INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700  800/521-0600
ACKNOWLEDGEMENTS

I wish to give Professor Myers my heartiest thanks for acting as my dissertation advisor and for the help he has given me in the past 4 years. I enrolled at University of Hawaii at Manoa in 1990. I started taking reading courses from Professor Myers in 1992 and began work on this dissertation in 1993. Professor Myers has been always patient and encouraging. He has assisted me in choosing the research topics, in proofreading and rewriting my drafts many times, and in applying for the RCUH assistantship.

I thank Professor Ross for his suggestions and enlightenment. We met regularly in our logic seminar. He was always kind enough to listen to my premature reports on my research and to give suggestions.

I also thank Professors Tom Craven, Ralph Freese, and Stephen Itoga for serving on my committee.
This dissertation introduces new theorem-proving strategies and uses these strategies to solve a wide variety of difficult problems requiring logical reasoning. It also shows how to use theorem-proving to solve the problem of learning mathematical concepts.

Our first algorithm constructs formulas called Craig interpolants from the refutation proofs generated by contemporary theorem-provers using binary resolution, paramodulation, and factoring. This algorithm can construct the formulas needed to learn concepts expressible in the full first-order logic from examples of the concept. It can also find sentences which distinguish pairs of nonisomorphic finite structures.

We then apply case analysis to solve hard problems such as the zebra problem, the pigeonhole problem, and the stable marriage problem. The case analysis technique we use is the first to be fully compatible with resolution and rewriting and powerful enough to solve these problems.

Our primary new theorem-proving strategies generate subgoals and efficient sets of rules. We show how to divide problems into smaller parts with intermediate goals by reversing logical implications. We solve these subdivided parts by discovering efficient subsets of rules or by generating efficient new rules.

We apply these and other new search strategies to solve difficult problems such as the 15-puzzle, central solitaire, TopSpin, Rubik’s cube, and masterball.
strategies apply universally to all such problems and can solve them quite efficiently: the 15-puzzle, Rubik's cube and masterball can all be done in 300 seconds.

Finally we apply our search strategies to solve real-world problems such as sorting, solving equations and inverting nonsingular matrices.
# TABLE OF CONTENTS

Acknowledgements ................................................................. iii

Abstract ................................................................................. iv

List of Figures ................................................................. x

Chapter 1. Introduction ....................................................... 1
  1.1 Background ................................................................. 1
  1.2 First-order Logic .......................................................... 2
  1.3 The Binary Resolution Principle ...................................... 7
  1.4 Other Inference Rules and Strategies ................................. 10
  1.5 OTTER ................................................................. 13
  1.6 Prolog ................................................................. 14
  1.7 Puzzles ................................................................. 16
  1.8 Overview of this Dissertation ......................................... 17

Chapter 2. The Construction of Craig Interpolants ...................... 19
  2.1 Background and Introduction .......................................... 19
  2.2 Constructing Interpolants from Refutations ....................... 20
  2.3 Examples ................................................................ 34
  2.4 Applications of Craig Interpolants ..................................... 39

vi
Chapter 3. Solving Problems by Case Analysis

3.1 The Zebra Problem ............................................. 44
3.2 Case Analysis .................................................. 48
3.3 Solving the Zebra Problem .................................... 50
3.4 The Stable Marriage Problem ................................. 52
3.5 The Pigeonhole Problem ....................................... 54
3.6 The Instant Insanity Problem ................................. 56

Chapter 4. First-Order Representation ............................... 59

4.1 Introduction .................................................... 59
4.2 A Simple Puzzle Example ...................................... 60
4.3 Reducing Puzzle Solving to Theorem Proving ............... 62

Chapter 5. Search Strategies ......................................... 66

5.1 One-Way Search vs. Two-Way Search .......................... 66
5.2 The Subgoal Strategy .......................................... 67
5.2.1 The Basic Idea ............................................. 67
5.2.2 Goals with Constants ...................................... 71
5.2.4 Goals with Conjunctions .................................. 72
5.2.4 Disjunctive Subgoals ...................................... 72
5.3 Using Multiple Subgoal Solutions ............................. 74
5.4 The Rule Restriction Strategy .................................. 75
5.5 Discovering New Rules ........................................ 78
6.6.1 The Representation .............................................. 117
6.6.2 Solving Masterball .............................................. 119
6.6.3 Using Derived Rules ............................................ 121

Chapter 7. Applications of Search Strategies to Real-World Problems ...... 122
7.1 Sorting................................................................. 122
7.2 Solving Systems of Linear Equations ............................... 123
7.3 Finding Inverses of Matrices ....................................... 127

Appendices
A. The CASE Input File for Solving the Zebra Problem ............... 129
B. A Solution for the Central Solitaire Problem ........................ 134
C. An OTTER Input File for Solving a System of Linear Equations 137

References ........................................................................... 140
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. An Example of Resolution</td>
<td>7</td>
</tr>
<tr>
<td>2. An Example of Resolution with Substitution</td>
<td>9</td>
</tr>
<tr>
<td>3. An Example of Hyperresolution</td>
<td>11</td>
</tr>
<tr>
<td>4. Refutation of Example 1</td>
<td>36</td>
</tr>
<tr>
<td>5. Refutation of Example 2</td>
<td>38</td>
</tr>
<tr>
<td>6. Two Nonisomorphic Structures</td>
<td>40</td>
</tr>
<tr>
<td>7. An Example of the Insanity Problem</td>
<td>56</td>
</tr>
<tr>
<td>8. Comparison of Search Spaces</td>
<td>67</td>
</tr>
<tr>
<td>9. A Configuration of Central Solitaire</td>
<td>88</td>
</tr>
<tr>
<td>10. A Coloring of the Central Solitaire Board</td>
<td>90</td>
</tr>
<tr>
<td>11. A Configuration of Triangular Solitaire</td>
<td>95</td>
</tr>
<tr>
<td>12. A Coloring of the Triangular Solitaire Board</td>
<td>96</td>
</tr>
<tr>
<td>13. A Configuration of TopSpin</td>
<td>99</td>
</tr>
<tr>
<td>15. A Configuration of Masterball</td>
<td>116</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

1.1 Background

The focus of this dissertation is on problems in automated reasoning and machine learning. Here *reasoning* means logical reasoning, the process of drawing conclusions from available facts by logical rules, rather than common sense reasoning. Logical reasoning is the basis of many artificial intelligence (AI) endeavors such as automated theorem proving, machine learning, expert systems and advanced robotics.

Since the construction of the first modern computer, computers have been used to solve computationally difficult problems in a variety of fields. In mathematics for example, they played a key role in solving the famous four-color problem [AH]. Only recently, though, have they had success with problems requiring reasoning or learning. Theorem-proving programs such as AURA, its successor OTTER and others have solved several less well-known open problems in mathematics and formal logic. They have found proofs, generated models and found counterexamples (see [Qu], [Sl], [Wa], [WW]). But these programs often fail on reasoning problems which are solved by intelligent teenagers. Most notably, they fail to solve puzzles ranging from the modestly difficult 15-puzzle to the very difficult central solitaire and Rubik's cube.
Here we introduce new reasoning strategies which have been able to solve all these puzzles and several related programming problems. Besides solving individual problems (e.g., unscrembling a particular Rubik's cube), these strategies also produce programs which efficiently solve all instances of the problem (e.g., all scrambled cubes). The performance of these artificially generated programs is within an order-of-magnitude of that of manually written programs.

In this chapter we define the basic concepts which will be used in later chapters. We also briefly describe the theorem provers OTTER and Prolog which we use to generate logical references.

In Chapter 2 we introduce an algorithm for constructing Craig interpolants from refutation proofs generated by resolution-based theorem provers. We then show how these interpolants can be used to solve the difficult problem of learning mathematical concepts from examples. Also given a pair of nonisomorphic finite structures, we show how to use the algorithm to find sentences which distinguish the pair.

In Chapters 3, 4, and 5 we present several automated reasoning strategies. We then use these strategies to solve the puzzles mentioned above such as the 15 puzzle, central solitaire and Rubik's cube in Chapter 6. None of these puzzles have been solved by previous general-purpose reasoning programs.

1.2 First-order Logic

In automated theorem proving one states axioms and theorems in some formal logic and searches for proofs using some set of rules of inference. The logic we use
will be first-order logic with sentences translated into clause form. Our primary rule of inference will be resolution.

To formalize a language, we need the following logical symbols:

punctuation: ",", "("", ")";
variables: $v_1, v_2, v_3, \ldots$;
connectives: $\land, \lor, \neg, \rightarrow$ (and, or, not, implies);
quantifiers: $\exists, \forall$ (there exists, for all);
equality: $=$.

All other symbols are nonlogical symbols. Nonlogical symbols are either relation symbols, operation symbols or constant symbols.

**Definition.** A language $\mathcal{L}$ is a set of nonlogical symbols. Each relation or operation symbol is assigned an *arity*, i.e., its number of places.

For example, $PA = \{\leq, +, *, 0, 1\}$ is the language for number theory (or Peano arithmetic) [CK], where $\leq$ is a two-place relational symbol; $+, *$ are two-place operation symbols; and $0, 1$ are constant symbols.

The *terms* of $\mathcal{L}$ are the strings of symbols generated by the following rules:

(i). A variable is a term;
(ii). A constant symbol is a term;
(iii). If $f$ is a $n$-place operation symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

The *atomic formulas* of $\mathcal{L}$ are the strings of symbols generated by the following
rules:

(i) If $s$, $t$ are terms of $\mathcal{L}$, then $s = t$ is an atomic formula;

(ii) If $P$ is an $n$-placed relation symbol and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an atomic formula.

The formulas of $\mathcal{L}$ are the strings of symbols generated by the following rules:

(i). An atomic formula is a formula;

(ii). If $\phi$, $\psi$ are formulas, then $(\phi \land \psi)$, $(\phi \lor \psi)$, $(\phi \rightarrow \psi)$, and $(\neg \phi)$ are formulas;

(iii). If $v$ is a variable and $\phi$ is a formula, then $(\exists v)\phi$ and $(\forall v)\phi$ are formulas.

An occurrence of a variable in a formula is bound if and only if the occurrence is within the scope of a quantifier for that variable. An occurrence of a variable is free if it is not bound.

A literal is an atomic formula (a positive literal), or the negation of an atomic formula (a negative literal). Two literals are complementary if and only if one is the negation of the other. A clause is a disjunction of literals. A one-literal clause is a unit clause. A Horn clause is a clause that contains at most one positive literal. The clause containing no literals is the empty clause and is denoted by $\diamond$. Since the empty clause has no literals that can be satisfied in an interpretation, it is always false. Thus $\diamond$ may also be used to denote a contradiction.

A formula is in conjunctive normal form if and only if it is a conjunction of clauses.

A formula is in prenex form if and only if it is of the form $(Q_1x_1)\ldots(Q_nx_n)(M)$
where every \( Q_i \) is either \( \forall \) or \( \exists \), and \( M \) is a formula containing no quantifiers. In this case \((Q_1x_1)...(Q_nx_n)\) is the quantifier prefix and \( M \) is matrix.

A formula is in *Skolem form* if and only if it is in prenex form and has no \( \exists \).

Two formulas or a formula and a set of formulas are *equiconsistent* if and only if the consistency of one is equivalent to the consistency of the other.

**Skolem Form Theorem** (Skolem). *Every first-order formula is equiconsistent with a formula in Skolem form.*

**Proof.** First convert the formula to prenex form. Then working left-to-right, successively remove each existential quantifier and replace each occurrence of its variable with the term consisting of a new Skolem function applied to the previous universally quantified variables. \( \Box \)

For example, \( \forall x \exists y \forall z \phi(x, y, z) \) is equiconsistent with \( \forall x \forall z \phi(x, f(x), z) \). The Skolem function \( f(x) \) which replaces the variable \( y \) depends on the earlier universally quantified variable \( x \).

**Clause Form Theorem** ([DP]). *Every formula is equiconsistent with a set of clauses.*

For example, suppose a formula has a conjunctive normal form matrix:

\[
\forall x \exists y \forall z \exists w [R(x, y, z) \land \neg R(y, z, w)].
\]

We first convert it to Skolem form:

\[
\forall x \forall z [R(x, f(x), z) \land \neg R(f(x), z, g(z, x))].
\]
And finally convert it to a set of clauses:

\[ \{ R(x, f(x), z), \neg R(f(x), z, g(x, z)) \}. \]

The free variables of a clause are understood to be universally quantified. Thus, the problem \( \Sigma \vdash \phi \) of proving a general first order formula \( \phi \) from a set \( \Sigma \) of axioms can be reduced to proving \( \Sigma \cup \{ \neg \phi \} \) is inconsistent which, by the above theorem, can be reduced to the problem of proving a finite set of clauses is inconsistent.

**Definition.** A sentence is a formula without free variables. A theory of \( \mathcal{L} \) is a collection of sentences (called axioms) of \( \mathcal{L} \).

A structure for \( \mathcal{L} \) is a pair \( \mathfrak{A} = (A, \mathcal{I}) \) where \( A \) is the universe and \( \mathcal{I} \) is an interpretation function that maps the symbols of \( \mathcal{L} \) to constants in \( A \), or relations or operations on \( A \) of the appropriate number of places. \( \mathfrak{A} \) is a model of theory \( T \) if every sentence of \( T \) is true in \( \mathfrak{A} \). In this case, we write \( \mathfrak{A} \models T \).

A theory \( T \) is satisfiable if and only if it has a model; \( T \) is unsatisfiable if and only if it does not.

Let \( \mathfrak{A} \) be a structure for \( \mathcal{L} \). Let \( \mathcal{L}_A = \mathcal{L} \cup \{ c_a : a \in A \} \) be the result of adding a new constant symbol \( c_a \) for each element \( a \in A \). Let \( \mathfrak{A}_A = (\mathfrak{A}, a)_{a \in A} \) be the expansion of \( \mathfrak{A} \) to \( \mathcal{L}_A \) in which each new constant \( c_a \) is interpreted as the element \( a \).

The diagram of \( \mathfrak{A} \), denoted by \( \Delta_{\mathfrak{A}} \), is the set of all atomic sentences and negations of atomic sentences of \( \mathcal{L}_A \) which hold in the model \( \mathfrak{A}_A \).
1.3 The Binary Resolution Principle

The resolution principle is due to J. A. Robinson [Ro]. It is essentially an extension of the one-literal rule of Davis and Putman (i.e., \( \{L, \neg L \lor Q\} \vdash Q \)). In the propositional case, the binary resolution principle is:

For any two clauses \( C_1 \) and \( C_2 \), if there is a literal \( L \) in \( C_1 \) that is complementary to a literal \( \bar{L} \) in \( C_2 \), then one may infer the resolvent obtained by deleting \( L \) and \( \bar{L} \) from \( C_1 \) and \( C_2 \) and taking the disjunction of the remaining clauses.

For example, \( \{L \lor C \lor D, \neg L \lor E \lor F\} \vdash C \lor D \lor E \lor F \).

Given a set \( S \) of clauses, a resolution deduction of \( C \) from \( S \) is a finite tree or acyclic graph labeled with clauses so that \( C \) is at the root, the leaf clauses are input clauses from \( S \) and the nonleaf clauses are resolvents of their parents. A deduction of \( \diamond \) from \( S \) is called a refutation of \( S \).

The following is a deduction of \( \{P \lor Q, \neg P \lor Q, P \lor \neg Q, \neg P \lor \neg Q\} \vdash \diamond \)

![Figure 1. An Example of Resolution.](image)

In order to introduce the resolution principle for the full first-order logic, we need some more definitions.
**Definition.** A *substitution* is a function from the set of variables to the set of terms. Let \( \{ t_1/v_1, \ldots, t_n/v_n \} \) be the substitution \( \pi \) such that \( \pi(v_i) = t_i \) and \( \pi(x) = x \) for \( x \notin \{ v_1, \ldots, v_n \} \).

Let \( \pi \) be a substitution and \( \phi \) be a formula. Then \( \phi\pi \) is the formula obtained from \( \phi \) by replacing simultaneously each variable \( v \) by \( \pi(v) \). We call \( \phi\pi \) an *instance* of \( \phi \).

A substitution \( \pi \) is a *unifier* for a set of formulas \( \{ \phi_1, \ldots, \phi_n \} \) if and only if \( \phi_1\pi = \cdots = \phi_n\pi \). \( \pi \) is called the *most general unifier* (mgu) if for any other unifier \( \theta \) of \( \{ \phi_1, \ldots, \phi_n \} \), \( \phi_1\theta \) is an instance of \( \phi_1\pi \).

A *first-order resolvent* of clauses \( C_1 \) and \( C_2 \) is a resolvent of \( C_1\pi' \) and \( C_2\pi \) for some substitution \( \pi \). We call \( C_1 \) and \( C_2 \) the *parent clauses* of the resolution.

**Factoring:** Let \( C \) be a clause. Suppose there is a pair of literals \( L, L' \) in \( C \) such that \( L\theta = L'\theta \) for a unifier \( \theta \). Then applying \( \theta \) to \( C \) and removing the duplicate \( L'\theta \) from \( C\theta \) produces a *factor* of \( C \).

For example, \( P(a) \lor P(x) \lor P(f(b)) \) can be factored to \( P(a) \lor P(f(b)) \) by either \( \theta = f(b)/x \) or \( \theta = a/x \).

In the propositional case, factoring is simply deleting duplicate literals.

For example, if \( \phi = R(x, f(a)) \lor P(g(x)), \psi = \neg R(g(b), y) \lor Q(y), \) and \( \pi = \{ g(b)/x, f(a)/y \} \), then by the resolution principle, we have

\[
\{ \phi\pi, \psi\pi \} \vdash P(g(g(b))) \lor Q(f(a)).
\]
Figure 2. An Example of Resolution with Substitution.

*Paramodulation:* Given clauses $C(r)$ and $s = t \lor D$ with no variables in common and a unifier $\pi$ such that $r\pi = s\pi$ or $r\pi = t\pi$, paramodulation infers the paramodulant $(C(t) \lor D)\pi$ or $(C(s) \lor D)\pi$ respectively.

Let $\Sigma \vdash C$ mean that clause $C$ can be deduced from the set of clauses $\Sigma$ by resolution, factoring, and paramodulation.

**Completeness Theorem for the Resolution Principle** [RW]. Resolution, factoring, and paramodulation are *refutation complete*. That is, a set $S$ of clauses is unsatisfiable if and only if there is a deduction of the empty clause from $S$. They remain complete even if all substitutions are required to be most general unifiers.

As a corollary we have $\Sigma \vdash \phi$ if and only if $(\Sigma \cup \neg\phi)' \vdash \Diamond$ where $(\Sigma \cup \neg\phi)'$ is the set of clauses equiconsistent with $\Sigma \cup \neg\phi$. 

9
1.4 Other Inference Rules and Strategies

Automated reasoning programs are expected to find proofs by themselves with minimal assistance from the user. However, even for simple problems, the set of clauses generated while searching for a proof can be unmanageably large. Hence it is almost always necessary to employ various strategies to restrict the search or to eliminate redundant clauses. We review two such strategies: subsumption which eliminates redundancy and set-of-support which places a restriction on which pairs of clauses may be resolved. We also review three restricted varieties of resolution: unit resolution, input resolution, and hyperresolution.

A clause \( C \) subsumes a clause \( D \) if there is a substitution \( \theta \) such that every literal in \( C\theta \) appears in \( D \) ([Ro], [Lo]). For example, \( P(x) \) subsumes \( P(a) \lor Q(b) \), and \( R(a,x) \lor R(y,b) \) subsumes \( R(a,b) \).

The subsumption procedure discards clauses which are subsumed by previously generated clauses (forward subsumption) or clauses which are subsumed by subsequently generated clauses (backward subsumption).

If binary resolution is restricted so that at least one of the parent clauses is a unit (a clause with a single literal), the resulting inference rule is called unit resolution. If binary resolutions are restricted so that at least one of the parent clauses is an input clause (a clause from the set of axioms), the resulting inference rule is called input resolution. Unit resolution and input resolution are equivalent in the sense that there is a unit refutation from a set \( S \) of clauses if and only if there is an
input refutation from $S$. They both are refutation complete when $S$ is a set of Horn clauses [CL]. For Horn clauses, these highly selective restrictions of binary resolution are often effective in reducing the combinatorial explosion of generated clauses.

**Definition.** A clause is *positive* if it contains only positive literals; it is *negative* if it contains only negative literals. A clause is *mixed* if it contains both positive and negative literals.

*Hyperresolution:* Let $C_1, ..., C_n$ be a set of $n$ ($\geq 1$), not necessarily distinct, positive clauses, and let $D$ be a negative or mixed clause containing exactly $n$ negative literals. Suppose there is a substitution $\pi$ such that each negative literal in $D\pi$ is complementary to a literal in one of $C_i\pi$. Hyper-resolution then infers the hyperresolvent produced by removing the complementary literals from each of the substituted clauses and then forming the disjunction of the remaining literals.

For example, the following figure shows a propositional hyperresolution from positive clauses $\{A \lor B, C \lor D, E\}$ against the mixed clause $\neg A \lor \neg D \lor \neg E \lor G$ with a positive resolvent $B \lor C \lor G$. 
Hyperresolution always produces a positive resolvent. So it is also called positive hyperresolution. Similarly, we can define negative hyperresolution as using a set of negative clauses to resolve against a mixed clause and to produce a negative resolvent.

Hyperresolution was introduced by J. R. Robinson. Loveland proved that hyperresolution is refutation complete. Note that hyperresolution is a combined inference rule that accomplishes several binary resolutions in one step. In hyperresolution and negative hyperresolution, mixed clauses are never generated and the number of clauses generated is significantly reduced. For Horn clauses, positive hyperresolution amounts to forward chaining from positive clauses and negative hyperresolution amounts to backward chaining from negative clauses. Pure binary resolution does both. We will use both binary resolution and hyperresolution.

Another goal-directed strategy is the set-of-support strategy introduced by Wos et al. in 1965 [WCR]. A set of clauses is designated as the set-of-support, and all resolvents are required to have at least one ancestor from the set of support.
Like hyperresolution, resolution with the set-of-support strategy restricts the set of clauses generated. Binary resolution with the set-of-support strategy is refutation complete, provided the complement of the set-of-support is satisfiable [CL].

1.5 OTTER

OTTER is a general-purpose automated reasoning program written by W.W. McCune at Argonne. OTTER features the inference rules of binary resolution, hyperresolution, ur-resolution, factoring and binary paramodulation. It offers subsumption and the set-of-support strategy to restrict the application of inference rules, and weighting to direct their applications. It works with clauses of first-order logic with equality [Mc1].

We will use OTTER as the basic logical inference engine for many of our problems. OTTER is not completely automatic. It allows the user to choose the inference rules, the set-of-support set, and a large variety of options to control processing. Discovering a proof with OTTER often depends on finding the right combination of options.

In this dissertation we include some examples of input and output files for OTTER. In these files we use the notation used in OTTER to represent clauses: every clause ends with a period, the logical connective \( \neg \) is represented by \( - \), and the logical connective \( \lor \) is represented by \( | \). Since OTTER treats any string starting with \( u, v, w, x, y, \) or \( z \) as a variable, we add the character 'a' in front of our constants which would otherwise start with these characters.
1.6 Prolog

The inspiration for Prolog was Robinson's resolution principle. In Prolog, a computational problem is represented by rules and facts ([CM], [SS]). These rules and facts are more closely related to the clauses of OTTER than to the commands of procedural languages such as C.

Our Prolog programs (run with Turbo Prolog) will consist of 4 parts: domains, predicates, clauses, and goal.

(i). The Domain section defines the types of arguments, including built-in types such as integer, real, symbol, string and user-defined types.

(ii). The Predicate section defines the relations and their argument types. For example, like(name, obj) defines a predicate like with two arguments, the first argument has type name, the second argument has type obj.

(iii). The Clause section enumerates the facts and the rules. For example, the clause like(John, computers) represents the fact that John likes computers. The rule like(x, computers) :- professor(x) states that all professors like computers. This is the Prolog way of writing

\[ \forall x \text{ professor}(x) \Rightarrow \text{like}(x, \text{computers}). \]

(iv). The Goal section states what the program is going to prove.

For example, if A :- B, C and B :- E, F are the first two rules in a Prolog program with A as the goal, then Prolog will try to prove A by establishing B and C. To
prove $B$ it will try using the rule $B:-E,F$. Logically this corresponds to replacing $B$ in the first clause by $E, F$ from the second clause. This gives a new clause $A:-E,F,C$. In terms of resolution, this corresponds to the derivation:

$$\{A \lor \neg B \lor \neg C, \quad B \lor \neg E \lor \neg F\} \vdash A \lor \neg E \lor \neg F \lor \neg C.$$ 

Prolog can be identified as a resolution prover for Horn clauses using ordered-linear resolution and depth-first search. It starts with the goal and works backward using a depth-first search to construct a derivation of the goal from the given facts and rules.

