

AN ABSTRACT OF THE THESIS OF

Nicholas S. Flann for the degree of Doctor of Philosophy in
Computer Science presented on December 12, 1991.

Title: Correct Abstraction in Counter-planning: A Knowledge Compilation Approach

Redacted for Privacy

Abstract approved: _____

Tom G. Dietterich.

Knowledge compilation improves search-intensive problem-solvers that are easily specified but inefficient. One promising approach improves efficiency by constructing a database of problem-instance/best-action pairs that replace problem-solving search with efficient lookup. The database is constructed by reverse enumeration—expanding the complete search space backwards, from the terminal problem instances. This approach has been used successfully in counter-planning to construct perfect problem-solvers for subdomains of chess and checkers. However, the approach is limited to small problems because both the space needed to store the database and the time needed to generate the database grow exponentially with problem size.

This thesis addresses these problems through two mechanisms. First, the space needed is reduced through an abstraction mechanism that is especially suited to counter-planning domains. The search space is abstracted by representing problem states as equivalence classes with respect to the goal achieved and the operators as equivalence classes with respect to how they influence the goals. Second, the time needed is reduced through a heuristic best-first control of the reverse enumeration. Since with larger problems it may be impractical to run the compiler to completion, the search is organized to optimize the tradeoff between the time spent compiling a domain and the coverage achieved over that domain.

These two mechanisms are implemented in a system that has been applied to problems in chess and checkers. Empirical results demonstrate both the strengths and weaknesses of the approach. In most problems and 80/20 rule was demonstrated, where a small number of patterns were identified early that covered most of the domain, justifying the use of best-first search. In addition, the method was able to automatically generate a set of abstract rules that had previously required two person-months to hand engineer.

**Correct Abstraction in Counter-planning:
A Knowledge Compilation Approach**

By Nicholas S. Flann

A Thesis submitted to
Oregon State University

in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

Completed April 11, 1992

Commencement June 1992.

Approved:

Redacted for Privacy

Thomas G. Dietterich

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis presented December 12, 1991
Typed and formatted by Nicholas S. Flann

ACKNOWLEDGMENTS

There are many people that contributed in one way or another to this thesis. I am indebted to my advisor, Tom Dietterich, who has been a constant source of ideas, encouragement and support throughout my long career as a graduate student. He has shown me by example how to be a good advisor, researcher and teacher.

I thank Prasad Tadepalli, who as a member of my thesis committee and earlier as a fellow graduate student, encouraged me to pursue my interest in games. He was always able to see the big picture and identify the important research issues. I thank Barney Pell, who I met while at NASA and whose enthusiasm and ideas have always been an inspiration. I have enjoyed and benefited from discussions with many others including Devika Subramanian, Oren Etzioni, Steve Minton, Haym Hirsh, Smadar Kedar-Cabelli, Chris Tong, Craig Knoblock and John Bresina.

My office mates over the years have been a constant source of encouragement and friendship: Ritchie Ruff, Cerbone Giuseppe, Hussein Almuallim, Tony Fountain and Caroline Koff.

I thank my friends with whom I have enjoyed many an adventure: Kevin Krefft, Mike Gross, Rob Cline, Steve Poet and especially Tony Fountain. I also thank Diane McKean for her patience and support.

I thank the Kalmiopsis and the Middle Santiam—two special places that after weeks of sitting in front of a computer, reminded me that we still live on a wild Earth.

Finally, I would like to thank my parents, Jean and John, whose support and encouragement make this thesis possible.

Table of Contents

1	Introduction	1
1.1	Previous Approaches	3
1.1.1	Explanation-based Approaches	3
1.1.2	Database Approaches	6
1.2	Overview of the Approach	9
1.2.1	Correct Abstractions	10
1.2.2	The Coverage/compile-time Tradeoff	14
1.3	Statement of Thesis	15
1.4	Reading Guide	15
2	Abstraction Based on Influence	17
2.1	Goal Achievement in Counter-planning	17
2.2	Influence Proofs for Goal Achievement	18
2.2.1	Influence Relations	20
2.2.2	Influence Axioms	21
2.2.3	Chess Proofs	23
2.3	Examples of Influence Proofs	25
2.3.1	Fork	26
2.3.2	Skewer	26
2.3.3	Half-Pin	27

2.3.4	Overworked Piece	28
2.4	Optimal Goal Achievement	29
2.4.1	When <i>LOSS</i> is to Play	30
2.4.2	When <i>WIN</i> is to Play	31
3	Compiling Influence Proofs	33
3.1	Compiling Influence Relations	35
3.1.1	Goal Patterns as Literals	35
3.1.2	Goal Patterns as Conjunctions	37
3.1.3	Goal Patterns as Conjunctions with Exceptions	54
3.1.4	Chess Examples	62
3.2	Compiling Influence Proofs	71
3.2.1	When <i>LOSS</i> is to Play	74
3.2.2	When <i>WIN</i> is to Play	83
4	Analysis of the Compiler	85
4.1	Soundness	86
4.2	Completeness	90
4.3	Complexity Analysis	91
4.3.1	Number of Proof Sentences Generated	91
4.3.2	Complexity of Compilation	93
4.3.3	Complexity of Performance	94
4.4	Summary	95
5	A Theory of Geometry for Efficient Abstractions	97
5.1	A Geometric Representation of Patterns	102
5.2	Geometric Influence Relations	105
5.2.1	Representation of Operator Sets	105

5.2.2	Compiling Geometric Influence Relations	109
5.3	Example of Compiling <i>maintain-true</i>	112
5.4	Geometric Intersection	119
6	Analysis of Geometric Abstractions	122
6.1	Completeness	122
6.1.1	Proof of Termination	122
6.1.2	Proof of Compiler Completeness	123
6.2	Complexity Analysis	124
6.2.1	Number of Proof Sentences	124
6.2.2	Number of Pattern Computations	124
6.2.3	Complexity of Pattern Computations	125
6.3	Summary	125
7	Techniques for Searching the Space of Abstractions	126
7.1	Breadth-first Search	129
7.1.1	The <i>Compile-all-losses(Depth,NewWin)</i> Algorithm	131
7.1.2	The <i>Compile-all-wins(Depth,NewLoss)</i> Algorithm	141
7.2	Best-first Search	143
7.2.1	Summary	147
8	Experimental Evaluation	148
8.1	Evaluation Criteria	148
8.2	Abstraction Effectiveness	149
8.3	Compiler Effectiveness	153
8.4	Analysis	156
8.5	Utility of Compiled Knowledge	158
8.6	Generality of Approach	159

9	Related Work	162
9.1	Knowledge Compilation	162
9.2	Abstraction in Counter-planning	165
9.2.1	Abstraction in Problem-Solving	165
9.2.2	Abstraction Form and Derivation	168
9.3	Summary	172
10	Conclusions and Future Work	173
10.1	Summary	173
10.2	Contributions	173
10.3	Limitations of the Thesis	175
10.4	Future Work	176
10.4.1	Pragmatic Issues	176
10.4.2	Research Issues	178
	Bibliography	182
	Appendices	190
A	Chess Domain Theory	190
B	King-Rook-King Problem Specification	196
C	Example run for KRK Problem	203

List of Figures

1.1	Two similar chess positions both with black to play. (a) is a loss position, while (b) is a draw	5
1.2	(a) The sufficient condition learned by EBL from the example in Figure 1.1(a) expressed as a decision tree. If the condition tested at a node is true the left branch is taken. (b) The key for the symbols used to illustrate the patterns learned in chess.	7
1.3	The correct decision tree that correctly classifies the examples in Figure 1.1	13
1.4	(a) The behavior of the database compiler without abstraction: there is a linear relationship between compile time and coverage (b) The behavior of the compiler with abstraction, when the most general patterns are learned first.	14
2.1	Examples of different chess concepts that can be described by influence proofs	25
3.1	Prolog definition of the Room domain	38
3.2	Results of compiling <i>make-true</i> , <i>make-false</i> and <i>maintain-true</i> , for the <i>stuffy-room</i> goal pattern	40
3.3	The evaluation tree generated when partially evaluating <i>maintain-true</i> ($\nu^{stuffy}(S), Op, S$).	45
3.4	The evaluation tree generated when partially evaluating <i>make-true</i> ($\nu^{stuffy}(S), Op, S$).	49

3.5	The evaluation tree generated when partially evaluating <i>make-false</i> ($\nu^{stuffy}(S), Op, S$). Note the computation of $\text{not } c(\text{do}(Op, S))$.	52
3.6	Definition of the Chess domain	63
3.7	Definition of the frame axioms for the Chess domain	65
3.8	Prolog definition and graphical representation of the termination pattern rook-takes-king	66
3.9	Prolog definition and graphical representation of the termination pattern rook-takes-knight1 . Note that exceptions are denoted by a bold outline around the objects of the exception. Hence, the black king must <i>not</i> be adjacent to the black knight.	67
3.10	Prolog definition and graphical representation of the termination pattern rook-takes-knight2	68
3.11	Pattern/action pairs that maintain the termination pattern capture-king(S) , with black moving	69
3.12	Pattern/action pairs that make false the termination pattern safe-capture-knight1(S) , with black moving	72
3.13	Pattern/action pairs that make true the termination pattern safe-capture-knight2(S) , with white moving	73
3.14	Compilation of a rook fork losing goal pattern with black to play. We intersect those pattern/action pairs from <i>make-false</i> ($\lambda s.\text{rook-takes-king}(s), Op, S$) (illustrated along the side) with those pattern/action pairs from <i>make-false</i> ($\lambda s.\text{safe-capture-knight1}(s), Op, S$) (illustrated across the top). Crosses denote an empty intersection	81
3.15	The final pattern in Prolog for a rook fork in the rook-king, knight-king end-game in chess. Derived from compiling a <i>Many-threat</i> proof with two threats, one where the rook captures the king, the other where the rook threatens to safely capture the knight.	82

5.1	A geometric representation of the pattern <i>rook-takes-king</i> . See Figure 3.8 for the logical and graphical representation. Note the components of the pattern are marked on the right	103
5.2	A geometric representation of the pattern <i>rook-takes-knight1</i> , illustrated in graphical form in Figure 3.9	104
5.3	Geometric representation of <i>make-false(rook-takes-king(s), Op, S)</i>	106
5.4	Operator <i>Sub-spaces</i> and linear constraint sets for the king, knight and rook	107
5.5	Geometric representation of the king operator subspaces taking into account edge affects	109
5.6	Determining consistent moves of the black king that <i>maintain-true rook-takes-king</i>	115
5.7	Detail of <i>Simplify</i> determining the consistent solution to <i>maintain-true rook-takes-king</i> marked (a) in Figure 5.6	116
5.8	Detail of <i>Simplify</i> determining an inconsistent case during compilation of <i>maintain-true rook-takes-king</i> marked (b) in Figure 5.6	117
5.9	Detail of <i>Simplify</i> tightening the region bounds in the <i>X</i> dimension during compilation of <i>maintain-true rook-takes-king</i> marked (c) in Figure 5.6	118
5.10	The final clustering stage combining individual solutions to compiling <i>maintain-true rook-takes-king</i>	118
5.11	A graph showing the time in mS needed to lookup all intersecting patterns as a function of the size of the database. The top line is where the database is represented as a list and lookup is linear search. The lower line is where the database is indexed using a rectangle tree. Execution times for all calls to <i>Lookup-Intersection($\nu(s), PLoss$)</i> are illustrated during compilation of the KRK chess ending.	120

7.1	Dynamic program deriving a loss pattern from a <i>one-threat</i> proof for the king-rook-king chess endgame	135
7.2	Dynamic program deriving a loss pattern from a <i>no-threat</i> proof for the king-rook-king chess endgame	137
7.3	Dynamic program deriving two loss patterns (4) and (5) from <i>many-threat</i> proofs for the king-rook-king-knight chess endgame.	139
7.4	Dynamic program deriving two optimal win patterns for the king-rook-king chess endgame. Patterns (1) and (2) are new black-to-play loss patterns that are used to generate the new white-to-play win patterns (3), (5) and (6) through the influence relations shown. The patterns (4) and (7) are previously determined white-to-play wins.	142
8.1	The number of <i>instances</i> as a function of ply for the problem KRK with black-to-move and lose compared to the number of <i>patterns</i> generated by the compile. Note the log vertical scale. .	150
8.2	The number of <i>instances</i> as a function of ply for the problem KRK with white-to-move and lose compared to the number of <i>patterns</i> generated by the compile. Note the log vertical scale. .	150
8.3	The number of <i>instances</i> as a function of ply for the problem KRKN with black-to-move and lose compared to the number of <i>patterns</i> generated by the compile. Note the log vertical scale.	151
8.4	The number of <i>instances</i> as a function of ply for the problem KRKN with white-to-move and lose compared to the number of <i>patterns</i> generated by the compile. Note the log vertical scale.	151
8.5	The average ratio between the branching factors of the abstract search space and the extensional search space as a function of ply for black-to-move. Both KRK and KRKN endings are shown.	152

8.6	The percentage of extensional instances covered as a function of the compile time for the KRK problem when using <i>Breadth-first search</i> and <i>Best-first search</i> . The numbers marked on <i>Breadth-first search</i> are the Ply achieved.	154
8.7	The percentage of extensional instances covered as a function of the compile time for the KRKN problem when using <i>Breadth-first search</i> and <i>Best-first search</i>	155
8.8	The average lookup time (in mS) as a function of the number of patterns produced by the compiler for the KRK ending . . .	158
8.9	The coverage achieved as a function of run time for a king-man ending in checkers.	159
8.10	The coverage achieved as a function of run time for a king-king ending in checkers.	160
10.1	Example of a white-to-move-and-win position from the KRKN ending.	179
10.2	A pattern that is generated by the compiler that describes the position illustrated in Figure 10.1. Note that the white king, black knight and white rook are in fixed positions and the black king is prohibited from the gray region.	179

List of Tables

2.1	A definition of the influence relations	20
3.1	The inputs and outputs of algorithm that computes the influence relations	36
3.2	Compiling <i>maintain-true</i> ($\lambda s.\nu(s), Op, S$), when $\nu(s) \Leftrightarrow c_0 \wedge \neg c_1$	55
3.3	The <i>Set-Difference</i> (l_0, l_1) function	57
3.4	Compiling <i>make-false</i> ($\lambda s.\nu(s), Op, S$), when $\nu(s) \Leftrightarrow c_0 \wedge \neg c_1$	59
3.5	Compiling <i>make-true</i> ($\lambda s.\nu(s), Op, S$), when $\nu(s) \Leftrightarrow c_0 \wedge \neg c_1$	60
3.6	The <i>Set-Intersection</i> (l_0, l_1) function	62
5.1	Representation of geometric patterns	102
7.1	The databases used by the dynamic program	129
7.2	A definition of <i>Cover-Pproofs</i>	134
7.3	The <i>Compile-all-wins</i> algorithm	141
8.1	Time in minutes needed to reach a coverage goal when using best-first or breadth-first search	157

Correct Abstraction in Counter-planning: A Knowledge Compilation Approach

Chapter 1

Introduction

A major goal of Machine Intelligence is to develop techniques that enable users to easily construct effective problem solving systems. Ideally, such problem solvers will be complete, correct, optimal and efficient. A *complete* problem solver is one that can solve all possible problem instances from a given class of problems. A *correct* problem solver is one that always returns the correct solution. An *optimal* problem solver always returns the best solution, such as the shortest solution or the one that achieves the most valuable goal. An *efficient* problem solver is one that returns the answer within a reasonable amount of time and using a reasonable amount of space.

Constructing such ideal problem solvers is difficult because of the strong tradeoffs that exist among these desired characteristics. In particular, ensuring efficiency imposes the most difficulty on system builders. This can be demonstrated by considering how easy it is to construct problem solvers that are inefficient yet satisfy the other requirements. When ignoring efficiency, the system designer can describe the problem solver as a simple search-intensive procedure, that given enough time, can solve any problem in the class correctly and optimally. For instance, to ensure optimality, the procedure can perform exhaustive search through the

space of all possible solutions and return the best solution. In the terminology in [McCarthy and Hayes, 69] the initial problem solver is *epistemologically adequate* but not *heuristically adequate*. Since it is so easy to specify epistemologically adequate problem solvers, there is much interest in developing *knowledge compilation* techniques that can automatically compile such search-intensive specifications into efficient form.

Most work in knowledge compilation for problem solvers has assumed that the problem solving domain involves only one agent who acts in an otherwise static world. While this simplification is useful in some domains, it is unrealistic for dynamic worlds that have independent actors each with their own goals that may conflict with the goals of the problem solving agent. These other agents increase the complexity of problem solving, because to ensure correctness, the problem solving procedure must now consider all possible actions of the other agents. For example, in chess where we have two agents with conflicting goals, solving a check mate problem correctly will *necessarily* involve exploring all possible actions that the opponent could make, because if there exists an action by the opponent that will prevent the loss, the opponent will take it.

This thesis addresses the problem of compiling *correct* knowledge for domains—such as counter planning above—that necessitate exhaustive search for correct behavior. The requirement of learning correct knowledge complicates the knowledge compilation process, because the intractability of the search required during problem solving cannot be avoided during learning.

Our approach to overcoming the intractability of problem solving and learning can best be understood by first considering how it is that human experts perform so efficiently and accurately in these difficult domains. A human chess expert, for example, is capable of quickly solving hard problems that when solved by computers require millions of nodes to be searched [Anantharaman Campbell and Hsu 88]. How is it possible that the human can avoid the massive search, yet still achieve

correct performance? The answer lies in the expert’s use of an *abstracted* search space that is smaller, yet equivalent to the exhaustive search of the computer [Chase and Simon 72].

This thesis introduces a new domain-independent abstraction mechanism that identifies useful abstractions in counter-planning domains. The emphasis in this work is on abstractions that preserve correctness while effectively simplifying learning and problem solving. Hence, the abstraction mechanism differs from previous work in *weak abstraction* [Knoblock 90], [Mostow and Prieditis 89], [Tenenbergs 87] in that the reduction in the search space is not gained by simply removing selected literals from the space.

Since the focus of this thesis is compiling useful knowledge when dealing with adversaries, many of the complexities of real world multi-agent planning such as incomplete information and robotics are not considered. Rather, we limit our investigation to domains that involve complete information and only two agents, such as chess.

1.1 Previous Approaches

There have been two quite different approaches to compiling counter-planning domains in the past. One approach—explanation-based learning—has the advantage that the method provides an abstraction mechanism [Cohen 90]. However, as we will later show, this approach leads to either intractability of explanations or incorrect generalizations. The other approach—referred to as the database approach—does not employ any abstraction or training examples. Rather, a complete database of problem instance/solution pairs is constructed during compilation. The principal advantage of this approach is that the compiled problem solver is ideal in that it is guaranteed to be correct, optimal, complete and efficient. However, the complexity of generating and storing the lookup table prevents the method from scaling to large domains.

1.1.1 Explanation-based Approaches

Explanation-based learning (EBL) is a knowledge intensive technique that has demonstrated success in speeding up performance in a variety of problem solving situations [Minton 88a], [Mitchell Keller and Kedar-Cabelli 86]. EBL is a three step process that takes as input a problem instance to be solved, a target goal (the goal to be achieved) and a specification of the problem solver. First, the problem solver specification is applied to solve the given problem instance. This search tree generated can be thought of as a proof that the instance achieves the goal. In the second step, the proof is generalized by eliminating components of the tree that refer only to the particular example, while retaining those parts of the tree that relate to the domain theory. Finally, the weakest precondition of the proof is extracted and simplified. This precondition describes a set of problem instances that, if solved by the problem solver, would all achieve the same goal and generate the same proof tree. Problem solving performance can be improved because the next time that any instance from this set is encountered, there is no need to run the problem solver again, we can simply match the instance against the precondition.

Although EBL has been successful in many planning and problem solving domains, it is difficult to apply successfully in domains that involve exhaustive search such as counter-planning. The problem arises because of the complexity of two steps in the EBL method: (a) constructing a complete proof and (b) extracting a correct and efficient weakest precondition from the proof.

