top of the goal. As a result, the proof should be read from bottom to top. The chronology of the proof in Figure 1.4 is as follows. First $\rightarrow$ FALSE (which is equivalent to $[\ ] \rightarrow$ FALSE) is the goal. This creates $\rightarrow P$ as one of the two subgoals (using clause rule R1), which creates $\rightarrow \neg Q$ as a subgoal (using clause rule R4). $\rightarrow \neg Q$ is solved by adding an assumption to the assumption list (using assumption axiom A2). So the solution to $\rightarrow \neg Q$ is $\neg Q \rightarrow \neg Q$. This leads to the solution $\neg Q \rightarrow P$ to the subgoal $\rightarrow P$. This leads to the second subgoals $\neg Q \rightarrow Q$ of $\rightarrow$ FALSE. To solve $\neg Q \rightarrow Q$, subgoal $\neg Q \rightarrow P$ is created (using rule R2), which in turn creates subgoal $\neg Q \rightarrow \neg Q$. $\neg Q \rightarrow \neg Q$ is solved by the first assumption axiom (A1). As a result, $\neg Q \rightarrow P$ is the solution to subgoal $\neg Q \rightarrow P$ and $\neg Q \rightarrow Q$ is the solution to subgoal $\neg Q \rightarrow Q$. This leads to the solution $\neg Q \rightarrow$ FALSE to the top-level goal $\rightarrow$ FALSE. We then apply a case analysis rule (S), creating a subgoal $Q \rightarrow$ FALSE. From subgoal $Q \rightarrow$ FALSE, one of two subgoals $Q \rightarrow P$ (using clause rule R1) is created. $Q \rightarrow P$ is proven using clause R3 and assumption axiom A1. The second subgoal $Q \rightarrow Q$ to the subgoal $Q \rightarrow$ FALSE is created and proven using assumption axiom A1. From the two solutions $\neg Q \rightarrow$ FALSE and $Q \rightarrow$ FALSE, we conclude $\rightarrow$ FALSE.

The modified problem reduction format is a *back chaining* inference system. It starts with the conclusion it wants to establish. In this case, it starts with the top-level goal [ ] $\rightarrow$ FALSE (or $\rightarrow$ FALSE) (this is to establish the inconsistency of the input clause set). To establish a goal, it generates a set of subgoals using either clause rules or the case analysis rule; the goal will become the *parent goal* of the subgoals; if the set of subgoals are proven, the goal itself will be proven. In general, a goal or subgoal may have several *ancestor goals*. This style of back chaining is the same as the back chaining of Prolog. Since the modified problem reduction format also handles non-Horn clauses, it can be regarded as an extension of Prolog-style Horn clause logic programming [Lloyd 84] to

---

false : – P, Q.
Q : – P.
P : – Q.
P : – ¬Q.

Figure 1.2: Horn-like Clauses for Clause Set $\{\neg P \vee \neg Q, \neg P \vee Q, \neg Q \vee P, P \vee Q\}$

$$\text{R1:}\quad \frac{[\Gamma_0 \to P => \Gamma_1 \to P],\ [\Gamma_1 \to Q => \Gamma_2 \to Q]}{\Gamma_0 \to \text{false} => \Gamma_2 \to \text{false}} \qquad (\text{false} :- P, Q.)$$

$$\text{R2:}\quad \frac{\Gamma_0 \to P => \Gamma_1 \to P}{\Gamma_0 \to Q => \Gamma_1 \to Q} \qquad (Q :- P)$$

$$\text{R3:}\quad \frac{\Gamma_0 \to Q => \Gamma_1 \to Q}{\Gamma_0 \to P => \Gamma_1 \to P} \qquad (P :- Q)$$

$$\text{R4:}\quad \frac{\Gamma_0 \to \neg Q => \Gamma_1 \to \neg Q}{\Gamma_0 \to P => \Gamma_1 \to P} \qquad (P :- \neg Q)$$

$$\text{A1:}\quad \Gamma \to L => \Gamma \to L \ \text{ if } L \in \Gamma \qquad L \text{ is a literal}$$

$$\text{A2:}\quad \Gamma \to \neg L => \Gamma, \neg L \to \neg L \qquad L \text{ is positive}$$

$$\text{S:}\quad \frac{[\Gamma_0 \to L => \Gamma_1, \neg M \to L],\ \ [\Gamma_1, M \to L => \Gamma_1, M \to L] \qquad |\Gamma_0| \leq |\Gamma_1|}{\Gamma_0 \to L => \Gamma_1 \to L}$$

Figure 1.3: Inference Rules for the Example of Figure 1.2

$$\text{(S)}\ \frac{\text{(R1)}\ \dfrac{\text{(R4)}\ \dfrac{\text{(A2)}\ \to \neg Q => \neg Q \to \neg Q}{\to P => \neg Q \to P} \qquad \text{(R2)}\ \dfrac{\text{(R3)}\ \dfrac{\text{(A1)}\ \neg Q \to \neg Q => \neg Q \to \neg Q}{\neg Q \to P => \neg Q \to P}}{\neg Q \to Q => \neg Q \to Q}}{\to \text{FALSE} => \neg Q \to \text{FALSE}} \qquad \text{(R1)}\ \dfrac{\text{(R3)}\ \dfrac{\text{(A1)}\ Q \to Q => Q \to Q}{Q \to P => Q \to P} \qquad \text{(A1)}\ Q \to Q => Q \to Q}{Q \to \text{FALSE} => Q \to \text{FALSE}}}{\to \text{FALSE} => \ \to \text{FALSE}}$$

Figure 1.4: A Proof for the Example of Figure 1.2

full first order logic. As a matter of fact, the modified problem reduction format without the assumption axioms and case analysis rule is complete for Horn clause logic. As we have noted, modified problem reduction does not need contrapositives and handles non-

Horn clauses using case analysis.

There are other extensions [Loveland 87, Stickel 88] of Prolog-style Horn clause logic programming to full first order logic. One of the earliest is *model elimination* [Loveland 69]. This method is very similar to Prolog but has an extra inference rule, called *basic reduction*, which tests whether a subgoal (a literal in this case) is complementary to one of its ancestor goals. Model elimination has been efficiently implemented by Stickel on a Symbolics [Stickel 88]. Stickel's implementation has achieved about 2,000 LIPs. This method has also been implemented by a group at CMU on a multiprocessor using or-parallel Prolog technology [Bose&al 88]. Another extension is Loveland's near-Horn Prolog [Loveland 87, Loveland 88] which is an extension of Prolog to "near-Horn" clauses but may be applied to more general clause sets as well. Informally, "near-Horn" refers to a clause set that does not contain many non-Horn clauses and those non-Horn clauses do not contain very many extra positive literals. One disadvantage of model elimination is its use of all the contrapositives of the clauses, since contrapositives may incur loss of control over the search. Loveland's near-Horn Prolog does not need contrapositives, nor does the modified problem reduction format.

1.5. Search Strategies

All theorem provers involves search, a systematic trial-and-error exploration of alternatives, except where the problem domain is limited and a special decision procedure is available. Search takes place in a *problem space*. A problem space consists of a set of states[1] of the problem and a set of operators that change the state of the problem. A *problem instance* is a problem space together with an initial state and some goal states. The *search problem* is to find a sequence of operators that map the initial state to a goal state. Such a sequence is called a *solution* to the problem.

There are many formalizations of the search problem [Nilsson 80, VanderBrug&Minker 75] and these formalizations can usually be transformed into one another. We will formalize the search problem as a path-finding problem in a graph following [Nilsson 80]. In this formalization, problem spaces are represented by graphs in which the states of the space are represented by nodes, and the operators by arcs between nodes. Solutions to the problems are represented by paths between nodes. We will first introduce some graph theory terminology [Nilsson 80]. Then we will briefly describe some

[1] A *state* may represent a goal, a clause, a problem, etc, depending on the context. In this thesis, the term state usually represents goals or subgoals.

search strategies [Pearl&Korf 87].

A *directed graph* consists of a countable set of *nodes* and a set of pairs of nodes called *arcs*. The arc $<s_1, s_2>$ is said to be *directed* from node s_1 to node s_2, node s_2 is called a *successor* of node s_1, and node s_1 is called a *parent* of node s_2. We are only interested in the graphs in which each node can have only a finite number of successors.

A sequence of nodes $(s_0, s_1, s_2, \ldots, s_n)$, with each node s_i a successor of s_{i-1} for i = 1, 2, $\cdots$, n, is called a *path of length n* from node s_0 to node s_n. If a path exists from node s_1 to node s_2, then node s_2 is called *accessible* from node s_1. State s_2 is a *descendant* of node s_1, and node s_1 is an *ancestor* of node s_2. A *cycle* is a path of length greater than 1 from a node to itself. A graph is *acyclic* if it has no cycles; otherwise it is *cyclic*.

Often it is convenient to assign positive *costs* to arcs. We use the notation $c(s_1, s_2)$ to denote the cost of an arc directed from node s_1 to node s_2. The cost of a path between two nodes is the sum of the costs of all the arcs connecting the nodes on the path. In some problems, we want to find that path having *minimal* cost between two nodes.

A *tree* is a special case of an acyclic graph in which each node has at most one parent. A node in the tree having no parent is called a *root node*. There may be more than one root nodes in a tree. A node in the tree having no successors is called a *leaf*. We say the root node is of *depth zero*. The depth of any other node in the tree is defined to be the depth of its parent plus 1. We define the *node branching factor* of a node to be the number of successors of the node.

A graph may be specified either explicitly or implicitly. In an explicit specification, the nodes and arcs (with associated costs) are explicitly given by a table. The table might list every node in the graph, its successors, and the costs of the associated arcs. Explicit specification is obviously not practical for large graphs and impossible for those having an infinite set of nodes.

A *search strategy* generates (makes explicit) part of an implicitly specified graph. This implicit specification is given by the *initial node*, s_0, and a *successor operator* that is applied to a node to give *all* of the successors of that node (and the costs of the associated arcs). We call this process of applying the successor operator to a node, *expanding* the node. Expanding s_0, the successors of s_0, ad infinitum, makes explicit the graph that is implicitly defined by s_0 and the successor operator.

Simply stated, the path-finding problem is to find a path (perhaps having minimal cost) between a given node s_0 (the initial node) and any member of a set of nodes (final nodes). A path between the initial node and a final node is called a *solution* and the length (cost) of the path is the *length (cost)* of the solution. For our discussion, we will assume that all the graphs are trees with the root node representing the initial state, leaves representing the goal states, and each node in the tree having the same node branching factor. We call such graphs *search trees*. In a search tree, each node has a unique depth.

Breadth-First Search. Breadth-first search generates the states of the search tree in order of their depth. Breadth-first search never generates a deeper state in a tree until all the shallower nodes have been generated. If the solution length is d and the state branching factor is b, the average time complexity of breadth-first search is $O(b^d)$, assuming that the successor operator takes constant time for all nodes. Since all states at each level of the search tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of states stored, the average space complexity of breadth-first search is also $O(b^d)$. As a result, breadth-first search is severely space-limited in practice, which is its main drawback.

Best-First Search and **A* Search**. Best-first search is one of the simplest and most general search strategies. Best-first search always expands the state that is most promising according to some *priority function*, or *evaluation function*. Breadth-first search is a special case of best-first search where we always select the state with the smallest depth, for example. Best-first search maintains two data structures, the Open and Closed lists. The Closed list contains those states already visited in the search, and disallows revisiting old nodes, thereby preventing infinite loops. The Open list contains the currently active states, in order of their priority values, from which the next state to expand is selected. The search strategy begins with just the root state. At each cycle, the best node on the Open list is expanded, generating all its successors, and is then placed on the Closed list. After eliminating those successors already on the Closed list, the priority function is applied to the remaining successors, and they are placed on the Open list in order of their priority values. The search terminates when a final state is selected from Open list for expansion. A* search [Hart&al 68, Nilsson 80] is a special case of best-first search. A *heuristic function*, is used to order the states in the Open list. The heuristic value of a state s is of the form g(s)+h(s), where g(s) is the cost so far in reaching the state s from the initial state and h(s) is the heuristic estimate of the cost remaining to reach a goal state for the state s. At each point, the state with the lowest heuristic value is chosen for expansion. An important result is that A* search always finds an optimal (cheapest) path to a final state if the heuristic function h(s) never overestimates the actual cost from any state s to a goal state [Hart&al 68].

The main drawback of A* search, and indeed of any best-first search, including breadth-first search, is its memory requirement [Huyn 80]. Since the entire Open list must be saved, A* search is severely space-limited in practice and is no more practical than breadth-first search on current machines. This limitation is removed by the iterative-deepening-A* search.

Many resolution-based theorem provers employ some kind of best-first search strategy [Greenbaum 86, Overbeek&al 76, Wang&Bledsoe 87]. The priority functions usually measure the "syntactic complexity" of the states (recall that states can represent goals, clauses, etc, depending on the problem). A simple example of the measurement of the complexity would be the number of symbols in the clauses. Much more sophisticated measurements of complexity have been studied in [Overbeek&al 76, Wang&Bledsoe 87]. These priority functions are largely *ad hoc*, but they are absolutely essential for the success of these theorem provers.

Depth-First Search. Depth-first search expands the states in a last-in, first-out order. It can be seen as a special case of best-first search where the state to be expanded is always the state with the biggest depth. Since depth-first search stores only the current search path, it does not suffer from the disadvantage of requiring too much memory. In practice, depth-first search is time-limited rather than space-limited. But depth-first search has one problem – it may not be complete, i.e., it may fail to find any solution even if one exists. This limitation is overcome by depth-first iterative deepening search.

Depth-First Iterative Deepening. To remedy the incompleteness of depth-first search, a new search strategy, depth-first iterative deepening (henceforth DFID), has been formalized [Korf 85, Stickel&Tyson 85]. This strategy involves repeatedly performing exhaustive depth-first search with increasing cut-off bounds. Depth-first iterative deepening search approximates breadth-first search in depth-first fashion. It does not suffer neither the drawbacks of breadth-first or depth-first search. Since at any given point DFID is executing a depth-first search, the space complexity of DFID is only $O(d)$ where d is the length of the solution. The asymptotic time complexity of DFID is $O(b^d)$.

Iterative-Deepening-A*. To approximate A* search in depth-first fashion, iterative-deepening-A* (IDA*) is proposed in [Korf 85]. Each iteration of IDA* is a complete depth-first search that keeps track of the cost $f(n) = g(n) + h(n)$ of each node generated. As soon as this cost exceeds the cut-off bound, that branch is cut off, and the search backtracks to the most recently generated state. The cut-off bound starts with the heuristic estimate of the initial state and in each successive iteration is increased to the minimum value that exceeded the previous cut-off bound. In the same way that depth-first iterative-deepening reduced the space complexity of breadth-first search, IDA* drastically reduces the memory requirement of A* search without sacrificing optimality of the solution found [Korf 85]. The memory requirement of IDA* is linear in the solution length.

2. Depth-first Iterative Deepening Search

In this chapter, we will show how depth-first iterative deepening search (henceforth DFID) is used to implement the modified problem reduction format. We will also discuss how caching is done with depth-first iterative deepening search and discuss how to deal with repeated solutions when caching is performed.

2.1. Depth-first Iterative Deepening Search

DFID has been the subject of some study recently [Korf 85, Stickel&Tyson 85]. This strategy involves repeatedly performing depth-first search with increasing cut-off bounds. Although this search strategy may appear to be naive and costly, analysis shows that depth-first iterative deepening search requires only $\frac{b}{b-1}$ times as many operations as breadth-first search, where b is the branching factor [Stickel&Tyson 85] of the search tree. An important advantage of this search strategy is that it requires minimal storage (linear to the solution length) to operate.

In this section, we show how the modified problem reduction format is implemented using DFID. We emphasize the fact that it is a simplified discussion. We will leave out discussions on how to detect loops in the search paths, how to simplify subgoals, how to control splitting, how to alternate between forward chaining and backward chaining, etc. The term rewriting facility to handle equality is not discussed either. For details see [Plaisted 88, Smith&Plaisted 88].

The implementation uses a Prolog clause to represent each inference rule. These Prolog clauses encode the information about the proof structures as well as information about the search control. We use not(L) to represent a negative literal $\neg L$ and $L :- \Gamma$ to represent the subgoal $\Gamma \rightarrow L$. If A is a literal and B is a list of literals, [A | B] represents the list obtained by attaching A at the front of B. The Prolog representations for all inference rules are shown in Figure 2.1. The code to implement depth-first iterative deepening search is shown in Figure 2.2.

The procedure *achieve* in the representation performs depth-first search with cut-off bound specified in one of its arguments. In a procedure call *achieve((*$L :- \Gamma_1$*),(*$L :- \Gamma_2$*), Start, End, Bound)*, *Bound* specifies the cut-off bound, *Start* indicates how much of the cut-off bound *Bound* has been used and *End* will be the sum of *Start* and the cost of solving $L :- \Gamma_1$ to obtain $L :- \Gamma_2$. Consequently, the difference between *End* and *Start* (*End* — *Start*) will be the *cost* of solving $L :- \Gamma_1$ to obtain $L :- \Gamma_2$. The iterative deepening behavior is controlled in the procedure *search* where, in case the current round of depth-first search fails, the procedure *increment* is called to increase the current search bound and *achieve* will be called with the increased search bound. The basic function of the procedure *match* is to perform the unification operations. The procedure *clause_cost* estimates the cost of an input clause, that is, how much the search bound will be decremented when using the input clause. For example, the cost of an input clause can be the number of literals in the clause body. The procedure *match_cost* estimates the cost of a unification operation, which is, similar to the cost of an input clause, how much is decremented as the result of the unification operation. Both *clause_cost* and *match_cost* can be defined by the user arbitrarily. See Chapter 6 for more discussions on this. The multiplier *breadth_factor* is used to balance the search. If *breadth_factor* is 0, the search is depth-first, but if *breadth_factor* is greater than 0, then the search will have a breadth-first component, since the cost of solving one subgoal will affect the search for solving its conjunctive subgoals.

We note two special cases, which we call the *size prover* and the *depth prover*. In the size prover, the procedure *clause_cost* returns the number of the literals in the clause body, *match* only performs the unification operation and *match_cost* returns 0. We set *breadth_factor* to 1. We call the provers using this setup the *size prover*, because we use the size of the proof trees to control the search. We note that the search in the size prover has a heavy breadth-first component since *breadth_factor* is set to 1. We show the Prolog representation for the inference rules of the size prover in Figure 2.3. The representations for the assumption axioms are the same as in Figure 2.1. For example, for the Horn-like clause FALSE :− P, Q, the corresponding Prolog clause will be as in Figure 2.4. Intuitively, the size prover uses the number of arcs or nodes in the proof tree constructed so far to control the search. In the depth prover, we set *breadth_factor* to 0. The depth provers use mainly the recursion level of subgoal generations, which is the depth of the

% Representation for input clause $L :- L_1, L_2, \ldots, L_n$.

$achieve((L_0 :- \Gamma_0),(L_0 :- \Gamma_n), Start, End, Bound) :-$

$\quad match(L_0, L, [L_1, L_2, \ldots, L_n], V, [V_1, V_2, \ldots, V_n]),$

$\quad E_0$ is $Start + clause\_cost(L :- L_1, L_2, \ldots, L_n) + match\_cost(L_0, V),$

$\quad E_0 \le Bound,$

$\quad F_1$ is $E_0 + match\_cost(L_1, V_1),$

$\quad achieve((L_1 :- \Gamma_0),(L_1 :- \Gamma_1), F_1, E_1, Bound),$

$\quad F_2$ is $E_0 + match\_cost(L_2, V_2) + breadth\_factor \times (E_1 - F_1)$

$\quad achieve((L_2 :- \Gamma_1),(L_2 :- \Gamma_2), F_2, E_2, Bound),$

$\quad \ldots$

$\quad F_i$ is $E_0 + match\_cost(L_i, V_i) + breadth\_factor \times (E_{i-1} - F_{i-1}),$

$\quad achieve((L_i :- \Gamma_{i-1}),(L_i :- \Gamma_i), F_i, E_i, Bound),$

$\quad \ldots$

$\quad F_n$ is $E_0 + match\_cost(L_n, V_n) + breadth\_factor \times (E_{n-1} - F_{n-1}),$

$\quad achieve((L_n :- \Gamma_{n-1}),(L_n :- \Gamma_n), F_n, E_n, Bound),$

$\quad End$ is $\max\{E_1, E_2, \ldots, E_n\}$.

% Representation for a unit clause L:

$achieve((L_0 :- \Gamma), (L_0 :- \Gamma), S, End, Bound) :-$

$\quad match(L_0, L, [\,], V, [\,]), S + match\_cost(L, V) \le bound,$

$\quad End$ is $S + match\_cost(L, V)$.

% Representations for the assumption axioms:

$achieve((L :- \Gamma), (L :- \Gamma), S, S, Bound) :- member(L, \Gamma).$

$achieve((\neg L :- \Gamma), (\neg L :- [\neg|\Gamma]), S, S, Bound).$

% Representation for the case analysis rule:

$achieve((L :- \Gamma_0), (L :- \Gamma), Start, End, Bound) :-$

$\quad achieve((L :- \Gamma_0),((L :- [not(M)|\,\Gamma]), Start, E_1, Bound),$

$\quad F_1$ is $Start + breadth\_factor \times (E_1 - Start),$

$\quad achieve((L :- [M \mid \Gamma]), (L :- [M \mid \Gamma]), F_1, E_2, Bound),$

$\quad length(\Gamma_0) \le length(\Gamma), End$ is $\max\{E_1, E_2\}$.

Figure 2.1: Representations for Inference Rules

```
search(S, Limit) :– S > Limit, !, fail.
search(S, Limit) :– achieve((false :- [ ]), (false :- [ ]), 0, Cost, S).
search(S, Limit) :– S1 is S + increment(S), search(S1,Limit).
```

Figure 2.2: Depth-first Iterative Deepening Search

proof tree constructed so far, to control the search. We show the representation of the inference rules in the depth prover in Figure 2.5. Consider again the clause FALSE :– P, Q. The Prolog clause for it is shown in Figure 2.6.

% Representation for input clause $L :- L_1, L_2, \ldots, L_n$:

$$
\begin{aligned}
&\text{achieve}((L :- \Gamma_0),(L :- \Gamma_n), \text{Start}, \text{End}, \text{Bound}) :- \\
&\quad E_0 \text{ is Start} + \text{clause\_cost}(L :- L_1, L_2, \ldots, L_n), \\
&\quad E_0 \le \text{Bound}, \\
&\quad \text{achieve}((L_1 :- \Gamma_0),(L_1 :- \Gamma_1), E_0, E_1, \text{Bound}), \\
&\quad \text{achieve}((L_2 :- \Gamma_1),(L_2 :- \Gamma_2), E_1, E_2, \text{Bound}), \\
&\quad \ldots \\
&\quad \text{achieve}((L_i :- \Gamma_{i-1}),(L_i :- \Gamma_i), E_{i-1}, E_i, \text{Bound}), \\
&\quad \ldots \\
&\quad \text{achieve}((L_n :- \Gamma_{n-1}),(L_n :- \Gamma_n), E_{n-1}, \text{End}, \text{Bound}).
\end{aligned}
$$

% Representation for splitting rule:

$$
\begin{aligned}
&\text{achieve}((L :- \Gamma_0), (L :- \Gamma), \text{Start}, \text{End}, \text{Bound}) :- \\
&\quad \text{achieve}((L :- \Gamma_0),((L :- [\text{not}(M) \mid \Gamma]), \text{Start}, E_1, \text{Bound}), \\
&\quad \text{achieve}((L :- [M \mid \Gamma]), (L :- [M \mid \Gamma]), E_1, \text{End}, \text{Bound}), \\
&\quad \text{length}(\Gamma_0) \le \text{length}(\Gamma).
\end{aligned}
$$

Figure 2.3: The Size Prover

$$
\begin{aligned}
&\text{achieve}((\text{FALSE} :- \Gamma_0),(\text{FALSE} :- \Gamma_2), \text{Start, End, Bound}) :- \\
&\quad E_0 \text{ is Start + clause\_cost(FALSE} :- \text{P, Q)}, \\
&\quad E_0 \leq \text{Bound}, \\
&\quad \text{achieve}((P :- \Gamma_0),(P :- \Gamma_1), E_0, E_1, \text{Bound}), \\
&\quad \text{achieve}((Q :- \Gamma_1),(Q :- \Gamma_2), E_1, \text{End ,Bound}).
\end{aligned}
$$

Figure 2.4: Clause Rule Representation in Size Prover

2.2. Caching

As we have said, DFID involves repeatedly performing depth-first search with increasing cut-off bounds [Stickel&Tyson 85]. A subgoal could be solved many times and, consequently, repeated work would be performed. The purpose of caching is to avoid repeated work by solving each subgoal only once. To implement caching, we should record which subgoals have been attempted and all the solutions that are derived for all the subgoals, which can be used later to solve other subgoals. Let's explain caching in more detail.