Prolog’s depth-first back-chaining search is very efficient with space compared to OTTER. For our problems, it is also faster than OTTER with a comparable unidirectional strategy such as hyperresolution. However on some problems, OTTER with the bidirectional binary resolution rule is faster than the unidirectional Prolog.

In order to improve efficiency and to control the direction of the search for a proof, Prolog interpreters include various ‘impure’ (non-logical) features. Most Prolog implementations omit the occurs check when performing unifications and unless care is exercised, non-theorems may be ‘proved’ by the Prolog interpreter. The ‘connective’ not (negation by failure) in Prolog is distinct from \(\neg\) in standard logic. The built-in predicates ! (cut) and fail control the search pattern. Prolog and OTTER also include many numerical functions for arithmetical calculations.
1.7 Puzzles

Most of us have puzzles of various types. Rubik’s cube and its spherical variant masterball, central solitaire, TopSpin, and the 15-puzzle are examples of challenging popular puzzles. Many of these puzzles have two or three dimensional boards with pieces inside the boards. A typical problem for these puzzles starts with a scrambled board. The goal is to find a sequence of moves which unscrambles the board or to prove that it can not be unscrambled.

The 15-puzzle is a simple example. It consists of fifteen tiles numbered from 1 to 15 which rest in a 4 by 4 tray. One position, the hole, is not occupied by a tile. The tiles can move up, down, left or right provided an adjacent hole allows the move. Given a tray with the numbered tiles in a scrambled arrangement, the goal is to find a sequence of moves which arrange the tiles in increasing numerical order with the hole in the last position. For example, from the first board below, we wish to get to the second:

```
15 14 13 12
11 10  9  8
  7  6  5  4
  3  2  1
```

```
  1  2  3  4
  5  6  7  8
  9 10 11 12
13 14 15
```

Another group of interesting problems require making a set of choices which satisfies some given restrictions. The zebra problem and the stable marriage problem are examples in this group. Related problems include scheduling and task assignment.

The puzzles mentioned above are entertaining and challenging. Finding a solu-
tion with a computer generally involves searching a very large space of potential solutions. There are $16!/2 \approx 10^{13}$ accessible configurations for the 15-puzzle. For the TopSpin puzzle, there are $20! \approx 2.4 \times 10^{18}$ accessible configurations. These search spaces are too large for brute-force methods to be feasible. Hence solving these puzzles requires some degree of intelligence.

For many of these puzzles there are computer programs which appear to solve them. Actually the program is implementing a solution which has been found by the program's author. Here we are interested in the problem of finding a solution, not in the relatively routine problem of implementing a known solution. Moreover, we are interested in finding solutions using general-purpose strategies which apply without modification to a wide variety of problems.

But why are we interested in puzzles? First puzzles are often abstractions of real-world problems. For example, the zebra problem is analogous to many scheduling problems. General purpose reasoning strategies which can solve these puzzles can also be expected to solve analogous real-world problems. Secondly, since puzzle-solving is an easy-to-formalize form of intelligent activity, puzzles are convenient examples for testing the capability and efficiency of reasoning strategies.

1.8 Overview of this Dissertation

This dissertation is divided into 7 chapters and three appendices. The appendices contain some detailed examples, and input and output files of some solved problems.

In Chapter 2 we give a mechanical method for constructing Craig interpolants
from refutation proofs and show how to find sentences which separate pairs of nonisomorphic finite structures.

In Chapter 3 we solve a group of puzzles using case analysis. The problems studied there include the zebra problem, the stable marriage problem, the pigeon-hole problem, and the instant insanity problem.

In Chapter 4 we give the first-order representation for a sample board puzzle.

In Chapter 5 we introduce efficient strategies for solving board puzzles.

In Chapter 6 we solve puzzles such as the 15-puzzle, Rubik’s cube, masterball, TopSpin, central solitaire, triangular solitaire, and we answer some open problems using our strategies.

In Chapter 7 we show how to apply these strategies to solve other typical real-world problems in mathematics and computer science.
CHAPTER 2
THE CONSTRUCTION OF
CRAIG INTERPOLANTS

2.1 Background and Introduction

Let \( \Sigma \) and \( \Pi \) be two inconsistent first-order theories. Then by Craig's Interpolation Theorem, there is a sentence \( \theta \), called a Craig interpolant, such that \( \theta \) is true in \( \Sigma \) and false in \( \Pi \) and every nonlogical symbol occurring in \( \theta \) occurs in both \( \Sigma \) and \( \Pi \). We say \( \theta \) separates \( \Sigma \) and \( \Pi \). Craig interpolants can be used to solve the problem of learning a first-order concept by letting \( \Sigma \) and \( \Pi \) be the lists of positive and negative examples of the concept to be learned.

The standard nonconstructive model-theoretic proof of Craig's Interpolation Theorem is in [CK]. Lyndon showed how to construct an interpolant from a special form of natural deduction (see [Ly]). We show how to construct an interpolant from a refutation proof which uses binary resolution, factoring and paramodulation. In our examples, we use OTTER (the standard text on OTTER is [WOLB]) to generate such proofs.

Craig interpolants can be used to find a sentence which distinguishes two nonisomorphic finite structures. Let \( \Sigma \) and \( \Pi \) be the atomic diagrams of the two structures. Then they are inconsistent and any Craig interpolant for them is a sentence which is true in one structure and false in the other.
2.2 Constructing Interpolation Formulas from Refutations

Let $L_\Sigma$ and $L_\Pi$ be two languages, $\Sigma$ a theory in $L_\Sigma$, and $\Pi$ a theory in $L_\Pi$ such that $\Sigma \cup \Pi$ is not consistent. In this paper we use $\Diamond$ to represent contradiction, use $\Box$ to indicate the end of a proof, and suppose $P$ is a refutation of $\Sigma \cup \Pi \models \Diamond$ involving only binary resolutions, paramodulations, and factorings. The input clauses (clauses at the top of the refutation) are required to be instances of clauses from $\Sigma$ and $\Pi$. For convenience, we will assume that different input clauses have disjoint sets of variables.

For any occurrence $L$ in the proof $P$ of a relational symbol in $L_\Sigma \cup L_\Pi$, we define $L$ is from $\Sigma$ recursively by:

(i). If the occurrence $L$ is in an input clause from $\Sigma$, we say it is from $\Sigma$; otherwise, it is not.

(ii). If the occurrence $L$ is in a non-input clause $C$, then it is from $\Sigma$ if the corresponding occurrence in some parent clause is from $\Sigma$.

Similarly, we can define $L$ is from $\Pi$. Since factoring is allowed in the proof, several occurrences of some literal may be factored into a single one. So it is possible that a literal in some clause may be from both $\Sigma$ and $\Pi$.

Let $T$ and $F$ be the truth values of 'truth' and 'falsehood'. For a binary resolution proof $P$ we use the following recursive procedure to assign formulas to the clauses in $P$. 

20
Interpolation Algorithm

(i). If \( C \) is an input clause from \( \Sigma \), its formula is \( F \); if \( C \) is an input clause from \( \Pi \), its formula is \( T \).

(ii). If \( \phi \) is assigned to \( L \vee C \) and \( \psi \) is assigned to \( \neg L' \vee D \), and if \( (C \vee D)\pi \) is the resolvent of \( L \vee C \) and \( D \vee \neg L' \) resolving against \( L\pi(= L'\pi) \), then the formula assigned to \( (C \vee D)\pi \) is:

(a). \( (\phi \vee \psi)\pi \) if the occurrences of both \( L \) and \( \neg L' \) are from \( \Sigma \) alone;

(b). \( (\phi \wedge \psi)\pi \) if the occurrences of both \( L \) and \( \neg L' \) are from \( \Pi \) alone;

(c). \( ((\neg L' \wedge \phi) \vee (L \wedge \psi))\pi \) if neither (a) nor (b).

Definition 2.1. A formula \( \theta \) is a relational interpolant separating \( \Sigma \) and \( \Pi \) relative to a clause \( C \) if and only if

1. all relational symbols of \( \theta \) are in \( L_\Sigma \cap L_\Pi \),
2. \( \Sigma \models \theta \vee C \), and
3. \( \Pi \models \neg \theta \vee C \).

Theorem 2.2. For each clause \( C \) of a binary resolution proof \( P \) of \( \Sigma \cup \Pi \models \Diamond \), the formula assigned by the above algorithm is a relational interpolant of \( \Sigma \) and \( \Pi \) relative to \( C \). In particular, the formula \( \theta \) assigned to the final empty clause of the proof \( P \) is a relational interpolant separating \( \Sigma \) and \( \Pi \).

Proof. It is obvious that any assigned formula contains only relation symbols from \( L_\Sigma \cap L_\Pi \). So condition (1) of the definition holds.

For any occurrence of a clause or subclause \( C \) in the proof \( P \), let \( C_\Sigma \) (let \( C_\Pi \))
be $C$ with all occurrences of literals not from $\Sigma$ (not from $\Pi$) deleted. Then $C_\Sigma \models C$, $C_\Pi \models C$, and $C_\Sigma \lor C_\Pi \iff C$ are valid and $(C \lor D)_\Sigma = (C_\Sigma \lor D_\Sigma)$ and $(C_\Sigma)_\pi = (C_\pi)_\Sigma$ for any unifier $\pi$.

We prove by induction on the depth of $C$ in $P$ the following strengthenings of (2) and (3):

(2)' $\Sigma \models \theta \lor C_\Sigma$,

(3)' $\Pi \models \neg \theta \lor C_\Pi$.

Suppose $C$ is an input clause from $\Sigma$. Then $\theta$ is $F$ and $C_\Sigma = C$. Thus (2)' and (3)' hold since $\Sigma \models F \lor C$ and $\Pi \models T$. The argument for an input clause from $\Pi$ is similar.

Suppose (2)' and (3)' are true for clauses $L \lor C$ and $\neg L' \lor D$ of $P$ whose resolvent in $P$ is $(C \lor D)_\pi$ where $\pi$ is a unifier such that $L\pi = L'\pi$. Assume $L \lor C$ is assigned the formula $\phi$ and $\neg L' \lor D$ is assigned $\psi$. Thus we have

\[ \Sigma \models \phi \lor (L \lor C)_\Sigma, \quad \Sigma \models \psi \lor (\neg L' \lor D)_\Sigma, \]

\[ \Pi \models \neg \phi \lor (L \lor C)_\Pi, \quad \Pi \models \neg \psi \lor (\neg L' \lor D)_\Pi. \]

Case (a). Suppose the occurrences of $L$ and $\neg L'$ are both from $\Sigma$ alone. Thus the formula assigned to $(C \lor D)_\pi$ is $(\phi \lor \psi)_\pi$, and $(L \lor C)_\Sigma = L \lor C_\Sigma$ and $(\neg L' \lor D)_\Sigma = \neg L' \lor D_\Sigma$. By resolution we get

\[ (2)' : \Sigma \models ((\phi \lor C_\Sigma) \lor (\psi \lor D_\Sigma))_\pi = (\phi \lor \psi)_\pi \lor (C \lor D)_\pi. \]

For (3)' we have

\[ (L \lor C)_\Pi = C_\Pi \text{ and } (\neg L' \lor D)_\Pi = D_\Pi. \]
so

\[ \Pi \models (\neg \phi \lor C_\Pi) \land (\neg \psi \lor D_\Pi) \quad \text{and} \quad \Pi \models \neg(\phi \lor \psi)\pi \lor (C \lor D)\pi_\Pi. \]

Case (b) for \( L \) and \( \neg L' \) from \( \Pi \) alone is similar.

Case (c). Otherwise the occurrences of \( L \) and \( \neg L' \) are from \( \Sigma \) and from \( \Pi \) and the formula assigned to \((C \lor D)\pi\) is \((\neg L' \land \psi) \lor (L \land \psi))\pi\). In any model of \( \Sigma \) with any assignment of variables, if both \( C_\Sigma\pi \) and \( D_\Sigma\pi \) are false, then \((\phi \lor L)\pi\) and \((\psi \lor \neg L')\pi\) are true. So if \( L\pi = L'\pi \) is true, then so is \( \psi\pi \); if \( L\pi \) is false, then \( \phi\pi \) is true. Either way,

\[ ((\neg L' \land \phi) \lor (L \land \psi) \lor (C \lor D)\Sigma)\pi \]

is always true.

Similarly, \( \Pi \models (((L \lor \neg \phi) \land (\neg L' \lor \psi)) \lor (C \lor D)\Pi)\pi. \)

Hence, by induction, the theorem holds. \( \Box \)

Resolution provers often use paramodulation to handle equality. Given clauses \( C(r) \) and \( s = t \lor D \) with no variables in common and a unifier \( \pi \) such that \( r\pi = s\pi \) or \( r\pi = t\pi \), paramodulation infers the paramodulant \((C(t) \lor D)\pi\) or \((C(s) \lor D)\pi\) respectively.

**Definition 2.3** For a deduction \( P \) in \( L_\Sigma \cup L_\Pi \), a **noncommon term** is a term which begins with a symbol not in \( L_\Sigma \cap L_\Pi \). Such a term is called a **\( \Sigma \)-term** if its initial symbol is from \( \Sigma \), a **\( \Pi \)-term** if its initial symbol is from \( \Pi \). An occurrence of a noncommon term is **maximal** if this occurrence is not a subterm of a larger
noncommon term. A maximal \( \Sigma \)-term (\( \Pi \)-term) is a \( \Sigma \)-term (\( \Pi \)-term) which is also a maximal noncommon term.

Now we extend the Interpolation Algorithm to proofs with paramodulation as follows:

(iii). If \( \phi \) is assigned to \( C(r) \) and \( \psi \) is assigned to \( s = t \lor D \) and if \( \pi \) is a unifier such that \( r \pi = s \pi \), then the formula assigned to the paramodulant \( (C(t) \lor D)\pi \) is:

(d). \( [(\phi \land s = t) \lor (\psi \land s \neq t)] \land [s = t \land h(s) \neq h(t)] \pi \), provided \( r \) occurs in \( C(r) \) as a subterm of a maximal \( \Pi \)-term \( h(r) \) and there is more than one occurrence of \( h(r) \) in \( C(r) \lor \phi \),

(e). \( [(\phi \land s = t) \lor (\psi \land s \neq t)] \land [s \neq t \lor h(s) = h(t)] \pi \), provided \( r \) occurs in \( C(r) \) as a subterm of a maximal \( \Sigma \)-term \( h(r) \) and there is more than one occurrence of \( h(r) \) in \( C(r) \lor \phi \),

(f). \( [(\phi \land s = t) \lor (\psi \land s \neq t)] \pi \), if neither (d) nor (e).

Lemma 2.4. If \( \phi \) and \( \psi \) are the interpolants relative to \( C(r) \) and \( s = t \lor D \), respectively, then the above formula is an interpolant relative to the paramodulant \( (C(t) \lor D)\pi \).

PROOF. Suppose \( \phi \) is assigned to \( C(r) \), \( \psi \) is assigned to \( s = t \lor D \), and \( r \pi = s \pi \), then \( \Sigma \models (C(r) \lor \phi)\pi \) and \( \Sigma \models (s = t \lor D \lor \psi)\pi \). Let \( A \) be a model of \( \Sigma \) and let \( \theta \) be the new formula assigned according to the cases. First assume \( s \pi = t \pi \) in \( A \), then we have

for case (d), \( A \models (C(t) \lor \phi)\pi \), so \( A \models (C(t) \lor D)\pi \lor \theta \);
for case (e), $A \models (C(t) \lor \phi)\pi$ and $A \models (h(s) = h(t))\pi$, so $A \models (C(t) \lor D)\pi \lor \theta$;

for case (f), $A \models (C(t) \lor \phi)\pi$, so $A \models (C(t) \lor D)\pi \lor \theta$.

So in all three cases, $A \models (C(t) \lor D)\pi \lor \theta$.

Next assume that $s \pi \neq t \pi$ in $A$, then

for case (d), $A \models (D \lor \psi)\pi$, so $A \models (C(t) \lor D)\pi \lor \theta$;

for case (e), $A \models (D \lor \psi)\pi$, and $A \models (s \neq t)\pi$, so $A \models (C(t) \lor D)\pi \lor \theta$;

for case (f), $A \models (D \lor \psi)\pi$; so $A \models (C(t) \lor D)\pi \lor \theta$.

So for any model $A$ of $\Sigma$, we have $A \models (C(t) \lor D)\pi \lor \theta$. Thus $\Sigma \models (C(t) \lor D)\pi \lor \theta$ for the assigned formula $\theta$.

Similarly, since $\Pi \models C(r) \lor \neg \phi$, and $\Pi \models (s = t \lor D) \lor \neg \psi$, for the assigned formula $\theta$, $\Pi \models (C(t) \lor D)\pi \lor \neg \theta$ in the two cases: for $s \pi = t \pi, \Pi \models (\neg \phi \lor C(t))\pi$ and $\Pi \models (h(s) = h(t))\pi$; for $s \pi \neq t \pi, \Pi \models (\neg \psi \lor D)\pi$. Therefore we always have $\Pi \models (C(t) \lor D)\pi \lor \neg \theta$. Thus the assigned formula satisfies the requirement. □

The final rule of inference we need is factoring. Given a clause $L \lor L' \lor C$ and a unifier $\pi$ such that $L\pi = L'\pi$, factoring infers the clause $(L \lor C)\pi$. We extend the Interpolation Algorithm to proofs with factoring as follows:

(iv). If $\phi$ is assigned to $L \lor L' \lor C$ and $\pi$ is a unifier as above, then we assign $\phi\pi$ to the factor clause $(L \lor C)\pi$.

Clearly $\Sigma \models L \lor L' \lor C \lor \phi$ and $\Pi \models L \lor L' \lor C \lor \neg \phi$ imply $\Sigma \models (L \lor C)\pi \lor \phi\pi$ and $\Pi \models (L \lor C)\pi \lor \neg \phi\pi$.

Thus for a refutation proof $P$ by a series of binary resolutions, factorings, and
paramodulations, applying the above extended algorithm gives a formula, say $\theta$, for the empty clause. Since $\Sigma \models \theta$ and $\Pi \models \neg \theta$, $\theta$ is a relational interpolant separating $\Sigma$ and $\Pi$. While $\theta$ does not contain any noncommon relational symbol, it may contain noncommon terms with constants or function symbols which are not in $\mathcal{L}_\Sigma \cap \mathcal{L}_\Pi$. We now show how to get a Craig interpolant by replacing all noncommon terms in $\theta$ with appropriately quantified variables.

First we define a binary tree deduction to be a deduction in which any clause is used at most once. Such a deduction involving only binary resolutions, factorings, and paramodulations forms a binary tree.

**Lemma 2.5.** Any refutation $P$ using only binary resolutions, paramodulations, and factorings, lifts to a binary tree deduction $P_b$ with the same conclusion.

**Proof.** We prove this lemma by induction on the number $k(P)$ of clauses which are used more than once in the deduction $P$. If $k(P) = 0$, $P$ is a binary tree deduction. Assume the lemma holds for all deductions with $k(P) \leq n$ and suppose $k(P) = n + 1$. Let $C$ be a clause such that $C$ is used $m \geq 2$ times in $P$ but all the ancestors of $C$ are used only once. We construct a new deduction $P'$ from $P$ such that $P'$ has $m$ copies of $C$ and its ancestors and each copy of $C$ and its ancestors is used exactly once in $P'$. Finally, variables may be renamed if necessary, so that different input clauses have disjoint sets of variables. Otherwise $P$ is the same as $P'$ and has the same conclusion. Since $P'$ is a deduction with $k(P') \leq n$, by induction, $P'$ lifts to a binary tree deduction $P_b$ with the same conclusion. $\square$
Suppose \( P \) is a binary tree deduction whose input clauses have disjoint sets of variables and whose substitutions are generated by the usual unification algorithm, then the following properties hold in \( P \):

1. Every variable of any noninput clause in \( P \) occurs in exactly one parent clause and thus traces back to a unique ancestral input clause.
2. Any two incomparable (neither is the ancestor of the other) clauses have disjoint sets of variables.
3. For any substitution \( \pi \) of \( P \) and any variable \( x \), either \( \pi \) is trivial on \( x \), i.e., \( \pi(x) = x \), or \( x \) does not occur in the term \( \pi(x) \).
4. If \( \pi \) is nontrivial on \( x \), \( x \) never appears in any clause below \( \pi \).

**Definition 2.6.** Given a binary tree deduction \( P \) as above, for any variable \( x \) occurring in \( P \), let \( \pi_p \), the composite substitution for \( P \), be the substitution such that \( \pi_p(x) \) is the term resulting from applying to \( x \) the composition of all the substitutions along the path from the unique input clause which contains \( x \) to the bottom of \( P \).

**Lemma 2.7.** For any clause \( G \) in such a binary tree deduction \( P \), \( G\pi_p \) is the clause obtained by applying to \( G \) the composition of all the substitutions along the path from \( G \) to the bottom of \( P \).

**Proof.** Suppose \( x \) is a variable of \( C \). Then \( x \) traces back to a unique ancestral input clause \( D \). All of the substitutions along the path from \( D \) to \( C \) are trivial on
since otherwise, \( x \) would not occur in \( C \). Hence \( \pi(x) \) = the composition of all substitutions from \( D \) to the bottom. □

We say a deduction is *propositional* if there are no nontrivial unifying substitutions involved in the deduction.

**Lemma 2.8.** The Boolean operations \( \lor, \land, \neg \) and propositional binary resolution, factoring, and paramodulation commute with substitution. That is, if \( \pi \) is a substitution, then \((A \lor B)\pi = A\pi \lor B\pi\), \((A \land B)\pi = A\pi \land B\pi\), \((\neg A)\pi = \neg(A\pi)\); and for any propositional binary resolution \( \{A \lor L, B \lor \neg L\} \models A \lor B \), we have \( \{A\pi \lor L\pi, B\pi \lor \neg L\pi\} \models A\pi \lor B\pi \); and for any propositional paramodulation \( \{C(s), s = t \lor D\} \models C(t) \lor D \), we have \( \{C(s)\pi, (s = t)\pi \lor D\pi\} \models C(t)\pi \lor D\pi \).

**Lemma 2.9.** Every binary tree proof \( P_b \) projects to a propositional proof \( P_p \).

PROOF. Given a binary tree proof \( P_b \), rename the variables if necessary so that the above properties (1), (2), (3), and (4) hold and let \( \pi_p \) be the composite substitution for \( P \). Let \( P_p \) be the result of replacing each clause \( C \) of \( P_b \) with \( C\pi_p \) and replacing each substitution with the trivial identity substitution. By Lemma 2.8, \( P_p \) is a projection of \( P_b \) and \( P_p \) is a propositional binary tree deduction. □

**Lemma 2.10.** Assume \( P_b \) projects to \( P_p \) as in Lemma 2.9. If we apply the Interpolation Algorithm to the propositional deduction \( P_p \), and if a clause \( C' \) in \( P_p \) is assigned the formula \( \phi' \), and if its corresponding clause \( C \) in \( P_b \) is assigned formula \( \phi \), then \( \phi' = \phi\pi_p \). In particular, the assignments to \( \Diamond \) from both deductions are the same.
PROOF. Any occurrence of a literal \( L \) in a clause \( C' \) of \( P_p \) is from \( \Sigma \) or \( \Pi \) or both if and only if its corresponding occurrence in \( P_b \) is from \( \Sigma, \Pi, \) or both. So the corresponding clauses of \( P_b \) and \( P_p \) are assigned interpolants by the same case of the Interpolation Algorithm. Lemma 2.8 gives the result. □

Let \( P_p \) be a propositional deduction in \( L_{\Sigma} \cup L_{\Pi} \), and \( t_1, \ldots, t_n \) be all the \( \Pi \)-terms with maximal occurrences in \( P_p \). Let \( x_1, \ldots, x_n \) be a set of new variables which do not occur in \( P_p \). For any term or formula \( \theta \) in \( P_p \), define \( \bar{\theta}(x_1, \ldots, x_n) \) to be the term or formula obtained by simultaneously replacing all maximal occurrences of the \( \Pi \)-terms \( t_j \)'s by the new variables \( x_j \)'s. We call \( \bar{\theta} \) the \textit{lifted} formula of \( \theta \) from \( \Pi \)-terms.