Proof generation: Since in counter-planning, proving goal achievement involves considering all possible defensive actions by the opponent, proof construction is exponential in the depth of the proof.

Generalized proof analysis: The goal of this analysis step is to extract a sufficient condition from the proof that is both efficient and correct. By efficient, we mean that the sufficient condition must be directly evaluable in the current

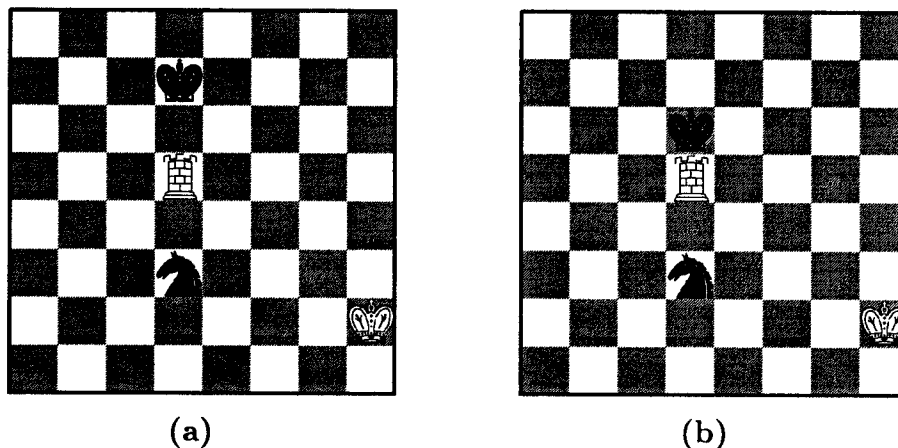


Figure 1.1: Two similar chess positions both with black to play. (a) is a loss position, while (b) is a draw

situation. Since the proof tree includes the application of operators, these operators must be excluded from the sufficient condition. This process is simple in single agent domains such as planning, because the operators are existentially quantified [Hirsh 87]. However, in counter-planning, because we must consider all possible defensive actions by the opponent, the proofs will also include *universal quantification* over operators. There is no simple solution to this problem (such as treating the \forall as an “and” node) since the quantification is over *all possible operators* not just those that occurred in the particular example.

This last point is important since even if we are prepared to invest the resources in constructing a complete search tree, the resulting generalization will not be correct. This problem can be easily demonstrated by considering applying EBL to learn a condition that can recognize lost positions from the example illustrated in 1.1(a). To prove that this position is a loss involves searching the 16 available moves by black and demonstrating that each either results in losing the king or losing the knight. EBL generalization can correctly compute the weakest precondition of each individual branch of this tree, since they involve only existential quantifi-

cation. However, the universal quantification at the root must be eliminated. One approach is to *assume* that the operators that were applicable to this position are all the operators that will ever be applicable in the resulting precondition, and replace the universal quantification with an *and* node. Simplifying the result produces the condition illustrated in Figure 1.2(a). However, this is an overgeneral and therefore incorrect rule, because it classifies the position shown in 1.1(b) as a loss when in fact it is a draw. The error arose because the assumption was wrong—the original explanation was incomplete—in fact there are other applicable operators (such as taking the rook with the king).

EBL approaches to learning in these domains have been forced to tolerate errors caused by this overgeneralization. The emphasis has been on developing techniques that learn how to avoid errors once they have been made [Tadepalli 89], [Chien 89]. Eventually, as mistakes are made and corrected, these systems can converge to correctness.

There are some significant problems with this approach. First, and foremost, the system will make mistakes. Secondly, the system will require careful training to reach a level of useful performance. Since our overall goal is to simplify the construction of problem solvers, there is a danger that the burden of teaching will undo any advantages gained by applying learning methods. This burden could be quite considerable, because it will be up to the teacher to detect and correct any error the system makes. Moreover, since these errors are due to incomplete searches made by the problem solver, the teacher must have made a more thorough search in order to detect the error. Hence, the responsibility for performing exhaustive search—which was the focus of our learning approach—has shifted to the teacher.

To summarize, although explanation-based learning provides an abstraction mechanism through the generalization of the explanations [Cohen 90], the method leads to unacceptable errors which must be corrected at great expense by a teacher.

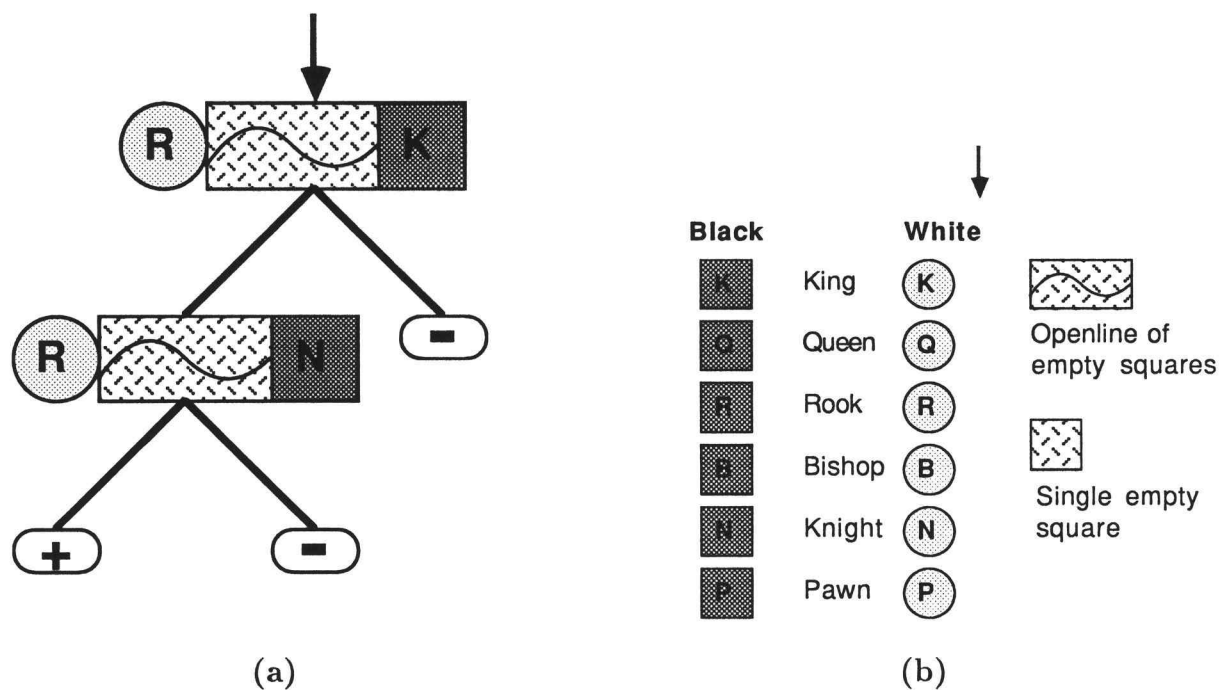


Figure 1.2: (a) The sufficient condition learned by EBL from the example in Figure 1.1(a) expressed as a decision tree. If the condition tested at a node is true the left branch is taken. (b) The key for the symbols used to illustrate the patterns learned in chess.

1.1.2 Database Approaches

In the database approach, the compiler constructs an exhaustive lookup table of all possible problem instances and computes the best action to take for each instance [Clark 1977], [Thompson 86]. The resulting lookup table can be considered the ideal problem solver, since correctness and optimality are preserved, all problem instances can be solved, and problem solving performance is very efficient (simple table lookup). In fact, the method, when applied to chess, has created “super experts” that are unbeatable by the best human players and can solve complex endings that require over 100 moves to win. In checkers, the large databases produced by the method form an essential part of a program that has earned the right to play for the world title [Schaeffer 91].

To ensure correctness and optimality of the lookup table, the compiler performs an exhaustive search of the problem solving space. Since this search could potentially be unbounded, it is important to organize the search effectively. One obvious, but impractical search strategy would be to work through each position in the table, perform the forward exhaustive search, then fill in the result in the table. Clearly, such a forward search strategy is too underconstrained since it is unbounded.¹ The insight reported in [Clark 1977] was to notice that, in contrast to the forward search, a *backwards* search from the known goal positions (i.e., termination positions) is much more constrained. In [Thompson 86], a dynamic programming method was described that effectively performs the backwards search.

To understand the method, it is sufficient to understand how the database would determine that the position in Figure 1.1(a) is a loss. To do this, the system must demonstrate that all possible actions that can be taken from this position lead to a win for the opponent. Initially, the table is filled with all the known white-win termination positions (in this case, all these consist of positions where the king or knight can be safely captured by white). Since the position in Figure 1.1(a) is not

¹How do we detect that the position being searched is a draw?

a termination position it is initialized with the count of possible moves that can be made by the players (black, 16; white 16). The backwards search is initiated by computing all possible *predecessors* of the termination positions by applying the standard moves of chess backwards (called unmoves) for black. Each time such an unmove generates the position in Figure 1.1(a), its count of black moves is decremented by 1. When the count reaches 0, we have shown that all available black moves lead to *successors* that are white wins—and hence the position is a loss for black. Each new loss position can be similarly “unmoved” for white to create new won positions. The process continues until there are no new won or lost positions. All remaining unclassified positions are labeled as draws.

The major advantage of this approach is that it produces the ideal problem solver by enumerating a *finite* space. However, the usefulness of the approach is limited by the computational complexity of searching and storing this space. In chess, with n pieces, both the compile time and the storage of the table² are lower bounded by 64^n . In general, when the problem instances are described by m binary features or sensors, the table size will be 2^m , hence the principal disadvantage of this method is that it cannot scale up to larger problem sizes.

1.2 Overview of the Approach

This thesis introduces a compilation method for adversary situations based on the database approach. The principal strengths of this approach—no training, and correct and efficient performance—are retained. The focus of this thesis has been overcoming the principal weakness of the approach: the exponential growth in database size and therefore compile time. Two techniques are explored in this thesis:

Correct abstractions: An abstraction mechanism has been developed that reduces the size of the database while retaining its correctness and efficiency.

²This size can be reduced by a factor of 8 for some combinations of pieces by exploiting rotational and reflexive symmetry.

Rather than learning individual position/action pairs, the compiler learns an equivalent but smaller set of pattern/action pairs. The reduced database is constructed by a process of *abstract enumeration*: a search backwards from termination *patterns* rather than *positions*.

Coverage/Compile time tradeoff: A strategy for constructing the database has been developed that best exploits the tradeoff between the coverage achieved over a domain and the time spent compiling the domain. This tradeoff exists because even with abstraction, it may be impractical to run the compiler to completion. Hence, it is wise for the compiler to make the best use of the limited time it has available. The approach explored here is to control the enumeration process so that the abstractions are generated best first: The pattern/action pairs with the highest coverage are generated before those with lower coverage.

These two techniques allow us to construct a near ideal problem solver that, given a problem instance, either provides a solution that is guaranteed to be correct or reports “solution unknown.” Let us consider these two techniques in more detail.

1.2.1 Correct Abstractions

The manual design of useful abstractions in counter-planning is very difficult. Campbell, [Campbell 88], earned a Ph.D. by developing useful abstractions in pawn-knight endings in chess. Michie, in [Michie, 82], describes several attempts at engineering abstractions for solving the king-rook versus king chess ending. Quinlan, [Quinlan 83], attempted the manual design of abstractions for lost-in- n -ply for the king-rook versus king-knight chess ending (of which Figure 1.1(a) is example). He spent 2 person-weeks developing abstractions for the case when $n = 2$, 2 person-months when $n = 3$, and gave up when $n = 4$. The difficulty of this task can be recognized when we consider that at each stage during the design process, Quinlan had a complete and correct database of positions with which to compare and

evaluate his designed abstractions. The problem is that each pattern initially engineered will likely have exceptions, and rules written to handle those exceptions will likely have exceptions too. For example, the incorrect decision tree illustrated in Figure 1.2 is lacking two exceptions that are encoded as additional tests in the correct decision tree illustrated in Figure 1.3.

Since manual design is so difficult, there has been much emphasis on developing methods that automatically identify useful abstractions. An effective methodology in this area is to define rigorously what it means to be a useful abstraction, then have the machine use this definition as a *generator* of abstractions. In [Knoblock 90], useful abstractions for planning have been automatically produced by having the machine exploit the property of ordered monotonicity—a property that minimizes the backtracking among levels in the abstract plan. While in [Subramanian 89], better conceptualizations of problems are automatically produced by exploiting a theory of irrelevance—a theory that allows the system to prove components of a conceptualization irrelevant and hence replace them.

Applying the same methodology in our domain immediately leads us to address the following questions: What is a useful abstraction in counter-planning? How can a definition of useful abstractions be processed to generate such abstractions?

Consider the first question: A useful abstraction in counter-planning is one that reduces the size of the search space while preserving correctness. Counter planning poses a distinct challenge for automating the abstraction process because of the need to consider all possible actions of the opponent to ensure goal achievement. Recall that the reason for the exhaustive search is because if there exists any means for the opponent to prevent goal achievement it will be taken. The naive way to ensure this is to search all possible actions by the opponent. However, many of these actions will be irrelevant, because they cannot affect or influence the outcome. For example in Figure 1.1(a), moves by the king along the line of the rook attack are

irrelevant, since they do not eliminate the rook threat. Hence, by focusing on how actions *influence* goal achievement, we can distinguish between relevant actions that must be considered during search and irrelevant actions that can be safely ignored. This thesis introduces an abstraction mechanism based on a domain-independent *theory of influence* that allows the system to reason about the effects of actions on goal achievement.

To understand how a theory of influence could reduce the search space while preserve correctness, it is useful to contrast two alternative proofs as to why the position in Figure 1.1(a) is lost. The first proof is the naive one, when all possible actions are considered:

Black is lost because all possible applicable actions lead to a situation where there exists an action for white that wins.

We can contrast this with a proof that exploits the theory of influence:

Black is lost because there are two threats, the capture of the king by the rook and the capture of the knight by the rook, and there exists no action by black that can simultaneously eliminate both threats.

Note that in the second explanation, we do not consider all possible actions by black. Rather we focus on the *relevant* actions—those that can influence the outcome by eliminating both threats. All other actions are irrelevant and hence ignored. This leads to a significant reduction in the search space.

More formally, the theory of influence consists of two components. First, we have *influence relations* that define four ways in which an action can influence the truth value of a goal: *make-false* (the actions above that eliminate the threat are making it false), *make-true*, *maintain-true* and *maintain-false*. These four relations describe equivalence classes of actions. All actions within a class influence the goal in the same way. For example, all actions by the king above that make the threat false are considered equivalent. Second, we have *influence proofs*, that define

goal achievement in counter-planning situations using the influence relations. The above explanation is an example of such a proof; in this case, the proof describes one strategy for goal achievement that is a generalization of the familiar concept of a fork. There are many other strategies for goal achievement that can be defined by influence proofs. Each influence proof defines an equivalence class of situations where the same goal is achieved using the same strategy.

Given that we have an abstraction mechanism based on influence relations and influence proofs, the question becomes, how can these definitions be applied to generate useful abstractions? The key lies in recognizing that the theory of influence defines a space of influence proofs each describing a distinct strategy for goal achievement. To generate useful abstractions, the system simply generates proofs from this space and compiles them into the desired pattern/action rules. To simplify this process, the compiler generates the shortest proofs first. For example, the first proofs generated and compiled are for strategies that achieve the goal immediately (such as capturing the king or knight). Then the proofs (such as the fork above) are generated, since they describe goal achievement after only 2 actions. The process continues working back from the simplest proof, each depth of proof using the previously compiled levels. The process is very like the dynamic program described earlier, but in this case, we are working with *patterns* rather than positions, and the proofs are *abstract* rather than extensional.

1.2.2 The Coverage/compile-time Tradeoff

Given that it may be impractical to run the compiler to completion, we must decide how best to organize the compilation during the limited time available, so as to maximize the coverage achieved. To understand how this coverage/compile time tradeoff can be best exploited, it is necessary to understand how coverage is accumulated during compilation. In Figure 1.4(a) we illustrate the behavior of the traditional database approach [Thompson 86]. Here coverage is accumulated

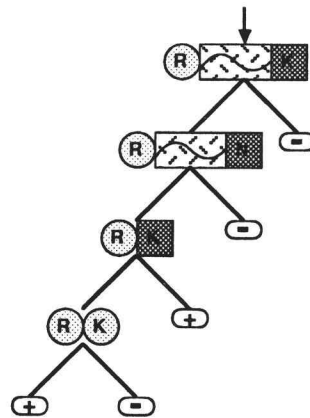


Figure 1.3: The correct decision tree that correctly classifies the examples in Figure 1.1

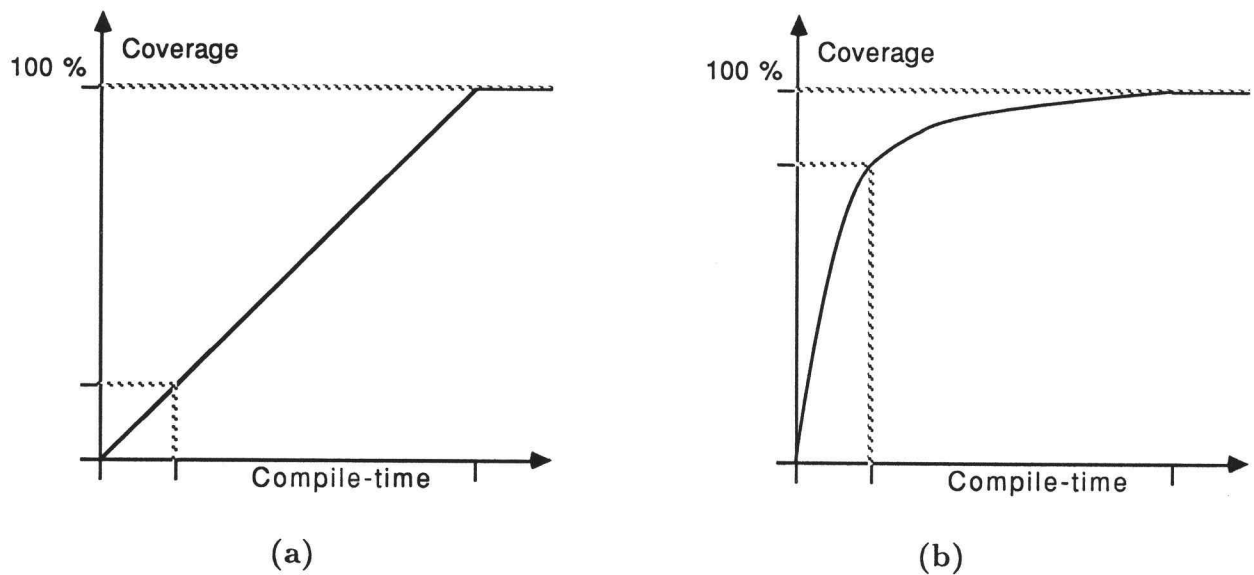


Figure 1.4: (a) The behavior of the database compiler without abstraction: there is a linear relationship between compile time and coverage (b) The behavior of the compiler with abstraction, when the most general patterns are learned first.

linearly³ with compile time, since the database is incrementally constructed one position at a time. With a linear relationship, the tradeoff is not very useful to exploit. By committing say 20% of the total time needed, we can only hope to achieve 20% coverage. Since compile time grows exponentially with problem size, we need to do much better than linear to make it worthwhile to exploit this tradeoff.

Abstraction changes the relationship between compile time and coverage that enable us to better exploit this tradeoff. Coverage can be accumulated at a rate faster than linear by organizing the compilation process to produce the most general patterns (i.e., those with the highest coverage) first. This process is illustrated in Figure 1.4(b). With such a relationship, by committing say 20% of the total time needed, we can achieve perhaps 80% or better coverage.