2.2.1. How Caching is Done

Suppose G is a subgoal. If a call *achieve(G, S, Start, End, Bound)* (see Figure 2.1) returns, the returned subgoal S is a solution to the subgoal G within the *effort bound* Bound − Start. All the solutions to the subgoal G within the effort bound D (= Bound − Start) will be derived if *achieve(G, S, Start, End, Bound)* is executed until it fails. A *done clause* done(D, G) indicates that the subgoal G has been worked on and all solutions for G have been derived with the *effort bound* D. When a subgoal $L :- \Gamma$ is to be attempted with an effort bound D, that is, *achieve(*$L :- \Gamma$*, S, Start, End, Bound)* is called with D being equal to Bound − Start, the prover first checks to see whether there is any solution which could be used to solve $L :- \Gamma$, that is, whether there is a solution $L_1 :- \Gamma_1$ that is cached earlier, that L unifies with L_1 and the cost of using $L_1 :- \Gamma_1$ (see below) is no greater than D. If all the solutions for $L :- \Gamma$ fail to lead to a proof, the prover is about to solve $L :- \Gamma$ using the input clauses. However, before it proceeds the prover checks whether there is a done clause done(D_1 , ($L_1 :- \Gamma_1$)) such that D_1 is greater or equal to D and $L :- \Gamma$ is an instance or a variant of $L_1 :- \Gamma_1$. If there is such a done clause, the prover does not need to work on $L :- \Gamma$ at all since all the possible solutions for it have been generated already when the subgoal $L_1 :- \Gamma_1$ was tried with effort bound D_1. If

% Representation for input clause $L :- L_1, L_2, \ldots, L_n$:

$\text{achieve}((L_0 :- \Gamma_0),(L_0 :- \Gamma_n), \text{Start}, \text{End}, \text{Bound}) :-$
$\quad \text{match}(L_0, L, [L_1, L_2, \ldots, L_n], V, [V_1, V_2, \ldots, V_n]),$
$\quad E_0 \text{ is Start} + \text{clause\_cost}(L :- L_1, L_2, \ldots, L_n) + \text{match\_cost}(L_0, V),$
$\quad E_0 \leq \text{Bound},$
$\quad F_1 \text{ is } E_0 + \text{match\_cost}(L_1, V_1),$
$\quad \text{achieve}((L_1 :- \Gamma_0),(L_1 :- \Gamma_1), F_1, E_1, \text{Bound}),$
$\quad F_2 \text{ is } E_0 + \text{match\_cost}(L_2, V_2),$
$\quad \text{achieve}((L_2 :- \Gamma_1),(L_2 :- \Gamma_2), F_2, E_2, \text{Bound}),$
$\quad \ldots$
$\quad F_i \text{ is } E_0 + \text{match\_cost}(L_i, V_i),$
$\quad \text{achieve}((L_i :- \Gamma_{i-1}),(L_i :- \Gamma_i), F_i, E_i, \text{Bound}),$
$\quad \ldots$
$\quad F_n \text{ is } E_0 + \text{match\_cost}(L_n, V_n),$
$\quad \text{achieve}((L_n :- \Gamma_{n-1}),(L_n :- \Gamma_n), F_n, E_n, \text{Bound}),$
$\quad \text{End is } \max\{E_1, E_2, \ldots, E_n\}.$

% Representation for the splitting rule:

$\text{achieve}((L :- \Gamma_0), (L :- \Gamma), \text{Start}, \text{End}, \text{Bound}) :-$
$\quad \text{achieve}((L :- \Gamma_0),((L :- [\text{not}(M) \mid \Gamma]), \text{Start}, E_1, \text{Bound}),$
$\quad \text{achieve}((L :- [M \mid \Gamma]), (L :- [M \mid \Gamma]), \text{Start}, E_2, \text{Bound}),$
$\quad \text{length}(\Gamma_0) \leq \text{length}(\Gamma),$
$\quad \text{End is } \max\{E_1, E_2\}.$

Figure 2.5: The Depth Prover

```
achieve((FALSE :− Γ0),(FALSE :− Γ2), Start, End, Bound):−
    match(L, FALSE, [P,Q], V, [V1, V2]),
    E0 is Start + clause_cost(FALSE :− P, Q) + match_cost(FALSE, V),
    E0 ≤ Bound,
    F1 is E0 + match_cost(P, V1),
    achieve((P :− Γ0),(P :− Γ1), F1, E1 ,Bound),
    F2 is E0 + match_cost(Q, V2),
    achieve((Q :− Γ1),(Q :− Γ2), F2, E2 ,Bound),
    End is max{E1, E2}.
```

Figure 2.6: Clause Rule Representation in Depth Prover

there is no such done clause, a done clause done(D, (L :− Γ)) will be asserted to indicate that subgoal L :− Γ has been worked on or is being worked on with effort bound D. Then the prover will proceed to solve L :− Γ, by continuing executing *achieve(*L :− Γ*, S, Start, End, Bound)*. Each solution to L :− Γ will be cached together with the effort of deriving the solution (End − Start) and the number of input clauses are used to derive the solution.

We show the part of Prolog code for implementing depth-first iterative deepening search with caching in Figure 2.7, in addition to the Prolog code for input clauses. Four procedures, *record_solution, use_solution, record_done* and *already_done*, implement the caching by asserting the appropriate Prolog clauses. The procedure *solution_cost* determines the cost of using a solution. The cost of a solution L :− Γ is

$$(S \times \text{solution\_size\_mult}) + (P \times \text{proof\_size\_mult}) + (C \times \text{clause\_count\_mult})$$

where S is the size of the solution (usually the number of symbols in the solution), P is the effort incurred in deriving this solution and C is the number of the input clauses used to derive this solution. All three multipliers in the formula can be set by the user.

2.2.2. Repeated Solutions

When caching is performed, all the solutions to the subgoals are recorded. It is possible that a newly derived solution is a repetition of a solution already cached. We call such solution a *repeated solution*. Formally speaking, a newly derived solution S (L :− B) for a subgoal G is a repeated solution if there is already a solution S1 (L1 :− B1) cached such that

```
% Use recorded solutions:
    achieve((L :– Γ), (L :– Γ1), Start, End, Bound) :–
        use_solution((L1 :– Γ1)),
        match(L, L1, [ ], V, [ ]),
        E1 is Start + solution_cost((L :– Γ1)),
        E2 is Start + match_cost(L, V),
        End is max{E1, E2}, End ≤ Bound.

% Check whether the subgoal has been worked on before:
    achieve((L :– Γ), (L :– Γ1), Start, End, Bound) :–
        E is Bound – Start,
        already_done(E, (L :– Γ)), !, fail.

% Assert the the necessary done clause:
    achieve((L :– Γ), (L :– Γ1), Start, End, Bound) :–
        E is Bound – Start,
        record_done(E, (L :– Γ)), fail.

% To solve a subgoal using the input clauses and
% record all the solutions generated:
    achieve((L :– Γ0), (L :– Γ), Start, End, Bound) :–
        achieve((L :– Γ0),((L :–[not(M) | Γ), Start, E1, Bound),
        F1 is Start + breadth_factor × E1,
        achieve((L :–[M | Γ]), (L :– [M | Γ]), F1, E2, Bound),
        length(Γ0) ≤ length(Γ),
        End is max{E1, E2}, E is End – Start,
        record_solution(E, (L :– Γ)).
```

Figure 2.7: Depth-first Iterative Deepening Search with Caching

1. L is an instance or variant of L1; and
2. Each literal in B is subsumed by a literal in B1; and
3. The effort of deriving S is no bigger than that of S1; and

4.1. S and S1 are solutions to the same subgoal G; or

4.2. S1 is derived before subgoal G is generated.

To detect whether a solution S is a repeated solution, we need to store, with each solution S, the identification of the subgoal for which S is a solution and the time S is generated, in addition to the effort of deriving S. For this purpose, we keep a counter which serves both as identification and time stamp for subgoals. Each newly generated subgoal will have the current value of the counter as its identification and the counter will be incremented by 1 whenever a new subgoal is generated. When a solution is derived, the current value of the counter is used as the time stamp for this solution.

When a repeated solution is generated, the search should fail on the current path. Otherwise the search will be repeated, because the subgoal is solved before, either by using the old solutions or the input clauses. The efficiency of the prover will suffer if repeated solutions are not handled properly on some problems. This is not very important for Horn problems because repeated searches will be prevented by caching. But caching will not prevent some of the repeated search performed by doing case analysis, thus it is specially important for non-Horn problems that search fail on repeated solutions.

2.2.3. Experimental Results

Table A.1 in Appendix A shows the performance results of the theorem prover on the test problems both with caching and without caching. The results are summarized in Table 2.1. The theorem prover gets proofs for 83 of the 95 problems while caching. The same prover without caching only gets 77. Note that the performance of the prover without caching degenerates rather quickly, especially for non-Horn problems.

Our experience shows that caching usually helps from our experience. It makes the theorem prover more reliable to use caching, in spite of the fact that the inference rate (an *inference* is performed when an input clause is used or an old solution is used in the

Table 2.1. Summary Data for Caching

	Prover with Caching	Prover without Caching
Average Time Per Theorem	224.30	491.41
Average Inferences Per Theorem	1245	129196

prover) is much lower when caching is performed. Of course, caching can take a lot of memory [Plaisted 88].

We have tested the theorem prover using different options dealing with repeated solutions. In one case, the search fails on repeated solutions and, in another case, the search does not fail on repeated solutions. We show the test results in Table A.2 in Appendix A. For most problems, there is little difference between using each of the two options. The prover with the option of failing on repeated solutions performs marginally better. However, for some problems (ls108, fex4t1, schubert, wos31), the improvements when failing on repeated solutions are significant. We note that all of the four problems are non-Horn problems. This indicates that the repeated search effort from doing case analysis is rather significant.

2.3. An Example

We give an example to illustrate the process of finding proofs. Consider the example in Figure 1.2

FALSE :– P, Q.
Q :– P.
P :– Q.
P :– ¬Q.

We give the complete trace for this problem. *Attempt(D, id(I), (L :– Γ))* means subgoal L :– Γ is attempted with effort bound D and identifier I. *Done(D, id(I), (L :– Γ))* is the done clause for subgoal L :– Γ with cost D. *Newly_solved(D, id(I), (L :– Γ))* means a solution L :– Γ has been generated within effort bound D and the solution is saved. *Already_seen(D, id(I), (L :– Γ))* means that the subgoal L :– Γ or a more general subgoal has been attempted with effort bound D, or greater, and we need not to work on L :– Γ any more. Given this explanation, it is not difficult to follow the trace and understand how caching is performed. For example, the subgoal q :– [ ] with id(11) will not be attempted with effort bound 4 since the same subgoal with id(7) has been worked on with effort bound 4.

```
(false:-p,q)
(p:-q)
(q:-p)
(p:- not q)

solution_size_mult(0.1) is asserted
proof_size_mult(0.3) is asserted
```

```
clause_count_mult(0.3) is asserted

search at size 5

attempt(5,id(1),(false:-[]))
attempt(3,id(2),(p:-[]))
attempt(2,id(3),(q:-[]))
done(2,id(3),(q:-[]))
attempt(2,id(4),(not q:-[]))
done(2,id(4),(not q:-[]))
done(3,id(2),(p:-[]))
done(5,id(1),(false:-[]))

search at size 7

attempt(7,id(5),(false:-[]))
attempt(5,id(6),(p:-[]))
attempt(4,id(7),(q:-[]))
done(4,id(7),(q:-[]))
attempt(4,id(8),(not q:-[]))
assumption(id(8),(not q:-[ not q]))
attempt_splits(5,id(6),(p:-[ not q]))
attempt(4,id(9),(p:-[q]))
attempt(3,id(10),(q:-[q]))
already_solved(id(10),(q:-[q]))

newly_solved(1,id(9),(p:-[q]))

newly_solved(1,id(6),(p:-[]))

attempt(4,id(11),(q:-[]))
already_seen(4,id(11),(q:-[]))
fail((not q:-[q]))
done(4,id(9),(p:-[q]))
repeated_solution(1,id(6),(p:-[ not q]))
attempt(4,id(12),(q:-[]))
already_seen(4,id(12),(q:-[]))
done(4,id(8),(not q:-[]))
```

```
done(5,id(6),(p:-[]))
done(7,id(5),(false:-[]))
retracting(solution(1,(p:-[q])))

search at size 9

attempt(9,id(13),(false:-[]))
attempt(7,id(14),(p:-[]))
already_solved(id(14),(p:-[]))
attempt(7,id(15),(q:-[]))
attempt(6,id(16),(p:-[]))
already_solved(id(16),(p:-[]))

newly_solved(1,id(15),(q:-[]))

newly_solved(3,id(13),(false:-[]))

proof found
```

3. Goal Generalization

In this chapter, we present Goal Generalization, a special case of Explanation-Based Generalization (EBG) which is a recent development in the field of machine learning. Goal generalization, which tries to find a proof for the most general solvable version of a specific goal while solving the specific goal, is an application of EBG methods in automatic theorem proving and is applicable to many goal-oriented theorem proving systems. We will describe Goal Generalization as an augmentation to the modified problem reduction format. Some experimental results are also given.

3.1. Motivation

A situation may arise that, during backward chaining, a goal to be solved is very specific and a proof for a more general version of the goal exists and has the same structure as the proof for the specific goal. Consider the example in Figure 3.1. The top-level goal is r(a). Obviously a proof exists for a more general goal r(X). It is beneficial to find the proof for the most general solvable version of a goal while the goal is being solved, especially when caching is performed. For the example in Figure 3.1, a goal r(b) will be declared as solved if r(X) instead of r(a) is solved and cached. Repeated work is avoided.

To accomplish the task of finding proofs for the most general solvable version of a goal while solving the goal, we need some generalization capability in the theorem prover. In this chapter, we will discuss our research to add one such generalization capability to the theorem prover. We will call our approach *goal generalization*. The basic idea is to augment the theorem proving system so that, when a goal G is attempted, two versions of G will be maintained. One of them is G itself, called the *specific goal*, and the other GG, called *general goal*, is the most general solvable version of the specific goal G

```
false :– r(a).
r(X) :– not s(X).
r(X) :– p(X), q(X), s(X).
p(X).
q(X).
```

Figure 3.1: An Example for Goal Generalization

(see Theorem 3.1). The general goal is at least as general as the specific goal. The general goal is constructed (along with its proof) as the specific goal is solved. When the specific goal is solved, the general goal will have been constructed and it can be instantiated to the specific goal.

3.2. Goal Generalization

This section shows how goal generalization is implemented. We will give a simplified representation of the inference rules to simplify our discussion. The inference rules will be represented as shown in Figure 3.2. A careful reader will notice that the representation in Figure 3.2 is indeed a simplified version of the representation in Figure 2.1. The L_0 in the representation of the clause rules in Figure 3.2 is a logical variable in Prolog. The procedure *unify* performs the unification operation. The procedure *member* is defined in Figure 3.3. We specify the unification operations explicitly so that we can explain an important optimization later.

If a call achieve((L :– B_0), (L :– B_1)) succeeds, the goal L :– B_1 has a proof. It is possible that a more general goal LG :– BG_1 has a proof with the same structure as that for L :– B_1. The more general goal LG :– BG_1 is a generalization in the sense that all goals that can be obtained from LG :– BG_1 by a substitution have proofs of the same structure. Goal generalization tries to find the most general solvable version LG :– BG of a goal L :– B while solving L :– B. We achieve this by augmenting the Prolog representation for the inference rules with extra arguments. Those extra arguments represent the more general versions of their counterparts. To be specific, the procedure

achieve((L :– B_0), (L :– B_1))

```
% Representation for input clause L :– L1, L2 , . . . , Ln:
   achieve((L0 :– B0), (L0 :– Bn)) :–
        unify(L0, L),
        achieve((L1 :– B0), (L1 :– B1)),
        . . .
        achieve((Li :– Bi–1), (Li :– Bi)),
        . . .
        achieve((Ln :– Bn–1), (Ln :– Bn)),

% Representation for a unit clause L:
   achieve((L0 :– B), (L0 :– B)) :– unify(L0, L).

% Representation for the assumption axioms:
   achieve((L :– B), (L :– B)) :– member(L, B).
   achieve((not(L) :– B), (not(L) :– [not(L)|B])).

% Representation for the case analysis rule:
   achieve((L :– B0), (L :– B1)) :–
        achieve((L :– B0), (L :– [not(M)|B1])),
        achieve((L :– [M|B1]), (L :– [M|B1])),
        length(B0) ≤ length(B1).
```

Figure 3.2: Simplified Representation of Inference Rules

```
member(L, [X|Y]) :– unify(L, X).
member(L, [X|Y]) :– member(L, Y).
```

Figure 3.3: Member Predicate Definition

will be replaced by

$$\text{achieve\_GG}((L :- B_0), (L :- B_1), (LG :- BG_0), (LG :- BG_1)),$$

where $L :- B_0$ and $L :- B_1$ are the goal to be solved and the goal solved, respectively, as the two arguments in the procedure achieve in Figure 3.1 are, and $LG :- BG_0$ and $LG :- BG_1$ are the more general versions of $L :- B_0$ and $L :- B_1$ respectively. The result is that a proof for $LG :- BG_1$ will be constructed which can be instantiated to be a proof for $L :- B_1$. The resulting representation for the clause rules will be as in Figure 3.4. The resulting representation for the assumption axioms and case analysis rule are in Figure 3.5.

In Figure 3.4, L_0 and LG_0 are logical variables and make_var(L_i, V_i) ($i = 1, 2, \ldots, n$) is such that V_i will be a distinct logical variable if L_i is a positive literal, a term

---

```
% Representation for input clause L :– L1, L2 , . . . , Ln:
   achieve_GG((L0 :– B0), (L0 :– Bn),(LG0 :– G0), (LG0 :– Gn)) :–
       unify(L0, L),
       variable_list(G0, VL0), make_var(LG1, V1),
       achieve_GG((L1 :– B0), (L1 :– Bn),(V1 :– VL0), (V1 :– G1)),
       unify(V1, LG1), unify(VL0, G0),
       . . .
       variable_list(Gi–1, VLi–1), make_var(LGi, Vi),
       achieve_GG((Li :– Bi–1), (Li :– Bi),(Vi :– VLi–1), (Vi :– Gi)),
       unify(Vi, LGi), unify(VLi–1, Gi–1),
       . . .
       variable_list(Gn–1, VLn–1), make_var(LGn, Vn),
       achieve_GG((Ln :– Bn–1), (Ln :– Bn),(Vn :– VLn–1), (Vn :– Gn)),
       unify(Vn, LGn), unify(VLn–1, Gn–1),
       unify(LG0, LG).

% Representation for unit clause L:
   achieve_GG((L0 :– B), (L0 :– B), (LG0 :– BG), (LG0 :– BG)) :–
       unify(L0, L), unify(LG0, LG).
```

Figure 3.4: Augmented Clause Rule

not(W_i) with W_i being a variable if L_i is a negative literal. LG : – LG_1, LG_2 , . . . , LG_n (or LG) is a copy of L : – L_1, L_2 , . . . , L_n (or L) (with new variables) made during the preprocessing, that is, when the Prolog clause is generated. The procedure *variable_list* assembles a list of distinct variables from a list of literals. For example, a list [X_1, X_2, X_3] will be returned by *variable_list*, given a list of three literals [L_1, L_2, L_3].

Similarly, Figure 3.5 shows the corresponding Prolog clause representations for the assumption axioms and the case analysis rule. The new, augmented procedure *member* is defined in Figure 3.6.

We will use the example in Figure 3.1 to illustrate how goal generalization works. Two proof trees will be constructed. We show the proof tree for the top-level goal false : – [] in Figure 3.7. In Figure 3.8, we show the proof tree for the generalized goal.

```
% Representation for assumption axioms:
   achieve_GG((L :- B), (L :- B), (LG :- BG), (LG :- BG)) :- member(L, B, LG, BG).
   achieve_GG((not(L) :- B), (not(L) :- [not(L)|B]), (not(LG) :- BG), (not(LG) :- [not(LG)|BG])).

% Representation for the case analysis rule:
   achieve_GG((L :- B0), (L :- B1),(LG :- BG0), (LG :- BG1)) :-
        achieve_GG((L :- B0), (L :- [not(M)|B1]), (LG :- BG0), (LG :- [not(MG)|BG1])),
        variable_list([MG|BG1], VL), make_var(L, V),
        achieve_GG((L :- [M|B1]), (L :- [M|B1]), (V :- VL), (V :- VL)),
        unify(VL, [MG|BG1]), unify(V, LG), length(B0) ≤ length(B1).
```

Figure 3.5: Augmented Assumption Axioms and Case Analysis Rule

```
member(L, [X|Y], LG, [XG|YG]) :- unify(L, X), unify(LG, XG).
member(L, [X|Y], LG, [XG|YG]) :- member(L, Y), member(LG, YG).
```

Figure 3.6: Augmented Member Predicate Definition

The goals in the proof tree of Figure 3.8 are the most general goals when their counterparts in Figure 3.7 are solved. We see that the proof tree for the more general goals has the exact structure as that of the specific tree. As a matter of fact, we update the more general proof tree as we construct the specific tree. However, we keep the goals in the more general proof tree as general as possible. This makes it possible to keep the input clauses as general as possible when they are used. Theorem 3.1 formalizes what goal generalization accomplishes.

Theorem 3.1: Given a set of input clauses S, if the call

$$\text{achieve\_GG}((L :- B_0), (L :- B_1), (V :- VL), (V :- BG_1))$$

succeeds, where V is a variable or a term not(W) with W being a variable depending on whether L is a positive or negative literal and VL is a list with each literal in B_0 replaced by a distinct variable, then the following are true:

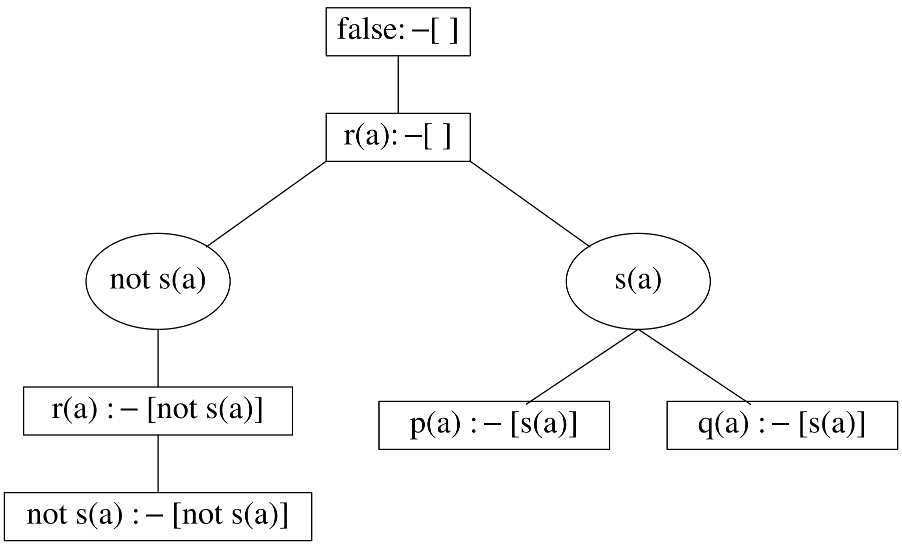

Figure 3.7: A Specific Proof Tree

(1) There exists a substitution θ such that $(V :- BG_1)\theta = (L :- B_1)$;

(2) $BG_1 \supset V$ is a logical consequence of S, where BG_1 is interpreted as a conjunction of the literals in it; and

(3) $V :- BG_1$ is the most general goal with the same proof; in another word, if there is a substitution π, a goal $(G :- M)$ which has the same proof as $(V :- BG_1)$ does, and $(G :- M)\pi = (V :- BG_1)$, then π only renames variables (Two goals $L_1 :- B_1$ and $L_2 :- B_2$ have the same proof if they use the same inference rules in the same order).

The proof is by induction on the size of the proof for $V :- BG_1$, where the size of a proof is the number of inference rules used to obtain the proof. We note that whenever mprf_GG is called, the third argument is always in the most general form (all variables).

3.3. Explanation-Based Generalization

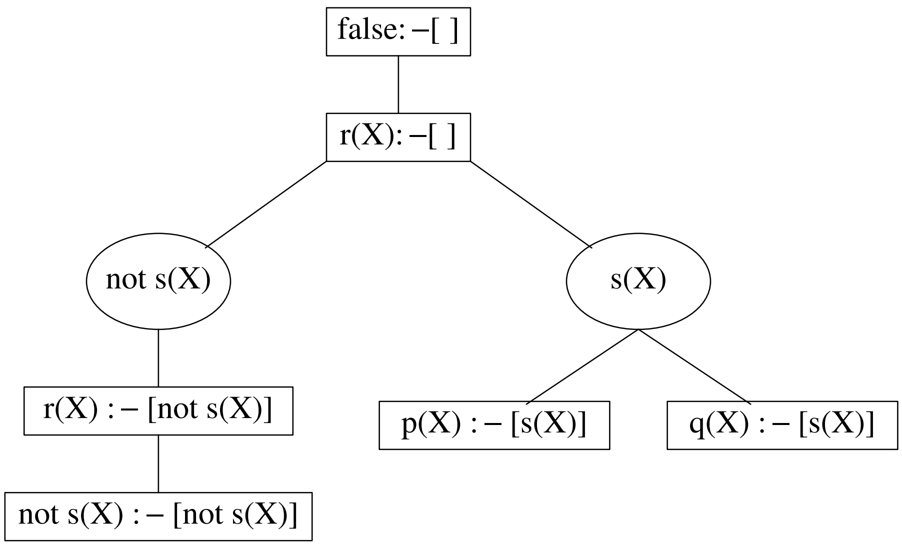

Figure 3.8: A More General Proof Tree

Explanation-Based Generalization (EBG) is a technique recently developed in the field of machine learning [Mitchell&al 86]. This technique deals with the problem of formulating general concepts on the basis of specific training examples. This technique has been shown to be the same as the technique in functional programming known as *partial evaluation* [Harmelen&Bundy 88]. In an Explanation-Based Generalization problem, we are given

- Goal concept — describing the concept to be learned;
- Training example — an example of the goal concept;
- Domain theory — a set of rules and facts about the domain;
- Operationality criterion — criterion to be met by the form of the learned concept definition;

And we are to determine a generalization of the training example that is a sufficient concept definition for the goal concept and that satisfies the operationality criterion.

Goal generalization can be regarded as a special case of Explanation-Based Generalization. We can reformulate goal generalization in terms of Explanation-Based Generalization:

- Goal concept — the general goal to be constructed;
- Training example — the specific goal;
- Domain theory — all the inference rules;
- Operationality Criterion — the learned concept (a goal) must satisfy the requirements (1)-(3) in Theorem 3.1.

Generally speaking, our work is one example of the research issues raised in [Mitchell&al 86]:

> ···how such methods for generalization will be used as subcomponents of larger systems that improve their performance at some given task. ··· One key issue to consider in this regard is how generalization tasks are initially formulated. In other words, where do the inputs to the EBG method (the goal concept, the domain theory, the operationality criterion) come from?···

In our approach, the attempt to solve each goal is formulated as a generalization task and this formulation bears a task-subtask structure similar to the goal-subgoal structure of the theorem proving system; the input clause set serves as the domain theory; and the concept of logical consequence serves as the operationality criterion.

EBG is presented as an augmentation of the SLD-resolution theorem proving system for Horn Clause Logic ([Kedar-Cabelli&McCarty 87]), where two proofs are constructed in parallel. One of the proofs is the specific proof that the training example is a member of the goal concept using the domain theory; the other proof, which is not unified with the facts of the training example, is the generalized proof from which an operational definition of the goal concept can be extracted. Goal generalization extends the idea to full first-order logic by augmenting a theorem proving system for full first-order logic.

3.4. Augmentation Related Refinements

We have modified the prover to incorporate the augmentation discussed above. There are two refinements in the implementation that deserve more elaboration. The first refinement concerns using the Prolog built-in unification in place of some calls to the procedure *unify*. The second refinement concerns how to eliminate unnecessary use of the case analysis rule.

Using Prolog built-in unification. We can replace the calls to *unify(R, L)* that involve the more general versions of the goals by $R = L$, which invokes Prolog built-in unification. It is well known that Prolog omits the occur-check in its unification for efficiency, and unification without occur-check is unsound ([Plaisted 84]). In our case, however, we use true unification for the specific goals and the unification operations involving the more general versions of the goals are always performed after the unification operations on the specific goals succeed. Thus unification operations on the general goals are guaranteed to succeed and use of Prolog built-in unification is sound. This refinement improves the efficiency of the augmented prover.