**Lemma 2.11.**

\[
\overline{(A \lor B)}(x_1, \ldots, x_n) \iff \overline{A}(x_1, \ldots, x_n) \lor \overline{B}(x_1, \ldots, x_n); \\
\overline{(A \land B)}(x_1, \ldots, x_n) \iff \overline{A}(x_1, \ldots, x_n) \land \overline{B}(x_1, \ldots, x_n); \\
\overline{(s = t)}(x_1, \ldots, x_n) \iff \overline{s}(x_1, \ldots, x_n) = \overline{t}(x_1, \ldots, x_n); \\
\overline{(\neg A)}(x_1, \ldots, x_n) \iff \neg \overline{A}(x_1, \ldots, x_n); \\
\theta = \bar{\theta}(t_1, \ldots, t_n).
\]

**Lemma 2.12.** If \( \theta \) is the relational interpolant of \( \Sigma \) and \( \Pi \) relative to \( C \) assigned by the Interpolation Algorithm to the clause \( C \) of the propositional deduction \( P_p \), then we have

\[
\Sigma \models \overline{(C \lor \theta)}(x_1, \ldots, x_n).
\]
In particular, if \( \theta \) is the relational interpolant relative to the empty clause, then

\[
\Sigma \models \bar{\theta}(x_1, \ldots, x_n).
\]

PROOF. We prove this lemma by induction on the position of \( C \) in \( P_p \). If \( \bar{C}(t_1, \ldots, t_n) \) is an instance of an input clause from \( \Sigma \), all of the \Pi\)-terms in \( C \) come from free variables by the unifying substitutions of the original deduction. So by the construction of \( P_p \) we know that \( \bar{C}(x_1, \ldots, x_n) \) is an instance of some input clause in \( \Sigma \), and \( F \) is assigned to this clause. Thus \( \Sigma \models \bar{C}(x_1, \ldots, x_n) \lor F \). If \( \bar{C}(t_1, \ldots, t_n) \) is an instance of input clause from \( \Pi \), then it has assigned the formula \( T \) and \( \Sigma \models \bar{C}(x_1, \ldots, x_n) \lor T \) holds.

Now assume \( \Sigma \models (C \lor L \lor \phi)(x_1, \ldots, x_n) \) and \( \Sigma \models (D \lor \neg L \lor \psi)(x_1, \ldots, x_n) \) and that \( C \lor L \) and \( D \lor \neg L \) resolving against \( L \) gives \( C \lor D \) with interpolant \( \theta \). We show that \( \Sigma \models (C \lor D \lor \bar{\theta})(x_1, \ldots, x_n) \).

Notice that by propositional deduction and Lemma 2.11 we have

\[
\{ \bar{C} \lor L \lor \bar{\phi}, \quad \bar{D} \lor \neg L \lor \bar{\psi} \} \models \bar{C} \lor \bar{D} \lor (\bar{\phi} \lor \bar{\psi}), \text{ and}
\]

\[
\{ \bar{C} \lor \bar{L} \lor \bar{\phi}, \quad \bar{D} \lor \neg \bar{L} \lor \bar{\psi} \} \models \bar{C} \lor \bar{D} \lor (\neg \bar{\phi} \land \bar{\psi}) \lor (\bar{L} \land \bar{\psi}).
\]

Using Lemma 2.11 again proves this lemma for case (a) and case (c) of the Interpolation Algorithm definition of \( \theta \). For case (b), \( \theta = \phi \land \psi \), and the occurrences of \( L \) and \( \neg L \) are not from \( \Sigma \). By the proof of Theorem 2.2 we know that \( \Sigma \models \bar{C} \lor \bar{\phi} \) and \( \Sigma \models \bar{D} \lor \bar{\psi} \) respectively. Thus we have \( \Sigma \models \bar{C} \lor \bar{D} \lor (\bar{\phi} \land \bar{\psi}) \).

Next assume that \( C(s) \) and \( s = t \lor D \) gives \( C(t) \lor D \) by paramodulation. Assume \( \Sigma \models (C(s) \lor \phi)(x_1, \ldots, x_n) \) and \( \Sigma \models (s = t \lor D \lor \psi)(x_1, \ldots, x_n) \). At first we consider
case (d) of the assignment for paramodulation in which \( s \) occurs in \( C(s) \) as a subterm of a maximal II-term \( h(s) \) which occurs more than once in \( C(s) \lor \phi \) and in which the assigned formula is:

\[
\theta = [(\phi \land s = t) \lor (\psi \land s \neq t)] \pi \lor [s = t \land h(s) \neq h(t)] \pi.
\]

Then since \( h(s), h(t) \) are distinct II-terms, they will be replaced by distinct new variables \( h(s), h(t) \) in \( C(t) \lor \phi \). For any model of \( \Sigma \) and any assignment of all free variables in the lifted paramodulant and its assigned formula, if \( C(s) \) and \( \bar{s} = \bar{t} \) are true but \( C(t) \) is false, then we must have \( \bar{h}(s) \neq \bar{h}(t) \). So in this case we have:

\[
\Sigma \models (s = t \land h(s) \neq h(t))(x_1, \ldots, x_n).
\]

And hence,

\[
\Sigma \models (C(t) \lor D \lor \theta)(x_1, \ldots, x_n).
\]

Finally, for case (e) and case (f), since \( h(s) \) is not a subterm of a maximal II-term which occurs more than once, it is not replaced by this step of lifting. Hence, by the proof of Lemma 2.4, we also have

\[
\Sigma \models (C(t) \lor D \lor \theta)(x_1, \ldots, x_n). \quad \Box
\]

We now assign a dual formula to each clause in the proof \( P \) as follows:

(i). If \( C \) is an input clause from \( \Sigma \), its formula is \( T \); if \( C \) is an input clause from \( \Pi \), its formula is \( F \).

(ii). If \( \phi \) is assigned to \( L \lor C \) and \( \psi \) is assigned to \( \neg L' \lor D \), and if \( (C \lor D) \pi \) is the resolvent of \( L \lor C \) and \( D \lor \neg L' \) against \( L \pi = L' \pi \), then the formula assigned to \( (C \lor D) \pi \) is:

31
(a). \((\phi \land \psi)\pi\) if the occurrences of both \(L\) and \(\neg L'\) are from \(\Sigma\) alone;

(b). \((\phi \lor \psi)\pi\) if the occurrences of both \(L\) and \(\neg L'\) are from \(\Pi\) alone;

(c). \(((\neg L' \land \phi) \lor (L \land \psi))\pi\) if neither (a) nor (b).

(iii). If \(\phi\) is assigned to \(C(r)\) and \(\psi\) is assigned to \(s = t \lor D\) and if \(\pi\) is a unifier such that \(r\pi = s\pi\), then the formula assigned to the paramodulant \((C(t) \lor D)\pi\) is:

(d). \([((\phi \land s = t) \lor (\psi \land s \neq t))\pi \land [s \neq t \lor h(s) = h(t)]\pi\), provided the \(r\) is a subterm of a maximal \(\Pi\)-term \(h(r)\) and there is more than one occurrence of \(h(r)\) in \(C(r) \lor \phi\).

(e). \([((\phi \land s = t) \lor (\psi \land s \neq t))\pi \lor [s = t \land h(s) \neq h(t)]\pi\), provided the \(r\) is a subterm of a maximal \(\Sigma\)-term \(h(r)\) and there is more than one occurrence of \(h(r)\) in \(C(r) \lor \phi\).

(f). \([((\phi \land s = t) \lor (\psi \land s \neq t))\pi\) if neither (d) nor (e).

(iv). If \(\phi\) is assigned to \(L \lor L' \lor C\) and \(\pi\) is a unifier such that \(L\pi = L'\pi\), then we assign \(\phi\pi\) to the factor clause \((L \lor C)\pi\).

By induction on the depth of a clause in the deduction we can show

**Lemma 2.13.** The formula assigned by the dual method is the logical negation of that assigned by the original Interpolation Algorithm.

Assume that \(\delta(s_1, \ldots, s_k)\) is the dual formula assigned in \(P_p\) by the dual assignment algorithm, where \(s_1, \ldots, s_k\) are all the \(\Sigma\)-terms with maximal occurrences in \(P_p\). We define \(\hat{\delta}(y_1, \ldots, y_k)\) to be the formula obtained by simultaneously replacing all maximal occurrences of the \(\Sigma\)-terms \(s_1, \ldots, s_k\) by the new variables \(y_1, \ldots, y_k\). Then
we have

**Corollary 2.14.** If $\delta$ is the dual formula relative to $C$ assigned by the dual assignment algorithm, then we have

$$\Pi \models (\hat{\delta} \lor \hat{C})(y_1, \ldots, y_k).$$

In particular, if $\delta$ is the dual formula assigned to the empty clause, then we have

$$\Pi \models \hat{\delta}(y_1, \ldots, y_k).$$

**Proof.** The proof is the dual to the proof of Lemma 2.12.

Now we are ready to quantify all the variables for noncommon terms in the relational interpolant $\theta$ of $\Sigma$ and $\Pi$ relative to the empty clause. Assume that the set of all the maximal noncommon terms is $\{t_1, \ldots, t_n\}$, ordered by length, i.e., if $i < j$, then $t'_i$'s length is $\leq t'_j$'s length. Assume $\{t_1, \ldots, t_n\} = \{r_1, \ldots, r_k\} \cup \{s_{k+1}, \ldots, s_n\}$ where the $r_i$'s are the maximal $\Pi$-terms and the $s_j$'s are the maximal $\Sigma$-terms. If lifting $\theta$ from $\Pi$-terms gives $\bar{\theta}(x_1, \ldots, x_k)$, and lifting $\bar{\theta}(x_1, \ldots, x_k)$ from the $\Sigma$-terms gives $\theta^*(z_1, \ldots, z_n)$ where the $z_i$'s are new variables for the $t_i$'s, then we have

**Theorem 2.15.** $Q_1z_1\ldots Q_nz_n\theta^*(z_1, \ldots, z_n)$ is a Craig interpolant separating $\Sigma$ and $\Pi$, where $Q_i$ is $\forall$ if $t_i$ is a $\Pi$-term, otherwise $Q_i$ is $\exists$.

**Proof.** Clearly $Q_1z_1\ldots Q_nz_n\theta^*(z_1, \ldots, z_n)$ is a formula in $L_{\Sigma} \cap L_{\Pi}$. By Lemma 12 we have $\Sigma \models \forall x_1\ldots x_k \bar{\theta}(x_1\ldots x_k)$.

Each maximal $\Sigma$-term of $\bar{\theta}(x_1, \ldots, x_k)$ is a lifting $\bar{s}_j(x_1, \ldots, x_k)$ of one of the maximal $\Sigma$-terms $s_{k+1}, \ldots, s_n$ of $\theta$. If $x_i$ occurs in $\bar{s}_j(x_1, \ldots, x_k)$ then the term $r_i$ which
$x_i$ replaces is a subterm of $s_j$ and thus $r_i$ occurs before $s_j$ in the list \{$t_1, \ldots, t_n$\} and the variable for $r_i$ occurs before the variable for $s_j$ in the prefix $Q_1z_1 \ldots Q_n z_n$. Hence $s_j$ is a witness for the quantifier $\exists y_j$ in $Q_1z_1 \ldots Q_n z_n \theta^*(z_1, \ldots, z_n)$. Therefore, $\Sigma \models Q_1z_1 \ldots Q_n z_n \theta^*(z_1, \ldots, z_n)$.

On the other side, notice that the set of maximal $\Sigma$-terms and maximal $\Pi$-terms are disjoint, so using the same order among the noncommon terms we know that $\theta^*$ is also the lifting from $\Pi$-terms of the lifting $\hat{\theta}$ of $\theta$ from $\Sigma$-terms. Therefore, for the dual formula $\neg \theta$, by Corollary 2.14 we also have

$$\Pi \models \overline{Q}_1z_1 \ldots \overline{Q}_n z_n \neg \theta^*(z_1, \ldots, z_n)$$

where $\overline{Q}_j = \forall (\exists)$ if and only if $Q_j = \exists (\forall)$.

Moving the negation symbol out we finally have

$$\Pi \models \neg Q_1z_1 \ldots Q_n z_n \theta^*(z_1, \ldots, z_n). \Box$$

The formula $\theta^*$ may contain free variables other than $z_1, \ldots, z_n$. We get a Craig interpolant sentence by quantifying these extra variables with the quantifier $Q_1$ or any other sequence of quantifiers.

**2.3 Examples**

A **simplifying logical rewrite rule** is a logically valid equivalence whose right side is shorter than its left side. We use such a rule as a rewrite rule when we use it in the left-to-right direction, i.e., when an occurrence of the left side is replaced by an occurrence of its right side.
The simplifying logical rewrite rules we use are

\[ A \lor \neg A \iff T, \]
\[ (A \land \neg B) \lor B \iff A \lor B, \]
\[ A \land A \iff A, \]
\[ A \land \neg A \iff F, \]
\[ A \lor A \iff A, \]
\[ \neg \neg A \iff A. \]

After constructing the Craig interpolant \( \theta^* \), we apply the above simplifying logical rewrite rules and their commutative variants such as \( \neg A \lor A \iff T \) to simplify \( \theta^* \).

Example 1. Let \( \Sigma = \{ R(x,a) \lor R(x,b) \} \), \( \Pi = \{ \neg R(c,y) \} \), where \( a \), \( b \) and \( c \) are distinct constants. An OTTER resolution refutation for \( \Sigma \cup \Pi \models \diamond \) is:

1. \( R(x,a) \lor R(x,b) \).
2. \( \neg R(c,y) \).
3. \([\text{binary},1,2] R(c,b)\).
4. \([\text{binary},3,2] \).

35
The refutation proof is depicted in the following figure:

![Refutation Proof Diagram](image)

*Figure 4. Refutation of Example 1.*

For each step of the deduction, the table below lists the unifier, the literal of the resolution or the equality of the paramodulation, and the formula assigned to the clause generated by this step. If $a$ is replaced by $b$ by a unifier, we denote it by $a/b$ in the table, $id$ is the trivial substitution. For noninput clauses, the substitutions from the left parent and right parent are listed. From the proof and the Interpolation Algorithm, we get the following:

<table>
<thead>
<tr>
<th>Clause No</th>
<th>Generated by</th>
<th>Unifier</th>
<th>Literal</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>$\Sigma$</td>
<td></td>
<td></td>
<td>$F$</td>
</tr>
<tr>
<td>(2)</td>
<td>$\Pi$</td>
<td></td>
<td></td>
<td>$T$</td>
</tr>
<tr>
<td>(3)</td>
<td>binary (1,2)</td>
<td>$x/c; y/a$</td>
<td>$R(c,a)$</td>
<td>$R(c,a)$</td>
</tr>
<tr>
<td>(4)</td>
<td>binary (3,2)</td>
<td>$id; y/b$</td>
<td>$R(c,b)$</td>
<td>$R(c,a) \lor R(c,b)$</td>
</tr>
</tbody>
</table>

The Interpolation Algorithm gives the formula $\theta = R(c,a) \lor R(c,b)$. The set of maximal noncommon terms is $\{a, b, c\}$. Since $a$ and $b$ are $\Sigma$-terms and $c$ is a $\Pi$-term,
we replace \( a \) and \( b \) by the existentially quantified variables \( x \) and \( y \), and replace \( c \) by the universally quantified variable \( z \). Since the lengths of \( a, b, c \) are all 1, the order among the quantifiers does not matter. Thus the following three formulas are all Craig interpolants between \( \Sigma \) and \( \Pi \):

\[
\forall z \exists x y (R(z, x) \lor R(z, y)),
\]

\[
\exists x \forall z \exists y (R(z, x) \lor R(z, y)),
\]

\[
\exists x y \forall z (R(z, x) \lor R(z, y)).
\]

**Example 2.** Let \( \Sigma = \{x \neq f(x), x \neq f(f(x))\} \), \( \Pi = \{y = x \lor y = g(x)\} \), where both \( f \) and \( g \) are functions. Any model of \( \Sigma \) has a universe of size at least 3, while any model of \( \Pi \) has a universe of size at most 2. So \( \Sigma \) and \( \Pi \) are inconsistent. An OTTER resolution refutation for \( \Sigma \cup \Pi \models \emptyset \) is:

1 \( x \neq f(x) \).
2 \( x \neq f(f(x)) \).
3 \( y = x \lor y = g(x) \).
4 \([\text{binary,3.1,2.1}] x = g(f(f(x)))\).
5 \([\text{binary,3.1,1.1}] x = g(f(x))\).
10 \([\text{para_into,4.1.2,5.1.2}] x = f(x)\).
11 \([\text{binary,10.1,1.1}] \).
The refutation proof is depicted in the following figure:

- \(1 \) \( x \not\equiv f(x) \)
- \(2 \) \( x \not\equiv f(f(x)) \)
- \(3 \) \( y = x \lor y = g(x) \)
- \(4 \) \( x = g(f(f(x))) \)
- \(5 \) \( x = g(f(x)) \)
- \(10 \) \( x = f(x) \)
- \(11 \) \( \diamond \)

\[ \{ x \rightarrow f(x), y \rightarrow x \} \]

- \(1 \rightarrow f(x) \)
- \(3 \rightarrow f(f(x)) \)
- \(4 \rightarrow f(f(f(x))) \)
- \(5 \rightarrow f(f(x)) \)
- \(10 \rightarrow f(x) \)

Figure 5. Refutation of Example 2.

And following the Interpolant Algorithm, we get the following table:

<table>
<thead>
<tr>
<th>Clause No</th>
<th>Generated by</th>
<th>Unifier</th>
<th>Literal</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(\Sigma)</td>
<td></td>
<td></td>
<td>(F)</td>
</tr>
<tr>
<td>(2)</td>
<td>(\Sigma)</td>
<td></td>
<td></td>
<td>(F)</td>
</tr>
<tr>
<td>(3)</td>
<td>(\Pi)</td>
<td></td>
<td></td>
<td>(T)</td>
</tr>
<tr>
<td>(4)</td>
<td>binary (3,2)</td>
<td>(x/f(f(x)), y/x; id)</td>
<td>(x = f(f(x)))</td>
<td>(x \neq f(f(x)))</td>
</tr>
<tr>
<td>(5)</td>
<td>binary (3,1)</td>
<td>(x/f(x), y/x; id)</td>
<td>(x = f(x))</td>
<td>(x \neq f(x))</td>
</tr>
<tr>
<td>(10)</td>
<td>para (4,5)</td>
<td>id; (x/f(x))</td>
<td>(g(f(f(x))) = f(x))</td>
<td>(\phi)</td>
</tr>
<tr>
<td>(11)</td>
<td>binary (10,1)</td>
<td>id; id</td>
<td>(x = f(x))</td>
<td>(\phi \land x \neq f(x))</td>
</tr>
</tbody>
</table>

where \(\phi = [x \neq f(f(x)) \land x \neq g(f(f(x)))] \lor [x = g(f(f(x))) \land f(x) \neq f(f(x))]\). So the formula \(\theta = [\phi \land x \neq f(x)]\) the Interpolation Algorithm assigns to the empty clause \(\diamond\) is

\[ ([x \neq f(f(x)) \land x \neq g(f(f(x)))] \lor [x = g(f(f(x))) \land f(x) \neq f(f(x))]) \land x \neq f(x). \]
The set of maximal noncommon terms, when sorted according to lengths, is 
\{f(x), f(f(x)), g(f(f(x)))\}, where \(g(f(f(x)))\) is a \(\Pi\)-term and the others are \(\Sigma\)-terms. Replacing these terms with the variables \(u, v, w\) and quantifying them gives the formula

\[
\theta^* = \exists uv \forall w[(x \neq v \land x \neq w) \lor (x = w \land u \neq v)] \land (x \neq u).
\]

Note that any model for \(\theta^*\) contains at least three elements: It can not contain only one or two elements, for \(x, u, v\) must be distinct. Thus \(\theta^*\) is a Craig interpolant separating \(\Sigma\) and \(\Pi\).

2.4 Applications of Craig Interpolants

Given two finite structures, to show they are isomorphic, one finds an isomorphism. To show they are not, one gives a statement that separates them, i.e., a sentence which is true in one structure but false in the other. The Interpolation Algorithm can be used to find such a sentence.

For structures \(S_1\) with elements \(\{a_1, \ldots, a_n\}\) and \(S_2\) with elements \(\{b_1, \ldots, b_n\}\), assume that the universes for \(S_1\) and \(S_2\) are disjoint, and all the elements \(a_i, b_j\) are named by distinct new constant symbols. Furthermore assume the diagrams (the collection of all atomic sentences and negations of atomic sentences which hold in the structure) for the structures are \(\Delta_1\) and \(\Delta_2\) respectively. If the two structures are not isomorphic, then the theories \(\Sigma = \Delta_1 \cup \forall x(x = a_1 \lor \ldots \lor x = a_n)\) and \(\Pi = \Delta_2 \cup \forall y(y = b_1 \lor \ldots \lor y = b_n)\) are inconsistent with each other, and by
completeness there is a proof of $\Sigma \cup \Pi \models \diamondsuit$. Applying the Interpolation Algorithm to this proof gives a first-order sentence which separates the structures.

For example, let $S_1$ and $S_2$ be directed graphs as depicted in the following figure:

![Diagram of two directed graphs](image)

Figure 6. Two Nonisomorphic Structures.

$S_1$ has vertices $\{a, b, c\}$ with edges $\{(a, b), (a, c)\}$. $S_2$ has vertices $\{a', b', c'\}$ with edges $\{(a',b'), (c',a')\}$. We use a binary relation $p$ to represent the edges of the graphs. Then the diagram for $S_1$ is

$$
\Delta_1 = \{p(a, b), p(a, c), \neg p(b, a), \neg p(b, c), \neg p(c, a), \neg p(c, b), a \neq b, a \neq c, b \neq c\}.
$$

And the diagram for $S_2$ is

$$
\Delta_2 = \{p(a', b'), p(c', a'), \neg p(a', c'), \neg p(b', a'), \neg p(b', c'), \neg p(c', b'), \\
\quad a' \neq b', a' \neq c', b' \neq c'\}.
$$

So $\Sigma = \Delta_1 \cup \forall x (x = a \lor x = b \lor x = c)$ and $\Pi = \Delta_2 \cup \forall x (x = a' \lor x = b' \lor x = c')$.

The following is a refutation proof found by OTTER:

1. $p(a, b)$.
2. $p(a, c)$. 

40
Applying the Interpolation Algorithm, and applying the simplifying logical rewrite rules, we get the following table:
<table>
<thead>
<tr>
<th>Clause No</th>
<th>Generated by</th>
<th>Unifier</th>
<th>Literal</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>$\Sigma$</td>
<td></td>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>$\Sigma$</td>
<td></td>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>(7)</td>
<td>$\Sigma$</td>
<td></td>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>(8)</td>
<td>$\Sigma$</td>
<td></td>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>(10)</td>
<td>$\Sigma$</td>
<td></td>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>(14)</td>
<td>$\Pi$</td>
<td></td>
<td>$T$</td>
<td></td>
</tr>
<tr>
<td>(15)</td>
<td>$\Pi$</td>
<td></td>
<td>$T$</td>
<td></td>
</tr>
<tr>
<td>(16)</td>
<td>$\Pi$</td>
<td></td>
<td>$T$</td>
<td></td>
</tr>
<tr>
<td>(19)</td>
<td>$\Pi$</td>
<td></td>
<td>$T$</td>
<td></td>
</tr>
<tr>
<td>(22)</td>
<td>$\Pi$</td>
<td></td>
<td>$T$</td>
<td></td>
</tr>
<tr>
<td>(30)</td>
<td>para (10, 7)</td>
<td>$x = b$</td>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>(34)</td>
<td>para (10, 1)</td>
<td>$x = b$</td>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>(44)</td>
<td>para (19, 16)</td>
<td>$x = a'$</td>
<td>$T$</td>
<td></td>
</tr>
<tr>
<td>(274)</td>
<td>para (30, 8)</td>
<td>$x = c$</td>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>(507)</td>
<td>para (34, 2)</td>
<td>$x = c$</td>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>(699)</td>
<td>para (44, 14)</td>
<td>$x = b'$</td>
<td>$T$</td>
<td></td>
</tr>
<tr>
<td>(711)</td>
<td>binary (699, 507)</td>
<td>$x/a; x/c'$ $p(a, c')$</td>
<td>$p(a, c')$</td>
<td></td>
</tr>
<tr>
<td>(783)</td>
<td>binary (711, 274)</td>
<td>id; $x/c'$ $a = c'$</td>
<td>$p(a, c') \land a \neq c'$</td>
<td></td>
</tr>
<tr>
<td>(794)</td>
<td>para (783, 22)</td>
<td>$c' = a$</td>
<td>$p(a, c') \lor a = c'$</td>
<td></td>
</tr>
<tr>
<td>(816)</td>
<td>binary (794, 507)</td>
<td>id; $x/b'$ $b' \neq a$</td>
<td>$(p(a, c') \lor a = c') \land b' = a$</td>
<td></td>
</tr>
<tr>
<td>(817)(◇)</td>
<td>binary (816, 797)</td>
<td>$p(a, b')$</td>
<td>$\theta$</td>
<td></td>
</tr>
</tbody>
</table>

where $\theta$ is $(c' = a \lor p(a, c')) \land (b' = a \lor p(a, b'))$. Thus, the relational interpolant is

$$(c' = a \lor p(a, c')) \land (b' = a \lor p(a, b'))$$

Note that $a$ is a $\Sigma$-term, while $b'$ and $c'$ are $\Pi$-terms. If we replace $a, b', c'$ by $x, y, z$, respectively, then Theorem 15 gives the following formulas. They all separate
the two graphs:

\[ \exists x \forall y z[(z = x \lor p(x, z)) \land (y = x \lor p(x, y))] , \]

\[ \forall y \exists x \forall z[(z = x \lor p(x, z)) \land (y = x \lor p(x, y))] , \]

\[ \forall y z \exists x[(z = x \lor p(x, z)) \land (y = x \lor p(x, y))] . \]

Note also that there is a shorter formula \( \exists x \forall y (x = y \lor p(x, y)) \) which separates the two given structures. The generation of minimal length separating sentences is an open problem. We also need more efficient proof strategies for such problems since current resolution provers can not find refutation proofs for pairs of structures with more than 6 elements.
CHAPTER 3
SOLVING PROBLEMS
BY CASE ANALYSIS

In this chapter we show how to use case analysis and resolution to solve problems such as the zebra problem, the pigeonhole problem and the stable marriage problem.