1.3 Statement of Thesis

Now that we have described the overall approach, we are in a position to state the central thesis of this dissertation:

Speeding up problem-solving in counter-planning domains while maintaining correctness can be effectively addressed by a combination of correctness-preserving abstractions and search heuristics that exploit the compile-time coverage tradeoff.

1.4 Reading Guide

This thesis is divided into ten chapters. Chapter 2 introduces the abstraction mechanism and illustrates how some well known abstractions in counter-planning can be defined within the mechanism. Chapter 3 details the compilation process that translates abstract proofs into efficient pattern/action pairs. Chapter 4 presents an

³This is assuming that we have a uniform distribution over the problem instances, if this is not the case, Thompson's method can do somewhat better.

analysis of the compilation approach proving that the method is sound, although incomplete and intractable. Chapter 5 introduces a geometric representation of the problem space with the goal of overcoming the previously identified problems. Chapter 6 analyses the geometric interpretation and proves that it maintains soundness while overcoming the problem with incompleteness and intractability. Chapter 7 explores strategies to search the space of abstractions. Chapter 8 presents an empirical study of the method compiling a selection of sub-problems in chess and checkers. Chapter 9 reviews related work in abstraction and counter-planning. Chapter 10 completes the thesis with the conclusions that can be drawn from this work and identifies some important open problems in this area of research.

Chapter 2

Abstraction Based on Influence

This chapter introduces the abstraction mechanism and provides some examples of its use in describing well known abstract concepts from counter-planning.

2.1 Goal Achievement in Counter-planning

To introduce the abstraction, let us first consider a simplified form of min/max search where the winning player is known, and optimality concerns, such as achieving the highest value or shortest path, are ignored. Here we consider the case when we have two *actors*, *WIN* and *LOSS*, where actor *WIN* can achieve an advantageous goal G , within n *ply* (i.e., n operator applications by *both* players). In this case we can define the search procedure as follows (note, all free variables are assumed to be universally quantified):

$$\begin{aligned} \text{achieve}(\lambda s.G(s), S, \text{LOSS}, n) &\Leftarrow \\ &\forall Op, o(S, \text{LOSS}, Op), \text{achieve}(G(s), do(Op, S), \text{WIN}, n - 1) \\ \text{achieve}(\lambda s.G(s), S, \text{WIN}, n) &\Leftarrow \\ &\exists Op, o(S, \text{WIN}, Op), \text{achieve}(G(s), do(Op, S), \text{LOSS}, n - 1) \\ \text{achieve}(\lambda s.G(s), S, \text{WIN}, 0) &\Leftarrow \\ &G(S) \end{aligned}$$

Note that the goal to be achieved, $G(s)$ employs *lambda binding* for its situation argument s . This is because we are employing situation calculus to describe

the application of operations to situations. Given an initial situation S_0 , we denote a situation which results from applying some operator Op in S_0 as $do(Op, S_0)$. The lambda binding is necessary, because during construction of the search tree, we need to evaluate the goal G at each leaf. To perform this evaluation, we employ the lambda mechanism to bind s , the argument of G , to the leaf situation, which is a composition of the operators that were applied to reach that leaf (see [Genesereth and Nilsson 87] Chapter 11 for a tutorial on situation calculus).

The first rule defines the situation when *LOSS* is to play. Since the losing player is to move, we must consider all possible operators that are available (returned by the relation $o(S, LOSS, Op)$) to ensure correctness. Hence we include universal quantification over the operators available. The second rule defines the situation when *WIN* is to play. Here, because the goal is advantageous, we need only show that any one of the operators available to *WIN* leads to goal achievement. Hence we include existential quantification over the operators available. The final rule describes the termination condition, where the advantageous goal is recognized as achieved.

These definitions can be used to solve problems in a domain by providing definitions of relation o , G and frame axioms. For example, to employ this definition to solve problems in chess such as the one given in Figure 1.1(a), o would be defined as a mapping from the current situation and the side moving to the legal moves of chess, while $G(s)$ would be defined as a predicate that was true when s described a situation where the game was won by white.

2.2 Influence Proofs for Goal Achievement

Although the definitions above can be used to solve problems, they are not useful for knowledge compilation because of the previously discussed problem—the proof sentences include quantification over all possible operators of the losing side. The abstraction process introduced in this chapter is based on exploiting alternative

proofs of goal achievement—called influence proofs—that include quantification over only those operators that are *relevant* to goal achievement. The proofs distinguish relevant from irrelevant operators by making explicit in the proofs the goals that are threatened to be achieved and how the operators affect or influence the truth value of these goals.

We define a space of influence axioms, where each axiom describes a distinct strategy for goal achievement. The axioms are like the axioms above in that we define goal achievement in depth i in terms of goal achievement at depth $i - 1$. However, unlike the axioms above, where we have only one sentence defining goal achievement, here there are many distinct influence axioms for each depth i .

The first step in defining the influence axioms is to introduce a simple abstraction mechanism where a set of situations, in which an advantageous goal can be achieved, is represented by a set of abstract *goal patterns*. For example, we can represent all the situations where the advantageous goal is achieved immediately (i.e., those defined by the predicate $G(s)$ above) as a small set of goal patterns, each denoted $\nu(s)^G$. Hence, rather than specifying $G(s)$ as a monolithic “black box,” which implicitly defines the set of terminating situations, let us define $G(s)$ explicitly as a disjunction of unique goal patterns: $G(s) \Leftrightarrow \nu(s)_1^G \vee \nu(s)_2^G \vee \dots \vee \nu(s)_m^G$. For example, to define $achieve(G(s), S, LOSS, 0)$ for a particular endgame in chess—the king-rook, king-knight ending (of which Figure 1.1(a) and (b) are examples)—we specify two goal patterns, one describing the situations where the black king is captured, and the other describing the safe capture of the knight (i.e., without a recapture by black).

Let us assume, for the moment, that the same abstraction mechanism can be applied to those situations where goals are achieved in any number of operator applications, rather than just at termination. Hence, we will assume that for some proof depth i , $achieve(G(s), S, -, i)$ can be defined as a disjunction of goal patterns $\nu(s)_1^{G_i} \vee \nu(s)_2^{G_i} \vee \dots \vee \nu(s)_k^{G_i}$. When $i = 0$ (i.e., at termination) the goal patterns

$$(1) \text{ make-true}(\lambda s.\nu^G(s), Op, S) \Leftrightarrow \neg\nu^G(S) \wedge o(S, Op) \wedge \nu^G(do(Op, S))$$

“*make-true*($\lambda s.\nu^G(s)$, Op , S) describes those cases in which the goal pattern ν^G is false in situation S , Op can legally be applied in S , and ν^G is true after applying Op . In other words, applying Op in S makes ν^G true (and it was not true before).”

$$(2) \text{ make-false}(\lambda s.\nu^G(s), Op, S) \Leftrightarrow \nu^G(S) \wedge o(S, Op) \wedge \neg\nu^G(do(Op, S))$$

“*make-false*($\lambda s.\nu^G(s)$, Op , S) describes those cases in which the goal pattern ν^G is true in situation S , Op can legally be applied in S , and ν^G is false after applying Op . In other words, applying Op in S makes ν^G false (and it was true before).”

$$(3) \text{ maintain-true}(\lambda s.\nu^G(s), Op, S) \Leftrightarrow \nu^G(S) \wedge o(S, Op) \wedge \nu^G(do(Op, S))$$

“*maintain-true*($\lambda s.\nu^G(s)$, Op , S) describes those cases in which the goal pattern ν^G is true in situation S , Op can legally be applied in S , and ν^G is true after applying Op . In other words, applying Op in S maintains ν^G true.”

$$(4) \text{ maintain-false}(\lambda s.\nu^G(s), Op, S) \Leftrightarrow \neg\nu^G(S) \wedge o(S, Op) \wedge \neg\nu^G(do(Op, S))$$

“*maintain-false*($\lambda s.\nu^G(s)$, Op , S) describes those cases in which the goal pattern ν^G is false in situation S , Op can legally be applied in S , and ν^G is false after applying Op . In other words, applying Op in S maintains ν^G false.”

Table 2.1: A definition of the influence relations

are provided by the user. We will proceed to show how for $i > 0$, the system itself, through a compilation process applied to the influence axioms, can automatically generate the goal patterns.

The following description of the abstraction process begins with a description influence relations, which define how the truth value of a goal pattern can be affected by an operator, then a description of influence axioms, which define goal achievement in terms of goal patterns and influence relations.

2.2.1 Influence Relations

In order to define these influence axioms, we exploit *influence relations* that

describe the four ways in which an operator can affect the truth value of a goal pattern: *make-true*, *make-false*, *maintain-true* and *maintain-false*. Each of these influence relations is a relation over a goal pattern, an operator set and a situation. They are defined in Table 2.1. Note that each definition employs *lambda binding* for the situations in the goal patterns. This notation is employed for a reason similar to its use before: to gain access to the situation variable of a goal pattern. In this case, we need access because the goal pattern is evaluated in two different situations, in the initial situation S , and the situation following the operator application, $do(Op,S)$. We call these primitives *influence relations* because they define the ways in which the application of operators influence the truth of goals.

2.2.2 Influence Axioms

We have introduced the two components of the abstraction mechanism, goal patterns and influence relations. We can now define the influence axioms for goal achievement in terms of these components. As we will see, the number of different axioms for a depth i depends upon the number of axioms for depth $i - 1$. We will give the simplest case first, and assume we have only one goal pattern for depth $i - 1$, denoted $\nu^{G_{i-1}}(s_1)$. In this case, with *LOSS* to play we define goal achievement for *WIN* for depth i as follows:

$$(5) \text{ achieve}(\lambda s.G(s), S, \text{LOSS}, i) \Leftarrow \\ \nu^{G_{i-1}}(S) \wedge \forall Op, o(S, \text{LOSS}, Op), \neg \text{make-false}(\lambda s.\nu^{G_{i-1}}(s), Op, S) \\ \vee \neg \nu^{G_{i-1}}(S) \wedge \forall Op, o(S, \text{LOSS}, Op), \text{make-true}(\lambda s.\nu^{G_{i-1}}(s), Op, S)$$

There are two cases: (a) $\nu^{G_{i-1}}(s_1)$ is already true (i.e., *WIN* is threatening to achieve the goal, all that is needed is a change of the side to move) and none of the operators make $\nu^{G_{i-1}}(s_1)$ false (i.e., the losing player cannot avoid the threat); or (b) $\nu^{G_{i-1}}(s_1)$ is not true and all the operators make it true. This proof differs from the previous non-influence proof in that the case when *WIN* is threatening to achieve a goal is made explicit. Under this condition, we need not consider all

possible operators; rather we can limit the quantification to only those operators that make this threat false. This is the simplest case when we have only one goal pattern at depth $i - 1$, and hence one possible threat situation. In general, there will be many goal patterns for *WIN* in $i - 1$ ply and hence many possible threat situations. The case when we have only 2 goal patterns is given below. Here goal achievement for depth $i - 1$ is defined as $achieve(\lambda s.G(s), do(Op, S), WIN, i - 1) \Leftrightarrow \nu_1^{G_{n-1}}(s) \vee \nu_2^{G_{n-1}}(s)$. The influence axioms for the case when we have more than 2 goal patterns can be easily induced from this case. Here, considering when *LOSS* is to move, goal achievement is defined:

$$\begin{aligned}
(6) \text{achieve}(\lambda s.G(s), S, LOSS, i) \Leftarrow & \\
& \vee \nu_1^{G_{i-1}}(S) \wedge \nu_2^{G_{i-1}}(S) \wedge \forall Op, o(S, LOSS, Op), [\neg \text{make-false}(\lambda s.\nu_1^{G_{i-1}}(s), Op, S) \\
& \qquad \qquad \qquad \vee \neg \text{make-false}(\lambda s.\nu_2^{G_{i-1}}(s), Op, S)] \\
& \vee \neg \nu_1^{G_{i-1}}(S) \wedge \nu_2^{G_{i-1}}(S) \wedge \forall Op, o(S, LOSS, Op), [\text{make-true}(\lambda s.\nu_1^{G_{i-1}}(s), Op, S) \\
& \qquad \qquad \qquad \vee \text{maintain-true}(\lambda s.\nu_2^{G_{i-1}}(s), Op, S)] \\
& \vee \nu_1^{G_{i-1}}(S) \wedge \neg \nu_2^{G_{i-1}}(S) \wedge \forall Op, o(S, LOSS, Op), [\text{maintain-true}(\lambda s.\nu_1^{G_{i-1}}(s), Op, S) \\
& \qquad \qquad \qquad \vee \text{make-true}(\lambda s.\nu_2^{G_{i-1}}(s), Op, S)] \\
& \vee \neg \nu_1^{G_{i-1}}(S) \wedge \neg \nu_2^{G_{i-1}}(S) \wedge \forall Op, o(S, LOSS, Op), [\text{make-true}(\lambda s.\nu_1^{G_{i-1}}(s), Op, S) \\
& \qquad \qquad \qquad \vee \text{make-true}(\lambda s.\nu_2^{G_{i-1}}(s), Op, S)]
\end{aligned}$$

Here we have 4 cases, each describing a unique threat situation. The first case defines the situation where the opponent is threatening goal achievement two distinct ways and there is no operator available that can eliminate both threats. This is an example of the well known tactic of *fork*. The second and third cases define situations where there is a single threat and all moves either maintain the threat or make some other threat true. In chess, these proofs can be used to describe the tactic of *skewer*. The final case describes the situation where there are no threats, but each move available makes one of two threats true.

In general, at depth i , we have n previous goal patterns for depth $i - 1$, $\nu_1^{G_{i-1}}(s) \vee \nu_2^{G_{i-1}}(s) \vee \dots \vee \nu_n^{G_{i-1}}(s) \Leftrightarrow achieve(G(s), do(Op, S), -, i - 1)$, then we have 2^n axioms. Each axiom describes how the goal can be achieved given that some subset of known goal patterns is true initially—that is, we describe goal achievement

under all possible threat combinations. To simplify this general case, let us partition these threat combinations into three cases:

No threat This axiom defines goal achievement when none of the goal patterns are true in the initial situation. Here all operators *make-true* some set of the n goal patterns. There will be one such axiom for depth i when we have n goal patterns for depth $i - 1$.

One threat This axiom defines goal achievement when only one goal pattern is true in the initial situation. Here all operators either *maintain-true* the one threat pattern or *make-true* some disjunction of other goal patterns. There will be n such axiom for depth i when we have n goal patterns for depth $i - 1$.

Many threat This axiom defines goal achievement when more than one goal pattern is true in the initial situation. Here none the operators make all the threat patterns false. There will be $2^n - (n + 1)$ such axiom for depth i when we have n goal patterns for depth $i - 1$.

These influence axioms are much more useful for learning than the simplified min-max definition given initially. The reason lies in limiting the quantification of the operators available in the proofs to only those that are relevant to goal achievement. The proofs make explicit the goals that are threatened to be achieved in each situation and how the operators affect the goals. Rather than considering all possible operators, we limit the quantification to only those operators that can affect the outcome of the search by influencing the truth value of those goals already known to be true.

2.2.3 Chess Proofs

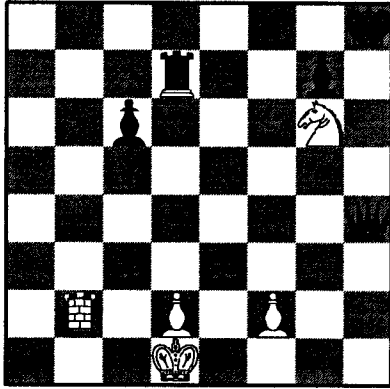
The previous analysis applies to any two agent counter-planning domain. In this section, we adapt the analysis for chess. The only detail to consider, which was

ignored in the above formalism, is the implementation of the function o , which generates the operators in the given situation, S . In chess, the operators available depend upon the side that is to move and whether that side is in-check or not. The complexity caused by the effect of check on the legal moves available can be simplified by dealing with *pseudo-moves* rather than legal-moves, where pseudo-moves are defined as those moves that are available if we ignore the in-check constraint. Then legal moves can be defined as those pseudo-moves that do not result in the moving side being in check. Hence, to define legal moves from pseudo-moves, we employ influence relations applied to the *in-check* constraint: to generate legal-moves, we first generate pseudo-moves. Then, if the moving side is *in-check*, we retain only those that make the *in-check* constraint *false*. Otherwise, if the moving side is not *in-check*, we retain only those pseudo-moves that do not make *in-check* true. Given that the relation $po(S, Side, Op)$ defines the mapping between the current situation and the side to play to the pseudo-moves available, and $in-check(S, Side)$ is true when $Side$ is in check in situation S , then we can define the legal moves as follows:

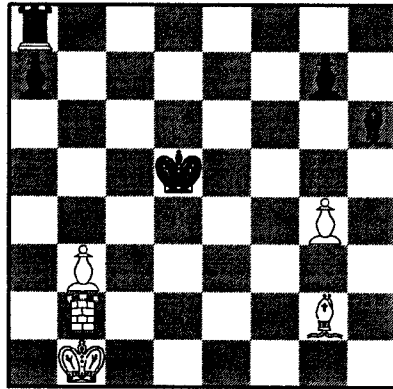
$$(7) \quad o(S, Side, Op) \Leftrightarrow \\ po(S, Side, Op) \wedge [\quad (\quad in-check(S, Side) \\ \quad \wedge \quad make-false(\lambda s.in-check(s, Side), Op, S)), \\ \quad \vee \quad (\quad \neg in-check(S, Side) \\ \quad \wedge \quad \neg make-true(\lambda s.in-check(s, Side), Op, S))]]$$

Incorporating this into the expressions above leads to a doubling of the number of axioms. We illustrate this by incorporating (8) into (5):

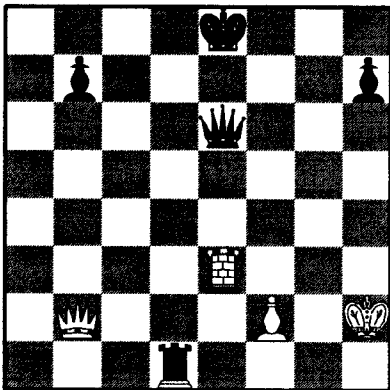
$$(8) \quad achieve(\lambda s.G(s), S, LOSS, i) \Leftarrow \\ \vee in-check(S, LOSS) \wedge \nu^{G_{i-1}}(S) \\ \wedge \forall Op, po(S, LOSS, Op) [\quad \neg make-false(\lambda s.in-check(s, LOSS), Op, S) \\ \quad \vee \quad \neg make-false(\lambda s.\nu^{G_{i-1}}(s), Op, S)] \\ \vee \neg in-check(S, LOSS) \wedge \nu^{G_{i-1}}(S) \\ \wedge \forall Op, po(S, LOSS, Op), [\quad make-true(\lambda s.in-check(s, LOSS), Op, S) \\ \quad \vee \quad maintain-true(\lambda s.\nu^{G_{i-1}}(s), Op, S)] \\ \vee in-check(S, LOSS) \wedge \neg \nu^{G_{i-1}}(S) \\ \wedge \forall Op, po(S, LOSS, Op), [\quad maintain-true(\lambda s.in-check(s, LOSS), Op, S) \\ \quad \vee \quad make-true(\lambda s.\nu^{G_{i-1}}(s), Op, S)]]$$



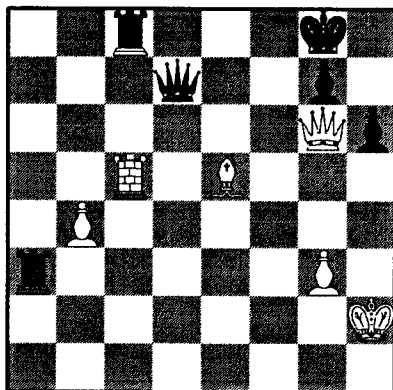
(a) Fork, Black-to-play



(b) Skewer, Black-to-play



(c) Pin, Black-to-play



(d) Overworked piece, White-to-play

Figure 2.1: Examples of different chess concepts that can be described by influence proofs

$$\begin{aligned} & \forall \neg \text{in-check}(S, \text{LOSS}) \wedge \neg \nu^{G_{i-1}}(S) \\ & \wedge \forall Op, po(S, \text{LOSS}, Op), [\text{make-true}(\lambda s. \text{in-check}(s, \text{LOSS}), Op, S) \\ & \quad \vee \text{make-true}(\lambda s. \nu^{G_{i-1}}(s), Op, S)] \end{aligned}$$

2.3 Examples of Influence Proofs

The influence proofs have been introduced as an abstraction mechanism for simplifying learning. However, the proofs also happen to be useful in defining many of the well known abstractions for achieving goals in counter-planning. These tactical devices for achieving goals have been identified over many years during the study of

games and other situations that involve conflict. Although they are usually thought of as a “bag of tricks” with little in common, the influence theory provides a uniform language with which to define them. To illustrate this power of the influence theory, there follows a description of four useful tactics in chess, each with an example position, an english description of the idea, and the relevant proof sentence.