Unnecessary case analysis. The augmentation makes it possible to detect when some splitting literals are not used during the proof and thus redundant. In the more general version of a goal LG :− BG, the assumption list BG starts as a list of logical variables. The only place where these variables can be bound to a literal is in the assumption axiom where the procedure *member* is called. If a call

$$\text{achieve\_GG}((\mathrm{L} :- \mathrm{B}_0), (\mathrm{L} :- \mathrm{B}_1), (\mathrm{LG} :- \mathrm{BG}_0), (\mathrm{LG} :- \mathrm{BG}_1))$$

succeeds and there are still unbound variables in BG_1, we know that there are redundant literals in the assumption list. In this case, we simply fail on this call to mprf_GG. This is a potentially powerful deletion strategy and is made possible by the augmentation. This strategy seems to be similar to the requirement in Near-Horn Prolog that there be cancellation within each restart block in a legal deduction ([Loveland 88]).

3.5. Experimental Results

We have tested the augmented prover, with the two refinements discussed above incorporated, on our problem set. Table A.3 in Appendix A shows our test results. To summarize the test results, we note that, in 72 out of the total 82 problems, the augmented prover generates fewer or equal number of solutions and, for the 35 problems on which the augmented prover generates fewer solutions, the number of solutions is reduced by 38.6 percent on the average. We also note that the inference rate of the augmented prover is much smaller than that of the original prover. The augmented prover performs 3.63 inferences per second while the original prover performs 5.00 inferences per second.

One benefit we have expected from the generalization capability is that the prover will generate and cache fewer solutions. For most problems, this is indeed the case. The reason for the fewer solutions is twofold. On one hand, the number of different solutions will be smaller in the augmented prover since the solutions generated are the most general possible. On the other hand, solutions will be more likely to be subsumed by other solutions in the augmented prover. The fact that the augmented prover generates and caches fewer solutions is probably the main reason why the augmented prover is so much faster on wos15. However, there are 10 of the 82 problems on which the augmented prover generates and caches more solutions, as the table shows (ls108, for example). This is probably because the augmentations alter the search behavior of the prover. The search behavior can be altered in several ways. First, the search behaviors of the two prover can differ because the size of the solutions is used to control the search, and the more general solutions usually have a smaller size. Secondly, the search behaviors of the two provers can be different when the problems contain non-Horn clauses, since the augmented prover fails if unnecessary splitting literals are detected while the original prover can not detect those literals. As a result, the two provers can perform case analysis (splitting) on possibly different literals in possibly different part of the search space.

The inference rate of the augmented prover is much smaller than that of the original prover. This is why the augmented prover is much slower on problems like wos4, ls65 and schubert, where the numbers of solutions generated and the numbers of inferences performed by the original prover and by the augmented prover differ very little. The slower inference rate is due to the need to maintain two extra arguments to implement goal generalization. Some structure sharing techniques may be used to solve this problem. But such techniques seem to be difficult to implement in Prolog. We want to point out that the augmented prover is even slower without using the Prolog built-in unification on the general goals.

3.6. Comments

We have presented goal generalization — a special case of Explanation-Based Generalization and showed how it can be incorporated in a goal-oriented theorem proving system. We think this work is interesting in several ways. First, goal generalization is a refinement to caching since it always derives the most general solutions. Consequently, the theorem prover using caching will be more efficient in term of space since fewer solutions are usually generated and cached. Second, goal generalization is an application of Explanation-Based Generalization. The most interesting point here may be the way in which the generalization tasks are formulated automatically during the proof process by introducing all-variables goals. Third, our work suggests that other techniques in machine learning, such as similarity-based methods, may also be useful in automatic theorem proving. Finally, we point out that our approach can be used for other goal-oriented systems to accomplish similar task.

4. Subgoal Reordering

In modified problem reduction format, goals are expressed as subgoals, which are similarly expressed. A goal is solved if all of its subgoals are solved; and a goal fails if any of its subgoals fails. Subgoal reordering is the problem of determining the order for solving the set of subgoals of a goal to improve the efficiency of DFID. In this chapter, we discuss our investigations into the effect of subgoal reordering based on simple syntactic heuristics on the performance of the theorem prover. We show that subgoal reordering using these simple heuristics has a considerable impact on the performance of the prover on a large set of test problems. Some heuristics provide equally good, and often better, performance than the hand ordering of the input clauses. The syntactic heuristics are simple in form, cheap in their evaluation and often provide good heuristics, as has been demonstrated by our results.

4.1. Introduction

In the modified problem reduction format, a clause rule corresponding to a Horn-like clause decomposes a goal into a set of subgoals. Furthermore, the subgoals can be attempted in any order. The work discussed in this chapter is based on this observation. We formally state this as follows:

Theorem 4.1. The modified problem reduction format is still sound and complete if, for a Horn-like clause $L :- L_1, L_2, \ldots, L_n$, the clause rule is

$$\text{permute}([L_1, L_2, \ldots, L_n], [M_1, M_2, \ldots, M_n]),$$

$$\frac{[\Gamma_0 \rightarrow M_1 => \Gamma_1 \rightarrow M_1], [\Gamma_1 \rightarrow M_2 => \Gamma_2 \rightarrow M_2], \ldots, [\Gamma_{n-1} \rightarrow M_n => \Gamma_n \rightarrow M_n]}{\Gamma_0 \rightarrow L => \Gamma_n \rightarrow L}$$

where permute($[L_1, \ldots, L_n]$, $[M_1, \ldots, M_n]$) produces an arbitrary permutation $M_1, \ldots, M_n$ of $L_1, \ldots, L_n$ each time the clause rule is used.

Proof. We note that the soundness and completeness proofs of the modified problem reduction format in [Plaisted 88] do not claim any order of the literals in the clause body each time a clause rule is used. We can conclude that the order produced by *permute* is immaterial. Q.E.D.

Theorem 4.1 enables us to order the subgoals in a clause rule during the proof process, when the rule is invoked. The order of attempting a set of subgoals can make great difference in the efficiency of search and it is important to consider carefully the problem of determining the order. In some cases such as logic programming, ordering subgoals automatically may not be as important since the user usually has a very good idea about what the order of the subgoals should be. But in automatic theorem proving, a user may not have the knowledge to specify a good order in the input. It then becomes relevant to order the subgoals automatically during the proof. We call the process of ordering the subgoals in a clause rule during the proof process *subgoal reordering*. There are a couple of issues involved and we will discuss each of them.

The first issue is how to measure the quality of an ordering. We will not give a precise quantitative answer in general; instead we will roughly say that an ordering is good if it makes the search more efficient. To be specific, we can order the subgoals so that the most important subgoal is attempted first or to reduce the branching factors of the search space. To this end, we have defined some *ordering functions* which measures the "quality" or "importance" of the subgoals and used the values of the ordering functions to select the ordering. This raises the question about what the orderings functions should measure. One requirement for the ordering functions is that the applications of them incur low overhead, since it is going to be a frequent activity to apply the ordering functions to the subgoals if the subgoals are ordered during the proof. We have considered several ordering functions([Nie&Plaisted 87a]), two of which provide good results.

F1 evaluates the *size* of the subgoals where the *size* of a subgoal is the number of occurrences of predicate symbols, function symbols and variables in the subgoal.

F2 To evaluate the *mass* of the subgoals. Given a set of clauses S, the *mass* of a symbol T (predicate or function symbol), denoted by mass(T) is defined to be

$$\text{mass(T)} = \frac{\text{Number of literals in S}}{\text{Number of occurrences of T in S}}$$

For a term t,

$$
\text{mass}(t) = \begin{cases} \text{mass}(C) & \text{if t is a predicate or function or constant symbol C} \\ 0 & \text{if t is variable} \\ \text{mass}(s) & \text{if } t = \neg s \\ \text{mass}(f) + \sum_{i=1}^{n} \text{mass}(t_i) & \text{if } t = f(t_1, t_2, \cdots t_n) \end{cases}
$$

The second issue concerns what algorithm to use to select the ordering for a set of subgoals. Given n subgoals, there are n! possible orderings. We will choose a specific ordering using a greedy algorithm. For a subgoal $\Gamma_0 \rightarrow L$, when a clause rule corresponding to the input clause $L :- L_1, L_2, \ldots, L_n$ is used, the algorithm will be called to determine an ordering among $L_1, L_2, \ldots, L_n$. The algorithm first applies an ordering function such as F1 or F2 to each of $L_1, L_2, \ldots, L_n$, then sorts them according to their ordering function values. The resulting order among $L_1, L_2, \ldots, L_n$ will be the order in which they will be attempted. We call this *static reordering* because it orders the subgoals prior to any attempt to solve any subgoal when a clause rule is invoked. A slight variation to the algorithm leads to the *dynamic reordering*. In dynamic reordering, no order will be determined prior to attempting any subgoal. Rather, each time a subgoal is to be attempted, a subgoal will be selected from the remaining subgoals based on an ordering function. To be specific, for any goal $\Gamma_0 \rightarrow L$, whenever a clause rule corresponding to the input clause $L :- L_1, L_2, \ldots, L_n$ is used, a subgoal L_1' among the n subgoals $L_1, L_2, \ldots, L_n$ will be selected and $\Gamma_0 \rightarrow L_1'$ attempted. After $\Gamma_0 \rightarrow L_1'$ returns with $\Gamma_1 \rightarrow L_1'$, another subgoal L_2' will be selected among the remaining n-1 subgoals and $\Gamma_1 \rightarrow L_2'$ will be attempted, etc.

Dynamic reordering can adjust the order based on the progress of the search, such as new variable bindings and newly derived solutions. A problem may arise from the overhead of repeatedly applying the ordering function. If there are n subgoals, the cost of performing static reordering would be O(nlog(n)) and the cost of performing dynamic reordering would be $O(n^2)$ for our algorithm. For short clauses, this would not make a big difference. This seems to be the case for most of our test problems.

We have studied many heuristics for performing subgoal reordering. We will explain two such heuristics.

H1 **Subgoal having largest size first**. This heuristic uses F1 as the ordering function. The heuristic is based on several considerations: (1) A larger subgoal usually has a smaller branching factor since the larger size imposes more constraints on unification. (2) A larger subgoal has a more complex structure. This can be regarded as containing more information, thus being more important. (3) As shown in Chapter 2, the solution size contributes to the cost of solving a subgoal in the prover. Attempting the larger subgoal first can make the potentially unsuccessful search path stop earlier since larger subgoals will use larger solutions, thus contributing more to the cost.

H2 **Subgoal having the biggest mass first**. This heuristic uses F2 as the ordering function. The heuristic is used in [Wang&Bledsoe 87] for the level-subgoal reordering in SHD-prover based on hierarchical deduction. The subgoal with largest mass is likely to contain non-variable symbols which occur less frequently or to contain more non-variable symbols. Non-variable symbols occurring less are more likely to be the symbols in the theorem or the skolem function symbol. Thus the subgoal with largest mass can be regarded as being the most important. Also a subgoal with large mass is likely to have a small branching factor.

4.2. Related Work

Similar problems are considered in some other goal-oriented theorem proving systems [Wang&Bledsoe 87, Kowalski&Kuehner 71]. In [Wang&Bledsoe 87], *level goal reordering* is performed during the proof process where the search process is controlled by suitable selection of the first literal to resolve upon in a goal clause[1]. The heuristic is to select the literal with the biggest *mass*, which is taken to be the one with the most complex structure. In SLR-based proof procedures, the choice of the literal can be made dynamically for the application of the extension operation [Kowalski&Kuehner 71]. One heuristic suggested is to select the literal which can be resolved upon with the least number of input chains.

The problem we consider here is similar in nature to the conjunctive problem in [Smith&Genesereth 85]. [Smith&Genesereth 85] discusses the problem of ordering a conjunction — a set of propositions which share variables and must be satisfied simultaneously — in order to reduce the size of the search. They use the size of the database to estimate the cost of solving a conjunct and determine an ordering of conjuncts which has the least cost by possibly searching through n! possible orderings for n conjuncts. An *adjacency theorem* is proven to cut down the size of the search and some heuristics are also suggested to avoid the search completely. While the basic problem is the same, some assumptions in [Smith&Genesereth 85] are not valid in our case. For example, the

[1]Here the term *goal clause* does not refer to an all-negative clause. See [Wang&Bledsoe 87].

assumption that all solutions to a conjunct are directly available in the database is not valid. This assumption makes it possible to estimate the cost of solving a conjunct rather easily. In our case, however, the solutions to a subgoal are rarely directly available and may require many inferences. And we do not know how many inferences would be required. Also, the cost of solving the same subgoal may vary if caching is performed. All these make good estimate of the cost of solving a subgoal very difficult. It is also pointed out in [Smith&Genesereth 85] that an optimal ordering of the conjuncts can not always be achieved by only considering the subgoals of a goal if inferences are required to obtain the solutions. This implies that a global data structure is needed to store all the unsolved subgoals and the optimal ordering is selected from all the possible orderings of those unsolved subgoals. A general best-first search will be required.

In our work, instead of estimating the cost of solving a subgoal, we quantify certain syntactic characteristics of the subgoals and use a cheap greedy algorithm to determine the ordering. We only deal with the subgoals belonging to one goal to make the subgoal reordering process compatible with the depth-first iterative deepening search used in the prover. If the best-first search strategy were used, subgoal reordering would not be necessary, but such a strategy requires a lot of memory,

4.3. Heuristics and Implementation

A convenient Prolog interface in the prover provides an easy vehicle to carry out subgoal reordering. In the input to the prover, a subgoal of the form prolog(L) represents a call to the Prolog procedure L. We write a Prolog subroutine, called *best_subgoal*, to order a list of subgoals according to the ordering function. Another Prolog subroutine is written to translate the standard input format into the format which includes the calls to the Prolog subroutine *best_subgoal*. For example, the input clause $L :- L_1, L_2, L_3$ is translated into the clause (X_1, X_2, X_3 and Y are logical variables)

$$\begin{aligned} L :- &\ \text{prolog(best\_subgoal}([L_1, L_2, L_3], [X_1|Y])), X_1, \\ &\ \text{prolog(best\_subgoal}(Y, [X_2, X_3])), X_2, X_3. \end{aligned}$$

to perform dynamic reordering; it is translated into the clause

$$L :- \text{prolog(best\_subgoal}([L_1, L_2, L_3], [X_1, X_2, X_3])),\ X_1,\ X_2,\ X_3.$$

to perform static reordering. The resulting clause will be the input to the prover.

We have performed tests on our test problem set, using the two heuristics. We test both static reordering and dynamic reordering using each heuristic on 82 problems. We show part of our experimental results in Table A.4.1 and Table A.4.2 in Appendix A. We summarize the data in the three tables Table 4.1, Table 4.2 and Table 4.3. As we have expected, no single heuristic, when used for subgoal reordering, performs better on all the test problems. Nevertheless, there are some interesting things revealed by the data.

Before we make any comment on the data, we first explain how to interpret Table 4.1, Table 4.2 and Table 4.3. Table 4.1 and Table 4.2 are self-explanatory. We explain

Table 4.1: Average Data for Subgoal Reordering

	Average Time Per Theorem	Average Inference Per Theorem	Average Inference Per Second
no reordering	232.05	1335.76	5.76
dynamic-H1	315.04	1648.72	5.23
static-H1	318.68	1649.38	5.18
dynamic-H2	154.44	1175.12	7.61
static-H2	168.82	1171.12	6.93

Table 4.2: Running Time Distribution for Subgoal Reordering

	(0, 10]	(10, 60]	(60, 300]	(300, 600]	(600, +∞)	Total
no reordering	52	16	8	1	5	82
dynamic-H1	51	21	7	0	3	82
static-H1	52	20	7	0	3	82
dynamic-H2	50	21	7	1	3	82
static-H2	50	20	6	1	5	82

Table 4.3: Comparing with no Reordering

functions	improvements		degeneration		Even
	number	average(%)	number	average(%)	number
dynamic-H1	33	15.73	20	53.4	29
static-H1	24	19.8	17	31.9	41
dynamic-H2	45	24.9	20	84.0	17
static-H2	38	24.6	24	108.0	20

Table 4.3 by an example. The two numbers 33 and 15.73 under *improvements* for *dynamic-H1* indicate that the prover with dynamic subgoal reordering using heuristic H1 does better on 33 of the 82 problems (takes fewer inferences) and the average speedup with respect to the performance of the prover without subgoal reordering is 15.73%. The two numbers 20 and 53.4 under *degenerations* for *dynamic-H1* indicate that the prover with dynamic subgoal reordering using heuristic H1 does worse on 20 of the 82 problems (takes more inferences) and the average slowdown with respect to the performance of the prover without subgoal reordering is 53.4%. The number 29 under *even* for *dynamic-H1* indicates that the prover with dynamic subgoal reordering using H1 performs equally well (takes equal number of inferences) as the prover without subgoal reordering on 29 of the 82 problems.

We first note that subgoal reordering incurs little overhead. This is because the ordering functions are easy to evaluate, the algorithm for selecting the ordering is simple and the input clauses in the problems are generally short (7 literals maximal). For the same reasons, dynamic reordering is not noticeably more expensive than static reordering. All these can be seen from the data in Table 4.1 and Table 4.2. The data in Table 4.3 suggest that, at least for our heuristics, dynamic reordering should be preferred if subgoal reordering is to be performed at all since dynamic reordering does better on more problems than static reordering using the same heuristics.

The data in Table 4.2 suggest that subgoal reordering does not affect the performance of the prover very much. But the data in Table 4.1 seem to suggest otherwise. This discrepancy results from the dramatic improvements or degeneration of the performance of the prover when performing subgoal reordering on several problems (ls108, wos15 and wos31). These problems are difficult for the prover without subgoal reordering. This suggests that subgoal reordering might be a valuable addition to the prover for solving hard problems if we can devise specific heuristics for them.

One general heuristic does suggest itself. It seems that subgoals with complex structures should be favored. The reasons are exactly those behind H1 and H2. Subgoals with complex structures tend to have small branching factors and can be seen as more important. Special attention should be paid to function symbols since they represent objects in the problem domain. The good performance of the prover when performing subgoal reordering using H2 enforces this rather strongly.

4.4. Comments

It requires domain dependent knowledge to find the optimal ordering for a set of subgoals. In case such knowledge is not available, we have to resort to general heuristics. We have tested several such heuristics and shown that they can have great impact, sometimes adverse, on the performance of the prover. But some heuristics seem to work better or equally well most of the time. Such heuristics are useful since they can make the theorem prover more automatic. We also point out that our heuristics are almost purely syntactic in nature. Heuristics of this sort are simple in form and impose low overhead in their evaluations; and they often provide performance improvements. In general, we think that the importance of the syntactic aspect of mechanical theorem proving is not to be ignored, although it may not play a decisive role in the success of theorem proving in the future.

5. Priority Systems

In Chapter 4, we discussed subgoal reordering, which considers ordering a set of subgoals, which must be resolved either by a failure on a single subgoal or by success on all of them. The selection of a subgoal in subgoal reordering is made locally within the set of immediate subgoals of a goal. Subgoal reordering considers subgoals that belong to one proof path.

In this chapter, we address the question of how to select among subgoals which are on different proof paths. The task is to find one successful proof path for one of these subgoals. The best-first search strategy can estimate the importance of ALL the available goals and always work on the best one based on the estimation. This feature is essential for the search strategy of a theorem prover. We will discuss a refinement to the depth-first iterative deepening search strategy. The refinement approximates best-first search using depth-first iterative deepening search, using the information provided by a priority function to control the search. We will introduce a new data structure, the *priority list*, into depth-first iterative deepening search for this purpose.

5.1. Motivation and the Basic Idea

One of the essential features in any automatic theorem proving system is for its search strategy to estimate the importance or relevance of its goals and always to work on the best available goal based on the estimation [Plaisted&Greenbaum 84]. The best-first search strategy is the most commonly used search strategy to achieve this. In best-first search, a *priority function* is defined which assigns priority values to the goals. A goal queue, sometimes called the *open list*, is also maintained in best-first search, which contains all the unfinished goals together with their priority values. The search always selects the goal from the goal queue with the "best" priority value to attempt next. It has been demonstrated that the best-first search strategy adds substantial power to a theorem prover [Wang&Bledsoe 87, Greenbaum 86, Overbeek&al 76]. The brute-force depth-first iterative deepening search we implemented in Chapter 2 does not have a good way of using the heuristic information provided by the priority function to control the search. We think this is a big disadvantage, and we tried to eliminate it by incorporating priorities into depth-first iterative deepening search. In the following discussion, we assume that a smaller priority value indicates a more important goal; we also use the more general term *operation* to refer an unit of work performed by the search process. In particular, an operation can refer to a goal.

Suppose we have a priority function and we would like to find a solution by solving only the goals with the smallest priority values possible. If we know that a solution exists and know the smallest bound B which allows the solution to be found, we can just set the priority bound to B and perform the depth-first iterative search until the solution is found, deleting the goals whose priority values exceed B. The problem is that we usually do not know the value of B unless we have found a solution. What we would like to do is to find the solution using some bound not much larger than the smallest bound B. We call the resulting strategy a *priority system*.

Assume that a solution exists. Then there will exist two numbers B and L_0 such that the solution can be found using L_0 operations with priority values smaller than B. These L_0 operations are the most important operations. But we know neither B nor L_0. However, we can start by assuming L_0 to be some fixed number K. That is, we will be interested in the K smallest operations performed in the search. At any moment of the search, we know the L smallest operations performed so far. Assume $C_1, C_2, \cdots C_K$ are the priority values of the K smallest operations performed so far. If an operation has a priority value P larger than or equal to the largest of $C_1, C_2, \cdots C_K$, we know that this operation will not be one of the K smallest operations we are interested in. If an operation has a priority value P smaller than one of $C_1, C_2, \cdots C_K$, this operation may be among the K smallest operations and we should replace the largest value in $C_1, C_2, \cdots C_K$ with C. This is the basic idea behind the priority system. In general, the priority system will repeatedly perform the search with increasing values K. The bound B will not be considered explicitly, but will be implicitly increased until it is larger enough to permit a solution to be found.

5.2. Formal Development

We propose a priority system. The priority system will operate as the depth-first iterative deepening search does. We call DFID the *underlying strategy*. The priority system will maintain a *priority list* of some fixed length K, which stores a sequence of priority values $C_1, C_2, \ldots, C_K$ in non-increasing order. The priority system records in the priority list the K smallest priority values of the operations performed so far. The priority list is initialized by performing the first K operations that are allowed by the underlying strategy and putting their priority values on the list. For each successive operation, we compute its priority value P. If P is less than C_1, the operation is performed and the priority list is updated by deleting C_1 and inserting P at the appropriate place. If P is greater or equal to C_1, the operation is rejected. This will continue until no operation is possible or a solution is found. If no solution is found before all the operations are exhausted, the length of the priority list will be increased and the process will be repeated. At any moment, the priority values C_1, C_2, $\cdots$ C_K in the priority list are the K smallest operations performed so far. But this fact is not important. What is important is that the search will favor small operations more and more as the search proceeds. We would also like to point out that, if a solution is found by the priority system with the largest value in the priority list being B, B may not be the smallest possible bound. This fact is not important either.

Let's consider an example of how the priority list is used and updated. Suppose that a priority list of length 5 is [10, 7, 6, 6, 3]. An operation with priority value 10 or greater will be rejected. An operation with priority value 5 will be accepted and the priority list will updated to [7, 6, 6, 5, 3]. Note that each operation is either rejected or updates the priority list to a strictly (lexicographically) smaller list.

The following theorem formalizes the basic idea behind the priority system.

Theorem 5.1: Given a problem P, suppose N operations are performed by the underlying strategy to find a solution S for P. Let B_0 be the smallest bound for the priority values which permits the solution S to be found. Let L_0 be the number of operations among the N operations whose priority values are less than B_0. The priority system will find the solution S when the priority list is of length greater than or equal to L_0, if the underlying strategy satisfies the *monotonicity condition*, which states

> for any bound B, if some or all of the operations with priority values $\geq$ B are deleted, the number of possible operations with priority value less than B will not increase.

Proof: At any given time after L_0 operations have been performed and before the solution is found, the largest element in the priority list will be greater than or equal to B_0. Since the priority list is of length greater than or equal to L_0 and the number of remaining operations with priority values smaller than B_0 will not increase by the monotonicity condition, all the remaining operations with priority values smaller than B_0 will be performed and the proof will be found. Q.E.D.

The monotonicity condition ensures that the region of the search space explored by the priority system is within the region of the same search space explored by the underlying strategy. This condition could be violated, for example, if the cut-off bound of the priority system is bigger than that of the underlying strategy when both strategies perform depth-first search.

We now analyze the priority system. We assume that a priority value is a positive integer. The question we ask is: Given that the length of the priority list is K, what is the number of operations that could be performed. Suppose the first successive K operations allowed by the underlying strategy have priority values $C_1, C_2, \ldots, C_K$ in non-increasing order and let $M_L = C_1$. Note that each operation is either rejected or changes the sequence to a strictly smaller list. With a priority list of length K, we can have a minimum of K operations (the first K operations allowed by the underlying strategy to initialize the list) and a maximum of $(M_K - 1) \times K$ operations afterwards, since a priority value is a natural number. We invoke successive trials of the procedure with the priority list being of length $L \times C^i$ ($i = 0, 1, \ldots,$ L is the initial length of the priority list and $C > 1$) respectively, until a solution is found. We show that the amount of work in each trial increases by a nearly constant factor in comparison with its preceding trial. We also show that the total amount of work is dominated by the last trial.

Letting w(i) denote the amount of work when the priority list is of length $L \times C^i$, we easily have

$$C^i \times L \le w(i) \le L \times C^i \times M_{C^iL}$$

where M_{C^iL} ($i = 0, 1, \ldots,$) is the largest priority of the first $C^i \times L$ operations. We can use this to bound the ratio of work of successive trials:

$$P_1(i) = \frac{w(i+1)}{w(i)} \le C \times M_{C^{i+1}L}.$$

This shows that the amount of work in each trial is increased by a nearly constant factor $(C \times M_{C^{i+1}})$ in comparison with its preceding trial.

Let I be the smallest i such that a solution is found by the priority system when the priority list is of length $L \times C^I$. We try to estimate the ratio

$$P_2(I) = \frac{\sum_{k=0}^{I} w(k)}{w(I)}$$

Note that $P_2(I)$ bounds the unnecessary work performed by the priority system.

$$\sum_{k=0}^{I} w(k) = \sum_{k=0}^{I} C^k \times L \times M_{C^k L} \leq M_{C^I L} \times L \sum_{k=0}^{I} C_k = M_{C^I L} \times L \times \frac{C^{I+1} - 1}{C - 1} \leq M_{C^I L} \times L \times \frac{C^{I+1}}{C - 1}$$

Let's simply take $\sum_{k=0}^{I} w(k) = M_{C^I L} \times L \times \frac{C^{I+1}}{C-1}$. Since we already have derived $C^i \times L \leq w(i) \leq L \times C^i \times M_{C^i L}$, we have

$$\frac{C}{C-1} \leq P_2(I) = \frac{\sum_{k=0}^{I} w(k)}{w(I)} \leq \frac{C}{C-1} \times M_{C^I L}$$

This shows that the amount of total work is dominated by the last trial.