3.1 The Zebra Problem

The following is the zebra problem taken from [SS].

There are five people: Englishman, Spaniard, Norwegian, Japanese, Ukranian.

Five houses: 1, 2, 3, 4, and 5.

Five drinks: orange, coffee, water, tea, and milk.


Five animals: dog, fox, horse, snail, and zebra.

Five colors: yellow, red, blue, ivory, and green.

Each person has a unique nationality lives in a unique house, drinks a unique drink, owns a unique animal, smokes a unique cigarette, and each house has a unique color.

(1). The Englishman lives in the red house.

(2). The Spaniard owns the dog.

(3). Coffee is drunk in the green house.

(4). The Ukrainian drinks tea.
(5). The green house is to the immediate right of the ivory house.

(6). The Winston smoker owns the snail.

(7). Kools are smoked in the yellow house.

(8). Milk is drunk in the middle house.

(9). The Norwegian lives in the first house on the left.

(10). The man who smokes Chesterfields lives in the house next to where the fox is kept.

(11). The Kools smoker lives next to where the horse is kept.

(12). The Lucky smoker drinks orange juice.

(13). The Japanese smokes Parliaments.

(14). The Norwegian lives next to the blue house.

And the question is “who owns the zebra?”

To reduce this puzzle to a theorem-proving problem we need to translate the problem into a set of sentences of a first-order language.

The zebra problem has six data types, each type has five distinct values. These values are naturally represented as first-order constants. These constants are related in various ways. The most straightforward representation would use a mix of relation symbol and constants. Thus, if housed(x, y) means person x lives in house y, and if color(x, y) means x is the color of house y, then condition 1 (The Englishman lives in the red house) could be translated as

$$\forall y (\text{housed(Englishman, } y) \iff \text{color(red, } y)).$$
Notice that *Englishman* and *red* are constants in the above representation.

While the relational representation is easy to write and understand, it is computationally inefficient. This is due to the unnecessary use of variables. When variables occur, theorem provers spend a lot of time searching for possible substitutions. For a problem as complicated as the zebra problem, the space of possible solutions for such a relational representation is too large and a search for a solution is likely to fail.

A more efficient representation uses only constants and equality. Since there are no variables and no relations other than the built-in equality relation, the size of the space of possible solutions is much smaller. The zebra problem involves 30 constants. We set two constants of different data types equal whenever they are related to each other. Thus we describe the problem using only equations. For example, the condition *The Englishman lives in the red house* becomes *Englishman* = *red*.

We use the constants 1, 2, 3, 4, and 5 for the five houses ordered from left to right. Likewise we have five constants for each of the other five data types. Using equality to represent associations, the first four conditions of the zebra problem are represented as:

\[
\begin{align*}
&\text{Englishman}=\text{red}, \\
&\text{Spaniard}=\text{dog}, \\
&\text{coffee}=\text{green}, \\
&\text{Ukrainian}=\text{tea}, \\
\end{align*}
\]
However, condition 5 (the green house is to the immediate right of the ivory house) can not be represented by a single clause. It is equivalent to the sentence

\[(ivory = 1 \iff green = 2) \land (ivory = 2 \iff green = 3) \land (ivory = 3 \iff green = 4) \land (ivory = 4 \iff green = 5).\]

where each \(\iff\) produces two clauses. Thus we have the following clauses for condition (5):

\[ivory \neq 1 \lor green = 2,\]
\[ivory \neq 2 \lor green = 3,\]
\[ivory \neq 3 \lor green = 4,\]
\[ivory \neq 4 \lor green = 5,\]
\[ivory = 1 \lor green \neq 2,\]
\[ivory = 2 \lor green \neq 3,\]
\[ivory = 3 \lor green \neq 4,\]
\[ivory = 4 \lor green \neq 5.\]

Similarly, conditions (7), (10), (11) and (14) also need several clauses to completely describe their meanings.

We must also explicitly state that the five constants of each of the five data types are distinct. For the houses, this can be done by the following clauses:

\[1 \neq 2, \quad 1 \neq 3, \quad 1 \neq 4, \quad 1 \neq 5,\]
\[2 \neq 3, \quad 2 \neq 4, \quad 2 \neq 5,\]
\[3 \neq 4, \quad 3 \neq 5, \quad 4 \neq 5.\]
Also we must state that each constant of one type is equal to one and only one constant of any other type. For house numbers and house colors the clauses are:

\[
\begin{align*}
\text{red} &= 1 \lor \text{red} = 2 \lor \text{red} = 3 \lor \text{red} = 4 \lor \text{red} = 5, \\
\text{green} &= 1 \lor \text{green} = 2 \lor \text{green} = 3 \lor \text{green} = 4 \lor \text{green} = 5, \\
\text{ivory} &= 1 \lor \text{ivory} = 2 \lor \text{ivory} = 3 \lor \text{ivory} = 4 \lor \text{ivory} = 5, \\
\text{yellow} &= 1 \lor \text{yellow} = 2 \lor \text{yellow} = 3 \lor \text{yellow} = 4 \lor \text{yellow} = 5, \\
\text{blue} &= 1 \lor \text{blue} = 2 \lor \text{blue} = 3 \lor \text{blue} = 4 \lor \text{blue} = 5.
\end{align*}
\]

The zebra problem has a solution if and only if the set of clauses listed above is satisfiable.

3.2 Case Analysis

With the above representation the set of clauses is large. What is worse, it contains many clauses such as these listed above which have five literals. For such problems a resolution prover such as OTTER will generate many long clauses. In practice, these clauses exhaust all the available memory before a solution is found.

The difficult clauses such as the clause

\[
\text{red} = 1 \lor \text{red} = 2 \lor \text{red} = 3 \lor \text{red} = 4 \lor \text{red} = 5
\]

are quite symmetric and in each at most one literal can be true. This kind of pattern occurs in many other problems, for example the pigeonhole problem and the stable marriage problem. These long clauses need to be reduced to shorter
clauses, preferably to unit clauses. To do this we use case analysis in place of resolution for some of the long clauses.

For any set of clauses $\Sigma$, and any literals $A_1, ..., A_n, ..., C_1, ..., C_n$,

$$\Sigma, A_1 \lor ... \lor A_n, B_1 \lor ... \lor B_n, C_1 \lor ... \lor C_n \vdash \emptyset$$

if and only if for all $1 \leq i, j, k \leq n$,

$$\Sigma, A_i, B_j, C_k \vdash \emptyset.$$

The latter problems are called cases of the original problem. The case analysis or path checking method consists of breaking a problem into its cases and then checking each of the cases.

To indicate that we wish to use case analysis on a clause $C_1 \lor C_2 \lor ... \lor C_n$ instead of resolution, we replace $\lor$ by $\parallel$ and write

$$C_1 \parallel C_2 \parallel ... \parallel C_n.$$

The above case clause has $n$ case literals, they are $C_1, ..., C_n$ respectively.

The case analysis algorithm CASE which we use chooses case literals, one at a time, from the set of case clauses. Each such set of case literals is combined with the noncase clauses to form a case of the original problem. A resolution prover (OTTER) is then used to determine if any of the cases of the original problem are satisfiable. The original problem is satisfiable if and only if some case of the original problem is satisfiable.
The idea behind the CASE algorithm is similar to the connection-matrix algorithm of [Bi]. In the matrix method, all the clauses for a problem are put into a matrix, the i-th row of which consists of the literals of the i-th clause. A problem is unsatisfiable if and only if the matrix representing the problem is complementary, that is, every vertical path through the matrix contains at least one pair of literals with complementary instances. The CASE program searches for vertical paths in a similar manner but instead of checking for complementary literals, it checks more generally for inconsistency using resolution.

3.3 Solving the Zebra Problem

We use the CASE program to attack the zebra problem. The time required depends on the set of clauses selected to be case clauses. Our best time was achieved using the following five clauses as the case clauses:

\[
\begin{align*}
\text{red}=1 & \quad | \quad \text{red}=2 & \quad | \quad \text{red}=3 & \quad | \quad \text{red}=4 & \quad | \quad \text{red}=5, \\
\text{ivory}=1 & \quad | \quad \text{ivory}=2 & \quad | \quad \text{ivory}=3 & \quad | \quad \text{ivory}=4, \\
\text{English}=1 & \quad | \quad \text{English}=2 & \quad | \quad \text{English}=3 & \quad | \quad \text{English}=4 & \quad | \quad \text{English}=5, \\
\text{Ukrainian}=1 & \quad | \quad \text{Ukrainian}=2 & \quad | \quad \text{Ukrainian}=3 & \quad | \quad \text{Ukrainian}=4 & \quad | \quad \text{Ukrainian}=5, \\
\text{Japanese}=1 & \quad | \quad \text{Japanese}=2 & \quad | \quad \text{Japanese}=3 & \quad | \quad \text{Japanese}=4 & \quad | \quad \text{Japanese}=5.
\end{align*}
\]

For this set of case clauses, CASE finds the following satisfiable case in 68 seconds after searching 61 vertical paths:

\[
\begin{align*}
\text{red} &= 3, & \text{Ukrainian} &= 2, & \text{English} &= 3, & \text{Japanese} &= 5, & \text{ivory} &= 4.
\end{align*}
\]
With these literals added to the noncase clauses of the zebra problem, OTTER generates the following solution within two seconds:

\[
\begin{align*}
green &= 5, & \text{dog} &= \text{Spaniard}, & \text{coffee} &= 5, \\
tea &= 2, & \text{snail} &= \text{Winston}, & \text{yellow} &= \text{Kools}, \\
milk &= 3, & 1 &= \text{Norwegian}, & \text{orange} &= \text{Luck}, \\
\text{Parliaments} &= 5, & \text{blue} &= 2, & \text{Spaniard} &= 4, \\
dog &= 4, & \text{Norwegian} &= \text{Kools}, & \text{Luck} &= 4, \\
\text{horse} &= 2, & 1 &= \text{Kools}, & \text{orange} &= 4, \\
\text{apple} &= \text{Kools}, & \text{Winston} &= 3, & \text{snail} &= 3, \\
\text{Chesterfields} &= 2, & \text{fox} &= \text{Kools}, & \text{zebra} &= 5.
\end{align*}
\]

Thus we get the following complete solution:

\[
\begin{align*}
1 &= \text{yellow} = \text{Norwegian} = \text{fox} = \text{water} = \text{Kools}, \\
2 &= \text{blue} = \text{Ukrainian} = \text{horse} = \text{tea} = \text{Chesterfields}, \\
3 &= \text{red} = \text{English} = \text{snail} = \text{milk} = \text{Winston}, \\
4 &= \text{ivory} = \text{Spaniard} = \text{dog} = \text{orange} = \text{Luck}, \\
5 &= \text{green} = \text{Japanese} = \text{zebra} = \text{coffee} = \text{Parliaments}.
\end{align*}
\]

The total run time is 70 seconds (on a Sun Sparc Station II), better than the most recent result of 185.72 seconds (see [LW]). The complete description of the CASE program for this problem is in Appendix A.
3.4 The Stable Marriage Problem

The stable marriage problem is an abstraction of a group of allocation problems. Assume there are $n$ men and $n$ women who have expressed mutual preferences, i.e., each man lists the $n$ women in order of his preference, and vice versa. We say a set of marriages is unstable if two people both prefer each other to their spouses. The problem is to find a set of $n$ marriages that is stable.

As an example we consider the following problem with 3 men A, B, C and 3 women a, b, c. Assume their preference lists, listed in order of decreasing preference, are as follows:

$$A: \ a\ b\ c$$
$$B: \ a\ c\ b$$
$$C: \ b\ a\ c$$
$$a: \ B\ C\ A$$
$$b: \ C\ A\ B$$
$$c: \ C\ B\ A.$$

We use constants $A, B, C, a, b, c$ for the 3 men and 3 women, and use constants $1, 2, 3$ for the 3 positions in the 3-place preference list. Then the preference lists can be represented with a 2-place function where $p(x,y) =$ the position of $y$ in $x$'s preference list. In this way the above preference lists are represented by the following first-order clauses:
\[ p(A, a) = 1, p(A, b) = 2, p(A, c) = 3, \]
\[ p(B, a) = 1, p(B, c) = 2, p(B, b) = 3, \]
\[ p(C, b) = 1, p(C, a) = 2, p(C, c) = 3, \]
\[ p(a, B) = 1, p(a, C) = 2, p(a, A) = 3, \]
\[ p(b, C) = 1, p(b, A) = 2, p(b, B) = 3, \]
\[ p(c, C) = 1, p(c, B) = 2, p(c, A) = 3. \]

We use a 2-place relation \( m \) to represent the marriage so that \( m(x, y) \) means \textit{woman} \( x \) \textit{marries man} \( y \). Then the fact that \textit{woman} \( b \) \textit{marries either} \( A \), or \( B \), or \( C \) is axiomatized by the clause

\[ m(b, A) \lor m(b, B) \lor m(b, C). \]

And the fact that \textit{man} \( A \) \textit{marries either} \( a \), or \( b \), or \( c \) is axiomatized by the clause

\[ m(a, A) \lor m(b, A) \lor m(c, A). \]

Similar clauses are needed for \( a, c, B, \) and \( C \).

For any two couples \( m(x_1, y_1) \) and \( m(x_2, y_2) \) of a stable marriage, one of the following must hold: either \( x_1 \) prefers \( y_1 \) to \( y_2 \), or \( y_2 \) prefers \( x_2 \) to \( x_1 \). This fact is axiomatized by the clause:

\[ [m(x_1, y_1) \land m(x_2, y_2)] \Rightarrow [p(x_1, y_1) \leq p(x_1, y_2) \lor p(y_2, x_2) \leq p(y_2, x_1)]. \]

In addition to the above axioms, we must explicitly state that the constants for
men and women are distinct:

\[ A \neq B, A \neq C, B \neq C, \]
\[ a \neq b, a \neq c, b \neq c. \]

Also a woman can marry only one man, and a man can marry only one woman:

\[ m(x, y_1) \land m(x, y_2) \Rightarrow (y_1 = y_2), \]
\[ m(x_1, y) \land m(x_2, y) \Rightarrow (x_1 = x_2). \]

We choose the clauses of the form \( m(b, A) \lor m(b, B) \lor m(b, C) \) to be the case clauses.

With these case clauses CASE finds the following set of stable marriages:

\[ m(a, B), m(b, C), m(c, A). \]

We encoded two other marriage problems in a similar way, one with 5 couples, another one with 7 couples. For the problem with 5 couples, CASE found a solution in 384 seconds. For the problem with 7 couples, CASE finds a solution in 20,000 seconds. Without cases, both problems exceed our memory resources and fail.

3.5 The Pigeonhole Problem

The pigeonhole principle states: if \( n + 1 \) pigeons are placed into \( n \) holes, then at least one hole is occupied by two or more pigeons.

In order to prove the pigeonhole principle for some particular number \( n \), it suffices to refute the statement that all \( n + 1 \) pigeons fit into the \( n \) holes and each hole is occupied by at most one pigeon.

For the simplest representation of the pigeonhole problem, we use propositional symbols. Assume \( a, b, c, d, \ldots \) are the pigeons and \( 1, 2, 3, \ldots \) are the holes. Then we
use propositions $a_1$, $b_2$, ... to denote the assertions that pigeon $a$ is in hole 1, pigeon $b$ is in hole 2, ... . Then the assertion that pigeon $a$ is in some hole is axiomatized by

$$a_1 \lor a_2 \lor ... \lor a_n.$$  

And the assertion that pigeon $a$ can occupy only one hole is axiomatized by the set of clauses $\{ \neg(a_i \land a_j) : i, j \text{ are distinct holes} \}$. And "hole 1 is occupied by at most one pigeon" is axiomatized by the set of clauses

$$\{ \neg(x_1 \land y_1) : x, y \text{ are distinct pigeons} \}.$$  

This gives a propositional representation of the pigeonhole problem. For the case of 4 pigeons and 3 holes, OTTER finds a refutation within 5.29 seconds. The proof has 68 steps. For the case of 5 pigeons and 4 holes, OTTER finds a proof in 3088 seconds, the proof has 502 steps. But OTTER fails to find proof for 6 or more pigeons.

When we choose several complex clauses as case clauses, CASE quickly finds a refutation. For the pigeon hole problem of 4 pigeons with 3 holes, if we choose only one clause such as $a_1 \lor a_2 \lor a_3$ as the case clause, we get a proof within 2 seconds. For the problem with 5 pigeons and 4 holes, we get a proof within 14 seconds using two case clauses. For 8 pigeons and 7 holes, with 4 case statements, CASE finds a proof within 8685 seconds, with 5 case statements, CASE finds a proof within 2509 seconds, with 6 case statements, CASE finds a proof within 39977 seconds.
This shows that a combination of case analysis and resolution can be much more effective than either case analysis or resolution alone.

3.6 The Instant Insanity Problem

The Instant Insanity puzzle (the Great Tantalizer in [Berlekamp], p. 784) has four given cubes with their faces colored in the way depicted in the following figure, where \( b, g, r, e \) represent the colors blue, green, red and yellow respectively. The problem is to assemble the four cubes into a vertical \( 1 \times 1 \times 4 \) tower so that each lateral or vertical wall displays all four colors.

For convenience we assume that the lateral sides of the cubes are perpendicular to the \( x \) and \( y \) axes. We name the four cubes \( p, q, s, t \) respectively. For any cube, say cube \( p \), let \( p(w, x, y, z) \) be the relation which is true if and only if \( w, x, y, z \) are the colors of its front, left, back, and right sides. Thus we have four 4-place relations, one for each cube.

\[
\begin{array}{cccc}
  & e & b & r \\
  b & r & e & e \\
  g & b & r & g \\
  & g & e & b \\
\end{array}
\]

\[
\begin{array}{cccc}
  & g & b & e \\
  b & r & g & e \\
  g & b & r & e \\
  & e & g & b \\
\end{array}
\]

Cube p Cube q Cube s Cube t

Figure 7. An Example of the Insanity Problem.

Since there are 24 possible orientations, for the cube \( p \), the following clause is always true. It lists the colorings of the four lateral faces for each possible orientation of \( p \).
\[ p(b, r, e, e) \lor p(b, g, e, e) \lor p(b, e, e, r) \lor p(b, e, e, g) \lor \]
\[ p(r, b, e, e) \lor p(r, e, e, g) \lor p(r, e, e, b) \lor p(r, g, e, e) \lor \]
\[ p(e, r, b, e) \lor p(e, g, b, e) \lor p(e, e, b, r) \lor p(e, e, b, g) \lor \]
\[ p(e, b, r, e) \lor p(e, e, r, g) \lor p(e, e, r, b) \lor p(e, g, r, e) \lor \]
\[ p(e, b, g, e) \lor p(e, r, g, e) \lor p(e, e, g, b) \lor p(e, e, g, r) \lor \]
\[ p(g, b, e, e) \lor p(g, r, e, e) \lor p(g, e, e, b) \lor p(g, e, e, r). \]

Similar clauses exist for cubes \( q, r \) and \( s \).

A solution of the puzzle requires that the four faces of any vertical wall have distinct colors. Hence for any two cubes say \( p \) and \( q \) and any lateral face, say the front, the cubes must have different colors in this face. Thus \( p(x_1, x, y, z) \) and \( q(x_1, u, v, w) \) can not both be true. To guarantee that cube \( p \) and \( q \) do not share the same color on any lateral face, we need the following clauses.

\[ \neg p(x_1, x, y, z) \lor \neg q(x_1, u, v, w), \]
\[ \neg p(x, x_1, y, z) \lor \neg q(u, x_1, v, w), \]
\[ \neg p(x, y, x_1, z) \lor \neg q(u, v, x_1, w), \]
\[ \neg p(x, y, z, x_1) \lor \neg q(u, v, w, x_1). \]

Similar sets of clauses are needed for all other pairs of cubes.

There is no unit clause in the above description, and resolving a nonunit clause with any of the four long clauses which describe a cube's 24 possible orientations will produce another clause of 24 or more literals. These numerous long clauses make
it impossible for OTTER to achieve in a reasonable amount of time a refutation of the claim that any assembling of the 4 cubes has at least one vertical wall with less than 4 colors.

The four 24-literal orientation clauses are good candidates for use as case clauses. We have found that when we designate two of the long clauses as case clauses, CASE finds a solution within 39 seconds. One solution is:

\[ p(e, b, g, e), \quad q(g, g, b, g), \quad s(r, e, r, r), \quad t(b, r, e, b). \]
CHAPTER 4
FIRST-ORDER REPRESENTATION

4.1 Introduction

To solve problems using general-purpose reasoning methods they must be formalized concisely in a unified way. If a problem can be formalized in any way at all, it can be formalized with first-order logic and thus be reduced to a first-order theorem-proving problem.

Representing a problem in first-order logic usually will not lead to a solution if axioms of set theory or number theory are involved. Current theorem provers have limited power in dealing with either of these undecidable theories. Fortunately the legal moves of all the puzzles in this chapter can be represented directly with first-order axioms which do not use set theoretic or number theoretic concepts and which are simple enough to be used with available theorem provers such as OTTER.

To represent a typical puzzle with \( n \) positions in its board, we number the positions in some order and then introduce an \( n \)-place relation, say \( r \), whose places correspond to the positions of the puzzle. Our variables range over the finite universe consisting of the pieces of the puzzle, each of which is named by some constant. Thus a configuration of the puzzle is represented by an instance \( r(a_1, a_2, \ldots, a_n) \) of the relation where \( a_i \) is the constant for the \( i \)-th piece on the board.

We shall call a configuration of the puzzle a board. We will often identify a board
with the literal \( r(a_1, a_2, \ldots, a_n) \) which represents it. The initial configuration and the final goal configuration of the puzzle are the initial and home boards. The home position or destination of a piece is its position on the home board. If board \( A \) can be transformed into board \( B \) by a sequence of legal moves, we say \( B \) is accessible or reachable from \( A \).

A move which carries a board \( r(t_1, t_2, \ldots, t_n) \) to a board \( r(s_1, s_2, \ldots, s_n) \) is represented by the if-then rule

\[
r(t_1, t_2, \ldots, t_n) \Rightarrow r(s_1, s_2, \ldots, s_n).
\]

We now introduce a simple puzzle which we will use to illustrate our terminology, strategies and theorems. Later we will apply our strategies to more interesting puzzles such as Rubik's cube and central solitaire.

### 4.2 A Simple Puzzle Example

The unscrambling puzzle we will use as an example has a board with 8 positions in a row. There are 8 pieces labeled 1, 2, 3, 4, 5, 6, 7 and 8 respectively in the board. A legal move consists of swapping the positions of any two adjacent pieces. In the move pictured below, pieces 3 and 2 are swapped.

\[
\begin{array}{cccccccc}
1 & 3 & 2 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\Rightarrow
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

Given any initial board, the goal is to produce the final unscrambled board with the pieces in ascending order.
For our first-order representation we need 8 constants 1, 2, 3, 4, 5, 6, 7, 8 for the pieces and an 8-place relation \( s \) to represent the board. The home board is \( s(1, 2, 3, 4, 5, 6, 7, 8) \). Given an initial board \( s(a_1, a_2, \ldots, a_n) \) the problem of getting from the initial board to the home board via the given rules can be formalized with the clauses:

\[ \text{initial} : s(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8), \]

\[ \text{home} : s(1, 2, 3, 4, 5, 6, 7, 8), \]

\[ \text{rules} : s(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) \Rightarrow s(x_2, x_1, x_3, x_4, x_5, x_6, x_7, x_8), \]

\[ s(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) \Rightarrow s(x_1, x_3, x_2, x_4, x_5, x_6, x_7, x_8), \]

\[ s(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) \Rightarrow s(x_1, x_2, x_4, x_3, x_5, x_6, x_7, x_8), \]

\[ s(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) \Rightarrow s(x_1, x_2, x_3, x_5, x_4, x_6, x_7, x_8), \]

\[ s(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) \Rightarrow s(x_1, x_2, x_3, x_4, x_6, x_5, x_7, x_8), \]

\[ s(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) \Rightarrow s(x_1, x_2, x_3, x_4, x_5, x_7, x_6, x_8), \]

\[ s(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) \Rightarrow s(x_1, x_2, x_3, x_4; x_5, x_6, x_7, x_8). \]

We use the usual convention that all variables in the rule clauses are to be universally qualified. Thus a rule \( \theta(x, y) \Rightarrow \theta(y, x) \) means

\[ \forall xy(\theta(x, y) \Rightarrow \theta(y, x)). \]

Notice that in each of the above rules most of the variables are inactive, i.e., they do not change their positions after applying the rule. For the sake of simplicity, we use stars for the inactive variables and omit commas when no confusion can occur.
For example, \( \theta(x,y,**) \mapsto \theta(y,x,**) \) represents \( \theta(x,y,z_1,z_2) \mapsto \theta(y,x,z_1,z_2) \). In this way, the above set of rules for our puzzle can be abbreviated to:

\[
\begin{align*}
& s(x,y,*****) \Rightarrow s(y,x,*****) , \\
& s(*,x,y,*****) \Rightarrow s(*,y,x,*****) , \\
& s(**,x,y,*****) \Rightarrow s(**,y,x,*****) , \\
& s(***,x,y,*****) \Rightarrow s(***,y,x,*) , \\
& s(*****,x,y,***) \Rightarrow s(*****,y,x,***) , \\
& s(*****,x,*) \Rightarrow s(*****,y,*) , \\
& s(*****,x,y) \Rightarrow s(*****,y,x).
\end{align*}
\]

4.3 Reducing Puzzle Solving to Theorem Proving

The problem of moving the pieces of the board \( s(a_1,a_2,\ldots,a_n) \) to their home positions can be reduced to the problem of proving

\[
\text{rules} \vdash \text{initial} \Rightarrow \text{home},
\]

or, for refutation theorem provers, to the problem of proving

\[
\text{rules} + \text{initial} + \neg \text{home} \vdash \Box.
\]

Suppose the initial board is \( s(3,1,4,6,5,2,8,7) \). Then the above can be encoded as follows for the OTTER theorem prover in which \( \neg \phi \mid \psi \) means \( \neg \phi \lor \psi \) which is equivalent to \( \phi \Rightarrow \psi \):
set(hyper_res).

list(usable).