2.3.1 Fork

The fork is a universal tactic found in many counter-planning situations. We illustrate an example from chess in Figure 2.1(a). Here the black knight on **g6** simultaneously threatens both the black king on **h8** and the black queen on **h4**. There does not exist a single move for black that can eliminate both threats, so the queen will be lost.

This tactic is defined by the following proof of depth 2:

$$\begin{aligned}
 & \text{achieve}(\lambda s.G(s), S, \text{LOSS}, 2) \Leftarrow \\
 & \quad \text{in-check}(S, \text{LOSS}) \\
 & \quad \wedge \nu^{G_1}(S) \\
 & \quad \wedge \forall Op, po(S, \text{LOSS}, Op) \\
 & \quad \quad [\neg \text{make-false}(\lambda s.\text{in-check}(s, \text{LOSS}), Op, S) \\
 & \quad \quad \vee \neg \text{make-false}(\lambda s.\nu^{G_1}(s), Op, S)]
 \end{aligned}$$

Where the pattern $\nu^{G_1}(S) \Rightarrow \text{achieve}(\lambda s.G(s), S, \text{WIN}, 1)$, and the goal $G(s)$ defines the condition when a queen is captured. The proof states that both $\text{in-check}(S, \text{LOSS})$ and $\nu^{G_1}(S)$ are true in the current situation, and for all moves that are available, they either make one threat or the other false. In other words, there is no move available that can make both threats false.

2.3.2 Skewer

The Skewer is a useful trick for capturing valuable material. In a Skewer, the king is attacked and forced to move out of the way, thereby opening an attack on another piece that is behind the king. We illustrate an example in Figure 2.1(b). Here the

white bishop on **g3** is attacking the black king on **d5**. The king is forced to move out of check, exposing the black rook on **a8** to capture by the bishop.

This tactic is defined by the following proof of depth 2:

$$\begin{aligned}
 \text{achieve}(\lambda s.G(s), S, \text{LOSS}, 2) &\Leftarrow \\
 &\text{in-check}(S, \text{LOSS}) \\
 &\wedge \forall Op, po(S, \text{LOSS}, Op) \\
 &\quad [\text{maintain-true}(\lambda s.\text{in-check}(s, \text{LOSS}), Op, S) \\
 &\quad \vee \text{make-true}(\lambda s.\nu^{G_1}(s), Op, S)]
 \end{aligned}$$

Where the goal pattern $\nu^{G_1}(S) \Rightarrow \text{achieve}(\lambda s.G(s), S, \text{WIN}, 1)$, and $G(s)$ defines the condition when the rook is captured. This proof states that $\text{in-check}(S, \text{LOSS})$ is currently true and all available operators either maintain the in-check threat on the king or make some new threat true.

2.3.3 Half-Pin

The Half-pin is similar to the Skewer, in that involves a threat along a line. However, in the half-pin, the king is not attacked directly. Rather, a valuable piece is attacked and it cannot move out of the way, because such a move exposes the king to attack. We illustrate an example in Figure 2.1(c). Here the white rook on **e3** is attacking the black queen on **e6**. The queen cannot move out of danger because such a move will expose the king to capture. Hence, the best black can do is to capture the white rook and be recaptured by the pawn on **f2**.

This tactic is defined by the following proof of depth 2:

$$\begin{aligned}
 \text{achieve}(\lambda s.G(s), S, \text{LOSS}, 2) &\Leftarrow \\
 &\wedge \nu_1^{G_1}(S) \\
 &\wedge \forall Op, po(S, \text{LOSS}, Op) \\
 &\quad [\text{make-true}(\lambda s.\text{in-check}(s, \text{LOSS}), Op, S) \\
 &\quad \vee \text{maintain-true}(\lambda s.\nu_1^{G_1}(s), Op, S) \\
 &\quad \vee \text{make-true}(\lambda s.\nu_2^{G_1}(s), Op, S)]
 \end{aligned}$$

Where $\nu_1^{G_1}(S) \Rightarrow \text{achieve}(\lambda s.G(s), S, \text{WIN}, 1)$, (rook captures black queen), $\nu_2^{G_1}(S) \Rightarrow \text{achieve}(\lambda s.G(s), S, \text{WIN}, 1)$, (pawn captures black queen) and $G(s) \Rightarrow$

Capture-queen(s). This proof defines a more complicated case of pin, where the pinned piece has the option to exchange. The proof states that the opponent is threatening to win immediately with $\nu_1^{G_1}(S)$, and for all the moves that are available they either maintain the threat, put the king in check or make true a new threat (the exchange).

2.3.4 Overworked Piece

An overworked piece is piece that is performing two distinct functions, and when called upon the implement one function, it is unable to carry out the other. This idea is illustrated in Figure 2.1(e). Here the black queen on **d7** is the overworked piece. The queen is protecting the rook on **c8** from capture and preventing the white queen from achieving *check-mate* by capturing the pawn on **g7**. White can exploit this weakness by capturing the black rook on **c8** with the rook on **c5**, thereby forcing the black queen to capture on **c8**. This move of the queen destroys its other function, allowing white to achieve *check-mate*.

This tactic involves a proof of depth 5, and so is more complex than the last proofs. First we define the winning proof of depth 5:

$$\begin{aligned} \text{achieve}(\lambda s.G(s), S, \text{WIN}, 5) \Leftarrow \\ \neg \nu^{G_4}(S) \wedge \exists Op, o(S, \text{WIN}, Op), \text{make-true}(\lambda s.\nu^{G_4}(s), Op, S) \end{aligned}$$

Where $\nu^{G_4}(S) \Rightarrow \text{achieve}(\lambda s.G(s), S, \text{LOSS}, 4)$ defined below:

$$\begin{aligned} \text{achieve}(\lambda s.G(s), S, \text{LOSS}, 4) \Leftarrow \\ \text{in-check}(S, \text{LOSS}) \\ \wedge \forall Op, po(S, \text{LOSS}, Op) \\ [\quad \text{maintain-true}(\lambda s.\text{in-check}(s, \text{LOSS}), Op, S) \\ \quad \vee \text{make-true}(\lambda s.\nu^{G_3}(s), Op, S)] \end{aligned}$$

This proof states that the king is currently in check and all moves either maintain the check threat or make a new threat true. This new threat is defined as $\nu^{G_3}(S)$ where $\nu^{G_3}(S) \Rightarrow \text{achieve}(\lambda s.G(s), S, \text{WIN}, 3)$ defined below:

$$\begin{aligned} \text{achieve}(\lambda s.G(s), S, \text{WIN}, 3) \Leftarrow \\ \neg \nu^{G_2}(S) \wedge \exists Op, o(S, \text{WIN}, Op), \text{make-true}(\lambda s.\nu^{G_2}(s), Op, S) \end{aligned}$$

This proof for *WIN* simply states that there is a move available that makes true a loss defined as $\nu^{G_2}(S) \Rightarrow \text{achieve}(\lambda s.G(s), S, \text{LOSS}, 2)$. This goal pattern defines a *check-mate* situation defined below:

$$\begin{aligned} \text{achieve}(\lambda s.G(s), S, \text{LOSS}, 2) \Leftarrow & \\ & \text{in-check}(S, \text{LOSS}) \\ & \wedge \forall Op, po(S, \text{LOSS}, Op) \\ & \quad \text{maintain-true}(\lambda s.\text{in-check}(s, \text{LOSS}), Op, S) \end{aligned}$$

2.4 Optimal Goal Achievement

The previously developed language of influence proofs for goal achievement focused on proving *correct* goal achievement. Since in this thesis we are also interested in *optimal* performance, this section extends the proofs to ensure optimal goal achievement. There are two considerations for optimality for counter planning in general: First, we wish our problem solver to always achieve the *best* goal available. Second, we wish our problem solver, when playing the winning side, to achieve the goal with the minimum number of operator applications, and when playing the losing side, to achieve the goal with the maximum number of operators.

To adapt the previous analysis to take optimality into account is straightforward. First, we must eliminate the simplification made in the previous proofs that the expanded proof or search tree is always balanced, that is, the goal patterns used to construct a proof of depth i are all of depth $i - 1$. In fact, they can be of any lower depth $j, j < i$ so long as they are winning patterns (i.e., even j) when we are constructing a proof for *LOSS* to play and losing patterns (i.e., odd j) when we are constructing a proof for *WIN* to play. This constraint is necessary to ensure correct counter-planning. Second, we must extend the notation for goal achievement to include the operator that leads to optimal goal achievement as an additional argument. Recall that $\text{achieve}(\lambda s.G(s), S, \text{Side}, \text{Depth})$ denotes that fact that the goal $G(s)$ is achieved in S for *Side* to play in depth *Depth*. We extend this notation with the argument *Op*, $\text{achieve}(\lambda s.G(s), S, \text{Side}, \text{Depth}, Op)$ where *Op* is the

operator leading to optimal goal achievement.

2.4.1 When *LOSS* is to Play

When *LOSS* is to play, the optimal operator is that which maximally extends the solution length. In other words, the optimal operator delays the loss as much as possible. Since the influence axioms for loss, given in Section 2.2.2, are written in terms of a set of goal patterns which are wins, the optimal operator is the one that leads to the goal pattern with the highest depth. Using the definitions given previously, we have three kinds of loss axioms:

No threat This axiom defines goal achievement when none of the goal patterns is true in the initial situation. Here all operators *make-true* some subset of the n goal patterns. The optimal operator is the one that *make-true* the goal pattern out of the set which has the highest depth.

One threat This axiom defines goal achievement when only one goal pattern is true in the initial situation. Here all operators either *maintain-true* the one threat pattern or *make-true* some disjunction of other goal patterns. The optimal operator is the one that either maintains the threat if the threat pattern has the highest depth or *make-true* the goal pattern from the set with the highest depth.

Many threat This axiom defines goal achievement when more than one goal pattern is true in the initial situation. Here none the operators makes all the threat patterns false. The optimal operator is the one that *make-false* the goal pattern with the lowest depth, thereby leaving the threat goal pattern with a higher depth true.

2.4.2 When WIN is to Play

When WIN is to play, the optimal operator is that which minimizes the solution length. Since we have not previously given influence proofs for the winning side, let us develop the optimal axioms from the correct axioms. Assume that we have n losing goal patterns that are used to define new winning axioms: $\nu_1^{G_{d_1}}(s) \vee \nu_2^{G_{d_2}}(s) \vee \dots \vee \nu_n^{G_{d_n}}(s)$ (note the superscripts d_j refer to the solution depth of losing pattern j). Clearly, correct goal achievement can be defined:

$$(9) \text{ achieve}(\lambda s.G(s), S, WIN, i) \Leftarrow$$

$$\begin{aligned} & \exists Op, o(S, WIN, Op), \nu_1^{G_{d_1}}(do(Op, S)), i = d_1 + 1 \\ & \vee \exists Op, o(S, WIN, Op), \nu_2^{G_{d_2}}(do(Op, S)), i = d_2 + 1 \\ & \vee \dots \\ & \vee \exists Op, o(S, WIN, Op), \nu_n^{G_{d_n}}(do(Op, S)), i = d_n + 1 \end{aligned}$$

These axioms simply state that the advantageous goal G can be achieved for WIN if there exists a legal operator of WIN that results in one of the known loss goal patterns being true. To modify this to take into account optimality we add an additional constraint to each axiom. This constraint states that there must not exist any other legal operator that results in a goal pattern of a lower depth being true. For simplicity, let us consider only one of the n possible axioms, denoted j :

$$(10) \text{ achieve}(\lambda s.G(s), S, WIN, i, Op) \Leftarrow$$

$$\begin{aligned} & \exists Op, o(S, WIN, Op), \nu_j^{G_{d_j}}(do(Op, S)), i = d_j + 1 \\ & \wedge \neg \exists Op_1, o(S, WIN, Op_1), \nu_j^{G_{d_1}}(do(Op_1, S)), j \neq 1, d_1 < d_j \\ & \wedge \neg \exists Op_2, o(S, WIN, Op_2), \nu_j^{G_{d_2}}(do(Op_2, S)), j \neq 2, d_2 < d_j \\ & \wedge \dots \\ & \wedge \neg \exists Op_n, o(S, WIN, Op_n), \nu_n^{G_{d_n}}(do(Op_n, S)), j \neq n, d_n < d_j \end{aligned}$$

This proof can be simplified if we assume that we have already compiled *winning* goal patterns for lower solution depths. Let this set be represented as $\hat{\nu}_1^{G_{d_1}}(s) \vee \hat{\nu}_2^{G_{d_2}}(s) \vee \dots \vee \hat{\nu}_m^{G_{d_m}}(s)$ (note the use of a accent $\hat{\nu}$ to denote *winning* goal patterns). Then the proof can be simplified to:

$$\begin{aligned}
(11) \text{achieve}(\lambda s.G(s), S, WIN, i, Op) \Leftarrow & \\
& \exists Op, o(S, WIN, Op), \nu_j^{G_{d_j}}(do(Op, S)), i = d_j + 1 \\
& \wedge \neg \hat{\nu}_1^{G_{d_1}}(S), d_1 < d_j \\
& \wedge \neg \hat{\nu}_2^{G_{d_2}}(S), d_2 < d_j \\
& \wedge \dots \\
& \wedge \neg \hat{\nu}_m^{G_{d_m}}(S), d_m < d_j
\end{aligned}$$

This states that the winning goal is optimally achieved if there is no way to achieve the goal in fewer steps. In other words, if none of the previously found optimal winning patterns apply. One further simplification is to eliminate the explicit operator application from the proof and reexpress it in terms of influence relations:

$$\begin{aligned}
(12) \text{achieve}(\lambda s.G(s), S, WIN, i, Op) \Leftarrow & \\
& \exists Op, o(S, WIN, Op), \text{make-true}(\lambda s.\nu_j^{G_{d_j}}(s), Op, S), i = d_j + 1 \\
& \wedge \neg \hat{\nu}_1^{G_{d_1}}(S), d_1 < d_j \\
& \wedge \neg \hat{\nu}_2^{G_{d_2}}(S), d_2 < d_j \\
& \wedge \dots \\
& \wedge \neg \hat{\nu}_m^{G_{d_m}}(S), d_m < d_j
\end{aligned}$$

We give only the case where WIN 's move makes true the losing goal pattern $\nu_j^{G_{d_j}}(S)$. There is also a similar axiom where WIN maintains $\nu_j^{G_{d_j}}(S)$. However, this is rarely optimal because it implies that WIN must make a waiting move. These kind of positions are known as *zugzwang* positions.

Chapter 3

Compiling Influence Proofs

The previous chapter has introduced an abstraction mechanism employing a theory of influence. With this theory, a space of proofs of goal achievement is defined, where each proof describes a unique strategy for achieving the given goal. This chapter explores using this space of proofs to improve the performance of a problem solver. In this study let us consider improving the performance of a simple “reactive” problem solver. Reactive problem solvers determine the next action to take by matching the given problem situation to a set of provided pattern/action rules. The principle advantages of this “reaction planning” [Schoppers 87] approach is that problem solving is quick, since no problem solving search is required to decide which action to take.

We investigate improving the performance of a reactive problem solver by increasing its coverage over a problem domain. Initially, given a domain specification, the problem solver can solve only those problems that are immediately recognized as wins. This is possible by matching the given problem to the termination patterns provided by the user. However, no other problems can be solved since they require a forward search.

To increase the coverage of the problem solver, we create new pattern/action pairs that recognize when goals are achieved within some number of operator applications and recommend the best action to take. These new patterns are generated

by compiling influence proofs. A given proof P , which proves that some Op optimally and correctly achieves the goal, is compiled into a pattern/action pair that recommends Op in exactly those situations where P is true.

Our approach is to incrementally generate proofs of goal achievement and compile each proof into a pattern/action pair. We organize the proof generation process so as to simplify compilation. Two policies are followed:

- Only proofs involving one step look ahead are compiled. In other words, proofs of depth i are defined once proofs of depth $i - 1$ have been compiled into goal patterns.
- Simple proofs are generated before complex proofs. The space of influence proofs for depth i can be arranged in a partial order with respect to the number of non-negated goal patterns included. Generation is controlled by generating proofs ordered by the count of non-negated patterns.

The compiler takes each generated proof and compiles it into an equivalent pattern/action pair. Compiling a proof containing goal patterns involves first compiling the relevant influence relations over the goal patterns, then combining them together to ensure the constraints of the proof are met. For instance, to compile a fork proof (given in Section 2.3.1), we first compile the *make-false* influence relations over the two threat goal patterns, then combine them together to ensure that there is no operator that can make both threats false.

The rest of the chapter is divided into two sections. The first section describes the compilation of influence relations over patterns. Here we describe in detail the method for compiling influence relations for increasingly complex representations of goal patterns, from simple literals to conjunctions with exceptions. We illustrate the methods both in chess and a simple planning domain. The second section describes the methods for compiling the influence proofs into pattern/action pairs. This section illustrates the techniques with examples from chess.

3.1 Compiling Influence Relations

The inputs and outputs of influence relation compilation is illustrated in Table 3.1. We compile influence relations over goal patterns of depth $i - 1$ as a subtask of compiling new proofs of depth i . The compilation goal is to replace the influence relation with a set of pattern/operator pairs that are equivalent to the influence relation.

The complexity of compiling the influence relations depends upon the representation of the goal patterns. We identify three cases of increasing complexity, and describe each in succession. The simplest case is when the goal patterns are literals in the state description, such as the blocks world goal of *clear*. A more complicated case is when the goal patterns are conjunctions of “operational literals.” We show how this case arises in planning domains with derived effects. The final case—which we show to be sufficient for counter-planning—represents a pattern as a conjunction with exceptions. We illustrate this case by computing influence relations over patterns that arise in chess.

3.1.1 Goal Patterns as Literals

When the goal patterns are simple literals, it is easy to compute the influence relations. For example, in the blocks world, given the goal pattern *on-table(Block)*, we can compute the *make-true* operations by selecting those operators that include this literal in their add lists, but not in their preconditions. The result is the *put-down* operator, which places a block on the table. Similarly, when we are considering the goal pattern *clear(Block)* and we are interested in *make-false*, we choose those operators that include the literal in the delete list and also in the preconditions. Here the result is *stack*, which places a block on top of another block. In general, it is easy to compute the influence relations when domains are described as STRIPS operators and the goal patterns are literals in the add and delete list of the operators.

Although the computation of influence relations in this case is easy, this

Given

- – $\forall Op \in o(Op, s), \text{make-true}(\nu^G(s), Op, S)$, or
- $\forall Op \in o(Op, s), \text{make-false}(\nu^G(s), Op, S)$, or
- $\forall Op \in o(Op, s), \text{maintain-true}(\nu^G(s), Op, S)$.
- A goal pattern $\nu^G(s)$, where $\nu^G(s) \Rightarrow G(s)$ and $G(s)$ is some goal.
- A function $o(S, Op)$ that takes a situation S and generates all the operators Op that are applicable in S .
- A set of $n \times m$ frame axioms for n dynamic literals and m operators in the domain.