The analysis above is a worst-case analysis and the result is admittedly weak. Note that the time complexity depends on the bound $M_{C^I L}$, and we do not know its expected value. Theoretically it can be arbitrarily large. The fact that the complexity depends on $M_{C^I L}$ also presents a practical problem: At the beginning of the search, we do not have any control over the complexity of the operations. The priority system will not have any effect until the priority list fills up. In practice, though, we believe that the values for $P_1(i)$ and $P_2(I)$ are almost constant with respect to C. Our experiments on our test problem set support this. Table 5.1 shows the average values for $P_1(i)$ and $P_2(I)$.

5.3. A Modification

We propose a modification of the priority system. In this modified priority system, we define the priority of an operation to consist of possibly multiple units of work. The number of units of an operation is called the *weight* of the operation. As before, we assume the priority list is of some length K and is represented by a sequence of priority values $C_1, C_2, \ldots, C_K$ in non-increasing order. An operation with priority value C and weight W will be rejected if one of the first W values (the W largest) in the priority list is less than or equal to C. If the operation is not rejected, the priority list will be updated by

Table 5.1. Average Values for $P_1(i)$ and $P_2(I)$

	C = 2	C = 3	C = 4
$P_1(i)$	2.10	3.13	4.24
C/(C-1)	2.00	1.50	1.33
$P_2(I)$	2.88	1.98	1.93

deleting the first W entries from the priority list, then inserting W copies of C into the priority list.

If we invoke successive trials of the procedure with the priority lists being of length $L \times C^i$ ($i = 0, 1, \ldots$, L is the initial length of the priority list and $C > 1$) respectively, until a solution is found, we can show that the amount of work in each trial increases by a constant factor in comparison with its preceding trial. We can show that the total amount of work is dominated by the last trial. Let's first determine how many operations can be performed when the priority list is of length K. This is, again, a worst-case analysis. For the sake of simplicity and without loss of generality, we assume that an operation with priority value C will have weight C as well. The analysis is straight forward if we realize that it takes at most K/N operations of priority N to fill an empty priority list of length K with weight N. To perform the maximal number of operations, we should fill the list with the largest priority value possible first. So the first candidate is the operation with weight K, the second candidate is the operation with weight $K-1$, the third with weight $K-2$, etc. The last operations are those with weight 1. Therefore, the maximal number of operations is

$$\sum_{i=1}^{K} \frac{K}{i} = K \times \sum_{i=1}^{K} \frac{1}{i} = O(K \times \ln(K))$$

We invoke successive trials of the procedure with the priority lists being of length $L \times C^i$ ($i = 0, 1, \ldots$, L is the initial length of the priority list and $C > 1$) respectively, until a solution is found. Letting $v(i)$ denote the amount of work when the priority list is of length $L \times C^i$, we can show

$$Q_1(i) = \frac{v(i+1)}{v(i)} = \frac{C^{i+1} \times \ln(C^{i+1} \times L)}{C^i \times \ln(C^i \times L)} = C \times \frac{i \times \ln(C) + \ln(C) + \ln(L)}{i \times \ln(C) + \ln(L)} =$$

$$C \times (1 + \frac{\ln(C)}{i \times \ln(C) + \ln(L)}) \le C \times (1 + \frac{1}{i}) \le 2C \quad (i > 0)$$

That is, the work for successive trials of the procedure increases only by a constant factor. And if a solution is found when the priority list is of length $C^I \times L$, we have

$$Q_2(I) = \frac{\sum_{k=0}^{I} C^k \times L \times \ln(C^k \times L)}{C^I \times L \times \ln(C^I \times L)} = \sum_{k=0}^{I} \frac{1}{C^{I-k}} \times \frac{\ln(C^k \times L)}{\ln(C^I \times L)} \le \sum_{k=0}^{I} \frac{1}{C^k} \le \frac{C}{C-1}$$

That is, the work is dominated by the last trial, if the last trial performs the maximal number of operations before the solution is found. Table 5.2 shows the average values of $Q_1(i)$ and $Q_2(I)$ obtained from our experiment on our test problem set.

Table 5.2. Average Values for $Q_1(i)$ and $Q_2(I)$

	C = 2	C = 3	C = 4
$Q_1(i)$	2.09	3.12	4.18
C/(C-1)	2.00	1.50	1.33
$Q_2(I)$	2.90	2.19	1.90

5.4. Implementations of the Priority System

As illustrated in Figure 5.1, the implementation of the priority system consists of *stages*, denoted by a pair of natural numbers$\{n_1, n_2\}$. Each stage consists of several rounds of depth-first search with increasing bounds. To be specific, the stage $\{n_1, n_2\}$ will consist of the consecutive rounds of depth-first search with cut-off bound being m_1, $m_2, \ldots, m_k$, which are determined by n_1 and n_2 and $n_1 = m_1 < m_2 < \cdots < m_k \le n_2$. A simple example for a stage $\{n_1, n_2\}$ would be $n_1 = m_1, m_1 + 1, m_1 + 2, \ldots, m_1 + k = n_2$. In Figure 5.1, we usually have $S_i = S_j$ and $E_i < E_j$ for all i and j ($i < j$) and the priority list for stage i is of length $C^{i-1}L$ where C (> 1) and L are constants. We note that the monotonicity condition in Theorem 5.1 may be invalid in our implementations. We also note that the bounds P_1 (Q_1) and P_2 (Q_2) may not work unless $E_i = \infty$ and $S_i = 1$ for all i. Nevertheless they seem to work well in practice.

The Prolog code in Figure 5.2 specifies the control structure of the priority system in greater detail. Each stage is defined by the procedure *search* which performs several

stage 1	stage 2	stage 3	stage 4	......	stage i	......
$\{S_1, E_1\}$	$\{S_2, E_2\}$	$\{S_3, E_3\}$	$\{S_4, E_4\}$	......	$\{S_i, E_i\}$	......

Figure 5.1: Control Structure of Priority System

rounds of depth-first search until its termination is called for by the procedure *stage_end* or the discovery of a proof by the procedure *achieve*, which is defined in Figure 2.1 in Chapter 2. The bounds for a stage (the pair of numbers S_i, E_i in Figure 5.1) are determined by the procedure *initialize_stage*. With each stage are also associated some priority lists which are initialized by the procedure *initialize_priority_list*. L is the initial length of the priority lists and C (>1) is the constant. For each stage except the first one, the priority lists will be of the length $C \times P$ where P is the length of the priority list of the previous stage.

The priority list is represented as a multi-set, that is, a list of the tuples (P, N) where P is the priority value and N is the number of entries being P in the priority list. For example, [7, 6, 6, 5, 5, 3] is represented as [(7,1),(6,2),(5,2),(3,1)].

```
search_stage(L, CI, C, B) :–
    terminate(L, CI, B), !, fail.
search_stage(L, CI, C, B) :–
    initialize_stage(Start, Stage_Bound),
    initialize_priority_list(L, CI),
    search(Start, Stage_Bound).
search_stage(L, CI, C, B) :–
    CI1 is CI × C,
    search_stage(L, CI1, C, B).

search(Start, Stage_Bound) :–
    stage_end(Start, Stage_Bound), !, fail.
search(Start, Stage_Bound) :–
    achieve((false :– [ ]), (false :– [ ]), 0, Cost, Start).
search(Start, Stage_Bound) :–
    increment(Start, S1),
    search(S1, Stage_Bound).
```

Figure 5.2: Control Structure of Priority System

The control structure of the priority system illustrated above raises several important practical issues concerning the implementation of the priority system. How to resolve these issues can make a dramatic difference in the efficiency of the priority system. We will discuss these issues. First we note that some design decisions for resolving these issues may violate the monotonicity condition.

What will be defined as an operation and what will defined as a unit of work? In general, an operation should represent some "unit expansion" of the search space. For this prover in particular, an operation can be either generating a new subgoal, or performing an inference, or deriving a new solution. As to the definition of a unit of work in the prover, we can define it to be some number of symbols. For example, a subgoal containing 12 symbols will have 3 units of work if 4 symbols is defined to be a unit of work. Other possibilities exist also.

How to identify the end of a stage? To stop a stage prematurely will not take full advantage of the priority system since some important operations may be left out. To stop a stage too late will result in useless work. It can also lead to a infinite loop in the search if this is not properly done. Ideally a stage should be terminated if no progress can be made toward the proof. We can say some progress is being made if some new solution is derived. One terminating condition for a stage suggests itself: A stage can be terminated if no new solution is generated in a round of depth-first search in the stage. This condition, however, can lead to an infinite loop in the search when used alone. We need some auxiliary conditions to avoid infinite loops. The basic idea of the auxiliary conditions is to guarantee that, for any integer n_0, there is an integer n ($\geq n_0$) such that a round of depth-first search with cut-off bound n will be performed eventually.

If caching is done, which is usually the case, what can we do with the information from the previous stages? Assume we start each stage at the same search bound, that is, all S_i's in Figure 5.1 are equal. This, by the way, seems to favor shorter proofs and works well. Upon the termination of each stage, many solutions and subgoals may be cached. The question is what we do with the information. We usually remove all the cached subgoals since each stage starts at the same search bound. The question remaining is what to do with the solutions. What we want to achieve is to avoid repeated work on one hand and to avoid increasing the branching factor too much on the other hand. There are several choices. (1) We can delete all the solutions between stages. This is simple and will not increase the branching factors of the search space. But this choice may result in too much repeated work, which is what we are trying to avoid by caching. (2) We can save all the solutions between stages. This tends to complicate the search since the solutions can increase the branching factors significantly, since solutions are used to solve subgoals. (3) We can save only the solutions satisfying certain conditions. The conditions are devised to avoid some repeated work while not substantially increasing the branching factors. The priority functions can come into play here. One condition is that the saved solutions have to have priority values within a specified bound and only contain ground literals. Another condition is to save solutions whose proof size is relatively bigger than their priority values. If solutions are kept between stages, the monotonicity condition may be violated.

How should a priority function be chosen? A good priority function should be (1) a good measurement of the relevance of the operations (2) sufficiently problem independent to be applicable to a large class of problems and (3) easy to evaluate. We intend the priority functions to reflect the syntactic characteristics of the operations. Operations with smaller priority values are to be favored. The most commonly used priority function in our experiments is the number of symbols in the subgoals or solutions involved in an operation. The search will favor smaller subgoals if such priority function is used. We can design some specialized priority functions for individual "hard" problems using the following ideas. (1) We can use the priority function to measure the similarity of a subgoal or a solution to the intended theorem, using ideas like not counting the skolem constants occurring in the theorem. (2) We can consider the number of occurrences of a symbol in the problem input. The symbols in the axioms are likely to occur more in the input. These symbols may be regarded less important than the symbols which occur less in the input. Thus different symbols may be weighted differently based on their occurrences in the input. This idea is similar to the idea of the symbols' *mass* in [Wang&Bledsoe 87]. An elaborate and more general scheme is described in [Overbeek&al 76] for calculating the complexity of the clauses in a resolution theorem prover. That scheme could also be used to define our priority function. We emphasize that priority functions basically use information which is syntactic in nature and local to the operations.

5.5. A Subgoal-Based Priority System

We describe a particular implementation of the priority system based on subgoal. In this implementation, we primarily use the priority values of the subgoals. Each subgoal contributes one entry in a priority list. This seems to work quite well in practice in spite of some theoretical complications. The priority function returns the size of the largest literal in the subgoal G plus the number of literals in the assumption list G. The effect is that small subgoals or subgoals with shorter assumption lists are favored. The stages are determined statically, as shown in Figure 5.3. Although there is no theoretical justification for this static setup, it does work well in practice. All solutions are kept between stages. This does not seem to present a problem in general since the priority lists for solutions function as "solution filters". Also, the condition for rejecting a subgoal is less restrictive than in the formal presentation of the strategy. This will invalidate the complexity analysis but, again, it works well in practice.

The theorem prover alternates between backward chaining and forward chaining. Backward chaining starts with a goal clause. There may be more than one goal clause in some problems (non-Horn problems, typically). We have used several priority lists in this implementation. For each goal clause false $:- L_1, \ldots, L_k$, we maintain two priority lists which are called *solution list* and *subgoal list* respectively. These two lists are used when the theorem prover performs backward chaining starting with this goal clause. We also maintain a solution list and a subgoal list for when the prover performs forward chaining. The decision to maintain multiple priority lists is based partially on a practical observation. In backward chaining phase, the search always starts with the first goal clause in the input. If only one priority list is maintained, the list tends to soon fill up with small priority value. As a consequence, little room is left for working on other goal clauses. Of course, the user can order the input clauses so that the most promising goal clause comes first, but this is not always easy to do. Another purpose of using different priority lists for different goal clauses and for forward chaining is to balance the search effort spent on backward chaining and forward chaining and the search effort spent on each goal clause during backward chaining. We increase the length of the priority lists for backward chaining more quickly than we do the priority lists for forward chaining so that backward chaining is encouraged while some forward chaining is still performed. We use separate priority lists for forward chaining and for each goal clause during backward chaining so that the search effort spent on one part does not affect the search effort on other parts, as

stage 1	stage 2	stage 3	stage 4	stage 5	......
{5, 7}	{5, 11}	{5, 15}	{5, 19}	{5, 23}	......

Figure 5.3: Control Structure of A Priority System

different goal clauses would affect each other in backward chaining if one priority list were used.

The subgoal-based priority system works as follows. When a new subgoal G is to be attempted, the procedure *check_subgoal* will be called. The procedure first calls the priority function to determine the priority value of G. It then checks the priority value against the current priority list (subgoal list). If the priority value is bigger than the largest in the priority list, the subgoal G will be rejected. If the priority value is equal to or smaller than the largest value in the subgoal list, the priority list will be updated in case the priority value is smaller than the largest value in the subgoal list and G will be attempted. The solution lists are maintained differently, however. Whenever a new solution is generated or an old solution will be used, the procedure *big_solution* will be called. This procedure checks whether the priority value of the new solution or the old solution is smaller than the largest value in the current solution list. Only when *big_solution* succeeds, is the new solution saved or the old solution used. Note this procedure does not alter the solution list. In case a new solution is generated and the call to *big_solution* succeeds, the procedure *check_solution* is called. This procedure first calls the priority function to get the priority value of the solution. It then checks whether the priority value of the solution is smaller than the largest value in the solution list. If it is, the priority value of the solution will replace the largest value in the solution list; otherwise, the solution list will not be altered. The solution will be always be saved, however. Since we have carefully selected solutions during the stage using the two procedures, all the cached solutions will be kept between stages.

Next we show some performance statistics of the implementation on our test problems and compare them with the underlying strategy. The test results are given in Table A.5 and Table 5.3. We note that the prover using the priority system obtains proofs for three more problems and it has a better average time among all the problems whose proofs are obtained. Note also that the inference rate is much larger when the priority system is used. This is because the inferences tend to be smaller in the priority system, thus consuming less time for each inference. There are three problems, wos15, ls108 and wos31, on which the priority system is much faster than the underlying strategy. The reason is that all the subgoals in the proofs are small subgoals. There are, however, some problems on which the priority performs less well. fex6t1 is one of them. The reason is that the proof for fex6t1 has many large subgoals, as shown in Figure 5.4. If we use the subgoal size as the priority values of the subgoals, the priority lists have to be rather long before the proof is found. The priority system can prove fex6t1 much faster if it uses a new priority function which counts less for terms of the form $f(t_1, t_2)$. The same can be said about fex6t2 and wos19. In general, however, it is often a good idea to favor small subgoals.

We mention in passing that this priority system obtains a proof for SAM's Lemma [Pelletier 86] in a little over 100 seconds. 2340 inferences are performed. The theorem prover without the priority system takes over 4600 seconds for the proof after performing 17000 inferences. It is interesting to note that RRL [Zhang 88] takes nearly 300 seconds to solve this problem. This shows that the prover is pretty efficient handling the equality axioms.

5.6. Related Work

The priority system incorporates the use of priority into depth-first iterative deepening search. With a priority list of length ∞, the priority system will be the same as depth-first iterative deepening search. What is the relationship between the worst case complexity of the priority system and the complexity of depth-first iterative deepening search? The worst-case behavior of the priority system occurs when the priority list reaches its maximal length in Theorem 5.1 to find a solution. Consider the search space formalized as a tree. Assume that the minimal solution length is N, the branching factor is B. We also assume that S_i is 1 and E_i is $i \times S$ for all i in Figure 5.1, and at stage i, the priority list is of length $C^{i-1} \times L$. Suppose a solution is found at stage I. We note that $I = \max\{\frac{N}{S}, (N+1) \times \log_C B - \log_C(L(B-1)) + 1\}$ since we need to have $I \times S \geq N$ and $C^{I-1} \times L \geq \frac{B^{N+1}}{B-1}$. We use DFID(d) to denote the time complexity of the depth-first iterative deepening search when the final cut-off bound is d. The total number of operations performed by the priority system will be at most

$$PS(I) = \sum_{i=1}^{I} DFID(i \times S) = \sum_{i=1}^{I} E \times B^{i \times S} = E \times \sum_{i=1}^{I} B^{i \times S}$$

Table 5.3. Summary Data for Subgoal-Based Priority System		
	Underlying Strategy	Subgoal-Based System
Average Time Per Theorem	224.30	178.86
Average Inferences Per Theorem	1245	4038
Total Theorem	82	85

```
false: -[ ]
   p(a,d,d): -[ ]
      input(p(a,g(g(d)),f(a,g(g(d)))))
      input(p(a,d,f(a,d)))
      q(g(g(d)),d,d): -[ ]
         q(g(g(d)),d,d): -[ ]
            input(q(g(g(d)),g(d),d)
            input(q(g(d),d,d))
            input(q(d,d,d))
         input(q(d,d,d))
         input(q(d,d,d))
      q(f(a,g(g(d))),f(a,d),d): -[ ]
         q(g(f(a,d)),f(a,g(g(d))),f(a,g(g(d)))): -[ ]
            input(q(g(f(a,d)),f(a,d),d))
            q(f(a,d),f(a,g(g(d))),f(a,g(g(d)))): -[ ]
               input(p(a,d,f(a,d)))
               input(p(a,g(g(d)),f(a,g(g(d)))))
               input(q(d,g(g(d)),g(g(d))))
               input(p(a,g(g(d)),f(a,g(g(d)))))
            input(q(d,f(a,g(g(d))),f(a,g(g(d)))))
         q(f(a,g(g(d))),f(a,d),f(a,d)): -[ ]
            input(p(a,g(g(d)),f(a,g(g(d)))))
            input(p(a,d,f(a,d)))
            lemma((q(g(g(d)),d,d): -[ ]))
            input(p(a,d,f(a,d)))
         input(q(g(f(a,d)),f(a,d),d))
```

Figure 5.4: The Proof for fex6t1

$$\approx \frac{B^S}{B^S - 1} \times E \times B^{I \times S} = \frac{B^S}{B^S - 1} \mathrm{DFID}(I \times S)$$

where $E = (\frac{B}{B-1})^2$. This analysis is similar to that in [Stickel&Tyson 85] and uses one result $\mathrm{DFID}(d) = E \times B^d$ from it. We can see that under these assumptions, the worst case performance of the priority system is generally a constant factor $\frac{B^S}{B^S - 1}$ times as expensive as depth-first iterative deepening search. But the priority system can also be less efficient because the depth $I \times S$ can be much larger than the depth required for depth-first iterative deepening search. We point out that this comparison is based on worst-case analysis and does not consider the heuristic effect of the priority system. From our experience, the priority system generally performs better than pure depth-first iterative deepening search. Furthermore, the priority list can be implemented using much less space than the goal queue in the breadth-first [Pearl&Korf 87] search or A* search [Hart&al 68, Huyn&al 80], since only the priority values need to be stored, which seems to be a significant advantage.

What is the relationship between the priority system and the conventional best-first search strategy, or, more specifically, the A* strategy? We note that the priority system is especially designed for depth-first search. In best-first search, the best subgoal in the Open list will be worked on next. Thus the priority lists would be redundant since their purpose is served by the Open list. In best-first search, the search space occupied by some states will be searched first before the search space occupied by others. That is, we have an "unbalanced" expansion of the search space based on the priority function. We try to achieve this effect in our strategy in an iterative deepening fashion with the introduction of the priority list. In a sense this strategy is a compromise between the best-first search and the depth-first iterative deepening search. The priority list would usually dictate a much smaller set of states to explore than the pure depth-first iterative deepening search which explores all possible states. But the priority list is obviously much less informed than the global queue. The priority list will also require much less storage to represent than the subgoal queue would. We can regard the priority system as the iterative-deepening version of the best-first search. However, the well-known results about A* search, such as optimality, admissibility, etc, do not hold for the priority system, when the heuristic functions in A* search are used as the priority functions in the priority system.

What is the relationship between the priority system and the iterative-deepening-A* search? In iterative-deepening-A* search, the priority function returns the estimate of the path cost which is analogous to the concept of the depth in the depth-first search. Both are distance measures. The function of the priority lists can be accomplished by a single threshold value in the iterative-deepening-A* search. In this sense, the iterative-deepening-A* search can be regarded as a special case of the priority system where the priority functions are in a special form. Again, the well-known results about iterative-deepening-A* search, such as optimality, admissibility, etc, do not hold for the priority system.

6. Proof Complexity Measures

In this chapter, we will discuss another refinement to the depth-first iterative deepening search strategy. This refinement is based on the observation that the process of finding a proof is a process of incrementally constructing a ground instance of the input clauses. This incremental process can be quantified and used to control depth-first iterative deepening search.

6.1. The Basic Idea

Many applications in deductive database, logic programming and theorem proving require finding instances which satisfy a certain property. For example, a query p(X) to a database system directs the database system to find an instantiation x for X such that p(x) is a logical consequence of the facts in the database. In resolution-based theorem proving systems, proving a theorem is equivalent to finding an inconsistent set of ground clauses which are instances of the general clauses from the negation of the theorem. Our refinement is based on the observation that search for proofs can be viewed as an incremental process of building up the required instances. This viewpoint is especially natural for a back chaining theorem proving system and can be used to control the search process.

Let's consider the problem reduction format in its purest form [Loveland 78]. One is given a conclusion G to be established and a set of assertions of the form $L :- L_1, L_2, \ldots, L_n$ (implications) or L (premises). An implication $L :- L_1, L_2, \ldots, L_n$ is understood to mean $L_1 \wedge \cdots \wedge L_n \supset L$. The L_i's are the *antecedents* and L is the *consequent*. The top-level goal will be the conclusion G. To confirm a goal L, one begins with a search of the premises to see if any premise matches L. If there is such a premise, L is confirmed. Otherwise, the set of implications is searched and one implication whose consequent matches with L will be selected, if one exists. The antecedents in the implication will be considered as new goals to be confirmed, much in the same manner as L has been.

If a goal L contains some logical variables, a match with a premise or the consequent of an implication will bind these variables with other structures through unification. These bindings will increase the complexity of the proof if a variable in L is bound to a non-variable term, because a structure is more complex than a variable. On the other hand, if a function symbol in L matches with the same function symbol in a premise or the consequent of an implication, the complexity of the proof will not be increased. The complexity of the proof can also be increased if some of the new subgoals $L_1, L_2, \ldots, L_n$ generated to confirm a goal L are more complex than L.

The search should be controlled to avoid the increases in the complexity of the proof. This is particularly easy to do in depth-first iterative deepening search, if we can quantify the increase in the complexity of proofs. We propose a method based on quantifying the complexity of proofs using the *proof complexity measure*.

6.2. Implementation

To implement the proof complexity measure in the prover, we define the two procedures *match_cost* and *clause_cost* used in Figure 2.1. We first define the function *complexity* for a term t.

$$\text{complexity}(t) = \begin{cases} 0 & \text{if } t \text{ is a variable} \\ n + \sum_{i=1}^{n} \text{complexity}(t_i) & \text{if } t = f(t_1, t_2, \ldots, t_n) \text{ and } n \geq \end{cases}$$

For a positive literal $L = p(t_1, t_2, \ldots, t_n)$, we define

$$\text{complexity}(L) = \max\{\text{complexity}(t_1), \ldots, \text{complexity}(t_n)\}.$$

For a negative literal $N = \neg L$, we define

$$\text{complexity}(N) = \text{complexity}(L).$$

The procedure *clause_cost* measures the increase of proof complexity in subgoal generation using the input clauses. Let C denote the input clause $L :- L_1, L_2, \ldots, L_n$. We say that clause C satisfies the *variable condition*, denoted by *variable_condition(C)*, if there is a variable V in C such that V occurs more in one of $L_1, L_2, \ldots, L_n$ than it does in L. Let

$$B = \max\{\text{complexity}(L_1), \text{complexity}(L_2), \ldots, \text{complexity}(L_n)\}$$

and

$$H = \text{complexity}(L).$$

we give two definitions for *clause_cost*, which are called CC1 and CC2 respectively.

$$\text{CC1:} \quad \text{clause\_cost(C)} = \begin{cases} B - H & \text{if } B > H \\ 1 & \text{if } B = H \\ 1 & \text{if variable\_condition(C) and } B < H \\ 0 & \text{otherwise} \end{cases}$$

$$\text{CC2:} \quad \text{clause\_cost(C)} = \begin{cases} B - H & \text{if } B > H \\ 1 + \lfloor \log_3 n \rfloor & \text{if } B = H \\ 1 & \text{if variable\_condition(C) and } B < H \\ 0 & \text{otherwise} \end{cases}$$

These two definitions obviously favor the input clauses which reduce the complexity of the goals. The variable condition is introduced to take into consideration that more occurrences of a variable in a literal of the clause body will increase the complexity of the subgoals during the proof. The logarithmic term in CC2 is introduced to penalize clauses with large numbers of antecedents since they result in bigger branching factors.

The procedure *match_cost* can also be defined in different ways. Its purpose is to determine the complexity increase resulting from binding variables to complex terms as a result of unification. First we explain how the procedure *match* introduced in Figure 2.1 works. The procedure call

$$\text{match}(L_0, L, [L_1, L_2, \ldots, L_n], V, [V_1, V_2, \ldots, V_n])$$

collects the variables in L_0 in the list V, performs the unification operation between L_0 and L, then collects the variables in L_i in the list V_i ($1 \le i \le n$) after the unification operation. In a call *match_cost(L,V)*, L is the goal and V will be the list of terms bound to the variables in L. Let

$$V = [t_1, t_2, \ldots, t_n] \text{ and } S = [s_1, s_2, \ldots, s_m]$$

where S is the set of non-variable subterms of $t_1, t_2, \ldots, t_n$. We give two definitions for *match_cost*, which will be called MC1 and MC2 respectively.

$$\text{MC1:} \quad \text{match\_cost(L, V)} = \sum_{i=1}^{n} \text{complexity}(t_i)$$

$$\text{MC2:} \quad \text{match\_cost(L, V)} = \sum_{i=1}^{m} \text{complexity}(s_i)$$

Note that MC2 does not charge for repeated subterms. The idea is that we can regard a subterm as a piece of information about the proof. The multiple occurrences of the a subterm should be encouraged since this may indicate better concentration of the search process.