-\(s(x,y,z1,z2,z3,z4,z5,z6) \mid s(y,x,z1,z2,z3,z4,z5,z6)\).

-\(s(z1,x,y,z2,z3,z4,z5,z6) \mid s(z1,y,x,z2,z3,z4,z5,z6)\).

-\(s(z1,z2,x,y,z3,z4,z5,z6) \mid s(z1,z2,y,x,z3,z4,z5,z6)\).

-\(s(z1,z2,z3,x,y,z4,z5,z6) \mid s(z1,z2,z3,y,x,z4,z5,z6)\).

-\(s(z1,z2,z3,z4,x,y,z5,z6) \mid s(z1,z2,z3,z4,y,x,z5,z6)\).

-\(s(z1,z2,z3,z4,z5,x,y,z6) \mid s(z1,z2,z3,z4,z5,y,x,z6)\).

-\(s(z1,z2,z3,z4,z5,z6,x,y) \mid s(z1,z2,z3,z4,z5,z6,y,x)\).

-\(s(1,2,3,4,5,6,7,8)\).

end_of_list.

list(sos).

\(s(3,1,4,6,5,2,8,7)\).

end_of_list.

OTTER's proof for this problem is:

UNIT CONFLICT at 11.29 sec.

Length of proof is 7. Level of proof is 7.

PROOF

1 [] -\(s(x,y,z1,z2,z3,z4,z5,z6) \mid s(y,x,z1,z2,z3,z4,z5,z6)\).

2 [] -\(s(z1,x,y,z2,z3,z4,z5,z6) \mid s(z1,y,x,z2,z3,z4,z5,z6)\).

3 [] -\(s(z1,z2,x,y,z3,z4,z5,z6) \mid s(z1,z2,y,x,z3,z4,z5,z6)\).

63
This output says that OTTER took 11.29 seconds and generates 1498 clauses to find a proof. From the proof we can easily extract the following sequence of moves which solves the problem:

From $s(3,1,4,6,5,2,8,7)$, 

(by rule 7) $\Rightarrow s(3,1,4,6,5,2,7,8)$, 

(by rule 5) $\Rightarrow s(3,1,4,6,2,5,7,8)$, 

(by rule 4) $\Rightarrow s(3,1,4,2,6,5,7,8)$, 

(by rule 5) $\Rightarrow s(3,1,4,2,5,6,7,8)$,
(by rule 3) ⇒ s(3, 1, 2, 4, 5, 6, 7, 8),
(by rule 1) ⇒ s(1, 3, 2, 4, 5, 6, 7, 8),
(by rule 2) ⇒ s(1, 2, 3, 4, 5, 6, 7, 8).

The previous example, which took 12 seconds on a Sparc II workstation, required 7 steps to move the pieces of the initial board to their home positions. Unfortunately, not all boards can be solved as easily. The totally reversed board s(8, 7, 6, 5, 4, 3, 2, 1) requires \((1 + 2 + \ldots + 7) = 28\) steps. Since there are 7 rules which can be applied to any board, a simple breadth-first search would investigate an impossible \(7^{28} > 10^{23}\) boards before finding a solution. Redundancy checks such as subsumption which is used by OTTER reduce the number of boards investigated to the number of possible boards \(8! \approx 4 \times 10^4\). However the cost of storing and checking this many boards is still prohibitive. Thus we need effective strategies for reducing the size of the search space.
CHAPTER 5
SEARCH STRATEGIES

In this chapter we introduce several search strategies which have proven effective for search spaces with as many as $10^{20}$ nodes. We will apply these strategies to various problems and puzzles in later chapters.

5.1 One-Way Search vs. Two-Way Search

If solving a problem requires $2m$ steps and if applying the rules to any board generates a new boards, then a simple breadth-first search space will have size $a^{2m}$. But if we know the home board, we can accomplish the same objective by searching forward from the initial board for $m$ steps and then searching backward from the home board for $m$ steps until a common board is found. If the search directions are symmetrical, the search space for this two-way search has size $a^m + a^m = 2a^m$. The two-way search vs. one-way search ratio is $2a^m/a^{2m} = 2/a^m$. When $a = 7$ and $2m = 28$ as in our previous simple board puzzle with totally reversed initial board, this ratio is $3/343$. Thus two-way search is much more efficient unless the two search directions are extremely asymmetric. A graphic comparison of one-way and two-way search spaces is depicted in the following figure.
Figure 8. Comparison of Search Spaces.

The unscrambling problem at the end of the previous section used one-way search (hyperresolution in OTTER), took 11.29 seconds and generated 1498 clauses. Using two-way search (binary resolution in OTTER) the same problem was solved in 1.19 seconds and generated only 256 clauses. For the totally reversed board though, even two-way search fails.

5.2 The Subgoal Strategy

5.2.1 The Basic Idea

While moving all 8 pieces to their home positions in one search is hard, moving a single piece to its home position is not. It takes at most 7 steps to move piece 1 to its home position. As above, this will take about one second (by two-way search) since all 7-step searches take about the same time. This suggests breaking the goal of moving all pieces to their home positions into the subgoals of moving pieces into positions one at a time. After achieving the subgoal of putting 1 in its place, our next subgoal is to get 2 in place while preserving 1’s position, then to get 3 in place
while preserving 1’s and 2’s positions, etc. Formally, this subgoal strategy divides our puzzle problem into the following subproblems:

\[
\text{rules+initial} \vdash \exists x_2 x_3 x_4 x_5 x_6 x_7 x_8 \ s(1, x_2, x_3, x_4, x_5, x_6, x_7, x_8),
\]

\[
\text{rules+s}(1, a_2, a_3, a_4, a_5, a_6, a_7, a_8)
\]

\[
\vdash \exists x_2 x_4 x_5 x_6 x_7 x_8 \ s(1, 2, x_3, x_4, x_5, x_6, x_7, x_8),
\]

\[
\text{rules+s}(1, 2, b_3, b_4, b_5, b_6, b_7, b_8) \vdash \exists x_4 x_5 x_6 x_7 x_8 \ s(1, 2, 3, x_4, x_5, x_6, x_7, x_8),
\]

......

Where \( s(1, a_2, a_3, a_4, a_5, a_6, a_7, a_8) \) and \( s(1, 2, b_3, b_4, b_5, b_6, b_7, b_8) \) are the solutions for the first and the second subgoals, respectively. We will call these solutions to subgoals subgoal solutions.

For goals let \( s(1, 2, 3, 4, ****) \) stand for

\[
\exists x_5 x_6 x_7 x_8 \ s(1, 2, 3, 4, x_5, x_6, x_7, x_8).
\]

Hence for goals, *’s are to be replaced by distinct existentially quantified variables.

In this way the totally reversed board can be solved with the following three subgoals:
initial : $s(8, 7, 6, 5, 4, 3, 2, 1)$,

$subgoal_1 : s(1, \ast \ast \ast \ast \ast \ast \ast)$,

$subgoal_2 : s(1, 2, \ast \ast \ast \ast \ast \ast \ast)$,

$subgoal_3 : s(1, 2, 3, \ast \ast \ast \ast \ast \ast \ast)$,

$home : s(1, 2, 3, 4, 5, 6, 7, 8)$,

$rules : s(x, y, \ast \ast \ast \ast \ast \ast \ast) \Rightarrow s(y, x, \ast \ast \ast \ast \ast \ast \ast)$,

\[ s(\ast, x, y, \ast \ast \ast \ast \ast \ast \ast) \Rightarrow s(y, x, \ast \ast \ast \ast \ast \ast \ast) \]

\[ s(\ast \ast, x, y, \ast \ast \ast \ast \ast \ast \ast) \Rightarrow s(y, x, \ast \ast \ast \ast \ast \ast \ast) \]

\[ s(\ast \ast \ast, x, y, \ast \ast \ast \ast \ast \ast \ast) \Rightarrow s(y, x, \ast \ast \ast \ast \ast \ast \ast) \]

\[ s(\ast \ast \ast \ast, x, y, \ast \ast \ast \ast \ast \ast \ast) \Rightarrow s(y, x, \ast \ast \ast \ast \ast \ast \ast) \]

\[ s(\ast \ast \ast \ast \ast, x, y) \Rightarrow s(y, x, \ast \ast \ast \ast \ast \ast \ast) \]

Using OTTER to solve the goals, we get the following results:

$rules + s(8, 7, 6, 5, 4, 3, 2, 1) \vdash subgoal_1$ by finding $s(1, 8, 7, 6, 5, 4, 3, 2)$ in 15 seconds,

$rules + s(1, 8, 7, 6, 5, 4, 3, 2) \vdash subgoal_2$ by finding $s(1, 2, 8, 7, 6, 5, 4, 3)$ in 7 seconds,

$rules + s(1, 2, 8, 7, 6, 5, 4, 3) \vdash subgoal_3$ by finding $s(1, 2, 3, 8, 7, 6, 5, 4)$ in 3 seconds,

$rules + s(1, 2, 3, 8, 7, 6, 5, 4) \vdash s(1, 2, 3, 4, 5, 6, 7, 8)$ in 101 seconds.
Thus with just three subgoals, the totally reversed board can be solved in 126 seconds. Here our subgoals filled the first three positions in the order they occur in the relation. Harder puzzles such as Rubik’s cube may require a different ordering of the subgoals.

Subgoals do not always help and their use might lead to incompleteness. A solution to a subgoal may be a dead end. This in fact is a problem for puzzles such as central and triangular solitaire. However completeness is not lost for puzzles such as the 15-puzzle, TopSpin, masterball and Rubik’s cube. In all these puzzles the rules are invertible, i.e., the inverse of each rule is derivable. Thus any unhelpful solution to a subgoal can, theoretically, be undone.

The importance of using subgoals is well-known. Using subgoals is like using lemmas in proving mathematical theorems. But there are no general methods for producing useful subgoals. This is one of the major unsolved problems of theorem proving. However, for all our puzzles, a suitable list of subgoals can be produced by reversing logical implication.

If $\psi \vdash \theta$, then in some sense $\theta$ is a special case of $\psi$. For example $\psi(a) \vdash \exists x \psi(x)$ and $\exists x \psi(x)$ is a special case of $\psi(a)$. Hence proving $\theta$ might be an intermediate step toward the proof of $\psi$.

The following three methods of producing subgoals are based on the reversal of three fundamental logical implications.
5.2.2 Goals with Constants

Most of our puzzles and some of our applications have a final goal clause (the home clause) with one or more constants. If the home clause is in the form \( \phi(a, b, c, d) \), then

\[
\phi(a, b, c, d) \vdash \exists z \phi(a, b, c, z),
\]

\[
\exists z \phi(a, b, c, z) \vdash \exists y z \phi(a, b, y, z),
\]

\[
\exists y z \phi(a, b, y, z) \vdash \exists x y z \phi(a, x, y, z).
\]

Thus the problem

\[
\text{initial} : \phi(...),
\]

\[
\text{home} : \phi(a, b, c, d)
\]

can be divided into four subproblems

\[
\text{initial} : \phi(...),
\]

\[
\text{subgoal}_1 : \phi(a, ** *),
\]

\[
\text{subgoal}_2 : \phi(a, b, **),
\]

\[
\text{subgoal}_3 : \phi(a, b, c, *),
\]

\[
\text{home} : \phi(a, b, c, d),
\]

\[
\text{rules} : \quad ...
\]

When subgoals are ordered as above by the positions of the constants in the home clause, the subgoals are said to be in canonical order. In general the first constant instantiated need not be the first, e.g., \( \text{subgoal}_1 \) could be \( \phi(***, d) \). However for all the puzzles we solve, the subgoal sequences share the property of having more and more existentially quantified variables (represented by *'s) instantiated by the constants of the home goal.
5.2.3 Goals with Conjunctions

Sometimes the final clause is a conjunction in the form $\phi_1 \land \phi_2 \land \phi_3$. For example, the final clause for sorting 4 distinct integers is

\[ \exists x y z w (\phi(x, y, z, w) \land \phi_1 \land \phi_2 \land \phi_3) \]

where

\[ \phi_1 : x < y \land x < z \land x < w, \]
\[ \phi_2 : y < z \land y < w, \]
\[ \phi_3 : z < w. \]

Clause $\phi_1$ guarantees that the integer $x$ in the first position is the smallest of the four integers, $\phi_1 \land \phi_2$ guarantees that in addition, the second position integer $y$ is the next smallest, and so on. Since $\phi_1 \land \phi_2$ is weaker than $\phi_1 \land \phi_2 \land \phi_3$ and $\phi_1$ is weaker than $\phi_1 \land \phi_2$, $\phi_1$ and $\phi_1 \land \phi_2$ may be used as subgoals in proving $\phi_1 \land \phi_2 \land \phi_3$ as followings:

- initial : $\phi(...)$,
- subgoal\(_1\) : $(\phi \land \phi_1)(...)$,
- subgoal\(_2\) : $(\phi \land \phi_1 \land \phi_2)(...)$,
- home : $\psi(...)$.  

5.2.4 Disjunctive Subgoals

If the subgoals produced by the above methods are difficult to achieve, they themselves may need to be subdivided into subgoals. For example, achieving the first goal $s(1,**\ldots)$ of getting 1 into its home position in a 20-place unscrambling problem can require 19 moves using all 19 rules. This is not a feasible subgoal. A
feasible sequence of subgoals is to first move the 1 to within the first 15 places, then to within the first 10 places, then to within the first 5 places, and finally to its home position. In our 8-place unscrambling problem the subgoal of getting 1 in the first place can similarly be divided into two steps with an added disjunctive subgoal as follows:

\[
\text{initial} : s(\ldots),
\]

\[
\text{subgoal}_1' : s(1,\cdots) \lor s(\cdots,1,\cdots),
\]

\[
\text{subgoal}_1 : s(\cdots).
\]

If the initial board is the totally reversed board \(s(8,7,6,5,4,3,2,1)\), then achieving \(\text{subgoal}_1\) by itself takes 15 seconds. With the added \(\text{subgoal}_1'\), the time is 0.88 seconds for \(\text{subgoal}_1'\) and 0.82 seconds for \(\text{subgoal}_1\) for a total time of 1.7 seconds.

In general, since \(\theta \lor \phi\) is logically weaker than \(\phi\), \(\theta \lor \phi\) is always a potential subgoal for proving \(\phi\). It is most likely to be useful when \(\theta \implies \phi\) is provable. For example, if the final goal \(\phi\) can be achieved from \(\theta_2\) and if \(\theta_2\) or \(\phi\) can be achieved from \(\theta_1\), then the task of proving \(\phi\) can be divided into the subproblems:

\[
\text{initial} : \psi,
\]

\[
\text{subgoal}_1 : \theta_1 \lor \theta_2 \lor \phi,
\]

\[
\text{subgoal}_2 : \theta_2 \lor \phi,
\]

\[
\text{home} : \phi.
\]

Successful examples of applying this disjunction strategy include the 15-puzzle and Rubik's cube.

Note that even when \(\theta_1 \implies \phi\) and \(\theta_2 \implies \phi\) are provable, we still include the goal
clause $\phi$ in each disjunction since it is possible that it may be easier to achieve $\phi$ directly from initial board than from either $\theta_1$ or $\theta_2$.

5.3 Using Multiple Subgoal Solutions

As noted above, a solution to a subgoal is not necessarily useful in achieving the final goal. Some solutions to a subgoal may be farther from the final goal than the initial board. In other cases subgoal solutions may be dead ends from which the final goal can not be achieved.

An effective solution to this problem is to produce multiple solutions for each subgoal. For some of our problems, this multiple subgoal solution strategy greatly improves the odds of finding a useful subgoal solution.

As an example, suppose we have 10 subgoals and for each subgoal the probability of generating a dead-end solution is 0.3. If only one solution is generated per subgoal, the probability of achieving the final goal is: $(1 - 0.3)^{10} \approx 0.028$. On the other hand, if we generate 10 solutions to each subgoal, then the probability that all the 10 solutions are dead-ends is $0.3^{10}$. Thus after 10 steps, the probability that at least one solution achieves the final goal is

$$(1 - 0.3^{10})^{10} \approx 0.99994.$$ 

Using multiple solutions to each subgoal is more efficient than repeating the search multiple times with a single randomly generated solution for each subgoal. With the same 0.3 probability of getting a dead-end, the probability that a single-solution-per-subgoal search achieves the final goal is $(1 - 0.3)^{10} \approx 0.028$ and thus we
can expect to repeat the search for $\frac{1}{0.006} \approx 36$ times before achieving a final solution. When the overhead of repeated searches is factored in, repeating the search 36 times takes four times as long as finding 10 solutions per subgoal.

This multiple solution strategy is used in solving central solitaire and its triangular and continental versions. In all of these puzzles, pegs are removed by checker-like jumps with the goal of removing all but one peg. For these puzzles, solutions to subgoals often lead to dead-end boards with widely scattered pegs for which no jumps are possible. When we generate a dozen solutions for each subgoal, especially for the later subgoals, we usually generate at least one solution which reaches the final goal.

5.4 The Rule Restriction Strategy

A search with $n$ steps using $m$ rules can have a search space of size $m^n$. Two-way search and searches with subgoals reduce the exponent $n$. The size of the search space can also be reduced by reducing the base $m$, i.e., the number of rules used in the search.

In the unscrambling problem with the totally reversed initial board, the third subgoal $subgoal_3$ is solved with the solution $s(1, 2, 3, 8, 7, 6, 5, 4)$, and the final goal $s(1, 2, 3, 4, 5, 6, 7, 8)$ is reached from $subgoal_3$ by 8 moves in 101 seconds. The final goal can be reached from this solution without moving pieces 1, 2, and 3. Thus we can restrict the rules used from the 7-element set of all rules to the 4-element subset of rules which do not move the first three pieces. This reduces the potential
search space for an 8-step solution from $7^8$ to $4^8$ ($\frac{7^8}{4^8} \approx 88$) and it reduces the time required for achieving the final solution from $s(1, 2, 3, 8, 7, 6, 5, 4)$ from 101 seconds (with 7 rules) to 1.27 seconds (with the subset of 4 rules).

For most of our puzzles, the subgoal of moving a particular piece to its home position usually requires only a proper subset of the set of all rules. Rule-restriction strategies for finding and using these proper subsets play an important role in solving these puzzles.

A more complicated rule restriction is to add conditions which restrict the application of a rule. Randomly applying a 'legal move' to a board of the unscrambling problem is just as likely to scramble as unscramble the board. In the last stages of a solution for an unscrambling, a randomly applied rule has a very high probability of undoing a previously achieved goal. Now suppose the rules for the unscrambling problem are restricted so that they apply only when the affected pieces are not in the correct order. Thus the first two rules would become:

\[
\text{rules : } (x > y) \land s(x, y, * \ldots) \Rightarrow s(y, x, * \ldots),
\]

\[
(x > y) \land s(*, x, y, * \ldots) \Rightarrow s(*, y, x, * \ldots).
\]

Using these restricted rules greatly reduces the search space size for one-way forward-chaining search.

Selecting too small a subset of rules or the wrong conditional restrictions can lead to incompleteness. Not selecting a small enough subset can result in an unmanageably large search space. For many of our puzzles such as the unscrambling
problem, the 15-puzzle and the solitaire puzzles, reasonable rule subsets are not hard to find for each subgoal. For Rubik's cube, finding suitable subsets of rules is quite difficult. For these harder problems appropriate subsets of rules can be found by a learning process or by exhaustive search.

The subset of rules needed for a particular goal can be found by a simple learning algorithm by solving the problem with easy initial boards and noting which rules are never used for the given goal. Eliminating these unused rules can speed up the search process enough that harder initial boards can be solved. With Rubik's cube, such observation shows that only the rules which rotate the outer layers are needed for subgoals which put corner pieces in place and only the rules which rotate the middle layers are needed for subgoals which put the middle face pieces in place.

For Rubik's cube, the above subsets are not small enough to reach the hardest goals. The only way we were able to solve the problem of finding small enough complete rule subsets was by exhaustive search. One-element subsets were clearly too small to solve these difficult subgoals, so we started by trying all $\binom{6}{2} = 15$ two-element subsets of the 6 basic rules, then trying all $\binom{6}{3} = 20$ three-element subsets. This tactic succeeded. Every instance of each subgoal was always solvable with some two- or three-element subset of the set of 6 rules.
5.5 Discovering New Rules

Sometimes the sequence of moves to achieve a subgoal is so long that it takes a long time even with properly selected rule subsets. In many of these cases, replacing the original rules with appropriately selected new rules can reduce the search time and space significantly.

Consider our unscrambling problem with the totally reversed initial board:

\[
\text{initial : } s(8, 7, 6, 5, 4, 3, 2, 1),
\]

\[
\text{subgoal}_1 : s(1, * * * * * * *),
\]

\[
\text{subgoal}_2 : s(1, 2, * * * * * * *),
\]

......

To achieve \(\text{subgoal}_1\) the program needs to move 1 to the first position of \(s\). In this case the problem of achieving \(\text{subgoal}_1\) can be represented generically by the following problem:

\[
\text{initial : } s(8, 7, 6, 5, 4, 3, 2, 1),
\]

\[
\text{goal : } s(1, * * * * * * *).
\]

Assume the solution to the above problem is \(s(1, 8, 7, 6, 5, 4, 3, 2)\), then this solution proves a fact using the the axioms from the set of original rules. This proven fact can be represented as a new rule

\[
s(8, 7, 6, 5, 4, 3, 2, 1) \Rightarrow s(1, 8, 7, 6, 5, 4, 3, 2)
\]

and it can be reused in proving other unscrambling problems. Since the con-
stants $8,7,6,5,4,3,2$ do not occur in the goal, they may be replaced by variables $x_1, x_2, \ldots, x_7$. This gives a generic rule for moving 1 from position 8 to its home position 1:

$$s(x_1, x_2, x_3, x_4, x_5, x_6, x_7, 1) \Rightarrow s(1, x_1, x_2, x_3, x_4, x_5, x_6, x_7).$$

Similar generic rules exist for subgoal_1 for the cases with 1 in the 2nd, 3rd, 4th, 5th, 6th, and 7th positions in the initial board. This set of 7 generic rules achieves subgoal_1 in at most one step for any initial board. Similar sets of generic rules work for subgoal_2, subgoal_3, etc.

Note that with the generic rules in place of the original rules, each subgoal is achieved in one step, and the total number of generic rules for this problem is $7 + 6 + 5 + 4 + 3 + 2 + 1 = 28$. Each of the new rules represents a sequence of original rules. So a solution using the new rules can easily be translated into a solution using the original rules.

In general, for a board puzzle of $n$ positions, if we divide the problem into $n$ subgoals each corresponding to one position in the board, then for each subgoal, depending on the position of the piece to be moved to its home, we have at most $n - 1$ generic rules. For each subgoal, if we replace the original rules by these $o(n)$-many generic rules, then the subgoal can be solved in at most one step, and this step can be found in $o(n)$ time. Therefore the whole problem is solvable in $o(n^2)$ time.

Unless $n$ is small, the size of the set of new rules can be quite large. For the
above unscrambling problem of 8 positions, \( n = 8 \), there are 7 original rules and
\( (1 + 2 + 3 + 4 + 5 + 6 + 7) = 28 \) new rules. If \( n = 20 \), then there are 19 original rules
and \( (1 + 2 + \ldots + 19) = 190 \) new rules. It is not easy to generate and to manage all
these 190 new rules.