Find

- A set $L, \{\langle \nu_1(s), Op_1 \rangle, \langle \nu_2(s), Op_2 \rangle, \dots, \langle \nu_r(s), Op_r \rangle\}$.
For each $\langle \nu_i(s), Op_i \rangle \in L, 1 \leq i \leq r$, the following holds:
 - If we are computing a *make-true*, then for any situation S where $\nu_i(S)$ is true, Op is an applicable operator in S , $\nu^G(S)$ is false and $\nu^G(\text{do}(Op_i, S))$ is true.
 - If we are computing a *make-false*, then for any situation S where $\nu_i(S)$ is true, Op is an applicable operator in S , $\nu^G(S)$ is true and $\nu^G(\text{do}(Op_i, S))$ is false.
 - If we are computing a *maintain-true*, then for any situation S where $\nu_i(S)$ is true, Op is an applicable operator in S , $\nu^G(S)$ is true and $\nu^G(\text{do}(Op_i, S))$ is true.

Table 3.1: The inputs and outputs of algorithm that computes the influence relations

limitation on the representation of goal patterns as pre-defined literals strongly limits the applicability of the approach. The principal problem arises because of the recursive nature of the influence proofs. Recall that to compile an influence proof of depth n we must first compile the influence relations over the goal patterns that result from compiling proofs of depth $n - 1$. Hence, the representation of goal patterns must be expressive enough to describe the results of compiling the influence proofs. It is hard to see how limiting the representation of goal patterns to pre-defined literals could be effective in complex domains like chess, where we are interested in goal patterns like “can win in at least 11 moves.” The burden of pre-engineering the domain so that these goal patterns are defined as literals is prohibitive. Hence, it is *necessary for learning* that the compilation of influence relations be flexible enough so as to compile goal patterns that are derived by the proof compiler.

3.1.2 Goal Patterns as Conjunctions

In this section we are interested in developing algorithms that compile influence relations in a more complex case—when the goal patterns are defined as *conjunctions of literals*. More precisely, goal patterns are defined as follows:

$$\begin{aligned} \nu & ::= c \\ c & ::= l_1 \wedge l_2 \wedge \dots \wedge l_n \\ l_j & ::= opdl | \neg opl | opl \\ opdl & ::= \textit{dynamic literal} \\ opl & ::= \textit{other operational literal} \end{aligned}$$

We distinguish two cases of operational literals, *dynamic literals*, which are directly affected by the operators, and *other operational literals*, which are simple constraints that can be easily computed. Notice, that we limit the dynamic literals to be all positive in the conjunction. An operational literal is defined as one that can be directly evaluated by simple lookup (such as `object(Obj)`) or easily evaluated (such as `connected(LocF,Loc2,Dir)`). For more information on operational

```

stuffy(S,room):-
    blocked(S,duct1),
    blocked(S,duct2).

blocked(S,D):-
    on(S,Obj,D),
    object(Obj),
    duct(D).

o(S,Op):-
    (Op = move(Objm,From,To),
     on(S,Objm,From),
     From /= To,
     object(Objm),
     on(S,empty,To),
     in_hand(S,empty)
    | Op = swap(Obj1,Obj2,Loc),
     in_hand(S,Obj1),
     object(Obj1),
     on(S,Obj2,Loc),
     object(Obj2)
    ).

on(do(move(Objm,From,To),S),Obj,At):-
    (At = To -> %just moved to this loc
     Obj = Objm
    |At = From -> %just moved from this loc
     Obj = empty
    |otherwise -> %op has no affect
     on(S,Obj,At)
    ).

on(do(swap(Obj1,Obj2,Loc),S),Obj,At):-
    (At = Loc -> %just swapped at this loc
     Obj = Obj2
    |otherwise ->
     on(S,Obj,At)
    ).

in_hand(do(move(Objm,From,To),S),Obj):-
    Obj = empty.

in_hand(do(swap(Obj1,Obj2,Loc),S),Obj):-
    Obj = Obj1.

```

Figure 3.1: Prolog definition of the Room domain

literals, see the EBL literature such as [DeJong and Mooney 86], [Hirsh 87] and [Mitchell Keller and Kedar-Cabelli 86].

To introduce computing influence relations in this case let us use a simple household domain adapted from [Ginsberg and Smith 88]. We illustrate a domain theory for this domain in Figure 3.1. The domain defines a simple room in a house that contains various objects such as boxes, televisions, plants and air conditioning ducts, which provide the fresh air in the room. Notice that if both ducts become blocked by objects, the room will become stuffy.

This domain is used to illustrate compiling the influence relations over the simple conjunctive goal pattern which describes the condition when the room will become stuffy, that is, when both ducts are blocked. The resulting set of influence relations could be employed by a planning system to ensure that any plan produced is guaranteed never to make the room stuffy.

In this stuffy room world, there are two operators available: *move*, which *changes* the location of an object and *swap*, which changes an object at the same location. There are two dynamic literals: `on(Situation,Operator,Location)` and `in_hand(Situation,Held_object)`. There is one other operational literal, `object(Obj)` which is true when `Obj` is an object. The operators are defined by the relation `o(S,Op)`, and their effects are defined by the four frame axioms shown on the right side of Figure 3.1, one for each operator/dynamic-literal combination.

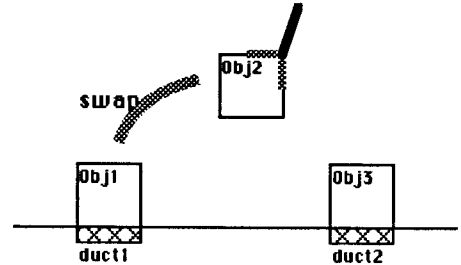
The goal we are interested in manipulating, `stuffy(S)`, is represented as a single conjunctive goal pattern, denoted $\nu^{stuffy}(s)$, which defines the condition where both ducts are blocked. This goal pattern is easily computed from the domain theory by partially evaluating `stuffy(S)`:

```

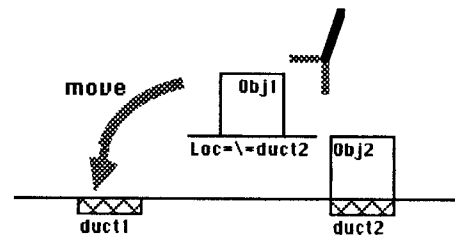
 $\nu^{stuffy}(S):-$ 
  on(S,Obj1,duct1),
  object(Obj1),
  on(S,Obj2,duct2),
  object(Obj2).

```

$\forall Op \in o(S, Op), \text{maintain-true}(\nu^{stuffy}(S), Op, S) \Leftrightarrow$
 $\langle [on(S, Obj2, duct1),$
 $in_hand(Obj1),$
 $object(Obj1),$
 $object(Obj2),$
 $on(S, Obj, duct2),$
 $object(Obj)],$
 $Op = \text{swap}(Obj1, Obj2, duct1) \rangle, \dots$



$\forall Op \in o(S, Op), \text{make-true}(\nu^{stuffy}(S), Op, S) \Leftrightarrow$
 $\langle [on(S, Objm, From),$
 $From \neq duct2,$
 $in_hand(empty),$
 $object(Objm),$
 $on(S, Obj, duct2),$
 $object(Obj)],$
 $Op = \text{move}(Objm, From, duct1) \rangle, \dots$



$\forall Op \in o(S, Op), \text{make-false}(\nu^{stuffy}(S), Op, S) \Leftrightarrow$
 $\langle [on(S, Objm, duct1),$
 $in_hand(empty),$
 $object(Objm),$
 $on(S, Obj, duct2),$
 $object(Obj),$
 $To \neq duct1],$
 $Op = \text{move}(Objm, duct1, To) \rangle, \dots$

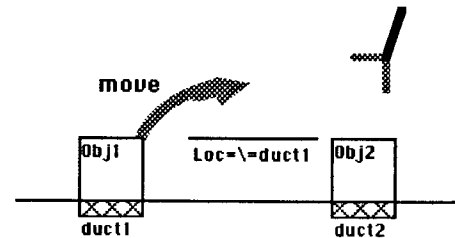


Figure 3.2: Results of compiling *make-true*, *make-false* and *maintain-true*, for the stuffy-room goal pattern

Given this goal pattern, compilation begins by compiling influence proofs of length 1. The results of this compilation process are illustrated in Figure 3.2. In each case the influence relations are compiled into a list of pattern/operator pairs $\langle \nu, Op \rangle$. At the top of Figure 3.2 we show the result of compiling *maintain-true* $(\nu^{stuffy}(s), Op, S)$. Here the compiler determined¹ that there are two different ways to maintain the stuffy condition, (1) swapping any object and (2) moving any object which is not on one of the ducts. We have shown the first of these two solutions as a logical expression and as a diagram. At the middle of Figure 3.2 we show the result of compiling *make-true* $(\nu^{stuffy}(s), Op, S)$. Two pattern/action rules were derived. The first, which is illustrated, is where `duct1` is empty and another object, which is not on `duct2`, is moved to `duct1` by the `move` operator. The other case has `duct1` and `duct2` reversed.

The result of compiling *make-false* $(\nu^{stuffy}(s), Op, S)$ is illustrated at the bottom of Figure 3.2. Again, two pattern/action rules are determined. The first, which is illustrated, is where the object on `duct1` is picked up and moved to another location (which is not `duct1`).

To compile influence relations we employ *partial evaluation*—a powerful technique for improving the efficiency of a computation [Kahn and Carlsson 84]. A partial evaluator is a program interpreter that, with only partial information about the inputs of a program, produces a specialized version that incorporates the partial information. More formally, given some function $f(\bar{x}, y)$ and the information that $y = a$, partial evaluation produces a specialized function $f_a(\bar{x})$. For example, in [Kahn 84] partial evaluation is used on the relation `append(Front, Rest, Total)`, to produce an efficient specialized version that can be applied when `Front` is known to be some list `[A|B]`. In this case partial evaluation produces a program, `[[A|B]]Rest`, which involves only unification.

Partial evaluation is applied to compile the influence relations by special-

¹In 15 seconds running compiled Quintus Prolog on a Sun Sparc 1.

izing their definition (see Table 2.1) with respect to a given goal pattern. More formally, partial evaluation takes a given pattern $\nu_1^{loss}(s)$, a domain theory DT and an influence relation such as $make\text{-}true(\lambda s.\nu^G(s), Op, S)$ and produces a specialized definition of $make\text{-}true$, represented as a list of pattern/action pairs, where $\nu^G(s) = \nu_1^{loss}(s)$.

In the remainder of this section, we present a detailed description of the compilation process. First we describe compiling $maintain\text{-}true$, since it is the most straightforward. Here we detail the partial evaluator, which is written in Prolog [Sterling and Shapiro 86]. Next we describe compiling $make\text{-}true$ and $make\text{-}false$. With each description, we include an algorithm that calls the partial evaluator and an example compilation in the stuffy room domain.

Compiling $maintain\text{-}true$

The goal of this stage is to produce a set of pattern/action pairs where the given pattern is true both before and after the action is applied. To produce this set we partially evaluate the definition of $maintain\text{-}true$ with the pattern instantiated to the given pattern. In the stuffy room domain this becomes the Prolog conjunction: $\nu^{stuffy}(S), o(S, Op), \nu^{stuffy}(do(Op, S))$. The partial evaluator is implemented as a meta-interpreter in Prolog and operates on conjunctions of Prolog literals. Disjunctions in the expression are handled by employing the standard backtracking depth first search of Prolog. More precisely, let us define the partial evaluator as a function PE that takes a conjunction of literals L_1, L_2, \dots, L_n and returns either fail or a simplified conjunction. PE comprises two principal actions: *unfolding*, where literals in the conjunction are expanded into their definitions, and *simplification*, where inconsistencies and redundancies are detected. PE computes the fixed point of an expression by repeatedly calling *unfold* and *simplify* until the expression is unaffected. We describe each process in detail:

- *Unfold*: Here we choose a “non-operational” literal L_i that unifies with the head H of some clause $H:- B$, with substitution θ . We replace L_i with B , instantiated with θ . B can be of two forms:
 - B is a conjunction. We call *simplify* over the new conjunction.
 - B is of the form $(T_1 \rightarrow B_1; T_2 \rightarrow B_2; \dots; \text{otherwise} \rightarrow B_n)$. This is the syntax of an if-then-else construct in Prolog. Here we must work through each disjunctive case in order, producing a simplified result for each case. For each clause $T_j \rightarrow B_j$ we first try to determine the truth value of T_j . If T_j is false, we simply try the next case $j+1$. If T_j is true or undetermined then we add $\text{not}(T_k)$, $1 \leq k \leq j-1$, T_j and B_j to the conjunction and call *simplify* on the result.
- *Simplify*: Here we try to determine if the conjunction is inconsistent or contains redundancy. We perform many simple operations to detect inconsistencies including the following:
 - *Apply functional dependencies*: Given information about functional dependencies which the literals in the domain obey, we ensure that the conjunction is consistent with those dependencies. For example, if we know that an object can be at only one location in a given situation, then if the conjunction includes both $\text{on}(S, \text{Obj1}, \text{Loc1})$ and $\text{on}(S, \text{Obj1}, \text{Loc2})$ we can conclude that $\text{Loc1} = \text{Loc2}$. This equality substitution can now be made throughout the conjunction, simplifying the result and possibly leading to other applicable simplifications.
 - *Look up ground literals*: If a static literal is found that has no variables, then its truth value can be directly determined by looking up the literal in the database. If the literal is true, then we can eliminate it. If the literal is false, the whole conjunction is false, due to Prolog’s *negation as failure* semantics. For example, while applying *PE* in the stuffy room domain,

the literal `object(empty)` is encountered. Here, a lookup determines that this is false, causing *PE* to return fail.

- *Look up partially instantiated literals:* When a static literal is partially instantiated, it can often be determined, through a lookup in the database, if the literal is false or has only one complete instantiation.
- *Compute functional attachments:* Literals with functional attachments allow the calculation of some of their arguments from other arguments. For example, the literal `plus(X,Y,Z)` has three functional attachments, each allowing the calculation of one argument from the other two. Whenever this literal is found with two arguments bound, its third argument can be computed and propagated, and the literal removed from the conjunction.
- *Detect inconsistencies:* We return fail if `L and not(L)` or `Var =/= Var` are found in the conjunction.

We illustrate the evaluation tree generated when partial evaluating *maintain-true*($\nu^{stuffy}(S), Op, S$) in Figure 3.3. Each unfolding of the rules in the domain theory is marked as a node in the tree. The partial evaluator explores each disjunctive case in the standard left-to-right top-down evaluation order of Prolog. The leaves of the tree are numbered in the order that they are explored. Each leaf represents a candidate solution. Those leaves with a cross, denote failure where no solution was found. Those leaves with a tick mark denote a successful solution. Below we itemize the computation at each leaf:

- (1) fail. We move To `duct1`, which implies that `duct1` is empty in *S* (from the constraints of `o(S,Op)`), since `duct1` is occupied by `Obj1` (from unfolding $\nu^{stuffy}(S)$), this case fails. The contradiction is detected by noticing that the conjunction contains both `on(S,empty,duct1)` and `on(S,Obj1,duct1)`, which implies that `Obj1 = empty`. Since the conjunction also includes the constraint

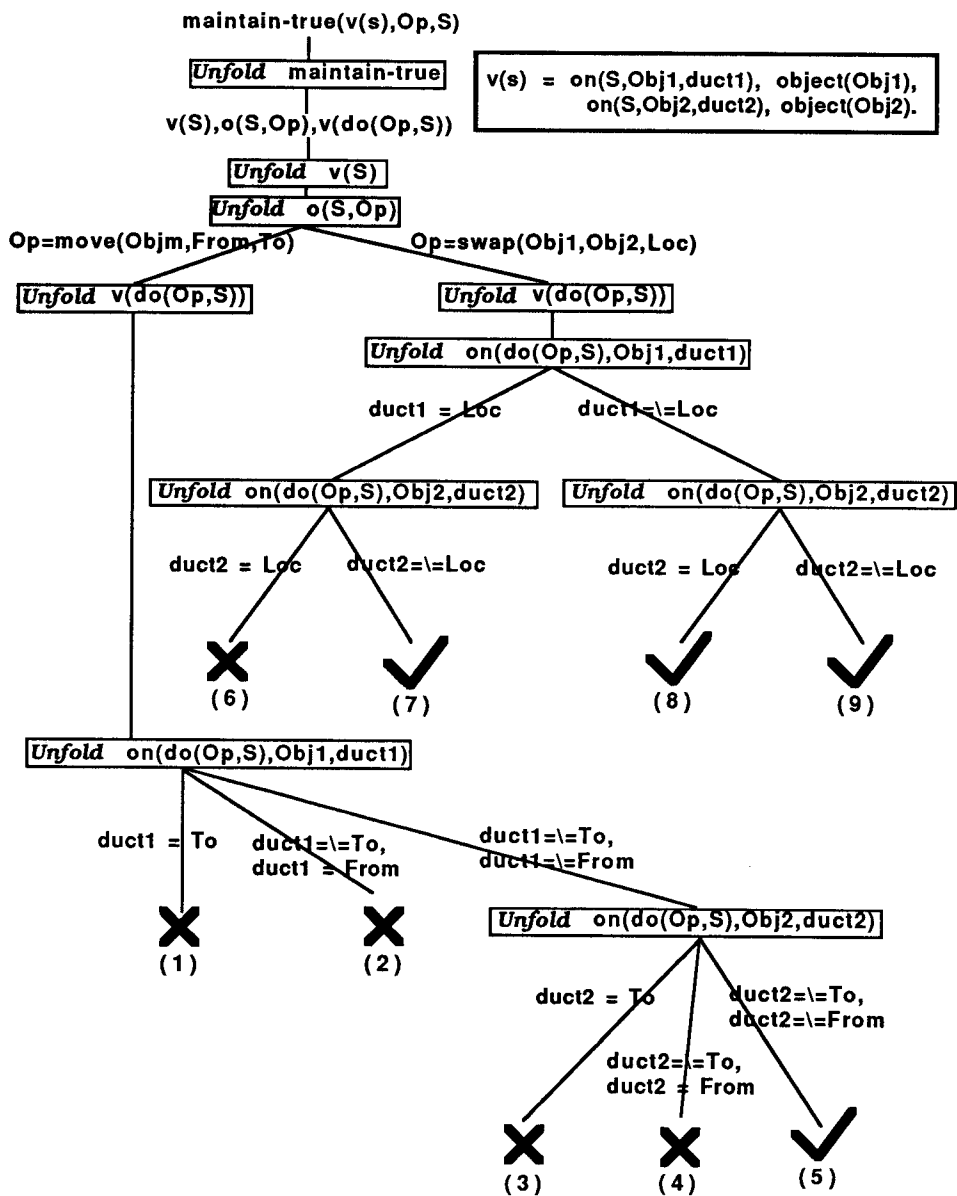


Figure 3.3: The evaluation tree generated when partially evaluating $\text{maintain-true}(v^{\text{stuffy}}(S), \text{Op}, S)$.

object(Obj1), this case fails. The constraint that the same object must occupy the same location and the related constraint that the same location must be occupied by the same object are imposed by functional dependencies:

$$\begin{array}{ll} \text{on}(S, \text{Obj1}, \text{Loc1}) & \text{on}(S, \text{Obj1}, \text{Loc1}) \\ \wedge \text{on}(S, \text{Obj2}, \text{Loc1}) & \wedge \text{on}(S, \text{Obj1}, \text{Loc2}) \\ \Rightarrow \text{Obj1} = \text{Obj2} & \Rightarrow \text{Loc1} = \text{Loc2} \end{array}$$

- (2) fail. We move an object From duct1, making it empty. The contradiction is detected by the constraint on the final situation that both ducts must be occupied by objects. In this case Obj1, the occupier of duct1, is bound to empty, which is not an object.
- (3) fail. We move an object To duct2, implying that duct2 is empty. This case fails like (1).
- (4) fail. We move an object From duct2. This case fails like (2).
- (5) succeed. We move an object that is not from either duct. Hence the goal pattern true after the operator application is also true in the initial situation S.
- (6) fail. We swap the object both at duct1 and duct2, implying that the swapped object is at two different locations. This case fails like (1).
- (7) succeed. We swap at duct1, while leaving duct2 unaffected. This case succeeds.
- (8) succeed. We swap at duct2, while leaving duct1 unaffected. This case succeeds.
- (9) succeed. We swap at a location that is not duct1 or duct2. This case succeeds.