Consider an example where the goal is p(f(g(a),X)) and the clause is p(f(X, Y)) :− q(f(X, Y)). The match_cost between p(f(g(a),X)) and p(f(X,Y)) is 0 since no variable in the goal is bound to a complex term. If the subgoal is p(Y) and the clause is p(f(g(a, X))) :− q(X), the match_cost between p(Y) and p(f(g(a,X))) is complexity(f(g(a,X))). In general, it is the variable bindings to complex terms in the goals that increase match_cost.

We have experimented with different definitions of *match_cost* and *clause_cost* using the depth prover in Figure 2.10. We use (CC1, MC1) to denote the prover using CC1 for *clause_cost* and MC1 for *match_cost*. The results are shown in Table A.6 in Appendix A and are summarized in Table 6.1. All four combinations perform well. The combination (CC2,MC2) appears to be the best in general. No special attention is given to any individual problem in these experiments.

6.3. Some Examples

We have customized the definitions for *match_cost* and *clause_cost* to solve several problems from [Wang&Bledsoe 87], including AM8, GCD, LCM, EXQ1 and EXQ2 [Wang 65]. The basic idea is to favor certain function symbols, certain clauses or certain terms by charging less for them. The prover without the proof complexity measures can not solve them as efficiently or can not solve them at all. We will give the details on two problems.

The greatest common divisor problem (GCD). Let gcd(a,b) be the greatest common divisor of two positive integers a and b, for any positive integer c, gcd($a \times c$,$b \times c$) = gcd(a,b) $\times$ c. The first computer proof of this problem is reported in [Wang&Bledsoe 87]. The input clauses are shown in Figure 6.1. The meanings of function and predicate symbols are: g(X,Y,Z) means Z is equal to gcd(X,Y), d(X,Y) means X divides Y, f(X,Y) is equal to X×Y, k(X,Y) is equal to gcd(X,Y), q(X,Y) is the quotient of X dividing Y, and h is a skolem function. All variables denote natural numbers. We added three clauses, d(V,f(X,Y)) :− d(V,f(Y,X)), d(Y,f(X,Y)) and d(f(Y,X), f(Z,X)) :− d(Y,Z), to handle the commutativity of multiplication function f.

Table 6.1. Summary Data for Proof Complexity Measures					
	underlying strategy	(CC1,MC1)	(CC1,MC2)	(CC2,MC1)	(CC2,MC2)
Average Time Per Theorem	224.30	154.95	158.57	191.95	110.26

We give two new definitions for the two functions *clause_cost* and *match_cost*. *Clause_cost* is defined as follows: (1) It returns 1 if the clause body has extra variables or the function nesting in the clause body is deeper than that in the clause head. (2) It returns 0 if the function nesting in the clause head is less or equal to that in the clause body. *Match_cost* is defined as follows:

$$\text{match\_cost}(t) = \begin{cases} 0 & \text{if t is a variable} \\ 0 & \text{if } t \in [f(a,c), f(c,a), f(b,c), f(c,b), f(e,c), f(c,e)] \\ 1 & \text{if t is constant symbol} \\ 1 + \text{complexity}(x) + \text{complexity}(y) & \text{if } t = k(x,y) \\ 3 + \text{complexity}(t_1) + \cdots + \text{complexity}(t_n) & \text{if } t = r(t_1, t_2, \ldots, t_n) \end{cases}$$

where r is any function symbol other than k. These definitions are a customization since they favor subgoals which are syntactically similar to the theorem, by virtue of containing the terms appearing in the theorem and terms which have meaning close to the theorem. Some rewrite rules (demodulators), e.g., *rewrite(f(c,a), f(a,c))* and *rewrite(k(X,X),X)*, are used to simplify the subgoals and to convert the equivalent terms to some uniform representations. The proof for GCD is shown in Figure 6.2. The prover takes about 310 seconds to get the proof after 38 solutions are generated.

The least common multiple problem (LCM). Let lcm(a,b) be the least common multiple of two positive integers a and b, lcm(a,b) = $\frac{a \times b}{\text{gcd}(a,b)}$. The input clauses for LCM are shown in Figure 6.3. To deal with the commutativity of multiplication function f, we added two clauses d(v,f(x,y)) :− d(v,f(y,x)) and d(f(y,x),f(z,x)) :− d(y,z). All the predicate and function symbols have the same meaning as in GCD except that k(X,Y,Z) means Z is equal to lcm(X,Y).

Similar ideas as those used for GCD are used to solve this problem. The procedure *clause_cost* is defined similarly and *match_cost* is defined as follows, favoring terms closely related to the theorem.

$$\text{match\_cost}(t) = \begin{cases} 0 & \text{if t is a variable} \\ 0 & \text{if } t \in [f(a,c), f(a,b), a, b, q(f(a,b),c)] \\ 1 & \text{if t is constant symbol} \\ \text{complexity}(x) + \text{complexity}(y) + \text{complexity}(z) & \text{if } t = k(x,y,z) \\ 1 + \text{complexity}(t_1) + \cdots + \text{complexity}(t_n) & \text{if } t = r(t_1, t_2, \ldots, t_n) \end{cases}$$

where r is any function symbol other than k. The proof for LCM is in Figure 6.4. The prover takes about 220 seconds to obtain the proof after 35 new solutions are generated.

```
false :– g(f(a,c),f(b,c),f(e,c)).
g(X,Y,U) :– d(U,X),d(U,Y),d(h(X,Y,U),U).
d(V,f(X,Y)) :– d(V,f(Y,X)).
d(X,f(Y,Z)) :– d(Y,X),d(q(X,Y),Z).
d(X,Z) :– d(X,Y),d(Y,Z).
d(V,U) :– g(X,Y,U),d(V,X),d(V,Y).
d(Y,f(X,Y)).
d(X,f(X,Y)).
d(X,f(Y,Z)) :– d(X,Y).
d(q(X,Y),Z) :– d(X,f(Y,Z)),d(Y,X).
d(k(Y,X),X).
d(k(Y,X),Y).
g(a,b,e).
d(V,k(Y,X)) :– d(V,X),d(V,Y).
d(h(X,Y,U),X) :– d(U,X),d(U,Y), not g(X,Y,U).
d(h(X,Y,U),Y) :– d(U,X),d(U,Y), not g(X,Y,U).
d(f(X,Y),f(X,Z)) :– d(Y,Z).
d(f(Y,X),f(Z,X)) :– d(Y,Z).
not g(f(a,c),f(b,c),f(e,c)).
d(X,q(Y,Z)) :– d(f(X,Z),Y),d(Z,Y).
d(U,Y) :– g(X,Y,U).
d(U,X) :– g(X,Y,U).
d(q(X,X),Y).
d(X,X).
```

Figure 6.1: Input Clauses of GCD Theorem

```
false: –[ ]
    g(f(a,c),f(b,c),f(e,c)): –[ ]
        d(f(e,c),f(a,c)): –[ ]
            d(e,a): –[ ]
                input(g(a,b,e))
        d(f(e,c),f(b,c)): –[ ]
            d(e,b): –[ ]
                input(g(a,b,e))
        d(h(f(a,c),f(b,c),f(e,c)),f(e,c)): –[ ]
            d(h(f(a,c),f(b,c),f(e,c)),k(f(a,c),f(b,c))): –[ ]
                d(h(f(a,c),f(b,c),f(e,c)),f(a,c)): –[ ]
                    lemma(d(f(e,c),f(a,c)): –[ ])
                    lemma(d(f(e,c),f(b,c)): –[ ])
                    input(not g(f(a,c),f(b,c),f(e,c)))
                d(h(f(a,c),f(b,c),f(e,c)),f(b,c)): –
                    lemma(d(f(e,c),f(a,c)): –[ ])
                    lemma(d(f(e,c),f(b,c)): –[ ])
                    input(not g(f(a,c),f(b,c),f(e,c)))
            d(k(f(a,c),f(b,c)),f(e,c)): –[ ]
                d(k(f(a,c),f(b,c)),f(c,e)): –[ ]
                    d(c,k(f(a,c),f(b,c))): –[ ]
                        input(d(c,f(a,c)))
                        input(d(c,f(b,c)))
                    d(q(k(f(a,c),f(b,c)),c),e): –[ ]
                        input(g(a,b,e))
                        d(q(k(f(a,c),f(b,c)),c),a): –[ ]
                            input(d(k(f(a,c),f(b,c)),f(c,a)))
                            lemma(d(c,k(f(a,c),f(b,c))): –[ ])
                        d(q(k(f(a,c),f(b,c)),c),b): –[ ]
                            input(d(k(f(a,c),f(b,c)),f(c,b)))
                            lemma(d(c,k(f(a,c),f(b,c))): –[ ])
```

Figure 6.2: Proof of GCD Theorem

```
not lcm(a,b,q(f(a,b),c)).
false :– lcm(a,b,q(f(a,b),c)).
lcm(X,Y,U) :– d(X,U),d(Y,U),d(U,k(Y,X,U)).
d(X,q(Y,Z)) :– d(f(X,Z),Y).
d(q(X,Y),Z) :– d(Y,X),d(X,f(Y,Z)).
g(f(X,Z),f(Y,Z),f(U,Z)) :– g(X,Y,U).
d(V,U) :– g(X,Y,U),d(V,X),d(V,Y).
d(U,f(X,Y)) :– d(U,f(Y,X)).
d(f(X,Y),f(X,Z)) :– d(Y,Z).
d(f(Y,X),f(Z,X)) :– d(Y,Z).
d(X,f(Y,Z)) :– d(X,Y).
d(U,Y) :– g(X,Y,U).
d(U,X) :– g(X,Y,U).
g(a,b,c).
d(X,k(Y,X,U)) :– d(X,U),d(Y,U),not lcm(X,Y,U).
d(Y,k(Y,X,U)) :– d(X,U),d(Y,U),not lcm(X,Y,U).
d(X,X).
d(X,f(X,Y)).
d(q(X,X),Y).
d(X,f(Y,Z)) :– d(q(X,Y),Z),d(Y,X).
d(h(Y,X,U),X) :– not g(X,Y,U),d(U,X),d(U,Y).
d(h(Y,X,U),Y) :– not g(X,Y,U),d(U,X),d(U,Y).
g(X,Y,U) :– d(h(Y,X,U),U),d(U,X),d(U,Y).
d(X,Z) :– d(X,Y),d(Y,Z).
```

Figure 6.3: Input Clauses for LCM Theorem

6.4. Comments

The idea of using proof complexity measures to control depth-first iterative deepening search is very intuitive. Although it requires more user interaction, proof complexity measures are easy to use. More importantly, they can be used to solve more difficult problems. The proof complexity measures that we use do not charge anything for the matches between constant symbols (function symbols or predicate symbols) in unification and usually do not charge anything for a match between variables and constants. As a result, proof paths which contain fewer variables are favored. This makes our method similar to

```
false: – [ ]
    lcm(a,b,q(f(a,b),c)): – [ ]
        d(a,q(f(a,b),c)): – [ ]
            d(f(a,c),f(a,b)): – [ ]
                d(c,b): – [ ]
                    input(g(a,b,c))
        d(b,q(f(a,b),c)): – [ ]
            d(f(b,c),f(a,b)): – [ ]
                d(f(b,c),f(b,a)): – [ ]
                    d(c,a): – [ ]
                        input(g(a,b,c))
        d(q(f(a,b),c),k(a,b,q(f(a,b),c))): – [ ]
            d(c,f(a,b)): – [ ]
                lemma(d(c,a) : – [ ])
            d(f(a,b),f(c,k(a,b,q(f(a,b),c)))): – [ ]
                g(f(a,k(a,b,q(f(a,b),c))),f(b,k(a,b,q(f(a,b),c))),f(c,k(a,b,q(f(a,b),c)))): – [ ]
                    input(g(a,b,c))
                d(f(a,b),f(a,k(a,b,q(f(a,b),c)))): – [ ]
                    d(b,k(a,b,q(f(a,b),c))): – [ ]
                        lemma(d(a,q(f(a,b),c)): – [ ])
                        lemma(d(b,q(f(a,b),c)): – [ ])
                        input(not lcm(a,b,q(f(a,b),c)))
                d(f(a,b),f(b,k(a,b,q(f(a,b),c)))): – [ ]
                        d(f(a,b),f(k(a,b,q(f(a,b),c)),b)): – [ ]
                            d(a,k(a,b,q(f(a,b),c))): – [ ]
                                lemma(d(a,q(f(a,b),c)): – [ ])
                                lemma(d(b,q(f(a,b),c)): – [ ])
                                input(not lcm(a,b,q(f(a,b),c)))
```

Figure 6.2: Proof of LCM Theorem

the idea of *twin symbols* in [Wang&Bledsoe 87], although we use depth-first iterative deepening search and [Wang&Bledsoe 87] uses best-first search.

7. Using Semantic Information

7.1. Overview

It is widely acknowledged that the syntactic approaches alone are not sufficient to produce a powerful theorem prover; problem domain knowledge must be used [Bledsoe 77, Bledsoe 86, Reiter 76]. For an excellent survey on the various techniques of using problem domain knowledge, see [Bledsoe 86]. One such technique is to encode problem domain knowledge in some *models* or *examples* which will be used to guide the proof. In this technique, only the *semantically provable paths* are pursued, that is, only the goals which are interpreted to be T in the models (or interpreted to be F if some refutation procedures are used) will be attempted. In the following, we will call problem domain knowledge *semantic information* or simply *semantics* since interpretations are used to define the semantics of first order logic.

Many inference systems that use semantic information have been studied. Gelernter's Geometry Theorem Prover [Gelernter 59], which proves theorems in plane geometry, is the earliest system, Gelernter's system uses back chaining and represents semantic information by diagrams. Subgoals are tested in the diagram and unachievable subgoals are discarded. The inference system in Gelernter's system is similar to that of Prolog and is only complete for Horn clauses. Reiter [Reiter 76] propose a natural deduction system which uses arbitrary interpretations to delete unachievable subgoals. Reiter's system is not complete for first order logic either. The set of support strategy [Wos 65] is a powerful and completeness preserving restriction strategy for resolution [Chang&Lee 73]. The set of support strategy uses some interpretations to divide the clauses set into two subsets, one of which is satisfiable and the other is called the *set of support*. Resolution is allowed between two clauses only if one of the two clauses depends on the set of support. Slagle

[Slagle 67] proposes semantic resolution which is a generalization of hyper-resolution to arbitrary models. Semantic resolution gives a semantic criterion for restricting which resolutions are performed. Sandford [Sandford 80] proposes hereditary lock resolution which combines lock resolution [Boyer 70] and the model strategy [Luckham 70] and also gives a semantic criterion for restricting resolutions. No implementation is described in [Sandford 80]. Wang [Wang 85] proposes semantically guided hierarchical deduction which uses only false resolvents in some model as goal clauses in the hierarchical deduction procedure. Wang's method is complete if the models are designed properly.

We will present a generalization of the modified problem reduction format. The resulting system, semantic modified problem reduction format, is in the spirit of Slagle's system and is developed based on the observations that the modified problem reduction format selects its inference rules according to a particular interpretation in which all the positive literals are true. The semantic modified problem reduction format uses an arbitrary interpretation for selecting its inference rules and deletes unachievable subgoals according to the interpretation. It can also use multiple models of a subset of the clause set to stop application of some inference rule (the case analysis rule). The advantage of our system over Slagle's semantic resolution is that the semantic modified problem reduction format has a natural goal-subgoal structure and supports back chaining. Our inference system can be regarded as a generalization of Gelernter's system to first order logic. It is compatible with the set of support strategy and can use interpretation to suggest instantiations for free variables in a natural way. Also, a technique similar to the splitting technique in [Bledsoe 71] can be incorporated into the system.

7.2. Semantic Modified Problem Reduction Format

We generalize the modified problem reduction format as follows. For the moment, we will assume that, for a set S of clauses, the input clauses include all the contrapositives of the clauses in S and all the clauses of the form false $:- L_1, L_2, \ldots, L_n$ where $\neg L_1 \vee \neg L_2 \vee \cdots \vee \neg L_n$ is a clause in S. Given an interpretation M for S, the inference rules for the semantic modified problem reduction format are in Figure 7.1. The inference rules in Figure 7.1 differ from those in Figure 2.1 in that, before a subgoal $\Gamma \rightarrow L$ can be solved using the clause rules or the case analysis rule, a semantic test has to be performed on L to make sure that L is satisfied by the interpretation M.

We will show the soundness and completeness of the semantic modified problem reduction format. We only deal with the ground case, which can be lifted to first order logic in the usual way.

Clause Rule for Horn-like Clause false : – L_1 , L_2 , . . . , L_n

$$\frac{M \mid =_E L, [\Gamma_0 \rightarrow L_1 => \Gamma_1 \rightarrow L_1] ,\ [\Gamma_1 \rightarrow L_2 => \Gamma_2 \rightarrow L_2] , \ldots , [\Gamma_{n-1} \rightarrow L_n => \Gamma_n \rightarrow L_n]}{\Gamma_0 \rightarrow L => \Gamma_n \rightarrow L}$$

Assumption Axioms

$\Gamma \rightarrow L => \Gamma \rightarrow L$ if $L \in \Gamma$ L is a literal.

$\Gamma \rightarrow \neg L => \Gamma , \neg L \rightarrow \neg L$ L is a positive literal.

Case Analysis Rule

$$\frac{M \mid =_E L, [\Gamma_0 \rightarrow L => \Gamma_1, \neg N \rightarrow L],\ [\Gamma_1 , N \rightarrow L => \Gamma_1 , N \rightarrow L] \quad |\Gamma_0| \leq |\Gamma_1|}{\Gamma_0 \rightarrow L => \Gamma_1 \rightarrow L}$$

Figure 7.1: Semantic Modified Problem Reduction Format

Theorem 7.1 (soundness): If $|\!\!-_S \Gamma_1 \rightarrow L => \Gamma_2 \rightarrow L$, then list Γ_1 is a prefix of list Γ_2 and $S| == \Gamma_2 \supset L$.

Proof. We can prove this by induction on the size of the proof, i.e., the number of times the inference rules are used, making use of the length restriction in the case analysis rule (This theorem is the same as the soundness theorem in [Plaisted 88]). Q.E.D.

Theorem 7.2 (completeness) If a set of clauses S is unsatisfiable and M is an interpretation for S which interprets false to be *T*, then $|\!\!-_S \rightarrow$ false => $|\!\!-_S \rightarrow$ false.

Proof. Let atom(S) denote the set of atoms in S and N-atom(S) denote the set of literals $\neg L$ where $L \in$ atom(S). Let P be the set of literals in S which are interpreted to be T by M, then $P \subseteq$ atom(S) $\cup$ N-atom(S) $\cup$ {false}. For every atom A ($\in$ atom(S)), either $A \in P$ or $\neg A \in P$, but not both. Specifically, false $\in$ M. Consider the following set of inference rules which includes the clause rules of all the contrapositives of clauses in S

$$\frac{L \in P, [\Gamma_0 \rightarrow L_1 => \Gamma_1 \rightarrow L_1] ,\ [\Gamma_1 \rightarrow L_2 => \Gamma_2 \rightarrow L_2] , \ldots , [\Gamma_{n-1} \rightarrow L_n => \Gamma_n \rightarrow L_n]}{\Gamma_0 \rightarrow L => \Gamma_n \rightarrow L}$$

plus the assumption axioms

$$\Gamma \rightarrow L \Rightarrow \Gamma \rightarrow L \;\; \text{if} \;\; L \in \Gamma$$

$$\Gamma \rightarrow L \Rightarrow \Gamma , L \rightarrow L \;\; \text{if} \;\; \neg L \in P$$

and the case analysis rule

$$\frac{L \in P,\ \neg N \in P,\ [\Gamma_0 \rightarrow L \Rightarrow \Gamma_1, N \rightarrow L],\ \ [\Gamma_1 , \neg N \rightarrow L \Rightarrow \Gamma_1 , \neg N \rightarrow L] \quad |\Gamma_0| \le |\Gamma_1|}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_1 \rightarrow L}$$

This system is complete from the proof of Theorem 1.2 [Plaisted 88], if we regard all the literals in P as positive and all other literals as negative. The completeness of our system follows if we observe that, if we use the assumption axioms and case analysis rule as shown in Figure 7.1, we are merely fixing the order of the two cases for each application of the case analysis rule. Obviously it does not matter in which order the case analysis is done. Q.E.D.

7.3. Discussions

The modified problem reduction format can be regarded as a special case of the semantic modified problem reduction format where M = atom(S) $\cup$ {false}. However, this generalization has some advantages over the modified problem reduction format. We will discuss them in this section.

7.3.1. Strengthening the System

The semantic modified problem reduction format can be made stronger than presented above in several ways. First, consider the input clause $L :- L_1 , L_2 , \ldots , L_n$. M is the interpretation used. Suppose a subgoal $\Gamma_0 \rightarrow L$ is being attempted during the proof. The use of the clause rule

$$\frac{M \models_E L,\ [\Gamma_0 \rightarrow L_1 \Rightarrow \Gamma_1 \rightarrow L_1] ,\ \ [\Gamma_1 \rightarrow L_2 \Rightarrow \Gamma_2 \rightarrow L_2] , \ldots , [\Gamma_{n-1} \rightarrow L_n \Rightarrow \Gamma_n \rightarrow L_n]}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_n \rightarrow L}$$

can be stopped if there exists a literal L_i among $L_1, L_2 , \ldots , L_n$ such that L_i is a positive literal and $M \not\models_E L_i$ and $L_i \notin \Gamma_0$. This is because when $\Gamma_{i-1} \rightarrow L_i$ is attempted, the only way to solve it is to use the assumption axiom. But it is impossible to have any extra positive literal in Γ_{i-1} other than those already in Γ_0, since the assumption axiom only adds negative literals and the case analysis rule, although adding positive literals, will remove them itself. Second, a new solution $\Gamma_0 \rightarrow L$ need not be recorded, in case caching is performed, if $M \not\models_E L$, since the subgoals of the form $\Gamma \rightarrow L$ will never be attempted except using the assumption axioms. Finally, consider the clause $L :- L_1, L_2 , \ldots , L_n$. If $M \not\models_E L$, the corresponding clause rule will never be used in the inference system. Thus we need not include such clauses in our input. In general, we only need to include those clauses whose heads are existentially satisfied by M.

7.3.2. Set of Support Strategy

The set of support strategy [Wos 65] is a powerful and completeness preserving restriction strategy for resolution. This strategy divides a clause set S into two sets A and T where $S = A \cup T$ and $A \cap T = \emptyset$. A usually represents the axioms and the special hypotheses and is satisfiable and T represents the negation of the theorem and is called the *set of support*. A resolution operation is only allowed for two clauses if at least one of them comes from the set of support T. The modified problem reduction format is not compatible with the set of support strategy because its goal clauses are always all-negative clauses which do not always come from the negation of the theorem. We will show that semantic modified problem reduction format is compatible with the set of support strategy. In fact, we will show in Theorem 7.3 that it is only necessary to have one goal clause according to some interpretation M (a goal clause is a clause whose clause head is the literal false), which is a somewhat stronger result. This means that we need not include all the clauses of the form false :– $L_1, L_2, \ldots, L_n$ where $\neg L_1 \vee \neg L_2 \vee \cdots \vee \neg L_n$ is a clause, contrary to what we stated earlier.

Theorem 7.3: If a set of clauses S is unsatisfiable, then there is an interpretation M, a subset S_1 of S and a clause $G \in S_1$, such that, $M \mid=_U S_1 - \{G\}$ and S_1 is unsatisfiable.

Proof. Assume that $S = A \cup T$, $A \cap T = \emptyset$, $T \neq \emptyset$ and A is satisfiable. Further assume $T = \{C_1, C_2, \ldots, C_n\}$. If there exists an interpretation I such that $M \mid=_U A$ and $M \mid=_U C_i$ $(i = 1, 2, \ldots, n-1)$ and $M \not\models_U C_n$, we are done since we can let $S_1 = S$, $G = C_n$ and $M = I$. S_1 is obviously unsatisfiable. If there is no such an interpretation, we have $|==$ $(A \cup \{C_1, C_2, \ldots, C_{n-1}\}) \supset C_n$, that is, $(A \cup \{C_1, C_2, \ldots, C_{n-1}\}) \supset C_n$ is valid. Thus $A \cup \{C_1, C_2, \ldots, C_{n-1}\} = S_0$ is unsatisfiable since S is. We can apply the same argument to S_0. Q.E.D.

7.3.3. Multiple Models

The inference system allows limited use of multiple models to delete unachievable subgoals. Consider the case analysis rule, where M is the interpretation used,

$$\frac{M \mid=_E L,\ [\Gamma_0 \rightarrow L => \Gamma_1, \neg N \rightarrow L],\ \ [\Gamma_1, N \rightarrow L => \Gamma_1, N \rightarrow L] \quad |\Gamma_0| \leq |\Gamma_1|}{\Gamma_0 \rightarrow L => \Gamma_1 \rightarrow L}$$

Suppose the clause set is $S = S_0 \cup \{G\}$ where G is the only goal clause. We may have several models $M_1, \ldots, M_k$ for S_0. In this event, the application of the case analysis rule can be stopped if there exists a M_j among $M_1, \ldots, M_k$ such that $M_j \not\models_E \Gamma_1 \rightarrow L$. This follows from the soundness theorem since $S_0 \mid== \Gamma_1 \rightarrow L$ and G is the only goal clause.

7.3.4. Gelernter's method

The earliest and perhaps most successful example of using semantics could be Gelernter's Geometry Engine[Gelernter 59], which proves theorems in plane geometry, where diagrams are used as interpretations and the subgoals unachievable in the diagram are discarded. Gelernter's Geometry Engine uses an interpretation, which is usually a diagram, and a set of inference rules of the form

$$\frac{G_1 \ , \quad G_2 \quad , \ldots , \quad G_n}{G}$$

where $G_1 \land G_2 \land \cdots \land G_n \supset G$ is an axiom or a hypothesis. We call G's goals or subgoals. The inference system is in problem reduction format of the purest form [Loveland 78]. and is that of Prolog and only complete for Horn Clause Logic [Lloyd 84]. A rule can be used only if the goal G is interpreted to be T by the interpretation.

If the clause set consists of only Horn clauses and the interpretation only interprets positive literals to be *T*, the semantic modified problem reduction format is the same as Gelernter's method. Since it is complete for first order logic, the semantic modified problem reduction format is a generalization of Gelernter's method to first order logic.