Thus we use generic rules to replace original rules only when the set of generic
rules for the subgoal is not large and when the set of original rules does not easily
achieve the subgoal. These cases often occur in the last stages of a solution. The
last subgoals are often difficult since all previously achieved subgoals must
be preserved. However, since there are relatively few positions for the pieces placed
by the final subgoals, the set of generic rules for these subgoals is relatively small.

We use generic rules in solving Rubik’s cube and masterball. In Rubik’s cube,
using a set of 5 generic rules reduces the search time for one subgoal from 2,500
seconds to less than 5 seconds.
Many puzzles, including all those in this chapter, consist of a two or three dimensional arrangement of objects, often on a board. The objects can be moved or removed by moves from a given set of legal moves. The pieces may be identical (central solitaire), or colored (Rubik's cube), or numbered (15-puzzle). For these board puzzles, there is an initial configuration and a final goal configuration. A solution consists of a sequence of moves which carries the initial configuration to the goal configuration.

All the puzzles studied here have been solved by hand. The methods of solution have usually been useful only for the puzzle they solve and not for other puzzles. Here we are interested in applying the general search strategies we developed in Chapter 5 to solve these puzzles.

6.1 The 15-Puzzle

The 15-puzzle was invented by Sam Loyd. William E. Story completely solved this puzzle using an invariant for the board (see American Journal of Mathematics, vol. 2, 399-404, 1879). Here we use our search strategies to solve this problem.

The 15-puzzle consists of 15 tiles numbered from 1 to 15. These tiles rest in a 4 × 4 tray where there is one unoccupied position. We call this unoccupied position
the hole. The tiles can be moved up, down, left, and right provided an adjacent hole allows the move. Given a tray with the numbered tiles in a scrambled arrangement, the goal is to find a sequence of moves which arrange the tiles in increasing numerical order with the hole in the last position. One particular problem we consider is that of finding a sequence of moves which carries the reversed board on the left to the unscrambled goal on the right:

\[
\begin{array}{cccc}
15 & 14 & 13 & 12 \\
11 & 10 & 9 & 8 \\
7 & 6 & 5 & 4 \\
3 & 2 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
13 & 14 & 15 & 0 \\
\end{array}
\]

6.1.1 The Set of Rules

In one move of the 15-puzzle one may move a tile to the hole provided the tile is adjacent to the hole. We use a 16-place relation \( r \) to represent the board, where the first 4 places of \( r \) represent the first row of the tray, the second 4 places of \( r \) represent the second row of the tray, and so on. We use constants 1, 2, ..., 15 to represent the tiles with labels 1, 2, ..., 15 respectively, and use the constant 0 to indicate the hole position. Thus, the above problem can be formalized as the problem of proving the implication

\[
\begin{align*}
r(15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0) \\
\Rightarrow r(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0).
\end{align*}
\]

Depending on the position of the hole (the constant 0), there are 2 legal moves if the hole is at one of the 4 corners in the tray, 3 legal moves if the hole is at one
of the eight edge positions in the tray, and 4 legal moves if the hole is in at any of the 4 center positions. In all there are 48 legal moves, each of which is formalized as an if-then rule. For example, the if-then rules for the two moves for the hole at the right bottom corner are:

\[
\begin{align*}
    r(\star \star \star, \star \star \star, \star \star, x, \star \star, 0) & \Rightarrow r(\star \star \star, \star \star \star, \star \star, 0, \star \star, x), \\
    r(\star \star \star, \star \star \star, \star \star, x, 0) & \Rightarrow r(\star \star \star, \star \star \star, \star \star, x, 0).
\end{align*}
\]

The first rule moves the hole up one position, the second rule moves the hole one position to the left.

6.1.2 Solving the 15-Puzzle

Since the home clause \(r(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0)\) for the final goal is a predicate with constants, we may apply the subgoal strategy for goals with constants. This gives the following list of 14 subgoals.

\[
\begin{align*}
    subgoal_1 & : r(1, \star \star \star \star \star \star \star \star \star \star \star \star \star \star \star \star \star \star), \\
    subgoal_2 & : r(1, 2, \star \star \star \star \star \star \star \star \star \star \star \star \star \star \star \star \star), \\
    subgoal_3 & : r(1, 2, 3, \star \star \star \star \star \star \star \star \star \star \star \star \star \star \star \star \star), \\
    \ldots \ldots \\
    subgoal_{14} & : r(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, *, *),
\end{align*}
\]

\[home : r(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0)\).

Each subgoal moves one particular tile to its home position. Once all the 15 tiles are at their destinations, the goal clause \(home\) is established.
These subgoals suffice for workstations with enough memory. For 16 Mbyte PCs, some of the subgoals are too difficult. The difficulty occurs in realizing the first subgoals and the last subgoals. The first subgoal, for example, is difficult when tile 1 is at the right bottom corner while the hole is at the upper left corner of the board. It takes around 20 steps to deduce \textit{subgoal}_{1}. This is difficult even with two-way search.

![Board with tiles and hole](image)

We overcome this difficulty using the disjunctive subgoal strategy to add an intermediate disjunctive subgoal before \textit{subgoal}_{1}. For example, while it can be hard to move tile 1 to its home position, it is relatively easy to move tile 1 to the nearest of the following 5 positions marked by X's depicted in the following board:

![Board with Xs](image)

Thus, the problem of achieving \textit{subgoal}_{1} can be split into steps by adding \textit{subgoal}'_{1} as follows:
subgoal_{1}^{'} : r(1\ast\ast,\ast\ast\ast,\ast\ast\ast) \lor r(\ast\ast\ast,1\ast,\ast\ast\ast) \lor r(\ast\ast\ast,\ast\ast\ast,1\ast\ast) \lor r(\ast\ast\ast,\ast\ast\ast,\ast\ast\ast,1\ast),

subgoal_{1} : r(1\ast\ast,\ast\ast\ast,\ast\ast\ast).}

With the above new subgoal, at most 12 steps are needed to achieve subgoal_{1}^{'} from any initial tray, and the deduction of subgoal_{1} from subgoal_{1}^{'} also needs no more than 12 steps. Tile 1 can be quickly moved to its home position with the new subgoals. Due to the exponential growth of the search space, two 12 step goals are much easier than one 20 step goal.

The difficulty of realizing the last subgoals is due to the need to preserve previously established subgoals. The rule restriction strategy solves this problem. It is reasonable to assume that the tiles in the first row do not need to be moved when moving third-row tiles into their home positions. This suggests that for the subgoals of placing tiles into third-row positions, the rules used can be restricted to rules which do not move first-row tiles. Similarly, for subgoals which place tiles in the fourth row, the rules used can be restricted to rules which do not move tiles in the first or second rows. By restricting the rules used to these proper subsets, the search time for these subgoals is greatly reduced.

6.1.3 The Reachability Problem for Trays

Now we can answer a more general question: when is a 15-puzzle board reachable from another?
Lemma 6.1. \textit{rules} + \textit{initial} $\vdash r(1, \blank, \blank, \blank, \blank, \blank, \blank, \blank)$ for any initial board \textit{initial} of the 15-puzzle.

\textbf{Proof.} Tile 1 has 16 possible positions in the initial tray and for each position of tile 1, the hole has 15 possible positions. So there are $16 \times 15 = 240$ possible pairs of positions for tile 1 and the hole. For each of these 240 cases, there is a deduction of \textit{Rules} $\vdash \textit{initial} \Rightarrow r(1, \blank, \blank, \blank, \blank, \blank, \blank)$. These generic rules for tile 1 guarantee the correctness of Lemma 6.1. $\Box$

Lemma 6.2.

\textit{rules} + \textit{initial} $\vdash r(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0) \lor r(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 14, 0)$.

\textbf{Proof.} Similar generic rules for tiles 2, 3, ..., 13 can be found for all possible boards. Applying those generic rules step by step until tile 13 moves tiles 1 through 13 to their home positions, leaving tiles 14, 15, and the hole in the last 3 positions. Moving the hole to the last position gives one of the above two final boards. $\Box$

Lemma 6.3.

\textit{rules} + $r(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0)$ \not\vdash r(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 14, 0).

\textbf{Proof.} William Story gave the following proof for Lemma 6.3. First color the tray in the red and white checkerboard pattern as indicated below.

\begin{center}
\begin{tabular}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c}
\hline
r & w & r & w \\
\hline
w & r & w & r \\
\hline
r & w & r & w \\
\hline
w & r & w & r \\
\hline
\end{tabular}
\end{center}
Then any move changes the color of the hole position. We define the color parity of the hole to be 0 if the hole is at a red position and 1 if the hole is at a white position. Every board is a permutation of $<0, 1, 2, 3, ..., 15>$ and has either an even (0) or an odd (1) permutation parity. Define the total parity of the board to be

$$(\text{parity of hole} + \text{parity of board}) \mod 2.$$  

Note that any legal move of the 15-puzzle changes both the parity of the hole and the parity of the board simultaneously, so the total parity is invariant. However, the two boards in this lemma do not have the same total parity so neither can reach the other by any sequence of legal moves. □

Since all moves for the 15-puzzle are invertible, board A is reachable from board B if and only if B is reachable from A. The above three lemmas and the transitivity of the reachability relation imply that any two boards of the 15-puzzle are reachable from each other if and only if they have the same total parity.

### 6.2 Central Solitaire

Central solitaire (also called Hi-Q) is a puzzle with 33 positions in a cross-shaped board. Initially there is one vacant position, the other 32 positions are occupied by pegs, one peg for each position. In a legal move, a peg may jump horizontally or vertically over an adjacent peg to a vacant position. The peg that has been jumped over is then removed. In the following figure the left board depicts a typical initial board of the puzzle, where '0' indicates a vacant position and '1' indicates an
occupied position. For the sake of reference, we name each position on the board 1, 2, ..., v, w, x as indicated in the right board.

\[
\begin{array}{cccccc}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1 \\
\end{array}
\quad
\begin{array}{cccccccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 & a & b & c & d \\
e & f & g & h & i & j & k \\
l & m & n & o & p & q & r \\
s & t & u \\
v & w & x \\
\end{array}
\]

*Figure 9. A Configuration of Central Solitaire.*

The standard problem is: given a central solitaire board with the center position vacant, is there a way of moving the pegs so that in the last board only a single peg at the center is left?

6.2.1 The Representation

We use a 33-place relation \( H \) to represent the central solitaire board. If a position is occupied by a peg, we fill the corresponding place with the constant ‘1’; if it is vacant, we fill the place with the constant ‘0’.

For any three horizontally consecutive positions in the board, there are two legal ways of moving pegs on them:

‘110’ \( \Rightarrow \) ‘001’, ‘011’ \( \Rightarrow \) ‘100’.

There are two rules which formalize these two moves. Similarly for any three
vertically consecutive positions in the board, there are two rules corresponding the
two legal moves on these positions. In all there are 76 such rules for the 76 possible
moves.

We can get a small set of rules by taking advantage of the symmetry of the puzzle. Since the pegs are identical, and the board is symmetric, we may consider only one orientation of moves on some half board, and include two rules for rotating the board and flipping the board. But experiments with this smaller set of rules show that even though the number of rules for this set is much smaller than that for the original set, the performance is actually worse.

6.2.2 The Negative Results

We first show when a final goal is not reachable from an initial board.

We say a board is an initial board if it has only one vacant position, a final board
if it has only one peg on it.

We alternately color the positions of the central solitaire board with 3 colors r
(red), w (white), and b (blue) along the parallel line of slope \(-1\) in the following way.
Each color colors 11 positions. If an initial board has its vacant position colored \( r \) (\( w \), or \( b \)), we call it an initial red (white, or blue) board. If a final board has its last peg at a position colored \( r \) (\( w \), or \( b \)), we call it a final red (white, or blue) board. Notice that any three consecutive positions (along the vertical or horizontal directions) have different colors, so any jump involves three positions of distinct colors. Thus, in any jump, we replace two pegs with distinct colors by a peg of the third color. We use a monomial to represent the distribution of the colors in a board: \( b^m r^n w^k \) means \( m \) blues, \( n \) reds, and \( k \) whites in the board. The change in the monomial produced by a jump corresponds to multiplying the monomial by one of the factors \( br^{-1} w^{-1} \), \( rw^{-1} b^{-1} \), or \( wb^{-1} r^{-1} \).

If we are given an initial red board, then the monomial for the initial board is \( b^{11} r^{10} w^{11} \). If this initial board can reach a final white board, there must be nonnegative integers \( x, y, z \) such that

\[
b^{11} r^{10} w^{11} (br^{-1} w^{-1})^x (rb^{-1} w^{-1})^y (wr^{-1} b^{-1})^z = w.
\]
Comparing the exponents we get

for b : \(11 + x - y - z = 0\),

for r : \(10 + y - x - z = 0\),

for w : \(11 + z - x - y = 1\).

But there is no integer solution \(x, y, z\) for the above equations. So no initial red board can reach any final white board. By the symmetry of the equations, no initial board of color \(x\) can reach a final board of a different color \(y\).

We can also color the board along parallel lines of slope 1. Since this coloring is a \(90^\circ\) rotation of the above one, all the above results hold for this new coloring by symmetry. Notice that two positions have the same color in the two colorings if and only if both their \(x\)-coordinates and \(y\)-coordinates differ by multiples of 3.

Thus we conclude:

**Theorem 6.4.** An initial board can not reach a final board if they are given different colors by the two colorings.

6.2.3 The Complete Solution for Central Solitaire

To finish the solution of this puzzle we use our reasoning strategies to show that an initial board can reach any final board which shares the same color in both colorings. In the following we use \(\tilde{B}\) to denote the board obtained by switching the filled and the vacant positions of board \(B\).

**Lemma 6.5.** If an initial board \(A\) can reach a final board \(B\), then the initial board \(\tilde{B}\) can reach the final board \(\tilde{A}\).
PROOF. Assume \( r_1, r_2, \ldots, r_\overline{30}, r_\overline{31} \) is a sequence of rules for deducing \( A \Rightarrow B \), then applying the rules in the sequence \( r_\overline{31}, r_\overline{30}, \ldots, r_2, r_1 \) accomplishes \( \overline{B} \Rightarrow \overline{A} \). □

Theorem 6.6. An initial board can reach a final board if and only if the two boards share the same color in the two colorings.

PROOF. Theorem 6.4 proves the 'only if' direction. For the 'if' direction we first prove the following lemma.

Lemma 6.7. The initial board with vacant position at \( h \) can reach the final boards with last pegs at positions 2, e, h, k, w respectively.

PROOF. In order to prove Lemma 6.7, it suffices to find a deduction for each of the following final boards where the potential positions 2, e, h, k, w of the last pegs are marked by X.

There are 76 rules for the puzzle. Except for the first and last several boards of a deduction, any intermediate board is likely to have more than a dozen possible
successors. Thus the search space grows rapidly and we need to use our search strategies to reduce its size.

The first strategy we use is the subgoal strategy. Since in the final board all but one positions are vacant, we set subgoals which clear the positions farthest from the last peg. For example, if the final peg is at center, we use 9 subgoals which successively clear the 9 outermost positions.

Since the moves in solitaire are not reversible, a solution for a subgoal may be a dead end which does not lead to any final board. This happens most often with solutions to the final subgoals. These often have several isolated pegs which can not be removed.

In order to overcome this difficulty, we use the multiple solution strategy. We generate multiple solutions (20 in this case) for each subgoal until there are about 10 pegs left on the board. Many of the last 20 subgoals do not lead to any final board, but usually there are some solutions which can reach the desired final boards. With comparable subgoals and multiple solutions the final boards with last pegs at positions 2, e, k, and w can also be proved to be reachable from the initial board. So Lemma 6.7 holds. □

Similarly, OTTER finds deductions for the following:
(1) The initial board with position o vacant can reach the final boards with last pegs at positions 5, l, o and r.

(2) The initial board with position p vacant can reach the final boards with last pegs at positions 6, m and p.

(3) The initial board with position t vacant can reach the final boards with last pegs at positions 7, a, d and t.

(4) The initial board with position u vacant can reach the final boards with last pegs at positions 8, b and u.

(5) The initial board with position w vacant can reach the final boards with last pegs at positions 2, e, h, k and w.

(6) The initial board with position x vacant can reach the final boards with last pegs at positions 3, f, i and x.

Since any position on the board can be rotated or reflected to one of positions h, o, p, t, u, w, x, Theorem 6.6 holds. □

In Appendix B we give a solution which carries the initial board with position w vacant to the final board with last peg at position h.

6.3 Triangular Solitaire

Triangular solitaire is a variation of central solitaire, and its solution is similar in many aspects. Therefore we will simplify our description whenever possible in this section. The layout of triangular solitaire is a triangular board consisting of 28
positions. The following figure depicts a typical initial board of triangular solitaire.

![Triangular Solitaire Board]

*Figure 11. A Configuration of Triangular Solitaire.*

As before, '1' indicates a peg in that position and '0' indicates that the position is vacant. In a legal move, a peg may jump over an adjacent peg to a vacant position and remove the jumped-over peg. Here the jump is along a direction parallel to one of the three bases of the triangular board. Thus there are up to 6 directions for a peg to jump. *Initial board, final board, etc.* are defined as in the central solitaire case. The standard problem for triangular solitaire is, can the initial board above reach a final board with the last peg at the center via some sequence of legal moves?

We use a 28-place relation to represent the board for triangular solitaire. Notice that in triangular solitaire a peg may jump in 6 directions rather than 4 as in central solitaire, so while triangular solitaire has fewer board positions than central solitaire does (28 vs. 33), it has more rules (90 vs. 76). So solving triangular solitaire is not necessarily easier than solving central solitaire.

Now we answer the general question: given an initial triangular solitaire board, what are the possible final boards it can reach?
We color the triangular solitaire board in the following way where \( r, w, b \) represent the red, white, and blue colors:

\[
\begin{array}{cccccc}
  r & w & b & r & w & b \\
  b & r & w & b & r & w \\
  w & b & r & w & b & r \\
  r & w & b & r & w & b \\
  b & r & w & b & r & w \\
  w & b & r & w & b & r \\
\end{array}
\]

\textbf{Figure 12. A Coloring of the Triangular Solitaire Board.}

There are 10 red positions, 9 white positions and 9 blue positions. Notice that with the above coloring, the colors of any three adjacent positions along any of the 6 possible legal move directions are distinct. So one legal move corresponds removing two pegs of different colors from the board and adding a peg of the third color.

We use monomials in \( r, w, b \) to represent the number of positions occupied by each of the three colors. By considering equations generated by a potential sequence of moves we can answer some questions about board reachability. For example, for an initial red board, if it can reach a final red board, then there should exist positive integers \( x, y, z \) such that

\[
r^9 w^9 b^9 (r^2 w^{-2} b^{-2})(w^y r^{-y} b^{-y})(b^z r^{-z} w^{-z}) = r.
\]

By comparing the exponents of both sides we get
for r: \( 9 + x - y - z = 1 \),

for w: \( 9 - x + y + z = 0 \),

for b: \( 9 - x - y + z = 0 \).

But the above system of equations has no solution. In particular, the standard problem is impossible because the center position in the standard problem is colored red.

Similar arguments show that:

(1) an initial red board can not reach any final board;

(2) an initial white board may reach only final blue boards;

(3) an initial blue board may reach only final white boards.

By finding sequences of moves from the initial board to the possible final boards for (2) and (3), OTTER proves that

(2') any initial white board can reach all final blue boards;

(3') any initial blue board can reach all final white boards.

The subgoal strategy and multiple solution strategy are used in the search.

Note: Irvin Roy Hentzel ([He]) studied the triangular solitaire problem in 1973. His paper partially solved the problem. His basic idea was to introduce a commutative group of four elements \( \{0, 1, p, q\} \) where 0 is the zero of this group, \( x + x = 0 \) for any element \( x \), and among nonzero elements \( \{1, p, q\} \), the sum of any two distinct elements is the third element. He then used a method similar to our coloring to assign each position a group element. In this way the parity of the board, defined as
the sum of elements on the board, is invariant throughout the jumps. His conclusion was that if the parity of a board is 0, then the board can not be reduced to a final board.

Our method of coloring and solving equations solves more problems than Henzel's method. Whenever Henzel's method rules out a reachability problem so does our method. However our method rules out some impossible reachability problems on which Henzel's method fails. For example, if there are only three occupied red positions in a board, our method correctly shows that this board can not reach any final board by showing the corresponding equations have no integer solution. But Henzel's method fails since if we assign the group element \( p \) to the red positions, then since the parity is \( p + p + p = p \), Henzel's answer would be "the board either can reach a final red board or it can not reach any final board".

6.4 TopSpin

TopSpin consists of a circular track with 20 pieces numbered 1, 2, ..., 20 placed in the track, and a turnstile which always holds 4 consecutive pieces. The following figure depicts the unscrambled board configuration of TopSpin.
There are three legal moves in TopSpin: slide all the pieces around the track one position in the clockwise direction, slide them one position in the counterclockwise direction, or flip the turnstile and its 4 pieces $180^\circ$. We will always picture the turnstile at the top of the circular track. Then sliding the pieces one position in either direction corresponds sliding pieces through the turnstile one position to the right or left. Thus we will refer to the moves as \textit{sliding left}, \textit{sliding right}, and \textit{flipping}; and denote them as L, R, and F respectively.

Some of the interesting problems for TopSpin are:

(1) unscramble the pieces, i.e., to put all the 20 numbers in order;

(2) turn the turnstile up side down while preserving the original order of the pieces;

(3) Swap two adjacent pieces without changing the positions of the other pieces.
6.4.1 The Representation

Starting from the leftmost position of the turnstile, from left to right we number the positions 1, 2, ..., 19, 20 in that order. We use a 20-place relation $S$ to denote the board for TopSpin and 20 constants 1, 2, 3, ..., 20 to name the 20 numbered pieces. In this way a configuration of TopSpin can be represented as $S(c_1, c_2, c_3, \ldots, c_{20})$ where $c_i \in \{1, 2, 3, \ldots, 20\}$. Corresponding to the three legal moves, we get the following three rules for TopSpin:

$$S(x_1, x_2, x_3, x_4, \ldots, x_{19}, x_{20}) \Rightarrow S(x_2, x_3, x_4, x_5, \ldots, x_{20}, x_1) \text{ (move L)}$$

$$S(x_1, x_2, x_3, x_4, \ldots, x_{19}, x_{20}) \Rightarrow S(x_{20}, x_1, x_2, x_4, \ldots, x_{18}, x_{19}) \text{ (move R)}$$

$$S(x_1, x_2, x_3, x_4, x_5, \ldots, x_{19}, x_{20}) \Rightarrow S(x_4, x_5, x_2, x_1, x_5, \ldots, x_{19}, x_{20}) \text{ (move F)}$$

6.4.2 Unscrambling the Pieces

Given any initial board with scrambled pieces on the track, the problem is to find a sequence of moves which unscrambles the pieces. Since the search space has $20!$ configurations in our representation, it is usually impossible to achieve the goal by a simple exhaustive search. For the unscrambling problem, we add the subgoals of moving piece 20 to its home position first, then moving piece 19 to its goal position, ..., and finally moving piece 2 (and so piece 1) to its home position.
Formally our subgoals are:

\[ \text{subgoal}_1 : S(**************, 20), \]
\[ \text{subgoal}_2 : S(**************, 19, 20), \]

......

\[ \text{subgoal}_{17} : S(**, 4, ..., 18, 19, 20), \]
\[ \text{subgoal}_{18} : S(**, 3, 4, ..., 18, 19, 20), \]
\[ \text{subgoal}_{19} : S(1, 2, 3, 4, ..., 18, 19, 20), \]

All but the last subgoal are easy to achieve. In solution of the next-to-the-last subgoal, \( \text{subgoal}_{18} \), OTTER found a sequence of moves \( F \ R \ F \ R \ F \ L \ F \ R \ F \ L \ F \ L \ F \ R \ F \ L \ F \ L \) which proves the following:

**Lemma 6.8.**

\[ S(a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t) \Rightarrow S(b, c, a, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t). \]

I.e., there is a sequence of moves which jumps piece \( a \) over pieces \( b, c \) without changing the position of any other piece.

Lemma 6.8 shows that one can cyclically permute three consecutive pieces without moving the remaining 17 pieces. This rule, written formally, is

\[ S(x, y, z, w_4, ..., w_{20}) \Rightarrow S(y, z, x, w_4, ..., w_{20}). \]

**Theorem 6.9.** Any two boards are reachable from each other.
PROOF: It is sufficient to show that any two adjacent pieces can be swapped without moving the other pieces. Without lost of generality assume we are given the board

\[ S(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20) \].

Then applying Lemma 6.8 repeatedly we can get

\[ S(1, 2, 3, 4, 5, 6, \ldots) \Rightarrow S(2, 3, 1, 4, 5, 6, \ldots) \Rightarrow S(2, 3, 4, 5, 1, 6, \ldots) \Rightarrow \ldots \]

\[ \Rightarrow S(2, 3, \ldots, 19, 1, 20) \Rightarrow S(2, 1, 3, 4, \ldots, 19, 20). \square \]

In the next section we will give a shortest path for swapping two adjacent pieces.