Compiling *make-true*

We compile *make-true* by employing partial evaluation in a manner similar to compiling *maintain-true*. Unfolding $make\text{-}true(c(s), Op, S)$ yields $not(c(S)), op(S, Op), c(do(Op, S))$ which includes the constraint that the given pattern must *not* be true in the initial situation S . This negated constraint makes the compilation more complex, because the partial evaluator described previously applies only to conjunctions of literals and not to negated conjunctions. To overcome these limitations we first re-order the evaluation so as to delay the evaluation of the negated constraint, leading to $op(S, Op), c(do(Op, S)), not(c(S))$. Partially evaluating $op(S, Op), c(do(Op, S))$ produces a set of candidate solutions that describe operators where $c(s)$ is true in $do(Op, S)$, following the operator application. Since we are compiling *make-true*, we must ensure that $c(s)$ is false in S , before the operator application.

One way to ensure that $c(S)$ is false is to just conjoin $not(c(S))$ onto each candidate solution. However, this is not viable, because it can lead to a proliferation of empty and redundant solutions:

- *Empty* solutions will be produced when $c(S)$ is always true in the candidate solution. This can occur, for example, when the operator in no way effects the expression $c(S)$. In this case we want to eliminate the candidate solution. Detecting and eliminating these empty candidates helps minimize the number of compiled patterns produced.
- *Redundant* solutions will be produced if $c(S)$ is already false in the candidate solution. In this case the simplest solution is just the original candidate solution. This analysis helps keep the compiled patterns succinct.

To detect these empty and redundant solutions we must *incorporate* the negated constraint into the candidate solution. This incorporation is performed by *PE* on an expression such as $c_0 \wedge \neg c_1$ by noticing that it can be equivalently rewritten as $c_0 \wedge \neg(c_0 \wedge c_1)$. This equivalence is easily shown by applying DeMorgan's

rule to $\neg(c_0 \wedge c_1)$ in the second expression, then distributing c_0 over the resulting disjunction. This yields $c_0 \wedge \neg c_0 \vee c_0 \wedge \neg c_1$, which is $c_0 \wedge \neg c_1$, our original expression. The advantage of the latter form is that PE can be applied to $(c_0 \wedge c_1)$.

There are three cases to consider:

1. $PE(c_0 \wedge c_1) \mapsto \text{fail}$. In this case, the $\neg c_1$ constraint is redundant, since c_1 is already false in c_0 . We return c_0 as the solution.
2. $PE(c_0 \wedge c_1) \mapsto c_0$. In this case, c_1 subsumes c_0 and the whole expression simplifies to $c_0 \wedge \neg c_0$, which is fail .
3. $PE(c_0 \wedge c_1) \mapsto c$, where c is simpler than c_1 . In this case we return $c_0 \wedge \neg c$ as the solution.

To apply this approach to compiling *make-true*, we first apply PE to $\text{op}(S, \text{Op})$, $c(\text{do}(\text{Op}, S))$ to generate candidate solutions, denoted $c_j^p(S)$. For each j we apply PE to $c_j^p(S), c(S)$ with the result as $c_j^r(S)$. We have three cases to consider as above:

1. $c_j^r(S)$ is fail , we know that the pattern is already false in the initial situation and we have found a solution $c_j^p(S)$.
2. $c_j^r(S)$ is $c_j^p(S)$, the pattern is always true in the initial situation and the solution is fail .
3. Otherwise, the pattern is true initially under some conditions and we must ensure those conditions are never true. The solution is $c_j^p(S)$, not $(c_j^r(S))$.

We illustrate this process of compiling $\text{make-true}(\nu^{\text{stuffy}}(S), \text{Op}, S)$ in Figure 3.4. Unfolding of rules from the domain theory are denoted as nodes in the tree, as in Figure 3.3. The difference in this figure is that the boxed evaluation trees denote the incorporation of the negated constraints as discussed above. Below we itemize the computation at each leaf:

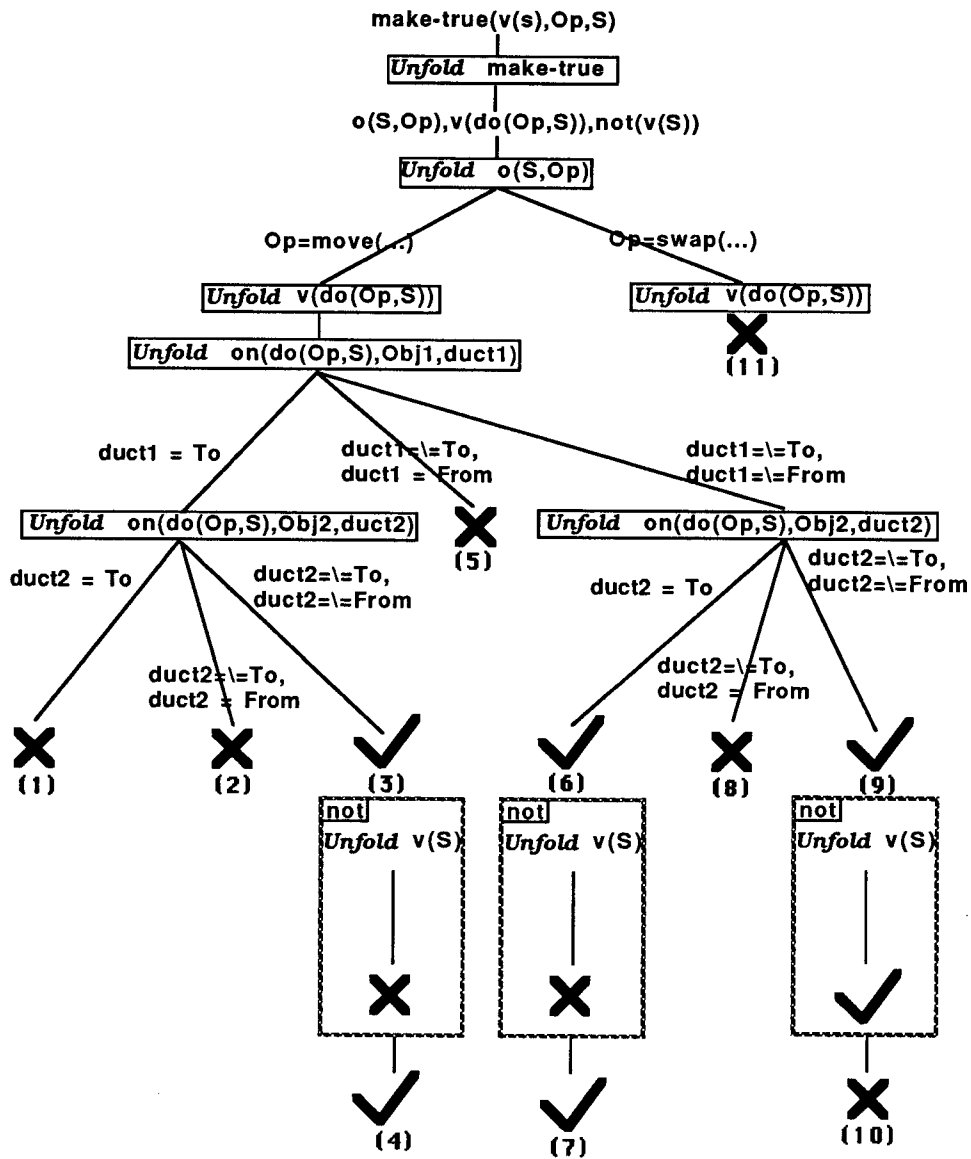


Figure 3.4: The evaluation tree generated when partially evaluating $\text{make-true}(v^{\text{stuffy}}(S), \text{Op}, S)$.

- (1) fail. We move To both duct1 and duct2, implying that the same object is at two different locations. This contradiction is detected by imposing the functional dependencies.
- (2) fail. We move To duct1, however we are moving From duct2, making it empty. The contradiction is detected by the constraint that both ducts must be occupied by objects after the operator has been applied. In this case Obj2, the occupier of duct2, is bound to empty, which is not an object.
- (3) succeed. We move an object To duct1 while leaving the blocked duct2 unaffected. We have our first candidate solution.
- (4) succeed. We must incorporate the constraint that the pattern be false before the operator is applied. We partially evaluate $\nu^{stuffy}(S)$ with the candidate solution from (3). The result is fail. because the candidate solution contains the constraint that duct1 is empty, while $\nu^{stuffy}(S)$ contains the constraint that it is occupied by an object. Since *empty* is not an object, this case fails. Hence, we have our first solution.
- (5) fail. We move an object From duct1. This case fails like case (2).
- (6) succeed. We move an object To duct2 while leaving the blocked duct1 unaffected. This case succeeds just like case (3). We have our second candidate solution.
- (7) succeed. This case succeeds like (4). We have our second solution.
- (8) fail. We move an object From duct2. This case fails like (2).
- (9) succeed. We move an object that is not from either duct. We have our third candidate solution.
- (10)fail. We partially evaluate $\nu^{stuffy}(S)$ with the candidate solution from (9). This results in a pattern that is exactly equivalent to $\nu^{stuffy}(S)$, since the

operator in (9) does not affect the pattern. Since the pattern is true initially, this case fails.

- (11) fail. This branch, where the operator is `swap`, was not expanded fully in the figure. All these cases fail, because in all situations where $\nu^{stuffy}(\text{do}(\text{Op}, S))$ is true (i.e., after a `swap` operator has been applied) the condition $\nu^{stuffy}(S)$ will also be true (before the operator is applied). Hence, all these cases fail like (10).

Computing $\text{make-false}(\lambda s.c(s), Op, S)$

Compiling make-false is similar to compiling make-true . Unfolding $\text{make-false}(c(s), \text{Op}, S)$ leads to $c(S), \text{op}(S, \text{Op}), \text{not}(c(\text{do}(\text{Op}, S)))$, which is similar to make-true in that a negated constraint is included. However, this time the negated constraint is following an operator application. We again delay the evaluation of the negated constraint, and first apply PE to $c(S), \text{op}(S, \text{Op})$ generating those operators that are applicable when the pattern is true initially. Let us denote these candidate solutions as $c_j^p(S)$. These candidate solutions do not take into account the constraint that the pattern must be false following the operator application. Hence we incorporate the negated constraints as before. For each j we apply PE to $c_j^p(S), c(\text{do}(\text{Op}, S))$ with the result as $c_j^r(S)$. We have three cases to consider as before:

1. $c_j^r(S)$ is fail, we know that the pattern is already false in the situation following the operator and we have found a solution $c_j^p(S)$.
2. $c_j^r(S)$ is $c_j^p(S)$, the pattern is always true following the operator application and the solution is fail.
3. Otherwise, the pattern is true following the operator under some conditions and we must ensure those conditions are never true. The solution is $c_j^p(S), \text{not}(c_j^r(S))$.

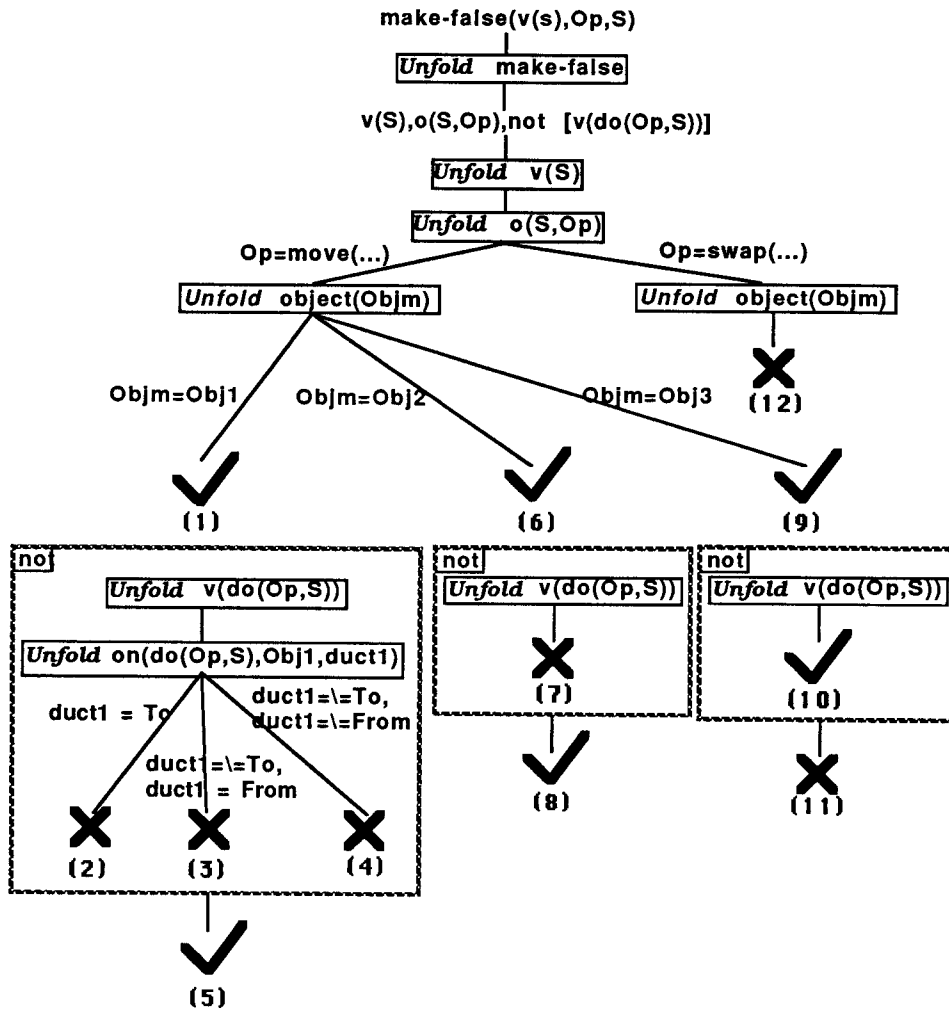


Figure 3.5: The evaluation tree generated when partially evaluating $\text{make-false}(v^{\text{stuffy}}(S), \text{Op}, S)$. Note the computation of $\text{not } c(\text{do}(\text{Op}, S))$

Figure 3.5 illustrates the evaluation tree generated when partially evaluating $make_false(\nu^{stuffy}(S), Op, S)$. Note the expanded trees when we are incorporating the negated constraints. This occurs because this process involves both unfolding and simplification. Below we summarize the computation at each leaf:

- (1) succeed. We move Obj1, the object currently occupying duct1. This case succeeds. We have our first candidate solution. We must incorporate the negated pattern after the operator application.
- (2) fail. We move To duct1, however we are moving From duct1. The contradiction is detected by the constraint that with the move operator, the location of the object must be changed.
- (3) fail. We move From duct1, making it empty. Hence, the constraint that both ducts must be occupied by objects fails.
- (4) fail. We are moving From duct1, hence this case implies that $duct1 \neq duct1$ which is false.
- (5) succeed. We have incorporated the constraint that the pattern must be false following operator application by apply PE to the candidate solution (1) and $\nu^{stuffy}(do(Op, S))$. Since all these cases failed, the pattern is already false after the operator has been applied. We have our first solution.
- (6) succeed. We move Obj2, the object currently occupying duct2. This case succeeds. We have our second candidate solution. We must incorporate the negated pattern after the operator application.
- (7) fail. Although not shown in the figure, this case fails very much like the combination of (2), (3) and (4) above, only this time, the move is from duct2.
- (8) succeed. This case succeeds just like (5).

- (9) succeed. We move Obj3, which is not currently occupying either duct1 or duct2. This case succeeds. We have our third candidate solution. We must incorporate the negated pattern after the operator application.
- (10) succeed. We have not expanded the evaluation tree fully in the figure. This case succeeds, since the pattern will be true after the operator application because the pattern is true initially and the object moved does not affect the pattern.
- (11) fail. Since (10) succeeded, this case fails.
- (12) fail. Here we explore the case when the operator is *swap*. All these cases fail because in all situations where $\nu^{stuffy}(S)$ is true (i.e., before a *swap* operator has been applied) the condition $\nu^{stuffy}(do(Op,S))$ will also be true (after the operator is applied). Hence, all these cases fail like (11).

3.1.3 Goal Patterns as Conjunctions with Exceptions

We have seen how partial evaluation can be effectively used to compile influence relations when the patterns are described as conjunctions of operational literals. This section extends the representational power of patterns to conjunctions with exceptions. More precisely, a pattern, denoted ν , is defined to have the following form:

$$\begin{aligned}
 \nu & ::= c_0 \wedge \neg c_1 \wedge \dots \wedge \neg c_m \\
 c_i & ::= l_1 \wedge l_2 \wedge \dots \wedge l_n \\
 l_j & ::= opdl | \neg opl | opl \\
 opdl & ::= \textit{dynamic literal} \\
 opl & ::= \textit{other operational literal}
 \end{aligned}$$

This extension in representational power is needed, because conjunctions are insufficient to represent the *result* of compiling the influence proofs. The problem with limiting patterns to simple conjunctions can be seen when we consider compiling patterns that arise from proofs of depth greater than 1. Here we can encounter

<ol style="list-style-type: none"> 1. Compile $maintain-true(\lambda s.c_0(s), Op, S) \mapsto maintain-true-c_0$. 2. Compile $make-true(\lambda s.c_i(s), Op, S) \mapsto make-true-c_i$. 3. Compute $Set-Difference(maintain-true-c_0, make-true-c_1) \mapsto maintain-true-\nu$.

Table 3.2: Compiling $maintain-true(\lambda s.\nu(s), Op, S)$, when $\nu(s) \Leftrightarrow c_0 \wedge \neg c_1$

the situation where we need to compile an influence relation over a pattern which is itself another influence relation. For example, we may need to compile an expression such as $make-true(\lambda s_1.make-false(\lambda s_2.\nu(s), Op_2, s_1), Op_1, S)$. This expression will only be compilable if the inner influence relation compiles into a conjunction. However, as we have seen, the result of compiling influence relations can lead to expressions that include exceptions. Hence, it is insufficient to limit the pattern representation to conjunctions, because the influence relations are not *closed over conjunctions*. However, as we will see, *influence relations are closed over conjunctions with exceptions*.

In this section we define the algorithms for compiling influence relations when the patterns are conjunctions with exceptions. Each algorithm works by decomposing the given pattern into its component conjunctions which are then compiled by the previously defined algorithms.

Compiling $maintain-true(\lambda s.\nu(s), Op, S)$

The goal of this compilation step is just as before with conjunctive patterns: replace the expression with an equivalent set of pattern/action pairs that implements the influence relation for the given pattern. For simplicity, let us assume that the given pattern has only one exception, thus $\nu(s) \Leftrightarrow c_0 \wedge \neg c_1$. To see how to decompose the compilation of this pattern, let us review the definition of *maintain-true* given in Table 2.1(3):

$$\text{maintain-true}(\lambda s.\nu(s), Op, S) \Leftrightarrow \nu(S) \wedge o(S, Op) \wedge \nu(\text{do}(Op, S)).$$

Substituting the definition of $\nu(s)$ into the right hand side gives us

$$c_0(S) \wedge \neg c_1(S) \wedge o(S, Op) \wedge c_0(\text{do}(Op, S)) \wedge \neg c_1(\text{do}(Op, S)).$$

This can be simplified to

$$\text{maintain-true}(\lambda s.c_0(s), Op, S) \wedge \neg \text{make-true}(\lambda s.c_1(s), Op, S).$$

This analysis tells us that to *maintain-true* a conjunctive pattern with one exception, we must maintain true the conjunction c_0 and not make true the negated conjunction c_1 . Hence, the problem of compiling $\text{maintain-true}(\lambda s.\nu(s), Op, S)$ is decomposed into compiling $\text{maintain-true}(\lambda s.c_0(s), Op, S)$ and compiling $\text{make-true}(\lambda s.c_1(s), Op, S)$. Let the result of compiling $\text{maintain-true}(\lambda s.c_0(s), Op, S)$ be *maintain-true- c_0* and the result of compiling $\text{make-true}(\lambda s.c_1(s), Op, S)$ be *make-true- c_1* . The compiler must determine a new list of pattern/action pairs, called *maintain-true- ν* , that implements *maintain-true- c_0* but not *make-true- c_1* . To compute this let us introduce a new function called *Set-Difference*(l_0, l_1), which takes two lists of pattern/action pairs, and computes their set difference. With this function we can define the compilation algorithm in Table 3.2.