7.3.5. Other Refinements

Limiting variable instantiations. When a subgoal $\Gamma \rightarrow L$ is attempted, if there are only a few, or possibly one, instantiations for some variables in $\Gamma \rightarrow L$ which make the subgoal true in the interpretation, we can limit all the future instantiations for these variables to those few.

Splitting technique. In [Bledsoe 71], the splitting technique is proposed where, to prove a theorem $A \land B$, A and B are proven as separate theorems. A similar technique can be used in our system by making use of the assumption list. The inference rule

$$\frac{\Gamma , \neg L \rightarrow L}{\Gamma \rightarrow L}$$

is sound since it is a special case of the case analysis rule

$$\frac{[\Gamma \rightarrow L => \Gamma, L \rightarrow L] \quad [\Gamma, \neg L \rightarrow L => \Gamma, \neg L \rightarrow L]}{\Gamma \rightarrow L => \Gamma \rightarrow L}$$

where $\Gamma \rightarrow L => \Gamma, L \rightarrow L$ is the assumption axiom in the semantic modified problem reduction format.

The splitting technique as presented has some advantages. First, we can make use of the unit fact which is the negation of the subgoal via the assumption axiom. This can be regarded as a limited use of the ancestry resolution as in the Model Elimination

[Loveland 69] or SL-Resolution [Kowalski&Kuehner 71]. Second, the search can be better directed since we can start a new search on the goal $\Gamma, \neg L \to L$, which is easy since back chaining is supported by the inference system. However, this technique is powerful only if we can effectively identify the important subproblems and is admittedly difficult to control, since it can be used to solve any goal. Some heuristics or user guidance is usually needed. One useful criterion for using the technique to solve a goal $\Gamma \to L$ is that L is interpreted to be T by the interpretation and L is a frequently encountered subgoal or is a subgoal of the top-level goal.

A set theoretic problem can be used to illustrate splitting as a proof technique: $A \cap B = B \cap A$ ([McCharen&al 76]). To show two sets A and B are equal, it suffices to prove $A \subseteq B$ and $B \subseteq A$. To prove $A \cap B = B \cap A$, the following subgoals are generated:

$$(1)\ \ \text{false} :- [\ ]$$

$$(2)\ \ A \cap B = B \cap A :- [\ ]$$

$$(3.1)\ \ A \cap B \subseteq B \cap A :- [\ ] \quad (3.2)\ \ B \cap A \subseteq A \cap B :- [\ ]$$

Subgoal (3.1) can be proven by proving $A \cap B \subseteq B \cap A :- [\neg(A \cap B \subseteq B \cap A)]$ and subgoal (3.2) can be solved by proving $B \cap A \subseteq A \cap B :- [\neg(B \cap A \subseteq A \cap B)]$.

7.3.6. Contrapositives

The Prolog-style extension to first order logic (non-Horn clauses) using the Model Elimination procedure [Loveland 69] requires the use of all the contrapositives if the clause set contains non-Horn clauses [Stickel 88]. The modified problem reduction format, on the other hand, does not need any contrapositive at all. Using contrapositives sometimes reduces efficiency since it increases the number of clauses. More importantly, though, it can lead to unnatural search behavior and make the search process difficult to control. See [Plaisted 88] for some examples. However, our experience has shown that, especially for non-Horn problems, some contrapositives can significantly improve the efficiency of the prover using the modified problem reduction format. The reason is that contrapositives may reduce the number of case analyses by performing unit resolution using negative literals [Chang 70, Overbeek&al 76]. For example, consider a ground version for part of the intermediate value theorem. Using the clause set in Figure 7.2, a proof will need two case analyses if the case analysis is done at the top-level (at the subgoal of the form false :– Γ). They correspond to the four cases ([¬ B, ¬ E], [B, ¬ E], [¬ B, E], [B, E]). A proof which needs no case analyses at all can be found if we use the set of clauses in Figure 7.3, which includes some contrapositives.

FALSE : – A.
A : – ¬B, C, D.
D : – ¬E.
E : – B.
A : – E.
C : – D.

Figure 7.2: An Example on Contrapositive

The problem is to decide which contrapositives to use. A semantics-based solution would be best since each input clause represents a potential proof path for some goals, and all the proof paths should be semantically provable. The semantic modified problem reduction format provides such a solution by making use of the problem domain knowledge represented in an interpretation. Given an interpretation M for a set S of clauses, we need only include the Horn-like clauses whose clause heads are existentially satisfied by M. We require existential satisfiability since the inference system is a refutational system. This point will become clearer in the discussion on interpretations.

Interpretations. As we have noted, given a set of clauses S and an interpretation M for S, we only need to have, as input clauses, those whose clause heads are existentially

FALSE : – A.
¬A.
A : – ¬B, C, D.
D : – ¬E.
¬B : – ¬E.
¬E : – ¬A.
C : – D.

Figure 7.3: An Example on Contrapositive (cont.)

satisfied by M. The problem of how to design a suitable interpretation for a given theorem is not a trivial one. It is difficult to automate, since problem domain knowledge is usually required and it is hard to give a precise description of what is a suitable interpretation. The difficulty for a human in designing an interpretation lies in the interpretation of the skolem functions.

A method for designing interpretations for a set of clauses is proposed in [Wang 85]. The basic idea of Wang's method is to put together all the clauses containing the same uninterpreted symbol, which are often skolem function symbols, and use some interpretation rules to interpret the uninterpreted symbol. It is a general method and can be slightly modified to select the input clauses for our system. We will briefly present the modified method.

Given a natural interpretation I for a theorem and the natural interpretations for the function symbols and predicate symbols, we need to interpret the uninterpreted symbols. We call a simplified first-order formula an *interpretation normal form* (INF) if it is in the following form:

$$L_1 \lor \cdots \lor L_k \lor [C_1 \land \cdots \land C_h]$$

where L_i's are literals and C_i's are clauses. The concept of INF is defined so that all the clauses containing the same uninterpreted symbol can be grouped together in one INF. Note that, corresponding to each INF $L_1 \lor \cdots \lor L_k \lor [C_1 \land \cdots \land C_h]$, there is an equivalent set of clauses $L_1 \lor \cdots \lor L_k \lor C_1$, $L_1 \lor \cdots \lor L_k \lor C_2$, . . . , $L_1 \lor \cdots \lor L_k \lor C_h$. There are two special cases of INF: a clause $L_1 \lor \cdots \lor L_k$ is in INF form where $h = 1$ and $C_1 = \Box$ ($\Box$ denotes the empty clause) and a formula $C_1 \land \cdots \land C_h$ is in INF form where $k = 1$ and $L_1 = \Box$.

There are two *interpretation rules* for a formula in INF form. These two interpretation rules can be used to select all the necessary contrapositives when the two interpretation rules are applied. For each interpretation instance of an INF form

$$L_1 \lor \cdots \lor L_k \lor [C_1 \land \cdots \land C_h]$$

R1 if $I \models_E [\neg L_1 \land \cdots \land \neg L_k]$, then for each i ($1 \le i \le h$), $I \models_U C_i$, except when C_i is the negation (or part of) of the theorem. This rule simply states that the interpretation should satisfy all axioms and hypotheses of the theorem. Note that each C_i ($1 \le i \le h$) is a clause. Let some C_i be the clause $N_1 \lor N_2 \lor \cdots \lor N_n$. Each N_j ($1 \le j \le n$) would be the head of some Horn-like clause. However, if for some N_s ($1 \le s \le n$), $I \not\models_E N_s$, we need not to have the Horn-like clause whose head is N_s for the simple reason that the clause would never be used if it were an input clause. If for some j, $I \not\models_E C_j$, we will have a goal clause false :– $\neg L_1$, . . . , $\neg L_k$, $\neg C_j$, with a little misuse of notation.

R2 if $I \models_U [L_1 \vee \cdots \vee L_k]$, then there should be at most one i $(1 \le i \le h)$, $I \models_E \neg C_i$. This rule simply states that there should be at most one proof path for any subgoal. Each L_i $(1 \le i \le h)$ would be the clause head of some Horn-like clauses. For the same reason, we need not have those Horn-like clauses whose head is L_s $(1 \le s \le h)$ where $I \not\models_E L_s$. If $I \models_E L_m$ $(1 \le m \le h)$, we would have, again, with a little misuse of notation, the following Horn-like clauses

$$
\begin{aligned}
&L_m :- \neg L_1 , \ldots , \neg L_k , \neg C_1 .\\
&L_m :- \neg L_1 , \ldots , \neg L_k , \neg C_2 .\\
&\quad \cdots\\
&L_m :- \neg L_1 , \ldots , \neg L_k , \neg C_h .
\end{aligned}
$$

Consider the task of proving L_m. Rule R2 states that there should be only one rule for proving L_m since there is at most one C_j such that $I \models_E \neg C_j$.

7.4. Some Examples

It is easy to implement the semantic modified problem reduction format, using the existing implementation of the modified problem reduction format in Chapter 2. We only need to add the necessary Prolog procedures to perform the semantic test. We will discuss several examples and show some experimental results.

Intermediate Value Theorem (IMV). If a function f is continuous in a real closed interval [a, b], where $f(a) \le 0$ and $f(b) \ge 0$, then $\exists x\, [(a \le x) \wedge (x \le b) \wedge (f(x) = 0)]$. The input clauses of IMV are shown in Figure 7.4. In Figure 7.4, p(X, Y) means $X \le Y$, a and b are the endpoints of a closed interval [a, b], f is a continuous function in [a, b], k, h and g are skolem functions. Some discussion of this problem can be found in [Ballantyne&Bledsoe 82, Bledsoe 82]. This problem is interesting for us because several techniques contribute to solving it.

We have designed an interpretation for IMV using the method described in Section 7.3. At the same time, we select the necessary contrapositives based on the resulting interpretation. Some contrapositives are determined to be unnecessary. For example, one of the clauses in Figure 7.4 is

$$p(f(x),0) \vee \neg p(a, x) \vee \neg p(x, b) \vee \neg p(x, h(x))$$

Note that any reasonable interpretation will not interpret $\neg p(a, x)$ or $\neg p(x, b)$ to be *T*. Thus the Horn-like clauses with $\neg p(a, x)$ or $\neg p(x, b)$ as heads are unnecessary.

The top-level goal for IMV is $\exists x\, [f(x) = 0]$, which is instantiated to $f(d) = 0$ using the interpretation. From the interpretation for the predicate p, we get two subgoals p(f(d),

```
p(a, b).
p(f(a), 0).
p(0, f(b)).
p(X, d) :– p(X, b), p(f(X), 0).
p(f(g(X)), 0) :– not(p(d, X)).
p(g(X), b) :– not(p(d, X)).
p(d, X) :– p(g(X), X).
p(X, X).
p(X, Z) :– p(X, Y), p(Y, Z).
p(X, Y) :– not(p(Y, X)).
p(X, Y) :– p(X, q(Y, X)).
p(X, Y) :– p(q(Y, X), Y).
p(f(X), 0) :– p(a, X), p(X, b), p(X, h(X)).
p(f(X), 0) :– not(p(Z, h(X))), p(a, X), p(X, b), p(Z, X), p(f(Z), 0).
p(0, f(X)) :– p(a, X), p(X, b), p(k(X), X).
p(0, f(X)) :– not(p(k(X), Z)), p(a, X), p(X, b), p(X, Z), p(0, f(Z)).
false :–  p(f(X), 0), p(0, f(X)).
```

Figure 7.4: Input Clauses for IMV Theorem

0) and p(0, f(d)). The splitting technique is used to solve these two subgoals separately, much like a human would do for this problem. Our prover fails to obtain a proof for IMV without the contrapositives. With the contrapositives and without using the splitting technique, a prover takes about 15,000 seconds to obtain the proof. More than 45,000 inferences are performed. The prover is able to obtain a proof in about 200 seconds using the splitting technique and the contrapositives. However, if we delete the ground subgoals interpreted to *F* in the interpretation, the prover obtains a proof in about 300 seconds using the splitting technique and the contrapositives.

Schubert's Statement [Walther 84]. This problem has been the subject of much study. We have done several experiments on it. The prover gets a proof for this problem in 748 seconds, without any contrapositive. 8921 inferences are performed. With all the contrapositives, the prover takes 730 seconds to get a proof. 7939 inferences are performed. For some reason, it does not makes a big difference if an interpretation is used to delete false subgoals. The proof obtained using contrapositives is much smaller than the one without contrapositives.

Attaining Minimum (or Maximum) Theorem (AM8). A continuous function f in a closed real interval [a, b] attains its minimum (or maximum) in this interval [Wang&Bledsoe 87]. Some contrapositives are added when we design an interpretation for the input clauses of this problem. Without the contrapositives, the prover did not obtain a proof. The prover finds a proof in about 2 hours with the contrapositives. A look at the proof tells why. The added contrapositives reduce the number of case analyses by solving many negative subgoals directly.

7.5. Comments

We have presented a generalization of the modified problem reduction format, the semantic modified problem reduction format. The resulting system supports back chaining and is compatible with the set of support strategy. It allows the deletion of false subgoals in some interpretation. What is most interesting about our system is probably that it provides an answer to the problem about how contrapositives are handled in similar systems. It seems that similar back chaining systems can not be compatible with the set of support strategy without using contrapositives. The semantic modified problem reduction format can still benefit from the set of support strategy while not needing to use all the contrapositives. This is not without cost, however. The user has to provide an interpretation and the necessary contrapositives, although the process of selecting the contrapositives can be automated if the user provides the interpretation.

8. Conclusions

8.1. Comparison with Other Provers

How does our theorem prover compare with other similar theorem provers? We have tested two other well known theorem provers using our test problem set. These two provers have been used to solve many problems efficiently, some of them are open problems. One of them is the resolution-based theorem prover OTTER [McCune 88]. The other is the SHD-prover developed by T.C. Wang [Wang 86]. We show the performance statistics of the two provers on the test problems in Appendix B and Appendix C.

Before we make any comment on the test results, we have to concede that it is difficult to compare different theorem provers. One reason is that different theorem provers are designed for different purposes. For example, the theorem prover by Boyer and Moore [Boyer&Moore 86] is primarily for proof checking, which requires a lot of user interaction. The theorem prover by Greenbaum [Greenbaum 87] is designed to be used with as little user interaction as possible. Another reason is that different theorem provers use different problem representations. While the most common is the clause form representation, other representations are also used. An example is the computational logic in [Boyer&Moore 79]. Yet another reason is that different theorem provers use different implementation techniques and different implementation languages. Finally, people who undertake the task of comparing different theorem provers may understand one prover better than others.

OTTER, SHD-prover and our theorem prover all use the clause form representation. None of them needs any user interaction during the proof process. During our experiments, we have only used the default strategies without any special attention to any individual problem. We have been also very careful to prepare the problem input to OTTER and SHD-prover so that the best possible results be obtained. We hope these precautions will offset some of the difficulties mentioned above to give us an objective view of how well our theorem prover performs.

We think the performance of our prover is comparable with OTTER and SHD-prover on our test problem set, in spite of the fact that our prover is implemented in Prolog. OTTER seems to perform best. However, we have to be very careful to select the right set of support when using OTTER. Failure to do so often leads to much poorer performance. SHD-prover seems to be erratic, especially when dealing with problems containing equality axioms. On the other hand, OTTER could not prove some problems (LCM, IMV, GCD, AM8) which are efficiently solved by SHD-prover. The reason is that SHD-prover uses a priority structure which suits these problems.

8.2. Summary

Larry Wos published his book "Automated Reasoning: 33 Basic Research Problems" [Wos 33] in order to promote research. All the topics in this thesis can be considered as an attempt to solve some problems in [Wos 88], although they are mainly concerned with the implementation of a particular theorem prover. As we have stated at the beginning of the thesis, and demonstrated throughout the thesis, that there are many important issues involved in the implementation of a theorem prover and quite a number of them are of general interest. For theorem prover to be successful, both syntactic knowledge and semantic knowledge should be used. It is interesting to note how far the pure syntactic approaches based on resolution has brought us in the field of automatic theorem proving and we think it is important to recognize the importance of using the syntactic knowledge.

Caching. The purpose of caching is to avoid repeated work in depth-first iterative deepening search. We have discussed how caching is implemented with DFID and have shown that the theorem prover using DFID with caching is very competitive in comparison with other theorem provers, in spite of the limitation of the implementation language. People who use DFID to implement theorem provers should consider caching a serious option, especially when the inference rate is slow, making it prohibitively expensive to repeat work. However, when the inference rate is high enough, caching may not help. We regard the work on goal generalization as an attempt to make caching more space-efficient, by deriving and recording the most general solutions possible. This is a technique that can be applied to other back chaining systems.

Subgoal Reordering. Subgoal reordering is made possible by the linear structure of the modified problem reduction format. However, it is usually difficult to perform subgoal reordering properly, due to the lack of knowledge about the problem domain

[Smith&Genesereth 85]. We have investigated and experimented with many different heuristics for performing subgoal reordering. One general heuristic suggests itself from our experiments: We should always favor the subgoal that contains more information (with larger size or more complex structure, e.g.). This is a good heuristic because (1) a subgoal containing more information is more important and should be attempted first, and (2) solving a subgoal containing the most information will often be easier because we can be better guide it and we can detect the failure of solving it earlier. Although the issue of ordering subgoals has been raised before, our work has demonstrated that, in the context of theorem proving, there is a general heuristic for performing the task.

Priority. The power of depth-first iterative deepening search is limited because it can not use priority. It is essential that the search strategy of a theorem prover be able to use priority to control the search. Although depth-first iterative deepening search can utilize the information given by the heuristic functions using iterative-deepening-A* search, it can not use the information given by the priority functions in the more general best-first search. Furthermore, good heuristic functions in A* search are rarely available. We propose two refinements to incorporate priority into depth-first iterative deepening search. The first refinement is the priority system, which can be regarded as a depth-first iterative version of the best-first search. The second refinement is the use of proof complexity measures, based on viewing the process of finding a proof as one of incrementally constructing an instance of the input clauses. These two refinements can be useful in many situations. Our experiments with priority systems suggest that we should favor subgoals that are simple, that is, those with small size or less complex structure.

The experiments with subgoal reordering suggest that we should prefer subgoals containing more information. The experiments with priority systems, on the other hand, suggest that we should prefer simple subgoals. This, however, is not a contradiction. Subgoal reordering considers ordering conjunctive subgoals, that is, subgoals that have to be solved simultaneously; these subgoals belong to one proof. A priority system, on the other hand, considers disjunctive subgoals, that is, subgoals of which only one needs to be solved. Thus, the underlying heuristics suggested by both subgoal reordering and priority system is that we should control the search so that it favors the simplest proof and it uses the information about the proof as early as possible.

Semantics. We developed a semantic version of the modified problem reduction format, the semantic modified problem reduction format. This inference system seems to be the first complete back chaining inference system which allows semantic deletion and is a set of support strategy. Using set of support strategy is important this strategy is still one of the most powerful in automatic theorem proving. An interesting point about this inference system is its treatment for contrapositives. As we have pointed out, the main problem with contrapositives is to identify which one to use. This inference system selects the contrapositives in a semantic way, by using a particular interpretation.

8.3. Future Research

Automatic theorem provers have already been used to solve open problems in mathematics, to help people prove difficult mathematical theorems, and to verify hardware and software designs. I believe that the power of automatic theorem provers will steadily increase, as old techniques are improved, new techniques and technology are developed and applied and new problems are solved. We will present several problems which can be topics for future research and development to increase the power of this theorem prover.

1 There are several ways of encoding problem dependent knowledge in the theorem prover. We can use problem domain knowledge to design priority functions, to perform subgoal reordering, to add contrapositives, to design special proof complexity measures and to design models to delete unachievable subgoals. It would be interesting to adapt the theorem prover to some special class of problems where concrete problem domain knowledge is available. Research on this problem could produce a powerful "special-purpose" theorem prover.

2 The inference rate of the theorem prover could be significantly increased if we used an appropriate data structure to implement caching. One such data structure is the discrimination net [Greenbaum 86]. However, the expected improvement in efficiency is not likely to be gained if the data structure is implemented in Prolog itself, due to the necessary and extensive use of assertions and retractions to implement the data structure [Nie&Plaisted 87b]. Possible alternatives are to implement the theorem prover in some language like C or to use the foreign function interface available in some Prolog systems.

3 Many applications in program and hardware verification require extensive use of induction. Adding an inductive capability to the theorem prover is another topic for future research.

Bibliography

[Ballantyne&Bledsoe 82] Ballantyne, A and W.W. Bledsoe, "On Generating and Using Examples in Proof Discovery", *Machine Intelligence*, Vol. 10, pp. 3-39, Harwood, Chichester 1982.

[Bledsoe 71] Bledsoe, W.W., "Splitting and Reduction Heuristics in Automatic Theorem Proving", *Artificial Intelligence*, Vol. 2, pp. 55-77, North-Holland Publishing Company, 1971.

[Bledsoe&al 72] Bledsoe, W.W., R.S. Boyer and W.H. Henneman, "Computer Proofs of Limit Theorems", *Artificial Intelligence*, Vol. 3, pp. 27-60, North-Holland Publishing Company, 1972.

[Bledsoe 77] Bledsoe, W.W., "Non-resolution Theorem Proving", *Artificial Intelligence*, Vol. 9, pp. 1-35, North-Holland Publishing Company, 1977.

[Bledsoe 82] Bledsoe, W.W., "Using Examples to Generate Instantiations for Set Variables", Technical Report No. ATP-67, Department of Computer Science, University of Texas at Austin, July 1982.

[Bledsoe 83] Bledsoe, W.W., "The UT Interactive Prover", Technical Report No. ATP-17B, Department of Computer Scienc, University of Texas at Austin, Apirl 1983.

[Bose&al 88] Bose, S., E.M. Clarke, D.E. Long and S. Michaylov, "Parthenon: A Parallel Theorem Prover for Non-Horn Clauses", Technical Report No. cmu-cs-88-137, Computer Science Department, Carnegie Mellon University, 1988.

[Boyer 70] Boyer, R., "Locking: A Restriction of Resolution", Ph.D. dissertation, University of Texas at Austin, Austin, TX, 1970.

[Boyer&Moore 79] Boyer, R.S. and J.S. Moore, *A computational logic*, Academic Press, New York, 1979.

[Boyer&Moore 86] Boyer, R.S. and J.S. Moore, "Overview of a theorem-prover for a computational logic", 8th International Conference on Automated Deduction, Oxford, England, July 1986.

[Chang 70] Chang, C.L., "The Unit Proof and the Input Proof in Theorem Proving", *JACM*, Vol. 17, pp. 698-707, 1970.

[Chang&Lee 73] Chang, C and R. Lee, "Symbolic Logic and Mechanical Theorem Proving", Academic Press, New York, 1973.

[Fleisig&al 74] Fleisig, S., D. Loveland, A.K. Smiley and D.L. Yarmush, "An Implementation of the Model Elimination Proof Procedure", *JACM*, Vol. 21, No. 1, pp. 124-139, 1974.

[Gallier 86] Gallier, J.H., "Logic for Computer Science: Foundations of Automatic Theorem Proving", Harper & Row, New York, 1986.

[Gelernter 59] Gelernter, H., "Realization of a Geometry Theorem-Proving Machine", *Proc. ICIP*, pp. 273-282, Paris UNESCO House, 1959.

[Greenbaum 86] Greenbaum, S., "Input Transformation and Resolution Implementation Techniques for Theorem Proving in First-Order Logic", Ph.D. Dissertation, Computer Science Department, University of Illinois at Urbana-Champaign, 1986.

[Harmelen&Bundy 88] Harmelen, F. and A. Bundy, "Explanation-Based Generalization = Partial Evaluation", *Artificial Intelligence* 36, pp. 401-412, 1988.

[Hart&al 68] Hart, P.E., N.J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Trans. on Sys. Sci. and Cybernetics*, July 1968.

[Henschen&al 74] Henschen, L. and L. Wos, "Unit Refutations and Horn Sets", *J. ACM* 21 (1974), pp 590-605.

[Huyn&al 80] Huyn, N., R. Dechter and J. Pearl, "Probabilistic Analysis of the Complexity of A*", *Artificial Intelligence*, Vol. 15, pp. 241-254, 1980.

[Kedar-Cabelli&McCarty 87] Kedar-Cabelli, S.T. and L.T. McCarty, "Explanation-Based Generalization as Resolution Theorem Proving", In P. Lanley, editor, *Proceedings of the 4th International Machine Learning Workshop*, pp. 383-389, Morgan Kaufmann, 1987.

[Kling 71] Kling, R.E., "A Paradigm for Reasoning by Analogy", *Artificial Intelligence*, Vol. 2, pp. 147-178, North-Holland Publishing Company, 1971.

[Korf 85] Korf, R.E., "Depth-first Iterative Deepening: an Optimal Admissible Tree Search", *Artificial Intelligence*, Vol. 27, 97-109, 1985.

[Kowalski 70] Kowalski, R., "Search Strategies for Theorem-Proving", *Machine Intelligence*, Vol. 5, pp. 181-201, 1970.

[Kowalski&Kuekner 71] Kowalski, R. and D. Kuekner, "Linear Resolution with Selection Function", *Artificial Intelligence*, Vol. 2, pp. 227-260, 1971.

[Lloyd 84] Lloyd, J.W., *Foundations of Logic Programming*, New York , NY, Springer-Verlag.

[Loveland 69] Loveland, D.W., "A Simplified Format for the Model Elimination Theorem-Proving Procedure", *J. ACM*, Vol. 16, No. 3, pp. 349-363, July 1969.

[Loveland 78] Loveland, D.W., "Automated Theorem Proving: a Logical Basis", North-holland Publishing Co., 1978.

[Loveland 87] Loveland, D.W., "Near-Horn Prolog", Technical report CS-1987-14, Department of Computer Science, Duke University, April, 1987.

[Loveland 88] Loveland, D., "Near-Horn Prolog and Beyond", Technical Report #CS-1988-25, Computer Science Department, Duke University, Durham, 1988.

[Luckham 70] Luckham, D., "Refinement Theorems in Resolution Theory", *Proc. IRIA Symp. Automatic Demonstration*, Versailles, France, 1968, Springer-Verlag, New York, pp. 163-190, 1970.

[McCharen&al 76] McCharen, J.D., R.A. Overbeek and L.R. Wos, "Problems and Experiments for and with Automated Theorem-Proving Programs", *IEEE Transactions on Computers*, Vol. C-25, No. 8, August 1976.

[McCune 88] McCune, W.W., "Otter 1.0 User's Guide", ANL-88-44, Mathematics and Computer Science Division, Argonne National Laboratory, 1988.

[Mitchell&al 86] Mitchell, T.M., R.M. Keller and S.T. Kedar-Cabelli, "Explanation-Based Generalization: A Unifying View", *Maching Learning*, Vol. 1, No. 1, pp. 47-80, 1986.

[Nevins 74] Nevins, A.J., "A Human Oriented Logic for Automatic Theorem-proving", *J. ACM*, Vol. 21, No. 4, pp. 606-621, Oct. 1974.