6.4.3 Two Hard Problems

One hard problem in TopSpin is turning the turnstile upside down without changing the order of the pieces. The brochure that came with the puzzle gave a 37-step solution for this problem. We have found a 31-step solution and have proved that it is the shortest possible. The rule-restriction and subgoal strategies do not help with this problem since there are no useful subsets of the three rules and no obvious intermediate subgoals.

Since we are interested in the orientation of the turnstile, we include one more flag place in the relation \( S \) (the 21-th place in the following clauses) to indicate the current orientation of turnstile. The flag place is filled with the constant ‘0’ or ‘1’ to indicate the two distinct orientations of the turnstile. The moves R and L do not change the flag, but move F always changes the value of the flag place. We found that the following sequence of moves accomplishes the task of inverting the
turnstile:

\[ RF L F R F L F L F R R F L F R F R F L L F R F L L F R F R. \]

That is, after applying this sequence of moves to the board

\[ S(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 0), \]

we get the board

\[ S(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 1) \]

with all the pieces remaining fixed while the flag changes.

Another hard problem is to switch two adjacent pieces while leaving the other pieces fixed. The brochure that came with the TopSpin puzzle gave a 49-step solution to this problem. We found a 41-step solution and have shown that it is the shortest. The following sequence of moves accomplishes this task:

\[ FL F R 17(F L) L FR. \]

We found the 41-step sequence and proved it to be minimal using Prolog. Since Prolog uses depth-first search strategy, we added another place in the relation to store the maximum search depth allowed for the program. To show that a path is of minimal length, we let the program exhaustively search all possible shorter paths, it found none.

103
6.5 Rubik’s Cube

Rubik’s cube, also called the magic cube, was invented by the Hungarian designer Erno Rubik ([BCR]). It consists of 26 small cubelets in a $3 \times 3 \times 3$ array, not counting the interior cubelet. The cube’s six faces are called front, back, left, right, top, and bottom, respectively, in this paper. For the unscrambled cube each of the six faces has a uniform color. Given a scrambled cube, the goal is to unscramble the cubelets so that all six faces are monocolored.

Many studies have been done about Rubik’s cube (see [BCR]). Here we solve it using our strategies.

6.5.1 The Representation of Rubik’s Cube

As depicted in the following figure, we use the Cartesian coordinate axes to indicate directions. Direction $x$ is from back to front, direction $y$ is from left to right, and direction $z$ is from bottom to top. We use $L, R, F, Bk, T, Bt$ to indicate the left, right, front, back, top, and bottom faces and layers, and use $M$ for the middle layer when the direction is clear.
Figure 14. A Configuration of Rubik’s Cube.

Of the 26 cubelets, 6 are central cubelets each having a single face, 12 are edge cubelets each having two faces, and 8 are corner cubelets each having 3 faces. The cube has, in total, 54 faces in 6 colors. We represent the three dimensional ‘board’ of Rubik’s cube with a 54-place relation $R$ which has a place for each face. For convenience we use the variables $x, y, z, u, v, w$ with subscripts to stand for the colors of the face positions at the top, left, front, right, bottom, and back. Thus the formula

$$R(x_1, x_2, ..., x_9, y_1, y_2, ..., y_9, z_1, z_2, ..., z_9,$$

$$u_1, u_2, ..., u_9, v_1, v_2, ..., v_9, w_1, w_2, ..., w_9)$$

represents a board of Rubik’s cube, where the order of the places for a given face in $R$ is the standard row-column matrix order of the faces from left to right then from top down when the face is rotated to the front around the $z$-axis or, for the top and bottom faces, around the $y$-axis. We use the constants $b, r, g, o, y, w$ to
represent the cube's six colors: blue, red, green, orange, yellow, and white. Thus the unscrambled Rubik's cube has the representation

\[ R(b, b, b, b, b, b, r, r, r, r, r, r, g, g, g, g, g, g, g, g, g, g, g, g, g, g, g, g, g, g, g, g, g). \]

There are \( 6 \times 4 = 24 \) ways a cube can be oriented with its axes parallel to the \( x, y \) and \( z \) axes. Thus each board configuration is equivalent, up to orientation, to 23 other configurations. To simplify our discussion we stipulate that the left-back-top corner cubelet (i.e., the \( x_1, y_1, w_3 \) cubelet in relation \( R \)) remain fixed. This stipulation selects a unique orientation for each configuration.

Each of our moves rotates a single layer a quarter of a turn. There are 6 possible quarter turns \( x \to y, x \to z, y \to x, y \to z, z \to x \) and \( z \to y \), each of which rotates one axis to another while leaving the third axis (the axis of rotation) fixed. For each of the 9 layers two opposite turns may applied. Thus there are \( 9 \times 2 = 18 \) moves. Since we have fixed the left-back-top corner, the moves that rotate the top layer, the left layer, and the back layer are not allowed. Thus we have just 12 basic moves. The following is the rule for the move that rotates the middle layer between the front and back layers in the clockwise \( z \to y \) direction:
We call this the MZY rule since it acts on a middle layer and rotates the positive z-axis to the positive y-axis. We use a similar notation to indicate the position of a cubelet in the cube. For example, corner LBkT means the corner position at the intersection of the left, back and top faces. This is the position of the fixed cubelet.

\[
\begin{align*}
& \quad s(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9, \\
& \quad z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, z_9, u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8, u_9, \\
& \quad v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8, w_9) \\
\implies & \quad s(x_1, x_2, x_3, y_8, y_5, y_2, x_7, x_8, x_9, y_1, y_4, y_3, y_4, v_5, v_6, y_7, v_6, y_9, \\
& \quad z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, z_9, u_1, x_4, u_3, u_4, x_5, u_6, u_7, x_6, u_9, \\
& \quad v_1, v_2, v_3, u_8, u_5, u_2, v_7, v_8, v_9, w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8, w_9).
\end{align*}
\]

Notice that any rotation of a particular layer is equivalent to three consecutive rotations of the same layer in the opposite direction, so 6 of the basic moves can generate the rest. But, for the sake of efficiency, we include all 12 basic moves in
our set of rules. They are

\{FZY, FYZ, MZY, MYZ, RXZ, RZX, \\
MXZ, MZX, BtXY, BtYX, MXY, MYX\}.

6.5.2 Solving Rubik’s Cube

Anne Scott (see [BCG]) proved that for every rearrangement of Rubik’s cube, the total number of edge-flips is always even, the total corner twisting is always zero mod 3, and the associated permutation of the cubelets is always even. Thus the total number of attainable configurations for Rubik’s cube is

\[
\frac{3^8 \times 2^{12} \times 8! \times 12!}{12} = 43,252,003,274,489,856,000 > 4.3 \times 10^{19}.
\]

Thus efficient search strategies are essential for solving Rubik’s cube.

Since we have fixed the position and orientation of the LBkT corner, the home position and orientation of any cubelet is uniquely determined. Hence forth ‘home position’ will mean ‘home position and orientation’. We will use our subgoal strategy with the subgoals of moving the cubelets into their home positions one by one.

The 6 central cubelets have fewer possible positions and orientations than the other cubelets. Their position relative to each other never changes. So it is convenient to achieve the goals for the 3 pairs of central cubelets first.

Experience shows that moving all the 8 corner cubelets to their home positions before moving the edge cubelets is better than moving the cubelets one by one in their natural order. This is because whenever a corner cubelet is moved, four edge cubelets are also moved. So achieving a goal for corners usually undoes some
previously achieved goal for an edge cubelet. But when we rotate only the middle
layers, we move edge cubelets without moving any corner cubelets. Thus when
unscrambling the whole cube, we are better off unscrambling all the corners before
moving the edge cubelets to their home positions. The LBkT corner is always in
its home position. Here are our first 10 subgoals:

\[ \text{subgoal}_1 : \text{move the top central face to its home position}, \]
\[ \text{subgoal}_2 : \text{move the left central face to its home position}, \]
\[ \text{subgoal}_3 : \text{move the front central face to its home position}, \]
\[ \text{subgoal}_4 : \text{move corner LFT to its home position}, \]
\[ \text{subgoal}_5 : \text{move corner RBkT to its home position}, \]
\[ \text{subgoal}_6 : \text{move corner RFT to its home position}, \]
\[ \text{subgoal}_7 : \text{move corner LBkBt to its home position}, \]
\[ \text{subgoal}_8 : \text{move corner LFBt to its home position}, \]
\[ \text{subgoal}_9 : \text{move corner RBkBt to its home position}, \]
\[ \text{subgoal}_{10} : \text{move corner RFBt to its home position}, \]

......

The first three subgoals are easy to achieve with the rule subset restricted to the
3 rules \{MXY, MYZ, MXZ\}. Once \text{subgoal}_3 is achieved, all 6 central cubelets
are in their home positions.

The next three subgoals are for the three nonfixed corners on the top layer. By
restricting the rule subset to the three basic rules \( \{FZY, RXZ, BtXY\} \), we can achieve these subgoals by a sequence of no more than 6 moves. These first six subgoals can be achieved very quickly.

Experience shows that subgoal_7 is still easy to achieve with the same rule subset restriction as for subgoal_6. But in many cases, subgoal_8 and subgoal_9 are very hard (subgoal_10 will be automatically achieved once subgoal_9 is achieved). With all 4 top corners and one bottom corners in place, it becomes harder to put the last 3 corners in place because these new subgoals must be achieved without undoing all the previously achieved subgoals. We need additional strategies to achieve the last 3 corner subgoals.

6.5.3 Strategies for the Corner Cubelets

Once the 4 top corners and one bottom corner are in their home positions, the last 3 corner cubelets have 3 possible positions and at each position the cubelets have 3 possible orientations. Thus the number of configurations for the last 3 cubelets is 
\[ 3! \times 3^2 = 54. \]

Attempts to achieve the last 3 corner subgoals using all the basic rules fail because too many clauses are generated. The rule-restriction strategy is one way to reduce the number of clauses generated. Since achieving the previous corner goals never required more than three rules, it is reasonable to hope that three or four rules always suffice for the last three corner goals. In fact three rules always suffice.
Since we do not know which 3-element rule subset works, we simply try them all. There are \( \binom{9}{3} = 20 \) such subsets. Trying each one for a bounded amount of time, say enough for 15 steps, takes a lot of time but less time than generating 15 levels of clauses using all 6 rules. Numerically, \( 20 \times 3^{15} \) is three orders of magnitude less than \( 6^{15} \).

After subgoal_7 has been achieved, there are 9 possible configurations (i.e. positions and orientations) for the cubelet that subgoal_8 moves into home position. Subgoal_8 can be achieved in at most 14 moves and 700 seconds (on a Sparc workstation II) with either rule subset \{BtXY, RXY, FZY\} or \{BtXY, RXZ, FYZ\}. After subgoal_8 has been achieved, there are 6 possible configurations for the last two corners. Subgoal_9, the last nontrivial subgoal for corners, can be achieved with one of the following 3-element rule subsets: \{BtXY, RXZ, FZY\}, \{BtXY, RXZ, FYZ\}, \{BtXY, RXZ, FZY\}, or \{BtXY, RXZ, FYZ\}. This subgoal is quite difficult: it can take as many as 15 steps and 2500 seconds.

The last two subgoals above take more time than all the previous subgoals combined. However there are a relatively small number of configurations for the corner pieces they move into place. If we store the solution for each configuration as a generic rule, then by using these generic rules in place of the original rules, we can reduce the time required to solve the last two subgoals from 700+2500 seconds to 10 seconds.
For subgoal\textsubscript{9}, besides the goal configuration, there are 5 other possible positions and orientations for the corner piece it moves into place. For each such configuration, the solution which moves the corner piece into position is a sequence of moves. The composition of these moves is a generic rule, a derived rule which can move the corner piece into position in one step. Generating a generic rule for each of the 5 possible configurations gives a set of 5 generic rules which can always solve subgoal\textsubscript{9} in one step. Thus for subgoal\textsubscript{9}, we can replace the original 6 basic rules with this 5-element set of generic rules. It takes no more than 5 \times 2500 seconds to generate these 5 generic rules, but once generated, the time required for subgoal\textsubscript{9} is reduced from 2500 seconds to 5 seconds (the latter is mostly loading and unloading time). Similarly, there are 8 generic rules for subgoal\textsubscript{8} which can solve it in 5 seconds.

6.5.4 Strategies for the Edge Cubelets

Once the 10 subgoals listed above are achieved, the 8 corners and 6 central cubelets are in their home positions. Next we work on the 12 edge cubelets. Each subgoal for an edge cubelet moves one edge cubelet to its home position while preserving all previously achieved subgoals. We first move the edge cubelets in the top layer then those in the middle layer and finally those in the bottom layer into their home positions. It is sometimes difficult to place the last edge cubelet of the top layer and the first edge cubelet of the second layer.

For the edge cubelet with home position LMT (left-middle-top), the initial position for that piece can be at any of the 12 edge places. So there are 24 initial
configurations (each edge position has 2 orientations) for that piece. If we use
generic rules to achieve this subgoal, the number of such new rules is 23. To avoid
deriving 23 generic rules, we choose to use the rule restriction strategy.

We achieve the subgoals for the edge cubelets in the top and middle layers by
discovering the appropriate rule subsets for each subgoal in the following way. For
each subgoal, we first list all the two-element and 3-element rule subsets of all the
6 original basic rules. Then each time we attempt to solve the subgoal, we run the
program on each of the rule subsets from the list for a bounded amount of time.
If a rule subset achieves the subgoal for this board configuration, we move the rule
subset to the front of the list. If no rule subset works, we increase the time bound
and try again.

Running this learning process on a dozen or more configurations typically pro-
duces a list of rule subsets and a time bound such that the given goal can always
be achieved with the first 4 or 5 rules subsets within the given time bound.

For the subgoal above for placing the proper edge cubelet in the $LMT$ position,
this learning process produced a list and a time bound such that the subgoal could
always be achieved within the time bound with one of the first four rule subsets of
the list. The four rule subsets were $\{BmXY, MZY\}, \{RZX, MZY\}, \{RXZ, MYZ,
MXY\}, \{BtYX, MZY, MZX\}$. The time bound was 25 seconds. Similarly, for each
of the other 7 edge positions in the top and middle layer, we found a time bound and
a small set of rule subsets which could always achieve the corresponding subgoal. In
this way, the time to achieve the subgoals for edge positions of the top and middle layers \((\text{subgoal}_{11} \text{ to } \text{subgoal}_{18})\) was reduced to no more than 60 seconds on average.

Next we consider the problem of achieving the last 4 subgoals for the edge positions of the bottom layer:

- \(\text{subgoal}_{19}:\) move edge \(\text{RMBt}\) to its home position,
- \(\text{subgoal}_{20}:\) move edge \(\text{FMBt}\) to its home position,
- \(\text{subgoal}_{21}:\) move edge \(\text{LMBt}\) to its home position,
- \(\text{subgoal}_{22}:\) move edge \(\text{BkMBt}\) to its home position,

The \(\text{subgoal}_{19}\) for placing the edge cubelet for the \(\text{RMBt}\) position is easy. With the rule subset \(\{\text{BtXY, MZY}\}\) this cubelet can always be moved to its home position within 16 moves and in 20 seconds in every case.

Placing the last 3 edges on the bottom layer again gets difficult due to the many previously established subgoals which must be preserved. Even with the optimal rule subsets some subgoals require up to 20 moves and 200 seconds. So in order to reduce the amount of time for these last subgoals, we use the generic rule method as with the cases of the last three corners.

Once \(\text{subgoal}_{19}\) is achieved, there are 3 edge pieces left in the bottom layer. For the piece whose home position is \(\text{FMBt}\), there are 6 possible positions and orientations. For each of the 5 positions other than the home position, there is a sequence of moves which carries the piece to its home position. The composition of these moves is a generic rule which moves the piece from that position to its home
position in one step. Each of these 5 generic rules can be generated in 200 seconds by rule subsets \{BtXY, Mxz\} or \{BtXY, Mzy\}. Thus to achieve subgoal\textsubscript{20} we replace the original basic rules with the 5-element set of these generic rules. With these generic rules, this subgoal can always be achieved in one step.

For the last two edges LMB\textsubscript{t} and BkMB\textsubscript{t}, due to the parity restrictions on the permutations and the total number of edge-flips, there are only 2 possible configurations. The 2 pieces will always be at their home places. In one configuration both pieces are in the right orientation. In the other configuration, both are in the wrong orientation. For the case that both pieces are in the wrong orientation, there is a sequence of 20 basic moves that changes the orientations of both pieces and preserves all the other pieces unchanged. The composition of these moves forms a generic rule for subgoal\textsubscript{21}. With this one generic rule, the last 2 subgoals can be achieved by at most one step (subgoal\textsubscript{22} is automatically achieved once subgoal\textsubscript{21} is achieved).

Using the above generic rules, we achieve the 4 subgoals for the bottom layer in 30 seconds.

6.5.5 Experimental Results

We implemented the above method using OTTER. On a Sparc II the program unscrambles a cube in about 200 to 400 seconds. Without the derived generic rules, the program needs more than 4000 seconds in many cases.
6.6 Masterball

Masterball is a creation of Dr. G. Gyovai. It is a variation of Rubik’s cube. A masterball is a sphere with 4 vertical cuts through the vertical axis which partition the ball into 8 equal-sized vertical parts and 3 horizontal cuts which divide the ball into 4 horizontal layers. These cuts divide the sphere into 32 pieces. In the unscrambled sphere, each of these 8 vertical parts is colored by a single color as depicted in the following figure:

![Masterball Diagram](image)

*Figure 15. A Configuration of Masterball.*

There are 16 basic moves for masterball: along the vertical cuts, any half of the ball can be rotated 180° around the horizontal axis perpendicular to the cut; along the horizontal cuts, any one of the 4 horizontal layers can be moved back or forth.
by one 8th of a revolution around the vertical axis (45° or -45°). For a scrambled ball, the goal is to unscramble it so that each of the 8 vertical parts is monocolored.

6.6.1 The Representation

We use a 32-place relation $B$ whose places represent the positions of masterball's 32 pieces. The first 8 positions are on the top layer, the next 8 positions are on the next layer, and so on. For simplicity we use the constants $a, b, c, d, e, f, g, h$ to represent the 8 colors as in the above figure.

Notice that the top layer and bottom layer are symmetric with respect to the center, as are the two middle layers. By rotating each of the two halves along any vertical cut, we rotate the ball upside down. This swaps the symmetric layers. Because of this symmetry, a rule for moving one layer can also be used to move its symmetric layer. To minimize the number of rules, we omit the derivable rotations of the top two horizontal layers and use only the 4 basic moves which rotate the third and bottom layer 45° in the clockwise and counter clockwise directions. They are:
Rules 1 and 2: rotating the third layer back and forth one 8th of a revolution:

\[
B(z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, \\
y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8) \Rightarrow \\
B(z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, \\
y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_1, w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8). \\
\]

\[
B(z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, \\
y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8) \Rightarrow \\
B(z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, \\
y_8, y_1, y_2, y_3, y_4, y_5, y_6, y_7, w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8). \\
\]

Rules 3 and 4: rotating the bottom layer back and forth one 8th of revolution:

\[
B(z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, \\
y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8) \Rightarrow \\
B(z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, \\
y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8). \\
\]

\[
B(z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, \\
y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8) \Rightarrow \\
B(z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, \\
y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, w_2, w_3, w_4, w_5, w_6, w_7, w_8). \\
\]

In addition to our 4 rules for rotating the two lowest horizontal layers, we need only one pair of rules for rotating the two half balls 180° along a vertical cut. The rotations of the half balls cut by the other vertical cuts can be generated from our 6 moves. So we include the following vertical moves in our set of rules:
Rules 5 and 6: rotating two vertically cut half balls 180° around a horizontal axis:

\[
B(x_1, x_2, z_3, z_4, z_5, z_6, z_7, z_8, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, \\
y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8) \Rightarrow \\
B(z_1, z_2, z_3, z_4, w_5, w_7, w_6, w_5, x_1, x_2, x_3, x_4, y_8, y_7, y_6, y_5, \\
y_1, y_2, y_3, y_4, x_8, x_7, x_6, x_5, w_1, w_2, w_3, w_4, z_8, z_7, z_6, z_5).
\]

\[
B(z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, \\
y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8) \Rightarrow \\
B(w_4, w_3, w_2, w_1, z_5, z_6, z_7, z_8, y_4, y_3, y_2, y_1, x_5, x_6, x_7, x_8, \\
x_4, x_3, x_2, x_1, y_5, y_6, y_7, y_8, z_4, z_3, z_2, z_1, w_5, w_6, w_7, w_8). 
\]

6.6.2 Solving Masterball

We use subgoals and rule restrictions in solving this problem. Our first 14 subgoals move the pieces of the top and bottom layers to their home positions. For these subgoals we do not need rules 1 and 2 since they only move middle-layer pieces.

Our last 14 subgoals move the pieces of the two middle horizontal layers to their home positions. For these subgoals we do not need rules 3 and 4 which only move the top and bottom layer pieces.

Thus our subgoals with their rule restrictions are as follow:
Using rules 3, 4, 5 and 6:

$subgoal_1 : B(a,**************,**************,**************),$

$subgoal_2 : B(a,**************,**************,a,**************),$

......

$subgoal_{14} : B(a,b,c,d,e,f,g,*,**************,***:***:***,a,b,c,d,e,f,g,*),$

Using the rules 1, 2, 5 and 6:

$subgoal_{15} : B(abcdefgh,a,**************,**************,abcdefgh),$

......

$subgoal_{26} : B(abcdefgh,a,b,c,d,e,f,*,*,a,b,c,d,e,f,*,*,abcdefgh),$

$subgoal_{27} : B(abcdefgh,a,b,c,d,e,f,*,*,a,b,c,d,e,f,g,*,abcdefgh),$

$subgoal_{28} : B(abcdefgh,a,b,c,d,e,f,g,*,a,b,c,d,e,f,g,*,abcdefgh).$

Note that once $subgoal_{28}$ is achieved, the two middle layers are in the expected order since the last two pieces of color $h$ must be in their home positions. A similar situation occurs for the top and bottom layers when $subgoal_{14}$ is achieved. Thus we need only 28 subgoals to move all 32 pieces to their destinations.

As we saw in the previous puzzles, the last subgoals are always among the hardest subgoals to achieve. In this problem, once $subgoal_{26}$ is achieved, there are 6 possible configurations for the last 4 pieces of the two middle layers, they are

$$
\begin{pmatrix}
  g \\
  h
\end{pmatrix}
\begin{pmatrix}
  g \\
  h
\end{pmatrix}
\begin{pmatrix}
  h \\
  g
\end{pmatrix}
\begin{pmatrix}
  h \\
  g
\end{pmatrix}
\begin{pmatrix}
  g \\
  g
\end{pmatrix}
\begin{pmatrix}
  h \\
  h
\end{pmatrix}
\begin{pmatrix}
  g \\
  g
\end{pmatrix}
\begin{pmatrix}
  h \\
  h
\end{pmatrix}
$$
The pieces of the first configuration are in their correct positions. The pieces of the remaining 5 need to be moved without scrambling the other 28 pieces. For each of these 5 cases our program moves the pieces to their home positions in at most 14 steps and in about 200-500 seconds (on a Sparc station II). Our experiments show that any masterball problem can be solved in about 2000 seconds with the above subgoals and rule restrictions.

6.6.3 Using Derived Rules

The last subgoals require most of the time. For each of the five nontrivial cases above, the composition of the sequence of moves which solves it is a generic rule for that case. With these five generic rules the last goal can be achieved in one step. Derived rules such as these generic rules which move only a few pieces are often useful for other goals as well. Similar derived rules exist for the top and bottom layers since they are symmetric with the two middle layers.

Adding two of these derived rules, the first and the fourth, to our rule subsets for the subgoals 26, 27, and 28 and a similar pair of rules for subgoals 12, 13 and 14, reduces the total time to 110-210 seconds, at least 10 times faster than the 2000 seconds required when only the basic rules are used.
As mentioned before, our search strategies can solve a variety of problems. Here we apply these strategies to sorting, solving systems of linear equations, and finding inverses of matrices.

### 7.1 Sorting

The sorting problem is, given \( n (n > 0) \) integers, say \( a_1, a_2, \ldots, a_n \), to get a permutation \( < b_1, b_2, \ldots, b_n > \) of \( a_1, a_2, \ldots, a_n \) so that \( b_1 \leq b_2 \leq \ldots \leq b_{n-1} \leq b_n \). There are many well-studied algorithms for sorting, such as quick sort, heap sort, shell sort, etc. None of these algorithms nor any algorithm of \( n \log n \)-time efficiency has been found with unaided artificial intelligence.