Since we have previously defined the compilation of conjunctions such as c_0 and c_1 , we focus on the computation of *Set-Difference*(l_0, l_1). We illustrate the definition of *Set-Difference* in Table 3.3. The condition (a) on l ensures that each pattern/action pair implements the same function as the pairs in l_0 . The condition (b) on l ensures that each pair does not compute the function of the pairs in l_1 .

To compute *Set-Difference*, we compute the cross product with respect to the \cap operator of the sets of pattern/action pairs illustrated below:

<p>Given:</p> <ul style="list-style-type: none"> • $l_0 = \{\langle \nu_0^1(S), Op_0^1 \rangle, \langle \nu_0^2(S), Op_0^2 \rangle, \dots, \langle \nu_0^m(S), Op_0^m \rangle\}$. • $l_1 = \{\langle \nu_1^1(S), Op_1^1 \rangle, \langle \nu_1^2(S), Op_1^2 \rangle, \dots, \langle \nu_1^m(S), Op_1^m \rangle\}$. <p>Find:</p> <ul style="list-style-type: none"> • $l = \{\langle \nu^1(S), Op^1 \rangle, \langle \nu^2(S), Op^2 \rangle, \dots, \langle \nu^k(S), Op^k \rangle\}$ Such that: $\forall \langle \nu^j(S), Op^j \rangle \in l$, (a) $\exists \langle \nu_0^i(S), Op_0^i \rangle \in l_0, Op^j = Op_0^i$ and $\nu^j(S) \cap \nu_0^i(S) \neq \emptyset$ (b) $\forall \langle \nu_1^i(S), Op_1^i \rangle \in l_1$, if $Op^j = Op_1^i$ then $\nu^j(S) \cap \nu_0^i(S) = \emptyset$, <p>where \cap computes the intersection of two patterns.</p>

Table 3.3: The *Set-Difference*(l_0, l_1) function

\cap	$\langle \nu_0^1(S), Op_0^1 \rangle$	$\langle \nu_0^2(S), Op_0^2 \rangle$...	$\langle \nu_0^m(S), Op_0^m \rangle$
$\langle \nu_1^1(S), Op_1^1 \rangle$	\emptyset	\emptyset		\emptyset
$\langle \nu_1^2(S), Op_1^2 \rangle$	\emptyset	\checkmark		\square
...				
$\langle \nu_1^m(S), Op_1^m \rangle$	\emptyset	\checkmark		

For each pair from l_0 and l_1 we compute the intersection (using a function *Pattern-Intersection* defined later) that first unifies the operators and then determines if the patterns intersect. There are three cases that result from this operation. They are illustrated by the three symbols in the table:

1. \emptyset The intersection is empty, either the operators do not unify or the patterns do not intersect.
2. \square The operators unify and the intersection of $\nu_0^i(S)$ with $\nu_1^j(S)$ is equal to $\nu_0^i(S)$. In other words, $\nu_0^j(S) \subset \nu_1^i(S)$.
3. \checkmark The operators unify, and the intersection is non-empty, but $\nu_0^j(S) \not\subset \nu_1^i(S)$.

For each column, i , we compute a new pattern/action pair as follows:

1. If the column has all \emptyset , then there are no conditions where $\langle \nu_0^i(S), Op_0^i \rangle$ makes true the exception c_1 . Hence $\langle \nu_0^i(S), Op_0^i \rangle$ forms part of the solution set.
2. If the column contains at least one \square , then this pattern/action pair is rejected. since it always makes c_1 true.
3. Otherwise the column contains some \surd . Here there are some conditions, given by the pattern intersection, where $\langle \nu_0^i(S), Op_0^i \rangle$ makes c_1 true. We must modify $\nu_0^i(S)$ so that these conditions are never true. This modification is performed by the function *Pattern-Difference* that is defined below.

In order to completely define this algorithm, all that remains is to define the two functions *Pattern-Intersection* and *Pattern-Difference*. Both are defined below in terms of the previously defined function *PE*.

Let us assume that we have two patterns, ν_1 and ν_2 :

$$\begin{aligned}\nu_1 &= c_0^1 \wedge \neg c_1^1 \wedge \neg c_2^1 \wedge \dots \wedge \neg c_q^1 \\ \nu_2 &= c_0^2 \wedge \neg c_1^2 \wedge \neg c_2^2 \wedge \dots \wedge \neg c_r^2\end{aligned}$$

To compute *Pattern-Intersection*, we must simplify the following:

$$\begin{aligned}\nu_1 \wedge \nu_2 &= c_0^1 \wedge c_0^2 \wedge (\neg c_1^1 \wedge \neg c_2^1 \wedge \dots \wedge \neg c_q^1) \\ &\quad \wedge (\neg c_1^2 \wedge \neg c_2^2 \wedge \dots \wedge \neg c_r^2)\end{aligned}$$

We have previously defined the function *PE* which simplifies conjunctions. We can employ this routine by noticing that the above expression can be rewritten:

$$\begin{aligned}\nu_1 \wedge \nu_2 &= (c_0^1 \wedge c_0^2) \wedge \neg(c_0^1 \wedge c_1^2) \wedge \neg(c_0^1 \wedge c_2^2) \wedge \dots \wedge \neg(c_0^1 \wedge c_q^2) \\ &\quad \wedge \neg(c_0^2 \wedge c_1^1) \wedge \neg(c_0^2 \wedge c_2^1) \wedge \dots \wedge \neg(c_0^2 \wedge c_r^1)\end{aligned}$$

Thus, computing *Pattern-Intersection*(ν_1, ν_2) decomposes into one call to *PE*(c_0^1, c_0^2), q calls to *PE*(c_0^1, c_i^2), $1 \leq i \leq q$, and r calls to *PE*(c_0^2, c_i^1), $1 \leq i \leq r$.

To compute *Pattern-Difference*, we must simplify the following:

1. Compile: $make\text{-}false(\lambda s.c_0(s), Op, S) \mapsto make\text{-}false\text{-}c_0$.
2. Compile: $make\text{-}true(\lambda s.c_1(s), Op, S) \mapsto make\text{-}true\text{-}c_1$.
3. Compute: $make\text{-}false\text{-}c_0 \downarrow make\text{-}true\text{-}c_1 \mapsto make\text{-}true\text{-}\nu$.

Table 3.4: Compiling $make\text{-}false(\lambda s.\nu(s), Op, S)$, when $\nu(s) \Leftrightarrow c_0 \wedge \neg c_1$

$$\nu_1 \wedge \neg \nu_2 = c_0^1 \wedge \neg c_1^1 \wedge \neg c_2^1 \wedge \dots \wedge \neg c_q^1 \wedge \neg [c_0^2 \wedge \neg c_1^2 \wedge \neg c_2^2 \wedge \dots \wedge \neg c_r^2]$$

Distributing the negation gives us

$$\begin{aligned} \nu_1 \wedge \neg \nu_2 = & c_0^1 \wedge \neg(c_0^1 \wedge c_0^2) \wedge \neg c_1^1 \wedge \neg c_2^1 \wedge \dots \wedge \neg c_q^1 \\ & \vee (c_0^1 \wedge c_1^2) \wedge \neg(c_1^2 \wedge c_1^1) \wedge \neg(c_1^2 \wedge c_2^1) \wedge \dots \wedge \neg(c_1^2 \wedge c_q^1) \\ & \vee (c_0^1 \wedge c_2^2) \wedge \neg(c_2^2 \wedge c_1^1) \wedge \neg(c_2^2 \wedge c_2^1) \wedge \dots \wedge \neg(c_2^2 \wedge c_q^1) \\ & \vee \dots \\ & \vee (c_0^1 \wedge c_r^2) \wedge \neg(c_r^2 \wedge c_1^1) \wedge \neg(c_r^2 \wedge c_2^1) \wedge \dots \wedge \neg(c_r^2 \wedge c_q^1). \end{aligned}$$

Thus, computing $Pattern\text{-}Difference(\nu_1, \nu_2)$ decomposes into one call to $PE(c_0^1 \wedge c_0^2)$ and $(q+1) \times r$ calls to $PE(c_i^1 \wedge c_j^2)$, $0 \leq i \leq q, 1 \leq j \leq r$

We have defined how to compile a pattern with only one exception. The algorithm can be easily extended, through multiple calls to $Set\text{-}Difference$, to apply to patterns with multiple exceptions. We give an example of applying this algorithm to a simple example from chess in Section 3.1.4.

Compiling $make\text{-}false(\lambda s.\nu(s), Op, S)$

To understand the appropriate decomposition of this pattern compilation into a set of conjunction compilations, let us review the definition of $make\text{-}false$ given in Table 2.1(2):

$$make\text{-}false(\lambda s.\nu(s), Op, S) \Leftrightarrow \nu(S) \wedge o(S, Op) \wedge \neg \nu(do(Op, S))$$

Considering the simple case when $\nu(S) \Leftrightarrow c_0 \wedge \neg c_1$ gives us

$$c_0(S) \wedge \neg c_1(S) \wedge o(S, Op) \wedge \neg [c_0(do(Op, S)) \wedge \neg c_1(do(Op, S))].$$

1. Compile $make\text{-}true(\lambda s.c_0(s), Op, S) \mapsto make\text{-}true\text{-}c_0$.
2. Compile $maintain\text{-}true(\lambda s.c_0(s), Op, S) \mapsto maintain\text{-}true\text{-}c_0$.
3. Compile $make\text{-}true(\lambda s.c_1(s), Op, S) \mapsto make\text{-}true\text{-}c_1$.
4. Compile $make\text{-}false(\lambda s.c_1(s), Op, S) \mapsto make\text{-}false\text{-}c_1$.
5. Compute:

$$\begin{aligned} & Set\text{-}Difference(make\text{-}true\text{-}c_0, make\text{-}true\text{-}c_1) \\ & \cup Set\text{-}Intersection(make\text{-}true\text{-}c_0, make\text{-}false\text{-}c_1) \\ & \cup Set\text{-}Intersection(maintain\text{-}true\text{-}c_0, make\text{-}false\text{-}c_1) \mapsto make\text{-}true\text{-}\nu. \end{aligned}$$

Table 3.5: Compiling $make\text{-}true(\lambda s.\nu(s), Op, S)$, when $\nu(s) \Leftrightarrow c_0 \wedge \neg c_1$

By distributing the negation over the conjunction, we obtain

$$\begin{aligned} & c_0(S) \wedge \neg c_1(S) \wedge o(S, Op) \wedge \neg c_0(do(Op, S)) \\ & \vee c_0(S) \wedge \neg c_1(S) \wedge o(S, Op) \wedge c_1(do(Op, S)). \end{aligned}$$

Simplifying gives

$$make\text{-}false(\lambda s.c_0(s), Op, S) \vee make\text{-}true(\lambda s.c_1(s), Op, S)$$

This analysis tells us that there are two alternative ways to make false a conjunction with one exception: (1) we make false c_0 or (2) we make true c_1 . Hence we have identified the appropriate decomposition for the compilation problem. We illustrate the algorithm in Table 3.4.

This description has focused on compiling a pattern with only one exception. We can easily extend the algorithm, through multiple calls to $make\text{-}false(\lambda s.c_j(s), Op, S)$ for each negated conjunction c_j , to apply to patterns with multiple exceptions. We give an example of applying this algorithm to a simple example from chess in Section 3.1.4.

Compiling $make\text{-}true(\lambda s.\nu(s), Op, S)$

To understand the appropriate decomposition of this pattern compilation into a set of conjunction compilations, let us review the definition of $make\text{-}true$ given in Table 2.1(1):

$$\text{make-true}(\lambda s.\nu(s), Op, S) \Leftrightarrow \neg\nu(S) \wedge o(S, Op) \wedge \nu(\text{do}(Op, S)).$$

Considering the simple case when $\nu(S) \Leftrightarrow c_0 \wedge \neg c_1$ gives us

$$\neg[c_0(S) \wedge \neg c_1(S)] \wedge o(S, Op) \wedge c_0(\text{do}(Op, S)) \wedge \neg c_1(\text{do}(Op, S)).$$

By distributing the negation over the conjunction we obtain

$$\begin{aligned} & \neg c_0(S) \wedge c_0(\text{do}(Op, S)) \wedge \neg c_1(\text{do}(Op, S)) \\ & \vee c_1(S) \wedge c_0(\text{do}(Op, S)) \wedge \neg c_1(\text{do}(Op, S)). \end{aligned}$$

Simplifying gives

$$\begin{aligned} & \text{make-true}(\lambda s.c_0(s), Op, S) \wedge \neg \text{make-true}(\lambda s.c_1(s), Op, S) \\ & \vee \text{make-true}(\lambda s.c_0(s), Op, S) \wedge \text{make-false}(\lambda s.c_1(s), Op, S) \\ & \vee \text{maintain-true}(\lambda s.c_0(s), Op, S) \wedge \text{make-false}(\lambda s.c_1(s), Op, S). \end{aligned}$$

This analysis tells us that there are three alternative ways to make true a conjunction with one exception: (1) we make true the c_0 and not make true c_1 , or (2) we make true the c_0 and make false c_1 , or (3) we maintain true c_0 and make false c_1 . These three cases correspond to all combinations of the initial truth values for the two conjunctions (except the one combination $c_0(S) \wedge \neg c_1(S)$, since the pattern must be false in S). Hence we have identified the appropriate decomposition for the compilation problem. We illustrate the algorithm in Table 3.5.

We introduce a new function *Set-Intersection* which is like *Set-Difference* previously defined, but in this case we compute the intersection of the two sets of pattern/action pairs. The function is defined in Table 3.6.

To compute *Set-Intersection*, we compute the cross product of the sets of pattern/action pairs as illustrated previously for *Set-Difference*. For each pair from l_0 and l_1 , we compute the intersection (using a function *Pattern-Intersection* defined earlier) that first unifies the operators and then determines if the patterns intersect. There same three cases can result from this operation as before. Whenever the intersection is non-null, we include the intersected pattern and action in the output solution.

<p>Given:</p> <ul style="list-style-type: none"> • $l_0 = \{\langle \nu_0^1(S), Op_0^1 \rangle, \langle \nu_0^2(S), Op_0^2 \rangle, \dots, \langle \nu_0^m(S), Op_0^m \rangle\}$. • $l_1 = \{\langle \nu_1^1(S), Op_1^1 \rangle, \langle \nu_1^2(S), Op_1^2 \rangle, \dots, \langle \nu_1^m(S), Op_1^m \rangle\}$. <p>Find:</p> <ul style="list-style-type: none"> • $l = \{\langle \nu^1(S), Op^1 \rangle, \langle \nu^2(S), Op^2 \rangle, \dots, \langle \nu^k(S), Op^k \rangle\}$ Such that: $\forall \langle \nu_0^i(S), Op_0^i \rangle \in l_0$, and $\forall \langle \nu_1^j(S), Op_1^j \rangle \in l_1$, If $Op_0^i = Op_1^j$ then $\langle \nu_0^i(S) \cap \nu_1^j(S), Op_0^i \rangle \in l$.

Table 3.6: The *Set-Intersection*(l_0, l_1) function

This section has focussed on describing the compilation step when the given pattern has only one exception. The algorithm can be extended to more exceptions. In general, when we have n exceptions, there will be $2^{n+1} - 1$ combinations of conjunctive influence relations that make up the solution, each corresponding to a unique initial setting of the truth values of the conjunctions. We give an example of applying this algorithm to a simple example from chess in Section 3.1.4.

3.1.4 Chess Examples

This section introduces a domain theory for chess, and illustrates our algorithms compiling influence relations for some complex patterns that arise in the chess domain. Figure 3.6 illustrates a slightly simplified definition of the chess² domain theory used in this work.

The theory employs a generic language for describing the chess positions and patterns. This language defines a situation as an arrangement of objects at locations in space. This is the same language that was used in the stuffy room domain. In particular, a situation S is defined by a set of $on(S, Loc, Obj)$ relations,

²Simplifications include ignoring case analysis on whether variables are bound for the openline relation. The complete domain theory is listed in the Appendix.

```

in-check(S0,Side1):-
    opside(Side1,Side2),
    on(S0,From,obj(Typet,Side2)),
    on(S0,To,obj(king,Side1)),
    legaldirection(Typet,Direct),
    reachable(S0,Typet,From,To,Direct).

opside(white,black).
opside(black,white).

sliding-piece(rock).
sliding-piece(queen).
sliding-piece(bishop).

o(S0,Op,Si1):-
    (Op = move(nm,SqF,SqT,obj(Tm,Si1),empty),
    on(S0,SqF,obj(Tm,Si1)),
    legaldirection(Tm,Direct),
    reachable(S0,Tm,SqF,SqT,Direct),
    on(S0,SqT,empty)
    ;Op = move(tm,SqF,SqT,obj(Tm,Si1),obj(Tt,Si2)),
    on(S0,SqF,obj(Tm,Si1)),
    legaldirection(Tm,Direct),
    reachable(S0,Tm,SqF,SqT,Direct),
    opside(Si1,Si2), %must be opposite side
    on(S0,SqT,obj(Tt,Si2))
    ).

legaldirection(king,( 1, 1)).
legaldirection(king,( 1, 0)).
legaldirection(king,( 1,-1)).
legaldirection(king,( 0, 1)).
legaldirection(king,( 0,-1)).
legaldirection(king,(-1, 1)).
legaldirection(king,(-1, 0)).
legaldirection(king,(-1,-1)).

legaldirection(knight,( 1, 2))...

reachable(S0,Type,SqF,SqT,Direct):-
    not(single-piece(Type)),
    openline(S0,SqF,SqT,Direct).

openline(S0,SqF,SqT,Direct):-
    (connected(SqF,SqT,Direct)
    ;connected(SqF,SqI,Direct),
    on(S0,SqI,empty),
    openline(S0,SqI,SqT,Direct)
    ).

reachable(S0,Type,SqF,SqT,Direct):-
    not(sliding-piece(Type)),
    connected(SqF,SqT,Direct).

```

Figure 3.6: Definition of the Chess domain

where Loc is the location (represented as an (X,Y) coordinate) and Obj is the object that is at Loc in situation S . Objects are either **empty**, denoting an empty location, or composite terms describing their properties. In chess, an object is defined by the composite term $obj(\text{Type},\text{Side})$ which denote its two properties Type (such as **bishop** or **knight**) and Side (**black** or **white**). Included in this generic language is the relation $openline(S,F,T,Dir)$, which defines a contiguous line of empty locations along direction Dir originating at location F and terminating at location T . Also included is the relation $connected(F,T,Dir)$ that constrains two locations to be connected in a given direction.

To define the chess domain, we must define the **in-check** constraint for use generating legal moves from pseudo-moves (see Section 2.2.3). We must also define termination patterns, which denote winning goals, such as capturing the king. The $in\text{-}check(S,Side)$ constraint is defined in Figure 3.6. It is true when $Side$ is in check in situation S . Also included is the definition of the operator generator $o(S,Op,Side)$, which is used to generate *pseudo-moves* given a situation S and a side $Side$. There are two kinds of moves, normal moves (denoted nm) and take moves (denoted tm). Legal moves are generated from pseudo-moves by taking into account whether the moving side is in check as described in Section 2.2.3.