[Newell&Simon 72] Newell, A. and H.A. Simon, "Human Problem Solving", Englewood cliffs, NJ, Prentice-hall, 1972.

[Nie&Plaisted 87a] Nie, X. and D.A. Plaisted, "Some Experimental Results on Dynamic Subgoal Reordering", TR-87-027, Dept. of Computer Science, UNC Chapel Hill, September, 1987.

[Nie&Plaisted 87b] Nie, X. and D.A. Plaisted, "Implementation of SPRFN - A Natural Deduction Theorem Prover", TR-87-28, Dept. of Computer Science, UNC Chapel Hill, September 1987.

[Nilsson 80] Nilsson, N.J., "Principles of Artificial Intelligence", Tioga Publishing Company, Palo Atlo, California, 1980.

[Overbeek&al 76] Overbeek, R, J. MeCharen and L. Wos, "Complexity and Related Enhancements for Automated Theorem-Proving Programs", *Comp. & Math. with Appls*, Vol. 2, pp. 1-16, 1976.

[Pearl 83] Pearl, J., "Heuristic Search Theory: Survey of Recent Results", *Proc. of IJCAI*, pp. 554-562, 1983.

[Pearl&Korf 87] Pearl, J., and R.E. Korf, "Search Techniques", *Ann. Rev. Comput.Sci.*, pp. 451-467, Vol 2, 1987.

[Pelletier 86] Pelletier, F.J., "Seventy-five problems for testing automatic theorem provers", *J. of Automated Reasoning*, Vol. 2, pp. 191-216, 1986.

[Plaisted 82] Plaisted, D.A., "A Simplified Problem Reduction Format", *Artificial Intelligence*, Vol. 18, pp. 227-261, 1982.

[Plaisted 84a] Plaisted, D.A., "The Occur-Check Problem in Prolog", *Journal of New Generation Computing* 2, pp. 309-322, 1984.

[Plaisted 84b] Plaisted, D.A., "Using Examples, Case Analysis, and Dependency Graphs in Theorem Proving", 7th International Conference on Automated Deduction, Napa, California, May 1984.

[Plaisted&Greenbaum 84] Plaisted, D.A. and S. Greenbaum, "Problem Representations for Back Chaining and Equality in Resolution Theorem Proving", First Annual AI Applications Conference, Denver, Colorado, December 1984.

[Plaisted 88] Plaisted, D.A., "Non-Horn Clause Logic Programming Without Contrapositives", *Journal of Automated Reasoning*, Vol. 3, No. 4, September 1988.

[Reiter 76] Reiter, R., "A Semantically Guided Deductive System for Automatic Theorem Proving", *IEEE Transaction on Computers*, Vol. C-25, No. 4, pp. 328-334, April 1976.

[Robinson 65] Robinson, J.A., "A Machine-Oriented Logic Based on the Resolution Principle", *Journal of ACM*, Vol. 12, No. 1, January 1965.

[Robinson 79] Robinson, J.A. , "Logic: Form and Function - the Mechanization of Deductive Reasoning", Edinburgh University Press, 1979.

[Sandford 80] Sandford, D.M., "Using Sophisticated Models in Resolution Theorem Proving", Lecture Notes in Computer Science, No. 90, G. Goos and J. Hartmanis eds, Springer-Verlag, 1980.

[Schoppers 83] Schoppers, M.J., "On A* as a Special Case of Ordered Search", *Proc. of IJCAI*, pp. 783-785, 1983.

[Slagle 67] Slagel, J.R., "Automatic Theorem Proving with Renamable and Semantics Resolution", *JACM*, Vol. 14, No. 4, pp. 687-697, October 1967.

[Smith&Genesereth 85] Smith, D.E. and M.R. Genesereth, "Ordering Conjunctive Queries", *Artificial Intelligence*, Vol. 26, pp. 171-215, 1985.

[Smith&Plaisted 88] Smith, M.P. and D.A. Plaisted, "Term-Rewriting Techniques for Logic Programming I: Completion", Technical Report No. TR88-019, Department of Computer Science, University of North Carolina at Chapel Hill, April 1988.

[Stickel&Tyson 85] Stickel, M.E. and M.W. Tyson, "An Analysis of Consecutively Bounded Depth-first Search with Application Automated Deduction", *Proc. of IJCAI*, pp. 1073-1075, 1985.

[Stickel 88] Stickel, M.E., "A PROLOG Technology Theorem Prover", *Journal of Automated Reasoning*, Vol. 4, No. 4, pp. 353-380, 1988.

[VanderBrug&Minker 75] VanderBrug, G.J. and J. Minker, "State-Space, Problem-Reduction, and Theorem Proving — Some Relationships", *CACM*, Vol. 18, No. 2, pp. 107 - 115, 1975.

[Walther 84] Walther, C., "A Mechanical Solution of Schubert's Steamroller by Many-sorted Resolution", *Proc. 8th AAAI*, pp. 30-334, 1984.

[Wang 60] Wang, H., "Toward Mechanical Mathematics", *IBM Journal*, Vol.4, pp. 2-22, 1960.

[Wang 65] Wang, H., "Formalization and Automatic Theorem Proving", *Proceedings of IFIP Congress 65*, Washington, D.C., pp. 51-58, 1965.

[Wang 85] Wang, T.C., "Designing Examples for Semantically Guided Hierarchical Deduction", *Proc. of IJCAI*, pp. 1201-1207, 1985.

[Wang 86] Wang, T.C., "SHD-prover at University of Texas at Austin", 8th International Conference on Automated Deduction, Oxford, England, July 1986.

[Wang&Bledsoe 87] Wang, T.C. and W.W. Bledsoe, "Hierarchical Deduction", *Journal of Automated Reasoning*, Vol. 3, No. 1, 1987.

[Wos 65] Wos, L.R., "Efficiency and Completeness of the Set of Support Strategy in Theorem Proving", *J. of ACM*, Vol. 12, No. 4, October 1965.

[Wos 88] Wos, L.R., *Automated Reasoning: 33 Basic Research Problems*, Prentice Hall, New York, 1988.

[Zhang 88] Zhang, H.T., "Reduction, Superposition & Induction: Automated Reasoning in an Equational Logic", Ph.D thesis, Department of Computer Science, Rensselaer Polytechnic Institute, August, 1988.

Appendix A: Performance Statistics of the Prover

We give the performance statistics of the theorem prover on our test problems. We note that several problems, such as apabhp, wos20, wos22, wos26, wos28, ls37 and ls76t2 are not listed here. Absence of a problem indicates that the theorem prover fails to get a proof for it. There are six tables which give the performance statistics for the theorem prover with various refinements. An entry with — also indicates the prover fail to get a proof for the problem in the entry. All the data are obtained on a SUN3/60 workstation with 12Mb memory. The Prolog system is the ALS Prolog Compiler (Version 0.60) from Applied Logic Systems, Inc. The † marker in the tables indicates that some contrapositives are added in the input clauses for the prover.

Table A.1: Prover with Caching and without Caching				
Theorem	Prover with Caching		Prover without Caching	
	cpu seconds	inferences	cpu seconds	inferences
ances1	2.93	22	1.25	67
burstall	3.62	91	2.27	307
Chang&Lee1	0.55	9	0.10	6
Chang&Lee2	5.47	295	6.17	1986
Chang&Lee3	0.80	27	0.85	214
Chang&Lee4	0.82	28	0.17	23
Chang&Lee5	0.13	4	0.05	5
Chang&Lee6	2.48	134	0.15	10
Chang&Lee7	0.65	13	0.18	7
Chang&Lee8	2.30	54	24.77	5097
Chang&Lee9	2.82	36	0.53	47
dbabhp	6.15	151	0.60	102
dm	0.47	8	0.07	6
ew1	0.48	6	0.30	12
ew2	0.37	4	0.25	12
ew3	1.23	11	2.50	381
example	10.08	238	1317.80	193547
fex4t1	590.18	2858	181.27	4550
fex4t2	126.02	1015	191.27	4040
fex5†	231.78	2970	—	—
fex6t1	23.18	881	271.45	86616
fex6t2	23.58	882	632.03	200043
group1	0.65	10	0.05	6
group2	5.43	295	6.08	1986
hasparts1	1.30	23	0.25	19
hasparts2	3.80	78	0.87	86
ls100	0.20	4	0.03	5
ls103	5.50	99	1325.25	168315
ls105	0.37	5	0.13	11
ls106	0.32	5	0.18	11
ls108	1557.38	8380	—	—
ls111	0.30	5	0.20	9
ls115	10.97	165	101.00	5105
ls116	10.63	131	23.70	1784
ls118†	7611.18	28722	—	—
ls121†	54.55	908	—	—
ls17	3.32	65	2.13	344
ls23	12.25	331	0.90	243
ls26	1.45	63	0.17	9
ls28	46.27	579	5.47	1533
ls29	45.13	564	3.63	985

Table A.1: Prover with and withoug Caching (Cont.)				
Theorem	Prover with Caching		Prover without Caching	
	cpu seconds	inferences	cpu seconds	inferences
ls35	7.92	350	17.20	4892
ls41	1.93	44	0.28	77
ls5	0.52	5	0.22	11
ls55	1.93	31	0.32	62
ls65	164.20	2957	1043.33	297377
ls68	4.92	122	0.95	257
ls75	25.48	548	29.82	6904
ls76t1	5.65	140	1.13	316
mqw	0.63	5	0.17	4
num1	0.62	14	0.17	8
prim	2.18	55	0.77	94
qw	0.75	10	0.20	10
rob1	0.30	2	0.13	16
rob2	5.23	284	5.67	1757
schubert	112.97	1488	3364.18	291645
shortburst	1.00	19	0.18	16
wos1	64.22	1059	3863.72	1350954
wos10	223.05	3950	3006.12	942882
wos11	235.53	4222	183.32	51706
wos12	0.57	27	0.22	39
wos13	8.55	304	0.65	171
wos14	8.32	310	0.37	68
wos15	6045.23	20556	—	—
wos17	37.90	1101	103.35	30297
wos19	62.58	1280	15897.00	5556974
wos2	4.78	159	1025.30	335456
wos23	1.57	46	0.33	72
wos24	15.65	431	2.32	541
wos25	31.83	752	2.67	587
wos27	18.08	473	3.00	758
wos29	45.17	963	586.12	108027
wos3	0.30	9	0.07	10
wos30	0.67	19	0.18	20
wos31†	7742.22	29783	—	—
wos32†	1.57	17	0.25	18
wos33†	7.33	81	6.18	597
wos4	656.17	7239	4389.60	227143
wos5	4.22	140	1.65	465
wos6	13.53	446	51.05	15666
wos7	9.25	375	139.60	42803
wos8	8.57	317	0.60	117
wos9	13.87	519	6.03	1729

theorem	not fail on repeated solution				fail on repeated solution			
	proof depth	running time	number of inference	number of solution	proof depth	running time	number of inference	number of solution
ances1	18	3.03	22	7	18	3.03	22	7
burstall	9	3.77	91	19	9	3.77	91	19
Chang&Lee1	7	0.65	9	2	7	0.63	9	2
Chang&Lee2	9	5.57	295	5	9	5.42	293	5
Chang&Lee3	7	0.82	27	4	7	0.82	27	4
Chang&Lee4	7	0.85	28	4	7	0.87	28	4
Chang&Lee5	5	0.15	4	2	5	0.13	4	2
Chang&Lee6	7	2.53	134	9	7	2.63	134	9
Chang&Lee7	7	0.65	13	6	7	0.67	13	6
Chang&Lee8	11	2.40	54	8	11	2.28	54	8
Chang&Lee9	9	2.83	36	9	9	2.67	35	9
dbabhp	9	6.28	151	45	9	6.40	151	45
dm	7	0.55	8	2	7	0.50	8	2
ew1	7	0.55	6	7	7	0.53	6	7
ew2	7	0.35	4	4	7	0.35	4	4
ew3	11	1.30	11	6	11	1.27	11	6
example	14	10.25	238	37	14	9.50	224	37
fex4t1	18	607.13	2865	245	18	263.23	1067	196
fex4t2	18	130.47	1022	17	18	165.20	873	152
fex5	9	237.80	2970	419	9	224.08	2911	418
fex6t1	18	23.78	881	36	18	23.38	881	36
fex6t2	18	24.10	882	31	18	23.60	882	31
group1	7	0.68	10	2	7	0.63	10	2
group2	9	5.58	295	5	9	5.28	293	5
hasparts1	9	1.23	23	6	9	1.22	23	6
hasparts2	18	3.93	78	12	18	3.90	78	12
ls100	5	0.22	4	3	5	0.20	4	3
ls103	14	5.62	99	11	14	5.52	99	11
ls105	5	0.32	5	4	5	0.33	5	4
ls106	5	0.35	5	4	5	0.37	5	4
ls108	24	1595.25	8380	71	24	454.55	3559	67
ls111	5	0.33	5	4	5	0.32	5	4
ls115	11	11.22	165	13	11	10.75	164	13
ls116	9	10.92	131	39	9	10.53	126	39
ls121	11	55.58	908	57	11	53.37	884	57
ls17	9	3.40	65	8	9	3.48	65	8
ls23	9	12.55	331	29	9	10.85	315	29
ls26	7	1.50	63	6	7	1.43	63	6
ls28	9	47.67	579	131	9	48.00	579	131
ls29	9	46.50	564	130	9	45.78	564	130
ls35	11	8.12	350	6	11	7.98	350	6

Table A.2: Test Result for handling Repeated Solutions

Table A.2: Test Result for handling Repeated Solutions (Cont.)								
theorem	not fail on repeated solution				fail on repeated solution			
	proof depth	running time	number of inference	number of solution	proof depth	running time	number of inference	number of solution
ls41	5	1.97	44	11	5	1.88	43	11
ls5	7	0.43	5	5	7	0.58	5	5
ls55	5	2.03	31	5	5	2.17	31	5
ls65	9	169.40	2957	285	9	153.75	2940	285
ls68	5	5.03	122	16	5	4.52	120	16
ls75	7	26.28	548	51	7	24.97	541	51
ls76t1	5	5.83	140	17	5	5.23	138	17
mqw	7	0.62	5	5	7	0.58	5	5
num1	7	0.70	14	6	7	0.67	14	6
prim	11	2.15	55	8	11	2.07	53	8
qw	9	0.80	10	4	9	0.80	10	4
rob1	7	0.30	2	2	7	0.28	2	2
rob2	9	5.40	284	5	9	5.17	282	5
schubert	32	114.03	1490	74	32	66.08	1124	67
shortburst	7	1.07	19	6	7	1.00	19	6
wos1	9	66.17	1059	145	9	63.23	1059	145
wos10	9	230.40	3950	251	9	221.98	3950	251
wos11	9	242.70	4222	292	9	231.50	4222	292
wos12	5	0.57	27	3	5	0.55	27	3
wos13	7	8.80	304	52	7	8.77	304	52
wos14	7	8.58	310	49	7	8.48	308	49
wos15	11	6321.13	20556	2303	11	6225.37	20535	2303
wos17	7	38.93	1101	123	7	36.00	1099	123
wos19	7	64.42	1280	203	7	62.32	1280	203
wos2	7	4.88	159	8	7	4.70	159	8
wos23	5	1.57	46	2	5	1.42	44	2
wos24	7	16.12	431	60	7	15.80	431	60
wos25	7	38.80	808	167	7	38.33	808	167
wos27	7	21.65	523	92	7	21.02	523	92
wos29	7	69.40	1133	248	7	68.37	1131	248
wos3	5	0.27	9	3	5	0.28	9	3
wos30	5	0.68	19	3	5	0.60	19	3
wos31	14	7455.30	28946	927	14	5553.23	22847	888
wos32	5	1.62	17	4	5	1.62	17	4
wos33	7	7.50	81	10	7	7.23	81	10
wos4	11	677.23	7244	4	11	664.05	7235	4
wos5	7	4.33	140	20	7	4.10	140	20
wos6	7	14.02	446	73	7	13.77	446	73
wos7	7	9.42	375	22	7	9.10	375	22
wos8	7	8.88	317	49	7	8.68	314	49
wos9	7	14.22	519	33	7	13.38	519	33

Table A.3: Test Result for Goal Generalization (GG)								
theorem	original prover				augmented prover			
	proof depth	running time	number of inference	number of solution	proof depth	running time	number of inference	number of solution
ances1	18	2.98	22	7	18	4.20	22	8
burstall	11	4.97	103	15	11	7.98	103	2
Chang&Lee1	9	0.95	14	2	7	0.77	9	2
Chang&Lee2	11	5.87	308	7	11	8.27	309	2
Chang&Lee3	9	1.13	33	4	7	1.02	27	4
Chang&Lee4	9	1.23	35	4	7	1.12	28	4
Chang&Lee5	7	0.33	6	2	7	0.42	6	2
Chang&Lee6	9	2.68	134	9	9	3.53	134	9
Chang&Lee7	9	0.87	13	6	9	1.07	13	6
Chang&Lee8	11	2.30	54	8	11	2.90	55	8
Chang&Lee9	11	3.02	37	8	11	3.77	37	8
dbabhp	11	8.60	163	51	11	13.47	163	51
dm	9	0.73	11	2	7	0.58	8	2
ew1	9	0.70	7	5	9	0.83	7	5
ew2	7	0.38	4	4	7	0.40	4	4
ew3	11	1.25	11	6	11	1.50	11	6
example	18	20.97	613	10	18	22.00	417	4
fex4t1	18	242.22	1033	196	18	383.58	1038	173
fex4t2	18	159.60	853	150	18	222.97	833	118
fex5	11	309.72	2967	297	11	1461.28	3710	318
fex6t1	24	26.93	935	27	18	35.72	881	36
fex6t2	24	25.20	895	24	18	37.88	887	31
group1	9	1.12	18	2	7	0.80	10	2
group2	11	5.90	308	7	11	7.80	309	2
hasparts1	11	1.45	24	6	11	2.05	25	6
hasparts2	24	4.30	81	11	24	5.90	80	10
ls100	7	0.40	6	3	7	0.50	7	3
ls103	18	6.67	115	9	18	10.18	116	5
ls105	7	0.70	11	4	7	0.75	11	4
ls106	7	0.67	11	4	7	0.75	11	4
ls108	24	375.07	3403	67	24	1358.47	3572	129
ls111	7	0.55	9	4	7	0.77	11	4
ls115	11	10.78	164	13	11	19.93	150	13
ls116	9	10.40	126	39	9	19.70	122	38
ls118	11	7611.18	28722	1382	14	37201.96	31371	2360
ls121	11	52.85	884	57	11	83.82	748	48
ls17	9	3.50	69	9	9	4.90	64	9
ls23	11	11.73	318	24	9	15.70	300	24
ls26	9	1.68	63	6	9	2.15	63	6
ls28	11	60.45	610	131	9	61.00	579	133
ls29	11	58.83	602	130	9	55.95	575	132
ls35	14	8.48	358	6	11	9.93	350	6

Table A.3: Test Result for Goal Generalization (Cont.)								
theorem	original prover				augmented prover			
	proof depth	running time	number of inference	number of solution	proof depth	running time	number of inference	number of solution
ls41	7	2.12	45	8	7	2.92	46	5
ls5	7	0.43	5	5	7	0.57	5	5
ls55	7	2.45	37	4	5	2.82	31	5
ls65	11	200.52	2944	281	11	489.05	2927	273
ls68	7	5.12	121	16	5	6.48	114	14
ls75	9	28.17	561	51	9	50.75	545	13
ls76t1	7	5.88	140	17	7	9.62	140	17
mqw	7	0.60	5	5	7	0.68	6	4
num1	9	0.80	14	6	9	1.10	14	6
prim	11	2.03	53	8	11	2.78	53	8
qw	9	0.77	10	4	9	0.85	10	4
rob1	7	0.35	2	2	7	0.35	2	2
rob2	11	5.88	299	4	11	7.90	298	2
schubert	32	65.30	1124	67	32	178.43	1124	68
shortburst	9	1.48	26	5	9	2.15	26	2
wos1	11	78.40	1154	90	9	201.72	1152	145
wos10	11	258.58	3981	233	11	1091.10	4143	115
wos11	11	281.20	4248	288	11	791.20	4336	130
wos12	7	0.73	27	3	7	0.97	27	3
wos13	7	8.43	301	51	7	11.17	301	51
wos14	9	10.87	313	49	9	13.57	313	48
wos15	14	8966.92	20553	2302	11	179.25	1594	102
wos17	9	47.52	1124	120	9	108.45	1124	12
wos19	9	89.05	1301	203	9	204.98	1291	194
wos2	9	5.00	166	8	7	7.00	159	8
wos23	7	1.85	48	2	7	2.77	48	2
wos24	9	19.45	443	60	9	29.23	419	56
wos25	9	57.22	828	167	9	75.77	793	129
wos27	9	28.22	542	94	7	51.97	522	92
wos29	9	106.13	1135	248	9	172.00	1135	235
wos3	7	0.63	14	3	7	0.85	14	2
wos30	7	1.62	40	4	5	1.08	19	3
wos31	18	5111.42	21510	552	18	6832.45	12120	250
wos32	5	1.73	21	7	7	8.13	67	6
wos33	9	7.85	90	7	9	12.53	86	6
wos4	11	653.38	7235	4	11	1988.20	7236	4
wos5	9	5.22	153	20	7	5.50	140	20
wos6	9	18.35	459	73	9	22.52	458	16
wos7	9	10.38	389	22	9	16.35	389	3
wos8	9	11.17	325	49	7	11.35	314	49
wos9	9	14.80	527	33	9	27.42	528	14

Table A.4.1: Subgoal Reordering Using H1

Theorem	No Reordering		Dynamic Reordering		Static Reordering	
	seconds	inferences	seconds	inferences	seconds	infereces
ances1	2.93	22	3.03	22	3.17	22
burstall	3.62	91	3.70	91	3.82	91
dbabhp	6.15	151	6.50	167	6.82	167
dm	0.47	8	0.50	7	0.50	7
ew1	0.48	6	0.52	6	0.62	6
ew2	0.37	4	0.35	4	0.43	4
ew3	1.23	11	1.28	11	1.40	11
fex4t1	590.18	2858	1635.57	3755	1637.50	3755
fex4t2	126.02	1015	126.07	1015	132.00	1015
fex5	231.78	2970	268.83	3710	269.07	3708
fex6t1	23.18	881	24.58	888	25.05	881
fex6t2	23.58	882	23.63	871	25.20	882
group1	0.65	10	0.72	9	0.63	9
group2	5.43	295	5.52	295	5.88	295
hasparts1	1.30	23	1.07	14	1.17	14
hasparts2	3.80	78	2.45	34	2.77	34
ls100	0.20	4	0.18	4	0.18	4
ls103	5.50	99	5.05	90	5.27	90
ls105	0.37	5	0.25	6	0.27	6
ls106	0.32	5	0.27	6	0.28	6
ls108	1557.38	8380	29.37	575	29.25	575
ls111	0.30	5	0.32	7	0.27	7
ls115	10.97	165	11.67	157	11.72	157
ls116	10.63	131	12.32	177	12.57	177
ls121	54.55	908	51.38	835	51.22	840
ls17	3.32	65	3.57	69	3.55	69
ls23	12.25	331	13.30	337	12.45	331
ls26	1.45	63	1.48	63	1.50	63
ls28	46.27	579	46.40	579	46.72	579
ls29	45.13	564	45.93	564	45.35	564
ls35	7.92	350	7.70	330	8.63	354
ls41	1.93	44	1.88	43	1.93	43
ls5	0.52	5	0.47	5	0.50	5
ls55	1.93	31	1.82	21	1.90	21
ls65	164.20	2957	140.72	2703	146.07	2703
ls68	4.92	122	5.07	122	5.32	122
ls75	25.48	548	17.35	410	18.58	410
ls76t1	5.65	140	5.47	134	5.85	134
mqw	0.63	5	0.62	5	0.65	5
num1	0.62	14	0.70	14	0.75	14
prim	2.18	55	1.82	41	1.97	41
qw	0.75	10	0.83	10	0.82	10

Table A.4.1: Subgoal Reordering Using H1 (Cont.)						
Theorem	No Reordering		Dynamic Reordering		Static Reordering	
	seconds	inferences	seconds	inferences	seconds	infereces
rob1	0.30	2	0.30	2	0.37	2
rob2	5.23	284	5.45	284	5.82	284
shortburst	1.00	19	1.02	19	1.13	19
wos1	64.22	1059	47.32	922	53.12	974
wos10	223.05	3950	200.65	3615	238.52	3950
wos11	235.53	4222	224.12	4070	247.23	4166
wos12	0.57	27	0.57	27	0.63	27
wos13	8.55	304	8.33	296	8.82	296
wos14	8.32	310	8.48	308	8.92	308
wos15	6045.23	20556	62.38	1403	66.80	1458
wos17	37.90	1101	33.20	974	32.75	970
wos19	62.58	1280	55.95	1106	55.97	1108
wos2	4.78	159	32.08	881	5.17	159
wos23	1.57	46	1.87	49	2.00	53
wos24	15.65	431	15.57	418	15.80	431
wos25	31.83	752	35.67	799	37.28	786
wos27	18.08	473	18.80	519	20.72	531
wos29	45.17	963	48.43	1004	53.05	1044
wos3	0.30	9	0.25	9	0.37	9
wos30	0.67	19	1.35	40	1.43	40
wos31	7742.22	29783	21079.78	78273	21162.65	78274
wos32	1.57	17	2.10	17	2.35	17
wos33	7.33	81	8.52	79	9.30	79
wos4	656.17	7239	704.83	8247	729.28	8388
wos5	4.22	140	4.00	132	4.57	140
wos6	13.53	446	13.70	444	14.60	446
wos7	9.25	375	8.98	361	9.95	375
wos8	8.57	317	8.58	310	9.45	314
wos9	13.87	519	13.68	508	14.93	519
chang&lee1	0.55	9	0.60	8	0.57	8
chang&lee2	5.47	295	5.57	295	6.08	295
chang&lee3	0.80	27	1.50	50	0.88	27
chang&lee4	0.82	28	0.80	27	0.85	28
chang&lee5	0.13	4	0.18	4	0.15	4
chang&lee6	2.48	134	2.48	134	2.75	134
chang&lee7	0.65	13	0.72	13	0.65	13
chang&lee8	2.30	54	2.33	54	2.52	54
chang&lee9	2.82	36	2.83	36	2.95	36
example	10.08	238	10.18	238	10.65	238
schubert	748.45	8921	649.52	11014	761.42	11014