We are given a sequence \( a_1, a_2, \ldots, a_n \) of \( n \) integers. We may compare any two items with respect to \( \leq \) and we may swap adjacent items of the list. For our first-order logic representation, we use an \( n \)-place relation \( S \) whose \( i \)th place corresponds to the \( i \)th position of the sequence. In addition we have the binary relation \( \leq \) and \( n \) constants \( a_1, a_2, \ldots, a_n \) for the \( n \) items. For the move which swaps the items in positions \( i \) and \( i + 1 \) we have the rule:

\[
S(x_1, \ldots, x_i, x_{i+1}, \ldots, x_n) \Rightarrow S(x_1, \ldots, x_{i+1}, x_i, \ldots, x_n).
\]
There are \( n - 1 \) such rules. So the problem description is:

\[
\text{initial} : S(a_1, ..., a_n),
\]

\[
\text{rules} : S(x_1, x_2, ...) \Rightarrow S(x_2, x_1, ...),
\]

\[\ldots\]

\[
S(\ldots, x_{n-1}, x_n) \Rightarrow S(\ldots, x_n, x_{n-1}),
\]

\[
\text{goal} : S(y_1, \ldots, y_n) \land (y_1 \leq y_2) \land \ldots \land (y_{n-1} \leq y_n).
\]

We use our subgoal strategy to solve this sorting problem. Since we have a conjunctive goal here, we set our subgoals using the idea in section 5.2, i.e., use the partial conjuncts as subgoals. In this way we have the following subgoals:

\[
\text{subgoal}_1 : S(y_1, y_2, y_3, \ldots, y_n) \land (y_1 \leq y_2),
\]

\[
\text{subgoal}_2 : S(y_1, y_2, y_3, y_4, \ldots, y_n) \land (y_1 \leq y_2) \land (y_2 \leq y_3),
\]

\[\ldots\]

\[
\text{subgoal}_{n-1} : S(y_1, \ldots, y_n) \land (y_1 \leq y_2) \land \ldots \land (y_{n-1} \leq y_n).
\]

This choice of subgoals corresponds to the behavior of insertion sort.

An exhaustive search without subgoals would take about \( 8! \) steps on a well scrambled list of 8 elements. With the above subgoals, every sequence of 8 numbers can be sorted within 5 seconds.

### 7.2 Solving Systems of Linear Equations

Now we apply our strategies to solve a system of linear equations. A system of \( n \) linear equations in \( n \) variables may be encoded as a matrix equation \( A_{n \times n}X_n = C_n \).
The matrix \((A, C)\) can be encoded as a relation of \(n(n+1)\) places. Here we assume that \(A\) is an invertible matrix so the equations have a unique solution. Our legal moves are the usual elementary row operations: exchanging two rows, multiplying a row by a nonzero constant, and adding to a row a multiple of another. The final goal is to translate matrix \((A, C)\) into a matrix of the form \((E, D)\) where \(E\) is the identity matrix of order \(n\). Then \(X = D\) is the solution to the original system of equations.

In the goal system the elements of matrix \(E\) are either the constant 1 (if the element is on the diagonal) or the constant 0 (if it is not on the diagonal). Hence we can set our rules so that they make elements become 0 and 1 in the coefficient matrix. As an example, we consider the case of 3 equations in 3 variables. So the system looks like:

\[
\begin{pmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{pmatrix}
\begin{pmatrix}
  x_1 \\
  x_2 \\
  x_3
\end{pmatrix}
=
\begin{pmatrix}
  c_1 \\
  c_2 \\
  c_3
\end{pmatrix}.
\]

We use a relation \(K\) of \(3 \times 4 = 12\) places to represent the system. Thus,

\[
K(a_{11}, a_{12}, a_{13}, c_1, a_{21}, a_{22}, a_{23}, c_2, a_{31}, a_{32}, a_{33}, c_3)
\]

represents the augmented matrix for the system with the usual subscription conventions. For the sake of readability, we write the relation \(K\) in the standard matrix format:

\[
K = \begin{pmatrix}
  a_{11} & a_{12} & a_{13} & c_1 \\
  a_{21} & a_{22} & a_{23} & c_2 \\
  a_{31} & a_{32} & a_{33} & c_3
\end{pmatrix}.
\]

There are two types of rules for each of the \(3 \times 3\) positions of coefficient elements, one is to make the position 0, the other is to make the position 1. For position \(a_{11}\),
we have
\[
\begin{pmatrix}
  a_{11} & a_{12} & a_{13} & c_1 \\
  a_{21} & a_{22} & a_{23} & c_2 \\
  a_{31} & a_{32} & a_{33} & c_3
\end{pmatrix}
\wedge (a_{11} \neq 0) \Rightarrow \begin{pmatrix}
  1 & a_{12} & a_{13} & c_1 \\
  a_{21} & a_{22} & a_{23} & c_2 \\
  a_{31} & a_{32} & a_{33} & c_3
\end{pmatrix},
\]

\[
\begin{pmatrix}
  a_{11} & a_{12} & a_{13} & c_1 \\
  a_{21} & a_{22} & a_{23} & c_2 \\
  a_{31} & a_{32} & a_{33} & c_3
\end{pmatrix}
\wedge (a_{11} \neq 0) \wedge (a_{21} \neq 0)
\Rightarrow \begin{pmatrix}
  0 & a_{12} - \frac{a_{11}a_{22}}{a_{21}} & a_{13} - \frac{a_{11}a_{23}}{a_{21}} & c_1 - \frac{a_{11}c_2}{a_{21}} \\
  a_{21} & a_{22} & a_{23} & c_2 \\
  a_{31} & a_{32} & a_{33} & c_3
\end{pmatrix}.
\]

Notice the numerical operations ÷, −, × in the above rules. Our intention here is that they be evaluated on numeric arguments. E.g. 3 ÷ 2 should be evaluated to 1.5.

OTTER supplies float-point evaluation functions $FDIV(x, y)$, $FDIFF(x, y)$, and $FPROD(x, y)$ which evaluate $x ÷ y$, $x - y$, and $x \times y$ respectively.

If we create for each of the $n^2$ positions two rules as above, the total number of rules will be of order $o(n^2)$. Since swapping rows is allowed in matrix operations, we need only apply the rules for making 0's and 1's to the first row and allow each of the other rows to swap with the first row. Thus we need only $2n$ rules for making 0's and 1's and $n - 1$ rules for swapping the other $n - 1$ rows with the first row.

For the above $3 \times 3$ case, we have the following two rules for swapping the first row with the second and third rows respectively:
At last, our goal is:

\[
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & c_1 \\
a_{21} & a_{22} & a_{23} & c_2 \\
a_{31} & a_{32} & a_{33} & c_3 \\
\end{pmatrix} \Rightarrow \begin{pmatrix}
a_{21} & a_{22} & a_{23} & c_2 \\
a_{31} & a_{32} & a_{33} & c_3 \\
\end{pmatrix} ,
\]

\[
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & c_1 \\
a_{21} & a_{22} & a_{23} & c_2 \\
a_{31} & a_{32} & a_{33} & c_3 \\
\end{pmatrix} \Rightarrow \begin{pmatrix}
a_{31} & a_{32} & a_{33} & c_3 \\
a_{21} & a_{22} & a_{23} & c_2 \\
a_{11} & a_{12} & a_{13} & c_1 \\
\end{pmatrix} .
\]

When this goal is achieved, \((x_1, x_2, x_3)\) will be a solution for the system.

Solving a system of equations may take many steps. We apply the subgoal strategy to decompose the problem into subproblems. One method is to set \(n\) subgoals so that the first subgoal puts the last row of the matrix into the form \((0, 0, \ldots, 0, 1, x_n)\). Then the second subgoal puts the next-to-the-last row of the matrix into the form \((0, 0, \ldots, 0, 1, 0, x_{n-1})\). And so on. Finally the \(n\)-th subgoal puts the relation into the form \((E, R)\).

If the order of \(n\) is large, it may be hard to achieve the above subgoals. A more detailed alternative subgoal strategy is to set a subgoal for each of the \(n^2\) coefficient matrix positions. For example, the first subgoal makes the \(a_{n1}\) position 0, then \(a_{n2}\) position zero, \ldots, the \(n\)-th subgoal makes \(a_{nn}\) position 1. And finally the last subgoal makes all \(n^2\) elements agree with the identity matrix of order \(n\).

Another strategy we find helpful in solving this problem is the rule restriction strategy. Notice that once the subgoals for the last row are achieved, we may exclude
the rule of swapping the last row with the first row. In this way, as the program progresses, more and more swapping rules can be excluded.

The above methods effectively lead to a solution whenever the coefficient matrix $A$ is invertible. In Appendix C we give the complete OTTER input file for solving a system of 3 equations.

### 7.3 Finding Inverses of Matrices

This method of solving equations can also be used to find the inverse of an invertible matrix. The basic idea is to transform a $n \times 2n$ matrix $(A, E)$ into the form $(E, B)$ with the same rules of row transformation. Once the goal is achieved, $B = A^{-1}$. For example, the rule of changing position $a_{11}$ into 0 is:

\[
\begin{pmatrix}
\begin{array}{cccc}
1 & a_{12} & a_{13} & e_{11} & e_{12} & e_{13} \\
a_{21} & a_{22} & a_{23} & e_{21} & e_{22} & e_{23} \\
a_{31} & a_{32} & a_{33} & e_{31} & e_{32} & e_{33}
\end{array}
\end{pmatrix}
\wedge (a_{11} \neq 0) \Rightarrow
\begin{pmatrix}
\begin{array}{cccc}
1 & a_{12} & a_{13} & e_{11} & e_{12} & e_{13} \\
a_{21} & a_{22} & a_{23} & e_{21} & e_{22} & e_{23} \\
a_{31} & a_{32} & a_{33} & e_{31} & e_{32} & e_{33}
\end{array}
\end{pmatrix}
\begin{pmatrix}
\begin{array}{cccc}
\frac{e_{11}}{a_{11}} & \frac{e_{12}}{a_{11}} & \frac{e_{13}}{a_{11}} \\
\frac{e_{21}}{a_{11}} & \frac{e_{22}}{a_{11}} & \frac{e_{23}}{a_{11}}
\end{array}
\end{pmatrix}.
\]

As before, we have rules for producing 0's and 1's in the first three columns and for swapping the first row with any other row.

As an example, consider the problem of finding inverse of

\[
\begin{pmatrix}
1.0 & 2.0 & 0.0 \\
0.0 & 1.0 & -1.0 \\
0.0 & 5.0 & 1.0
\end{pmatrix}.
\]

The solution found for the first goal was:

\[
\begin{pmatrix}
1.0 & 2.0 & 0.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 1.0 & -1.0 & 0.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 0.0 & -0.83 & 0.167
\end{pmatrix}.
\]
The solution for the second subgoal was:

\[
\begin{pmatrix}
1.0 & 2.0 & 0.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 0.0 & 0.0 & 0.167 & 0.167 \\
0.0 & 0.0 & 1.0 & 0.0 & -0.83 & 0.167
\end{pmatrix}.
\]

The solution for the final subgoal was:

\[
\begin{pmatrix}
1.0 & 0.0 & 0.0 & 1.0 & -0.33 & -0.33 \\
0.0 & 1.0 & 0.0 & 0.0 & 0.167 & 0.167 \\
0.0 & 0.0 & 1.0 & 0.0 & -0.83 & 0.167
\end{pmatrix}.
\]

The total time spent on the 3 subgoals was less than 6 seconds. The total length of the 'proof' was 12 steps. Thus the inverse of matrix $A$ is

\[
\begin{pmatrix}
1.0 & -0.33 & -0.33 \\
0.0 & 0.167 & 0.167 \\
0.0 & -0.83 & 0.167
\end{pmatrix}.
\]
APPENDICES

A. The CASE Input File for Solving the Zebra Problem

(In the following input file, we use abbreviated strings eng, parli etc. to represent
the constants Englishman, Parliments etc. . We add a character ‘a’ in front of
constant strings leaded by ‘u’, ‘v’, ‘w’, ‘x’, ‘y’, ‘z’. Therefore, we use azebra etc. to
represent zebra. Lines ending in ‘...’ are used as demodulators.

set(back_demod).
set(process_input).
set(binary_res).
set(printpaths).
list(sos).
(eng = red). (span = dog).
(coffee = green). (auakra = tea).
(old = snail). (kool = ayellow).
(milk = 3). (norw = 1).
(lucky = orange). (jap = parli).
end_of_list.
list(usable).
(red = 1) || (red = 2) || (red = 3) || (red = 4) || (red = 5)...

129
(green = 1) | (green = 2) | (green = 3) | (green = 4) | (green = 5).

(ivory = 1) || (ivory = 2) || (ivory = 3) || (ivory = 4) || (ivory = 5)...

(ayellow = 1) | (ayellow = 2) | (ayellow = 3) | (ayellow = 4) | (ayellow = 5).

(blue = 1) | (blue = 2) | (blue = 3) | (blue = 4) | (blue = 5).

(eng = 1) || (eng = 2) || (eng = 3) || (eng = 4) || (eng = 5)...

(span = 1) | (span = 2) | (span = 3) | (span = 4) | (span = 5).

(aukra = 1) || (aukra = 2) || (aukra = 3) || (aukra = 4) || (aukra = 5)...

(norw = 1) | (norw = 2) | (norw = 3) | (norw = 4) | (norw = 5).

(jap = 1) || (jap = 2) || (jap = 3) || (jap = 4) || (jap = 5)...

(dog = 1) | (dog = 2) | (dog = 3) | (dog = 4) | (dog = 5).

(snail = 1) | (snail = 2) | (snail = 3) | (snail = 4) | (snail = 5).

(fox = 1) | (fox = 2) | (fox = 3) | (fox = 4) | (fox = 5).

(horse = 1) | (horse = 2) | (horse = 3) | (horse = 4) | (horse = 5).

(azebra = 1) | (azebra = 2) | (azebra = 3) | (azebra = 4) | (azebra = 5).

(coffee = 1) | (coffee = 2) | (coffee = 3) | (coffee = 4) | (coffee = 5).

(tea = 1) | (tea = 2) | (tea = 3) | (tea = 4) | (tea = 5).

(milk = 1) | (milk = 2) | (milk = 3) | (milk = 4) | (milk = 5).

(orange = 1) | (orange = 2) | (orange = 3) | (orange = 4) | (orange = 5).

(apple = 1) | (apple = 2) | (apple = 3) | (apple = 4) | (apple = 5).

(old = 1) | (old = 2) | (old = 3) | (old = 4) | (old = 5).

(kool = 1) | (kool = 2) | (kool = 3) | (kool = 4) | (kool = 5).
(chest = 1) | (chest = 2) | (chest = 3) | (chest = 4) | (chest = 5).

(lucky = 1) | (lucky = 2) | (lucky = 3) | (lucky = 4) | (lucky = 5).

(parli = 1) | (parli = 2) | (parli = 3) | (parli = 4) | (parli = 5).

(ivory = 1) | (green = 2).  (ivory = 2) | (green = 3).

(ivory = 3) | (green = 4).  (ivory = 4) | (green = 5).

(ivory = 1) | (green = 2).  (ivory = 2) | (green = 3).

(ivory = 3) | (green = 4).  (ivory = 4) | (green = 5).

(1 != kool) | (2 = horse).  (1 != horse) | (2 = kool).

(2 != horse) | (1 = kool) | (3 = kool).

(2 != kool) | (1 = horse) | (3 = horse).

(3 != kool) | (2 = horse) | (4 = horse).

(3 != horse) | (2 = kool) | (4 = kool).

(4 != horse) | (3 = kool) | (5 = kool).

(4 != kool) | (3 = horse) | (5 = horse).


(1 != chest) | (2 = fox).  (1 != fox) | (2 = chest).

(2 != fox) | (1 = chest) | (3 = chest).

(3 != fox) | (2 = chest) | (4 = chest).

(4 != fox) | (3 = chest) | (5 = chest).

(2 != chest) | (1 = fox) | (3 = fox).

(3 != chest) | (2 = fox) | (4 = fox).
(4 !:= chest) \mid (3 = fox) \mid (5 = fox).

(5 !:= chest) \mid (4 = fox). \quad (5 !:= fox) \mid (4 = chest).

(1 !:= norw) \mid (2 = blue). \quad (1 !:= blue) \mid (2 = norw).

(2 !:= blue) \mid (1 = norw) \mid (3 = norw).

(3 !:= blue) \mid (2 = norw) \mid (4 = norw).

(4 !:= blue) \mid (3 = norw) \mid (5 = norw).

(2 !:= norw) \mid (1 = blue) \mid (3 = blue).

(3 !:= norw) \mid (2 = blue) \mid (4 = blue).

(4 !:= norw) \mid (3 = blue) \mid (5 = blue).

(5 !:= norw) \mid (4 = blue). \quad (5 !:= blue) \mid (4 = norw).

(x = x). \quad (1 !:= 2). \quad (1 !:= 3).

(1 !:= 4). \quad (1 !:= 5). \quad (2 !:= 3).

(2 !:= 4). \quad (2 !:= 5). \quad (3 !:= 4).

(3 !:= 5). \quad (4 !:= 5).

(red !:= green). \quad (red !:= ivory). \quad (red !:= ayellow).

(red !:= blue). \quad (green !:= ivory). \quad (green !:= ayellow).

(green !:= blue). \quad (ivory !:= ayellow). \quad (ivory !:= blue).

(ayellow !:= blue). \quad (ayellow !:= blue). \quad (eng !:= span).

(eng !:= aukra). \quad (eng !:= norw). \quad (eng !:= jap).

(span !:= aukra). \quad (span !:= norw). \quad (span !:= jap).

(aukra !:= norw). \quad (aukra !:= jap). \quad (norw !:= jap).

(dog !:= snail). \quad (dog !:= fox). \quad (dog !:= horse).
(dog !\= azebra). (snail !\= fox). (snail !\= horse).
(snail !\= azebra). (fox !\= horse). (fox !\= azebra).
(horse !\= azebra). (coffee !\= tea). (coffee !\= milk).
(coffee !\= orange). (coffee !\= apple). (tea !\= milk).
(tea !\= orange). (tea !\= apple). (milk !\= orange).
(milk !\= apple). (orange !\= apple). (old !\= kool).
(old !\= chest). (old !\= lucky). (old !\= parli).
(kool !\= chest). (kool !\= lucky). (kool !\= parli).
(chest !\= lucky). (chest !\= parli). (lucky !\= parli).

end_of_list.
B. A Solution for the Central Solitaire Problem

The following is a solution found by OTTER which carries board 0 to board 31. We display boards 0, 6, 12, 18, 24 and 31 in the figure below, followed by the complete step-by-step list of the configurations of the move sequence.
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.</td>
<td>( R(1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>12.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>16.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>17.</td>
<td>( R(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>18.</td>
<td>( R(0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>19.</td>
<td>( R(0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
<tr>
<td>20.</td>
<td>( R(0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1) )</td>
<td></td>
</tr>
</tbody>
</table>
21. $R(0,0,1,0,0,1,1,0,1,0,0,0,0,0,1,0,1,1,0,1,0,0,0,1,0,0)$.  
22. $R(0,0,0,0,0,1,1,1,0,0,0,0,0,0,1,0,1,0,1,0,0,0,0,1,0,0)$.  
23. $R(0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,1,0,0,1,0,0,0,1,0,0)$.  
24. $R(0,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0)$.  
25. $R(0,0,0,0,1,1,1,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,0,0,0,0)$.  
26. $R(0,0,0,0,1,1,1,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0)$.  
27. $R(0,0,0,0,1,1,1,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0)$.  
28. $R(0,0,0,0,1,0,0,1,1,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0)$.  
29. $R(0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0)$.  
30. $R(0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0)$.  
31. $R(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0)$.  

136
C. An OTTER Input File for Solving a System of Linear Equations

\begin{verbatim}
set(hyper_res).
set(para_into).
set(para_from).
list(sos).
s("1.0", "2.0", "0.0", "1.0", "0.0", "0.0", "0.0", "1.0", "0.0", "0.0",
"0.166667", "0.1666667", "0.0", "0.0", "1.0", "0.0", 
"-0.8333333333", 
"0.16666666667").

% Clause for the Initial Matrix
end_of_list.

list(usable).

-s("1.0", "0.0", "0.0", x, x1, x2, "0.0", "1.0", "0.0", y, y1, y2, "0.0",
"0.0", "1.0", z, zl, z2) | $ANS( x, x1, x2, y, y1, y2, z, zl, z2).

% Clause for the Goal Matrix
-s(z, x2, x3, x4, x5, x6, y1, y2, y3, y4, y5, y6, z1, z2, z3, z4, z5, z6) | 
-$FNE(z, "0.0") | s("1.0", $FDIV(x2, z), $FDIV(x3, z), $FDIV(x4, z),
$FDIV(x5, z), $FDIV(x6, z), y1, y2, y3, y4, y5, y6, z1, z2, z3, z4, z5, z6).

% Rule for Changing x1 to 1
-s(x1, z, x3, x4, x5, x6, y1, y2, y3, y4, y5, y6, z1, z2, z3, z4, z5, z6) | 
-$FNE(z, "0.0") | s($FDIV(x1, z), "1.0", $FDIV(x3, z), $FDIV(x4, z),
$FDIV(x5, z), $FDIV(x6, z), y1, y2, y3, y4, y5, y6, z1, z2, z3, z4, z5, z6).
\end{verbatim}
% Rule for Changing $x_2$ to 1
\[-s(x_1, x_2, z, x_4, x_5, x_6, y_1, y_2, y_3, y_4, y_5, y_6, z_1, z_2, z_3, z_4, z_5, z_6) \mid
-\text{FNE}(z, "0.0") \mid s(\text{FDIV}(x_1, z), \text{FDIV}(x_2, z), "1.0", \text{FDIV}(x_4, z), \text{FDIV}(x_5, z), \text{FDIV}(x_6, z), y_1, y_2, y_3, y_4, y_5, y_6, z_1, z_2, z_3, z_4, z_5, z_6).
\]

% Rule for Changing $x_3$ to 1
\[-s(x_1, x_2, x_3, x_4, x_5, x_6, y_1, y_2, y_3, y_4, y_5, y_6, z_1, z_2, z_3, z_4, z_5, z_6) \mid
s(y_1, y_2, y_3, y_4, y_5, y_6, x_1, x_2, x_3, x_4, x_5, x_6, z_1, z_2, z_3, z_4, z_5, z_6).
\]

% Rule for Switching Row 1 and Row 2
\[-s(x_1, x_2, x_3, x_4, x_5, x_6, y_1, y_2, y_3, y_4, y_5, y_6, z_1, z_2, z_3, z_4, z_5, z_6) \mid
s(z_1, z_2, z_3, z_4, z_5, z_6, y_1, y_2, y_3, y_4, y_5, y_6, x_1, x_2, x_3, x_4, x_5, x_6).
\]

% Rule for Switching Row 1 and Row 3
\[-s(x, x_2, x_3, x_4, x_5, x_6, "1.0", y_2, y_3, y_4, y_5, y_6, z_1, z_2, z_3, z_4, z_5, z_6) \mid
s("0.0", \text{FDIFF}(x_2, \text{FPROD}(x, y_2), \text{FDIFF}(x_3, \text{FPROD}(x, y_3)), \text{FDIFF}(x_4, \text{FPROD}(x, y_4), \text{FDIFF}(x_5, \text{FPROD}(x, y_5)), \text{FDIFF}(x_6, \text{FPROD}(x, y_6)), "1.0", y_2, y_3, y_4, y_5, y_6, z_1, z_2, z_3, z_4, z_5, z_6).
\]

% Rule for Changing $x_1$ to 0
\[-s(x_1, x, x_3, x_4, x_5, x_6, y_1, "1.0", y_3, y_4, y_5, y_6, z_1, z_2, z_3, z_4, z_5, z_6) \mid
s(\text{FDIFF}(x_1, \text{FPROD}(x, y_1), "0.0", \text{FDIFF}(x_3, \text{FPROD}(x, y_3)), \text{FDIFF}(x_4, \text{FPROD}(x, y_4), \text{FDIFF}(x_5, \text{FPROD}(x, y_5)), \text{FDIFF}(x_6, \text{FPROD}(x, y_6), y_1, "1.0", y_3, y_4, y_5, y_6, z_1, z_2, z_3, z_4, z_5, z_6).
\]

% Rule for Changing $x_2$ to 0

138
-s(x1, x2, x, x4, x5, x6, y1, y2, "1.0", y4, y5, y6, z1, z2, z3, z4, z5, z6) |

s($FDIFF(x1, $FPROD(x, y1), $FDIFF(x2, $FPROD(x, y2), "0.0", $FDIFF(x4, $FPROD(x, y4), $FDIFF(x5, $FPROD(x, y5)), $FDIFF(x6, $FPROD(x, y6)), y1, y2, "1.0", y4, y5, y6, z1, z2, z3, z4, z5, z6).

% Rule for Changing x3 to 0
end_of_list.

list(demodulators).

("1.000000000000"="1.0").

("0.000000000000"="0.0").

("1e0"="1.0").

("0e0"="0.0").

end_of_list.
REFERENCES