Table A.4.2: Subgoal Reordering Using H2						
Theorem	No Reordering		Dynamic Reordering		Static Reordering	
	seconds	inferences	seconds	inferences	seconds	infereces
ances1	2.93	22	2.42	13	2.45	13
burstall	3.62	91	3.72	91	3.77	91
dbabhp	6.15	151	40.82	598	41.55	598
dm	0.47	8	0.53	7	0.50	7
ew1	0.48	6	0.48	6	0.52	6
ew2	0.37	4	0.38	4	0.37	4
ew3	1.23	11	1.37	18	1.37	18
fex4t1	590.18	2858	1670.18	3755	1681.22	3755
fex4t2	126.02	1015	126.98	1015	129.27	1015
fex5	231.78	2970	259.57	3644	266.15	3632
fex6t1	23.18	881	24.07	822	23.97	875
fex6t2	23.58	882	21.30	762	23.90	876
group1	0.65	10	0.68	9	0.70	9
group2	5.43	295	5.12	254	5.35	291
hasparts1	1.30	23	1.15	14	1.18	14
hasparts2	3.80	78	2.62	34	2.50	34
ls100	0.20	4	0.17	4	0.17	4
ls103	5.50	99	5.22	89	5.15	89
ls105	0.37	5	0.27	6	0.23	6
ls106	0.32	5	0.28	6	0.25	6
ls108	1557.38	8380	29.43	575	30.37	575
ls111	0.30	5	0.35	7	0.37	7
ls115	10.97	165	11.57	156	11.77	156
ls116	10.63	131	12.45	181	12.68	181
ls121	54.55	908	43.71	724	52.13	805
ls17	3.32	65	3.53	69	3.65	69
ls23	12.25	331	3.60	103	3.75	107
|s26	1.45	63	1.52	57	1.48	63
ls28	46.27	579	46.42	579	47.53	579
ls29	45.13	564	45.22	564	46.45	564
ls35	7.92	350	7.63	326	8.40	354
ls41	1.93	44	2.20	46	2.25	46
ls5	0.52	5	0.48	5	0.55	5
ls55	1.93	31	1.90	21	1.87	21
ls65	164.20	2957	100.37	1786	102.78	1786
ls68	4.92	122	4.08	78	4.23	78
ls75	25.48	548	25.47	437	26.10	437
ls76t1	5.65	140	5.13	85	5.32	85
mqw	0.63	5	0.65	5	0.60	5
num1	0.62	14	0.68	14	0.68	14
prim	2.18	55	2.13	49	2.13	49

Table A.4.2: Subgoal Reordering Using H2 (Cont.)						
Theorem	No Reordering		Dynamic Reordering		Static Reordering	
	seconds	inferences	seconds	inferences	seconds	infereces
qw	0.75	10	0.78	10	0.80	10
rob1	0.30	2	0.30	2	0.32	2
rob2	5.23	284	5.20	254	5.35	285
shortburst	1.00	19	1.07	19	1.02	19
wos1	64.22	1059	681.03	5665	696.50	5680
wos10	223.05	3950	136.68	2282	138.40	2322
wos11	235.53	4222	239.52	3732	245.40	3905
wos12	0.57	27	0.63	29	0.65	29
wos13	8.55	304	7.07	239	7.35	254
wos14	8.32	310	8.23	295	8.60	310
wos15	6045.23	20556	62.08	1339	61.57	1382
wos17	37.90	1101	31.00	834	30.38	846
wos19	62.58	1280	47.93	904	48.83	930
wos2	4.78	159	20.77	525	21.78	542
wos23	1.57	46	1.62	42	1.57	44
wos24	15.65	431	12.42	335	12.80	338
wos25	31.83	752	27.77	601	28.50	633
wos27	18.08	473	14.02	385	14.48	391
wos29	45.17	963	42.32	804	2548.62	10471
wos3	0.30	9	0.28	9	0.32	9
wos30	0.67	19	1.00	29	1.03	29
wos31	7742.22	29783	5816.27	28086	5987.40	28136
wos32	1.57	17	2.00	19	2.03	19
wos33	7.33	81	11.77	165	12.00	165
wos4	656.17	7239	597.80	6302	640.72	6863
wos5	4.22	140	2.87	86	3.02	91
wos6	13.53	446	12.33	349	12.38	377
wos7	9.25	375	6.93	283	7.60	305
wos8	8.57	317	7.38	253	7.92	272
wos9	13.87	519	10.50	364	11.15	389
chang&lee1	0.55	9	0.62	8	0.58	8
chang&lee2	5.47	295	5.00	240	5.10	274
chang&lee3	0.80	27	1.43	44	1.60	54
chang&lee4	0.82	28	0.73	23	0.93	29
chang&lee5	0.13	4	0.12	4	0.13	4
chang&lee6	2.48	134	2.30	120	2.57	134
chang&lee7	0.65	13	0.68	14	0.65	14
chang&lee8	2.30	54	2.32	54	2.43	54
chang&lee9	2.82	36	2.82	36	2.92	36
example	10.08	238	15.87	368	16.30	368
schubert	748.45	8921	2306.73	24208	695.25	12680

Table A.5: Subgoal-Based Priority System				
Theorem	Default Prover		Priority System	
	cpu seconds used	number of inferences	cpu seconds used	number of inferences
ances1	2.93	22	10.07	49
burstall	3.62	91	4.10	48
Chang&Lee1	0.55	9	0.42	6
Chang&Lee2	5.47	295	3.32	125
Chang&Lee3	0.80	27	1.13	27
Chang&Lee4	0.82	28	1.10	28
Chang&Lee5	0.13	4	0.22	4
Chang&Lee6	2.48	134	1.67	54
Chang&Lee7	0.65	13	0.97	13
Chang&Lee8	2.30	54	3.73	60
Chang&Lee9	2.82	36	5.32	40
dbabhp	6.15	151	21.72	695
dm	0.47	8	0.40	6
ew1	0.48	6	0.77	6
ew2	0.37	4	0.53	4
ew3	1.23	11	1.67	11
example	10.08	238	18.53	393
fex4t1	590.18	2858	128.70	1029
fex4t2	126.02	1015	131.93	1415
fex5†	231.78	2970	398.28	7206
fex6t1	23.18	881	972.07	38419
fex6t2	23.58	882	1278.72	54022
group1	0.65	10	0.40	6
group2	5.43	295	3.33	125
hasparts1	1.30	23	2.83	31
hasparts2	3.80	78	3.87	48
ls100	0.20	4	0.33	4
ls103	5.50	99	13.63	197
ls105	0.37	5	0.50	5
ls106	0.32	5	0.60	5
ls108	1557.38	8380	27.87	332
ls111	0.30	5	0.52	5
ls112	—	—	643.23	9775
ls115	10.97	165	12.60	111
ls116	10.63	131	61.48	1190
ls118†	7611.18	28722	1501.58	17358
ls121†	54.55	908	90.22	1048
ls17	3.32	65	6.65	92
ls23	12.25	331	2.58	71
ls26	1.45	63	1.37	38
ls28	46.27	579	9.37	283
ls29	45.13	564	13.53	446
ls35	7.92	350	15.10	485
ls36	—	—	421.30	15130

Table A.5: Subgoal-Based Priority System (Cont.)				
Theorem	Default Prover		Priority System	
	cpu seconds used	number of inferences	cpu seconds used	number of inferences
ls41	1.93	44	3.45	67
ls5	0.52	5	0.75	5
ls55	1.93	31	3.18	53
ls65	164.20	2957	50.20	1360
ls68	4.92	122	5.58	133
ls75	25.48	548	60.20	1728
ls76t1	5.65	140	6.65	148
mqw	0.63	5	1.00	5
num1	0.62	14	0.92	14
prim	2.18	55	2.88	46
qw	0.75	10	0.98	10
rob1	0.30	2	0.38	2
rob2	5.23	284	3.38	122
schubert	112.97	1488	69.47	1154
shortburst	1.00	19	1.45	13
wos1	64.22	1059	8.60	257
wos10	223.05	3950	60.18	1718
wos11	235.53	4222	75.18	2039
wos12	0.57	27	2.32	55
wos13	8.55	304	2.83	89
wos14	8.32	310	5.32	174
wos15	6045.23	20556	50.43	1740
wos17	37.90	1101	47.85	1544
wos19	62.58	1280	5394.00	91791
wos2	4.78	159	5.18	149
wos23	1.57	46	2.70	70
wos24	15.65	431	33.48	922
wos25	31.83	752	58.63	1658
wos27	18.08	473	37.72	1060
wos29	45.17	963	38.37	1116
wos3	0.30	9	0.47	9
wos30	0.67	19	0.87	19
wos31†	7742.22	29783	3334.40	79398
wos32†	1.57	17	2.07	17
wos33†	7.33	81	11.58	143
wos4	656.17	7239	43.17	1637
wos5	4.22	140	4.60	143
wos6	13.53	446	17.27	618
wos7	9.25	375	20.47	750
wos8	8.57	317	20.17	716
wos9	13.87	519	21.62	800

Table A.6: Proof Complexity Measures					
theorem	Default	(CC1,MC1)	(CC1,MC2)	(CC2,MC1)	(CC2,MC2)
ances1	2.93	2.52	2.43	2.85	2.55
burstall	3.62	3.75	3.77	8.33	8.30
Chang&Lee1	0.55	3.78	3.70	4.55	4.77
Chang&Lee2	5.47	8.08	8.00	15.52	15.63
Chang&Lee3	0.80	2.43	2.38	2.15	2.35
Chang&Lee4	0.82	3.42	3.33	2.22	2.32
Chang&Lee5	0.13	0.20	0.22	0.23	0.20
Chang&Lee6	2.48	0.60	0.58	0.53	0.60
Chang&Lee7	0.65	1.78	1.80	2.40	2.33
Chang&Lee8	2.30	2.33	2.32	3.78	3.47
Chang&Lee9	2.82	3.92	3.83	5.30	4.87
dbabhp	6.15	2.58	2.52	4.30	4.60
apabhp	—	—	—	—	—
dm	0.47	1.72	1.70	2.53	2.27
ew1	0.48	0.87	0.87	1.65	1.55
ew2	0.37	0.52	0.55	0.80	0.70
ew3	1.23	1.20	1.18	1.08	1.17
example	10.08	16.35	14.98	44.60	41.92
fex4t1	590.18	—	—	—	—
fex4t2	126.02	—	—	—	—
fex5†	231.78	133.22	134.95	628.70	647.07
fex6t1	23.18	—	—	4559.22	546.37
fex6t2	23.58	—	—	4096.55	621.82
group1	0.65	7.67	7.37	8.55	8.80
group2	5.43	8.18	7.98	14.58	14.85
hasparts1	1.30	1.02	1.00	1.52	1.47
hasparts2	3.80	2.43	2.48	2.97	3.02
ls100	0.20	0.35	0.33	0.63	0.63
ls103	5.50	3.82	3.82	4.02	4.25
ls105	0.37	1.38	1.30	2.18	2.03
ls106	0.32	1.38	1.32	2.27	2.33
ls108	1557.38	51.00	49.97	—	—
ls111	0.30	1.43	1.27	2.05	2.02
ls112	—	226.20	222.07	44.48	42.38
ls115	10.97	12.33	11.92	6.58	7.27
ls116	10.63	36.15	34.65	19.32	20.02
ls118†	—	713.48	719.78	744.02	741.80
ls121†	54.55	121.60	120.98	37.97	37.90
ls17	3.32	5.20	5.15	4.98	5.10
ls23	12.25	255.87	309.60	44.90	46.12
ls26	1.45	0.70	0.57	0.63	0.57
ls28	46.27	74.45	74.70	87.48	91.10
ls29	45.13	92.20	92.83	44.47	46.15

Table A.6: Proof Complexity Measures (Cont.)					
theorem	Default	(CC1,MC1)	(CC1,MC2)	(CC2,MC1)	(CC2,MC2)
ls35	7.92	13.43	13.17	15.35	15.48
ls36	—	2129.52	2097.90	1105.08	1265.17
ls41	1.93	0.35	0.35	0.68	0.75
ls5	0.52	0.65	0.77	0.60	0.68
ls55	1.93	29.90	16.33	8.13	13.32
ls65	164.20	62.10	61.72	25.93	26.13
ls68	4.92	4.05	4.12	1.50	1.60
ls75	25.48	628.53	642.27	415.32	435.07
ls76t1	5.65	0.45	0.42	0.83	0.80
ls87	—	1030.07	1042.35	159.65	162.12
mqw	0.63	0.58	0.62	0.63	0.62
num1	0.62	1.67	1.52	1.85	1.87
prim	2.18	1.93	1.98	2.82	2.97
qw	0.75	0.75	0.63	0.82	0.62
rob1	0.30	0.82	0.98	0.83	0.87
rob2	5.23	7.75	7.77	14.33	14.28
schubert	112.97	220.65	214.38	309.78	313.18
shortburst	1.00	0.72	0.75	1.35	1.45
wos1	64.22	4640.13	2129.23	8.72	8.80
wos10	223.05	44.72	44.20	104.82	104.50
wos11	235.53	727.47	707.70	282.87	284.00
wos12	0.57	0.27	0.30	0.20	0.25
wos13	8.55	0.90	0.88	0.40	0.40
wos14	8.32	12.60	12.67	1.93	1.98
wos15	6045.23	304.23	127.10	111.43	663.48
wos17	37.90	53.83	52.63	77.70	80.22
wos19	62.58	905.92	929.40	—	—
wos2	4.78	10.58	10.43	8.15	8.20
wos23	1.57	22.87	22.63	2.88	2.88
wos24	15.65	90.22	87.08	40.95	40.73
wos25	31.83	1002.62	993.53	42.98	43.05
wos27	18.08	161.95	159.77	114.63	114.48
wos29	45.17	19.35	19.18	29.00	28.88
wos3	0.30	0.28	0.30	0.50	0.52
wos30	0.67	0.38	0.40	0.80	0.82
wos31†	7742.22	1921.17	1886.15	2049.17	2067.68
wos32†	1.57	3.93	3.97	34.00	34.25
wos33†	7.33	14.92	14.87	10.28	10.47
wos4	656.17	54.97	53.47	13.75	13.57
wos5	4.22	44.88	44.92	6.67	6.78
wos6	13.53	38.30	37.85	11.88	11.95
wos7	9.25	154.38	153.60	215.15	218.77
wos8	8.57	25.82	25.50	20.35	20.40
wos9	13.87	42.83	42.00	24.57	24.60

Appendix B: SHD-prover Performance Statistics

We show the performance statistics of the SHD-prover on our test problems. The data are obtained on a SUN3/60 workstation using compiled Kyoto Common Lisp. The letter n in the data-file colume indicates that the input to SHD-prover is a set of clause and all the parameters of SHD-prover have their default values. The letter y in the data-file colume indicates that the input to the SHD-prover is in the form of a data-file. A data-file for a problem is created after we have failed to obtain a proof for the problem using the set of clauses as input. Some parameters of the SHD-prover, such as function nesting limit, will be adjusted in the data-file. No locking is used.

Table B.1: Performance Statistics of SHD-prover				
Theorem	cpu time	accepted-resolvents	useful-resolvents	data-file
ances1	0.367	11	10	n
burstall	3.65	35	13	n
dbabhp	4.3667	15	12	y
ew1	0.217	6	7	n
ew2	0.2	6	6	n
ew3	0.383	9	8	n
Chang&Lee2	129.92	203	11	n
Chang&Lee3	67.35	125	14	y
Chang&Lee4	56.583	80	9	n
Chang&Lee5	44.267	50	8	n
Chang&Lee6	24.4	72	11	y
Chang&Lee8	0.933	12	10	n
Chang&Lee9	0.483	9	9	n
example	23.583	148	25	n
hasparts1	0.667	7	8	n
hasparts2	1.3333	11	12	n
ls103	5.0667	36	12	n
ls105	2.9667	18	6	n
ls106	23.25	93	6	n
ls111	2.8	18	6	n
ls115	131.32	481	8	n
ls17	133.4	192	11	n
ls23	79.15	127	9	n
ls26	22.87	53	11	y
ls28	135.05	521	10	n
ls29	365.53	1347	8	n
ls5	0.183	4	5	n
ls55	2.7	14	5	n
ls65	1464.5	2927	10	n
ls75	103.53	691	8	n
ls76t1	2.6167	16	4	n
mqw	0.417	8	8	n
num1	1.8167	11	7	n
prim	3.6833	28	19	n
rob1	0.683	8	9	y
wos14	6.9667	20	7	n
wos17	22.667	105	10	n
wos19	1097.2	3158	12	n
wos23	18.267	88	6	n
wos24	28.583	134	7	n
wos25	24.783	95	7	n
wos27	90.683	359	7	n
wos31	83.917	268	35	n
wos33	2157.8	3724	29	y
wos4	50.75	202	12	n

Table B.1: Performance Statistics of SHD-prover (Cont.)

Theorem	cpu time	proof	data-file
Chang&Lee1	0.93	proof obtained during forward chaining	n
Chang&Lee7	0.15	proof obtained during forward chaining	n
dm	0.65	proof obtained during forward chaining	n
fex6t1	11.07	proof obtained during forward chaining	n
fex6t2	11.08	proof obtained during forward chaining	n
group1	7.73	proof obtained during forward chaining	n
group2	12.02	proof obtained during forward chaining	n
ls100	0.08	proof obtained during forward chaining	n
ls35	81.55	proof obtained during forward chaining	n
ls41	0.20	proof obtained during forward chaining	n
ls68	1.80	proof obtained during forward chaining	n
rob2	11.65	proof obtained during forward chaining	n
shortburst	0.20	proof obtained during forward chaining	n
wos10	15.65	proof obtained during forward chaining	n
wos12	21.67	proof obtained during forward chaining	n
wos13	23.23	proof obtained during forward chaining	n
wos3	10.67	proof obtained during forward chaining	n
wos30	65.07	proof obtained during forward chaining	n
wos32	1.80	proof obtained during forward chaining	n
fex4t1	proof terminates and fails		y
fex4t2	proof terminates and fails		y
qw	proof terminates and fails		y
apabhp	stopped after 1700 goals generated		y
fex5	stopped after 4400 goals generated		y
ls108	stopped after 3000 goals generated		y
ls121	stopped after 4000 goals generated		y
ls36	stopped after 5500 goals generated		y
schubert	stopped after 4000 goals generated		y
wos1	stopped after 5000 goals generated		y
wos11	stopped after 5000 goals generated		y
wos15	stopped after 4500 goals generated		y
wos2	stopped after 5300 goals generated		y
wos21	stopped after 5000 goals generated		y
wos29	stopped after 4500 goals generated		y
wos5	stopped after 4000 goals generated		y
wos6	stopped after 4000 goals generated		y
wos7	stopped after 5000 goals generated		y
wos8	stopped after 8000 goals generated		y
wos9	stopped after 4000 goals generated		y

Appendix C: Otter Performance Statistics

We give the performance statistics of Otter on our test problems. We use the default weights on all the symbols. All the other systems flags are in their default states unless otherwise indicated in the tables. A set of support clauses will be selected for a problem if Otter fails to get a proof for the problem with all the clauses as support clauses. The fact that a set of support clauses is selected for a problem is indicated by a letter y under the colume SOS. The data are obtained on a SUN3/60 workstation.

Table C.1: Performance Statistics of Otter							
Theorem	hyper resolution forward + bacward subsumption			hyper res. + UR res forward + bacward subsumption			SOS
	run-time	generated	kept	run-time	generated	kept	
ances1	0.12	12	12	0.14	13	13	no
burstall	0.40	48	32	1.16	155	77	no
dbabhp	126.88	2619	2097	198.70	5523	2969	no
apabhp	117.86	2475	1630	13.54	946	356	y
dm	0.16	5	5	0.14	5	5	no
ew1	0.12	6	6	0.08	9	6	no
ew2	0.06	3	3	0.10	4	4	no
ew3	0.12	7	7	0.14	9	7	no
fex4t1	527.92	5936	1419	1482.38	5935	1418	no
fex4t2	522.96	5936	1413	1610.44	5935	1412	no
fex5	991.92	46722	1965	2932.54	61541	6738	no
fex6t1	1011.04	247965	3257	1070.62	295083	1879	y
fex6t2	1019.74	252992	3444	1067.86	295079	1872	y
group1	0.28	18	18	0.24	18	18	no
group2	0.36	88	5	0.48	116	5	no
hasparts1	0.30	14	12	0.38	20	12	no
hasparts2	0.40	18	16	0.40	15	13	no
ls100	0.22	11	9	0.20	13	9	no
ls103	0.84	80	30	0.74	63	26	no
ls105	0.32	18	11	0.36	25	12	no
ls106	0.36	18	11	0.40	25	12	no
ls108	285.02	4375	1616	293.92	4308	1579	y
ls111	0.38	20	11	0.42	29	13	no
ls112	—	—	—	—	—	—	y
ls115	128.28	5259	925	2.40	220	45	y
ls116	22.58	1748	212	3.54	293	64	y
ls118	—	—	—	—	—	—	y
ls121	671.42	8905	2590	284.00	3786	1249	y
ls17	0.34	26	13	0.54	55	24	no
ls23	1.62	327	124	0.58	67	18	no
ls26	0.20	12	8	0.20	7	5	no
ls28	32.68	1978	1215	3.36	429	141	no
ls29	—	—	—	—	—	—	y
ls35	1.62	572	7	2.92	879	17	no
ls36	12.52	1096	681	3.06	568	60	y
ls41	0.34	40	17	0.54	64	23	no
ls5	0.08	15	4	0.16	15	4	no
ls55	7.20	1498	326	2.26	365	99	no
ls65	4.34	946	207	1.52	318	49	y
ls68	1.66	362	98	10.68	1871	343	no
ls75	53.52	5864	1827	13.24	1645	625	y
ls76t1	1.90	383	116	14.54	2350	498	no
ls87	1379.06	34456	11143	19.46	2108	549	y
mqw	0.14	22	9	0.26	22	9	no

Table C.1: Performance Statistics of Otter (Cont.)							
Theorem	hyper resolution forward + bacward subsumption			hyper res. + UR res forward + bacward subsumption			SOS
	run-time	generated	kept	run-time	generated	kept	
num1	0.22	11	8	0.22	20	10	no
prim	0.38	65	29	0.74	115	33	no
qw	0.22	54	9	0.28	54	9	no
rob1	0.10	2	2	0.16	3	2	no
rob2	0.44	88	5	0.54	116	5	no
shortburst	0.14	10	10	0.32	22	17	no
wos1	2.58	477	165	4.48	887	182	y
wos10	0.84	178	6	1.04	267	6	y
wos11	17.24	4601	236	3.68	664	30	y
wos12	0.32	9	3	0.48	23	5	no
wos13	1.04	179	32	0.78	101	10	y
wos14	0.84	169	26	0.46	32	6	y
wos15	48.30	7780	2403	36.98	6272	466	y
wos17	17.90	2201	499	11.40	2114	218	y
wos19	2634.46	19485	6607	116.48	13397	1905	y
wos2	2.64	473	161	0.98	141	14	y
wos21	—	—	—	182.22	27093	2751	y
wos23	2.18	375	30	3.62	539	24	y
wos24	2.08	339	37	3.48	485	14	y
wos25	22.12	5392	169	53.54	10037	190	y
wos27	46.32	8868	1279	1.70	209	12	y
wos29	3.12	405	96	4.98	611	88	y
wos3	0.56	82	5	0.40	33	4	y
wos31	—	—	—	103.19	3943	517	y
wos30	1.02	102	4	1.76	170	14	y
wos32	0.52	50	14	0.66	70	15	no
wos33	265.34	3694	807	185.36	2569	590	no
wos4	6730.30	232096	32641	7818.76	429448	32693	y
wos5	2.58	473	159	0.98	115	22	y
wos6	2.06	433	73	2.06	349	39	y
wos7	1.92	456	44	1.40	281	25	y
wos8	5.72	1294	301	1.56	338	28	y
wos9	6.78	1349	392	3.96	940	67	y
Chang&Lee1	0.14	5	5	0.18	5	5	no
Chang&Lee2	0.40	88	5	0.52	116	5	no
Chang&Lee3	0.62	147	20	0.48	86	13	no
Chang&Lee4	0.72	147	20	0.26	27	7	no
Chang&Lee5	0.14	2	2	0.16	2	2	no
Chang&Lee6	0.16	7	4	0.14	5	4	no
Chang&Lee7	0.20	11	8	0.18	20	10	no
Chang&Lee8	0.48	65	29	0.70	115	33	no
Chang&Lee9	0.32	14	12	0.48	35	21	no
example	47.48	3599	665	145.20	3651	667	no
schubert	10.72	708	402	3.92	659	207	no

Abstract

XUMIN NIE. Automatic Theorem Proving in Problem Reduction Formats. (under the direction of DAVID A. PLAISTED).

This thesis explores several topics concerning the sequent-style inference system - the *modified problem reduction format*. Chapter 1 is the introductory chapter. In Chapter 2, we will present how *caching* is performed with the depth-first iterative deepening search to implement the modified problem reduction format, in order to avoid the repeated work involved in solving a subgoal more than once. In Chapter 3, we present the formalization of *goal generalization* and how it is implemented by augmenting the modified problem reduction format, where goal generalization is a special case of *Explanation-Based Generalization* in maching learning. In Chapter 4, we will present how subgoal reordering is performed in the modified problem reduction format and how it is implemented. In Chapter 5 and Chapter 6, we will present two refinements to the depth-first iterative deepening search strategy in the implementation. The first refinement, the *priority system*, concerns how to incorporate the use of priority of subgoals into the depth-first iterative deepening search. We show that the time complexity of the *priority systems* is within a constant factor of the complexity of the depth-first iterative deepening search. The second refinement is based on a syntactic viewpoint of proof development, which views the process of finding proofs as an incremental process of constructing instances with a certain property. In Chapter 7, we present how semantics, or domain dependent knowledge, can be used in the inference system. In

particular, we will present a semantic variant of the modified problem reduction format which selects its inference rules from any interpretation. This results in an inference system which is a true *set-of-support* strategy and allows *back chaining*. We will also discuss how *contrapositives* are used in the modified problem reduction format and its semantic variant. We will show that only some contrapositives are needed according to some interpretation.

Acknowledgements

I am most indebted to my advisor Professor David Plaisted for suggesting a fascinating dissertation topic, for continuously guiding me during the course of research, for carefully reading and correcting several drafts. His contributions to the dissertation are as much as mine, if not more. His support and motivation were essential in the completion of my graduate study. His academic excellence has been, and will always be, an inspiration to me.

I am thankful to Professor Bharat Jayaraman for his gentle, consistent support and encouragement during my graduate study. I have learned a lot from him, both as a fine teacher and outstanding scholar. I am equally thankful to Professor Donald Stanat, who has generously spent time with me, reading and correcting drafts of my dissertation, literally word by word. I also thank Professor Dean Brock and Professor Jan Prins for serving on my committee.

I would also like to thank my friends who have made my stay at University of North Carolina enjoyable and productive. I also thank the Computer Science Department for providing me the opportunity and good environment to pursue my graduate study. My research has been supported by the National Science Foundation under grants DCR-8516243 and CCR-8802282.

The love, understanding and support of my parents have been boundless. They have allowed their son to go to college far away from home and, even farther away from home, to graduate school.

Table of Contents

List of Tables

List of Figures