The theory introduces additional relations such as $opside(Side1,Side2)$, which declares that **white** and **black** are opposite sides; $sliding\text{-}piece(T)$, which declares that pieces of type T can move through multiple squares; $single\text{-}piece(T)$, which declares that pieces of type T can only move through single square; and $legaldirection(\text{Type},Dir)$, which maps the piece type to the legal directions along which it can move.

The frame axioms for the domain theory are given in Figure 3.7. We include frame axioms for the generic situation predicates on and $openline$ for both move types.

Now that we have introduced the chess domain theory, we introduce two

```

on(do(move(nm,SqF,SqT,Of,Ot),S0),Sq,O):-
  (Sq = SqF -> %just moved from
   O = empty
  ;Sq = SqT -> %just moved to
   O = Of
  ;otherwise ->
   on(S0,Sq,O)
  ).

on(do(move(tm,SqF,SqT,Of,Ot),S0),Sq,O):-
  (Sq = SqF -> %just moved from
   O = empty
  ;Sq = SqT -> %just moved to
   O = Of
  ;otherwise ->
   on(S0,Sq,O)
  ).

openline(do(Op,S0),Start,End,D):-
  Op=move(nm,SqF,SqT,Of,Ot),
  (inline(Start,SqT,End) ->
   fail %just moved into line
  ;inline(Start,SqF,End) ->
   % just moved from line
   openline(S0,Start,SqF,D),
   on(S0,SqF,Of),
   openline(S0,SqF,End,D)
  ;SqT = Start ->
   % just moved to start of line
   openline(S0,Start,End,D)
  ;SqT = End ->
   % just moved to end of line
   openline(S0,Start,End,D)
  ;otherwise ->
   openline(S0,Start,End,D)
  ).

openline(do(Op,S),Start,End,D):-
  Op=move(tm,SqF,SqT,Of,Ot),
  (inline(Start,SqT,End) ->
   fail %just moved into line
  ;inline(Start,SqF,End) ->
   % just moved from line
   openline(S0,Start,SqF,D),
   on(S0,SqF,Of),
   openline(S0,SqF,End,D)
  ;SqT = Start ->
   % just moved to start of line
   openline(S0,Start,End,D)
  ;SqT = End ->
   % just moved to end of line
   openline(S0,Start,End,D)
  ;otherwise ->
   openline(S0,Start,End,D)
  ).

```

Figure 3.7: Definition of the frame axioms for the Chess domain


```

capture-king(S) :-
  on(S,SqR,obj(rook,white)),
  legaldirection(rook,Direct),
  openline(S,SqR,SqBK,Direct),
  on(S,SqBK,obj(king,black)).

```



Figure 3.8: Prolog definition and graphical representation of the termination pattern `rook-takes-king`

example patterns that arise in compiling chess databases. For each pattern we give its definition written in the domain theory and then demonstrate compiling the influence relations over the pattern. The last section of this chapter introduces the influence proof compiler and illustrates the compilation of influence proofs which use these two patterns.

Examples of Chess Patterns

In this section, since we are interested in illustrating the influence compiler, we limit the patterns to termination patterns defining winning situations that are provided by the user. In particular, we consider those patterns that arise when specifying sub-domains of chess where only a few playing pieces are involved. These sub-domains represent challenging problems that arise during the endgame phases of complete chess games.

There are many different patterns that could be considered. Here we illustrate two simple patterns that arise in popular endgames. One pattern defines a special case of the ultimate goal in chess, when the opponents' king is captured by a rook. This pattern is used in all sub-domains that involve the rook on the winning side. Another pattern arises in the king-rook, king-knight sub-domain and defines the conditions where the black knight is safely captured by the white rook. This case is more complex than capturing the king, since we must ensure that the capture is safe, i.e., the capturing rook cannot itself be recaptured.

The first pattern describing the capture of the black king is defined as a cap-

```

safe-capture-knight1(S) :-
  on(S,SqR,obj(rook,white)),
  legaldirection(rook,DirectR),
  openline(S,SqR,SqN,DirectR),
  on(S,SqN,obj(knight,black))
  not(on(S,SqBK,obj(king,black)),
    legaldirection(king,DirectK),
    connected(SqBK,SqN,DirectK)).

```

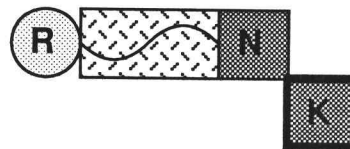


Figure 3.9: Prolog definition and graphical representation of the termination pattern `rook-takes-knight1`. Note that exceptions are denoted by a bold outline around the objects of the exception. Hence, the black king must *not* be adjacent to the black knight.

ture move of the black king by the white rook. The simplified pattern is illustrated in Figure 3.8 (see Figure 1.2(b) for key to graphical representation of the pattern). This pattern is true when there is a white rook on `SqR` and open line of empty squares along direction `Direct`, where `Direct` is legal for the rook, leading to location `SqBK`, which is occupied by the black king.

We consider two patterns which define different forms of the safe capture of the black knight, given that the only other black piece is the king. The first pattern, illustrated in Figure 3.9, defines the situation where the white rook can capture the black knight and the black king does not protect the black knight. The principal conjunction (that is, the non-negated conjunction) is similar to the previous pattern, only in this case the captured piece is a knight. The exception defines the situation where the black king is on some location `SqBK` which is connected to the location of the black knight `SqN` in a direction that is legal for the black king to move. Hence the exception defines the situation where the black king could recapture the rook.

The other pattern, illustrated in Figure 3.10, describes the case when the black king is adjacent the knight, but unable to recapture due to the white king also being adjacent to the rook. The principal conjunction of this pattern defines the situation where the rook can capture the black knight (just as before) and both

```

safe-capture-knight2(S) :-
  on(S,SqR,obj(rook,white)),
  legaldirection(rook,DirectR),
  openline(S,SqR,SqN,DirectR),
  on(S,SqN,obj(knight,black))
  on(S,SqBK,obj(king,black)),
  legaldirection(king,DirectBK),
  connected(SqBK,SqN,DirectBK)
  on(S,SqWK,obj(king,white)),
  legaldirection(king,DirectWK),
  connected(SqWK,SqN,DirectWK),
  not(connected(SqBK,SqWK,DirectK),
      legaldirection(king,DirectK)).

```

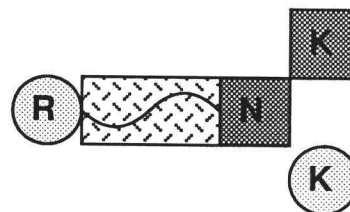


Figure 3.10: Prolog definition and graphical representation of the termination pattern rook-takes-knight2

the black king and the white king are adjacent to the black knight. The exception arises from the legal requirement that the two kings cannot be adjacent.

Example of *maintain-true*(capture-king(s),Op,S)

Let us consider the operators by black that maintain the pattern *capture-king*(S) defined above. This case illustrates the role of the frame axioms for *openline*, given in Figure 3.7, in compiling the influence relations. Given that the black pieces in play are limited to the king and knight, intuitively, we would expect two kinds of moves to result. First we have non-take moves of the king such that there is still an openline in a legal direction for the rook from the rook position to the new location of the king. Second we have non-take moves of the **black knight** which do not block the openline. The compiler determines this in 9 seconds. We illustrated in Figure 3.11 two of the cases derived.

The possible moves of the black king that maintain *king-capture*(S) are determined by unfolding and simplifying the expression *king-capture*(S), *o*(S,Op,black), *king-capture*(do(Op,S)) as described in Section 3.1.2. This analysis produces two cases, one where the king moves in the same direction as the openline away from

$\forall Op, o(S, \text{black}, Op), \text{maintain-true}(\text{capture-king}(S), Op, S) \Leftrightarrow$
 \langle
 $\{$
 $\text{on}(S, SqR, \text{obj}(\text{rook}, \text{white})),$
 $\text{legaldirection}(\text{rook}, \text{DirectR0}),$
 $\text{openline}(S, SqR, SqBK, \text{DirectR0}),$
 $\text{on}(S, SqBK, \text{obj}(\text{king}, \text{black})),$
 $\text{legaldirection}(\text{king}, \text{DirectK}),$
 $\text{connected}(SqBK, SqT, \text{DirectK}),$
 $\text{on}(S, SqT, \text{empty}),$
 $\text{legaldirection}(\text{rook}, \text{DirectR1}),$
 $\text{openline}(S, SqR, SqT, \text{DirectR1}),$
 $\text{DirectR1} \neq \text{DirectR0},$
 $Op = \text{move}(\text{nm}, SqBK, SqT, \text{obj}(\text{king}, \text{black}), \text{empty}) \},$
 \langle
 $\{$
 $\text{on}(S, SqR, \text{obj}(\text{rook}, \text{white})),$
 $\text{legaldirection}(\text{rook}, \text{DirectR0}),$
 $\text{openline}(S, SqR, SqBK, \text{DirectR0}),$
 $\text{on}(S, SqBK, \text{obj}(\text{king}, \text{black})),$
 $\text{on}(S, SqN, \text{obj}(\text{knight}, \text{black})),$
 $\text{legaldirection}(\text{knight}, \text{DirectN}),$
 $\text{connected}(SqN, SqT, \text{DirectN}),$
 $\text{on}(S, SqT, \text{empty}),$
 $\text{not}(\text{inline}(SqR, SqT, SqBK)),$
 $Op = \text{move}(\text{nm}, SqN, SqT, \text{obj}(\text{knight}, \text{black}), \text{empty}) \}.$

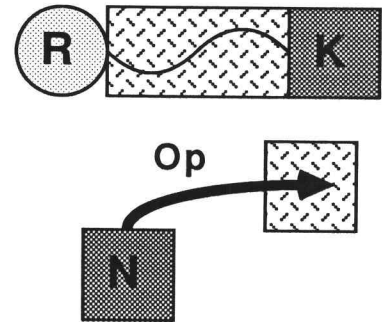
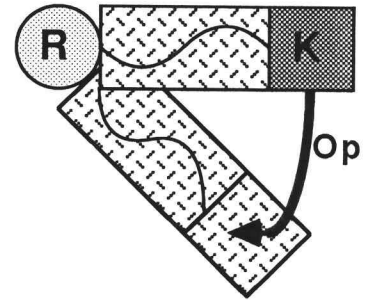


Figure 3.11: Pattern/action pairs that maintain the termination pattern capture-king(S), with black moving

the rook (not illustrated) and the other when the king moves in any other direction (illustrated). To understand why these two cases arise, it is necessary to study how the compiler processes expressions that involve openlines. Consider the last literal in the maintain expression above, `king-capture(do(Op,S))`. When this is unfolded, it includes `openline(do(Op,S),SqR,SqT,D)`.

Partially evaluating `openline(do(Op,S),SqR,SqT,D)` produces five cases (given in the lower left of Figure 3.7). The first case, where the object moves into the line, fails quickly. The second case, where the object moves from the line, produces the solution, which is not illustrated, where the king moves in the same direction as the rook openline. This solution is found during simplification when functional dependencies of the `openline` relation are applied. Note that the candidate solution in this case will include both `openline(S,SqR,SqBK,D)`, from unfolding the frame axiom (see Figure 3.7), and `openline(S,SqR,SqBK,DirectR)`, from the original pattern. Applying functional dependencies determines that `D`, the direction of the openline to the new location of the king, must be the same as `DirectR`, the direction of the original openline. Further simplifications constrain the direction of the king move to be the same as `DirectR`. The third case, where the move is to the start of the openline, fails because it constrains the destination location of the move (which must be empty) to be the same as the start of the openline (which is occupied by the rook). The fourth case, where the move is to the end of the openline, succeeds and produce the solution illustrated above. The final case fails, because it implies that the new openline from the rook to the king is independent of the move. This is clearly false, since we are considering the case when the king is moved.

The alternative solution, when the knight is moved, is determined similarly. In this case, we again have the five cases which arise from unfolding `openline(do(Op,S),SqR,SqBK,D)`. However, here all but the last case fails.

Example of `make-false(safe-capture-knight1(S),Op,S)`

Let us use the example of compiling `make-false(safe-capture-knight1(S),Op,S)` with

black to play to illustrate compiling influence relations for a pattern that has exceptions. In this case, since we have a pattern with one exception, we make it false one of two ways: (1) we make-false the positive conjunction (that a rook is inline with the knight) or (2) make-true the negated conjunction (that the black king is adjacent to the black knight). Compiling (1) leads to a two pattern/action pairs, one which is illustrated, where the black king captures the rook, and another where the black knight moves out the way; compiling (2) leads to a single pattern/action pair where the black king moves to become adjacent to the knight. The two cases are illustrated in Figure 3.12.

Example of *make-true*(safe-capture-knight2(S),Op,S)

Let us use the example of compiling *make-true*(safe-capture-knight2(S),Op,S0) for white to illustrate the introduction of exceptions in the resulting patterns. Here we expect at least two sets of pattern/operator pairs; those that move the white rook so that an openline from the rook to the knight becomes true, and those that move the white king to be adjacent to the black knight. The two cases are illustrated in Figure 3.13.

3.2 Compiling Influence Proofs

Once we have defined how influence relations are compiled, we can now turn our attention to compiling the influence proofs. The goal of this compilation stage is to compile a given influence proof into an equivalent set of pattern/action pairs, which recommend the same action in the same situations, but without the complexity of problem space search.

As we have previously described (in the introduction to Section 3) we only consider compiling a proof when proofs lower depths (i.e., shorter solution lengths) have already been compiled into goal patterns. The compilation process is further simplified by ordering process to compile simpler proofs before more complex proofs.

$$\forall Op, o(S, \text{black}, Op), \text{make-false}(\text{safe-capture-knight1}(S), Op, S) \Leftrightarrow$$

$$\langle \{ \text{on}(S, \text{SqR}, \text{obj}(\text{rook}, \text{white})),$$

$$\text{legaldirection}(\text{rook}, \text{DirectR}),$$

$$\text{openline}(S, \text{SqR}, \text{SqN}, \text{DirectR}),$$

$$\text{on}(S, \text{SqN}, \text{obj}(\text{knight}, \text{black}))$$

$$\text{on}(S, \text{SqBK}, \text{obj}(\text{king}, \text{black})),$$

$$\text{legaldirection}(\text{king}, \text{DirectK1}),$$

$$\text{connected}(\text{SqBK}, \text{SqR}, \text{DirectK1}),$$

$$\text{not}(\text{legaldirection}(\text{king}, \text{DirectK2}),$$

$$\text{connected}(\text{SqBK}, \text{SqN}, \text{DirectK2})) \rangle$$

$$Op = \text{move}(\text{tm}, \text{SqBK}, \text{SqR}, \text{obj}(\text{king}, \text{black}), \text{obj}(\text{rook}, \text{white})) \rangle,$$

$$\langle \{ \text{on}(S, \text{SqR}, \text{obj}(\text{rook}, \text{white})),$$

$$\text{legaldirection}(\text{rook}, \text{DirectR}),$$

$$\text{openline}(S, \text{SqR}, \text{SqN}, \text{DirectR}),$$

$$\text{on}(S, \text{SqN}, \text{obj}(\text{knight}, \text{black})),$$

$$\text{on}(S, \text{SqBK}, \text{obj}(\text{king}, \text{black})),$$

$$\text{legaldirection}(\text{king}, \text{DirectK1}),$$

$$\text{connected}(\text{SqBK}, \text{SqT}, \text{DirectK1}),$$

$$\text{on}(S, \text{SqT}, \text{empty}),$$

$$\text{legaldirection}(\text{king}, \text{DirectK2}),$$

$$\text{connected}(\text{SqT}, \text{SqN}, \text{DirectK2}),$$

$$\text{not}(\text{on}(S, \text{SqBK}, \text{obj}(\text{king}, \text{black})),$$

$$\text{legaldirection}(\text{king}, \text{DirectK3}),$$

$$\text{connected}(\text{SqBK}, \text{SqN}, \text{DirectK3})) \rangle$$

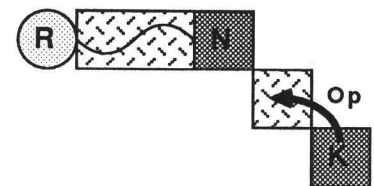
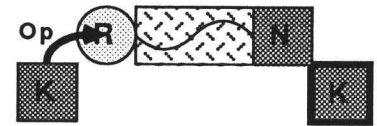
$$Op = \text{move}(\text{nm}, \text{SqBK}, \text{SqT}, \text{obj}(\text{king}, \text{black}), \text{empty}) \rangle.$$


Figure 3.12: Pattern/action pairs that make false the termination pattern `safe-capture-knight1(S)`, with black moving

$$\forall Op \in o(S, Op), \text{make-true}(\text{safe-capture-knight2}(S), Op, S) \Leftrightarrow$$

$$\langle [\text{on}(S, SqR, \text{obj}(\text{rook}, \text{white})),$$

$$\text{legaldirection}(\text{rook}, \text{DirectR1}),$$

$$\text{openline}(S, SqR, SqT, \text{DirectR1}),$$

$$\text{on}(S, SqT, \text{empty}),$$

$$\text{legaldirection}(\text{rook}, \text{DirectR2}),$$

$$\text{openline}(S, SqT, SqN, \text{DirectR2}),$$

$$\text{on}(S, SqN, \text{obj}(\text{knight}, \text{black}))$$

$$\text{on}(S, SqBK, \text{obj}(\text{king}, \text{black})),$$

$$\text{legaldirection}(\text{king}, \text{DirectBK}),$$

$$\text{connected}(SqBK, SqN, \text{DirectBK})$$

$$\text{on}(S, SqWK, \text{obj}(\text{king}, \text{white})),$$

$$\text{legaldirection}(\text{king}, \text{DirectWK}),$$

$$\text{connected}(SqWK, SqN, \text{DirectWK}),$$

$$\text{not}(\text{connected}(SqBK, SqWK, \text{DirectK})),$$

$$\text{legaldirection}(\text{king}, \text{DirectK})$$

$$\text{not}(\text{legaldirection}(\text{rook}, \text{DirectR3}),$$

$$\text{openline}(S, SqR, SqN, \text{DirectR3}))]$$

$$Op = \text{move}(\text{nm}, SqR, SqT, \text{obj}(\text{rook}, \text{white}), \text{empty}) \rangle,$$

$$\langle [\text{on}(S, SqR, \text{obj}(\text{rook}, \text{white})),$$

$$\text{legaldirection}(\text{rook}, \text{DirectR}),$$

$$\text{openline}(S, SqR, SqN, \text{DirectR}),$$

$$\text{on}(S, SqN, \text{obj}(\text{knight}, \text{black}))$$

$$\text{on}(S, SqBK, \text{obj}(\text{king}, \text{black})),$$

$$\text{legaldirection}(\text{king}, \text{DirectBK}),$$

$$\text{connected}(SqBK, SqN, \text{DirectBK})$$

$$\text{on}(S, SqWK, \text{obj}(\text{king}, \text{white})),$$

$$\text{legaldirection}(\text{king}, \text{DirectWK1}),$$

$$\text{connected}(SqWK, SqT, \text{DirectWK1}),$$

$$\text{on}(S, SqT, \text{empty}),$$

$$\text{legaldirection}(\text{king}, \text{DirectWK2}),$$

$$\text{connected}(SqT, SqN, \text{DirectWK2}),$$

$$\text{not}(\text{connected}(SqT, SqWK, \text{DirectK})),$$

$$\text{legaldirection}(\text{king}, \text{DirectK})$$

$$\text{not}(\text{connected}(SqBK, SqWK, \text{DirectK1}),$$

$$\text{legaldirection}(\text{king}, \text{DirectK1}))]$$

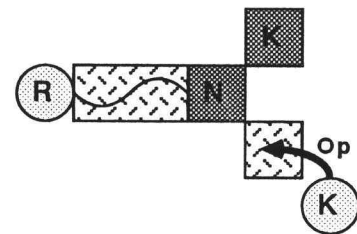
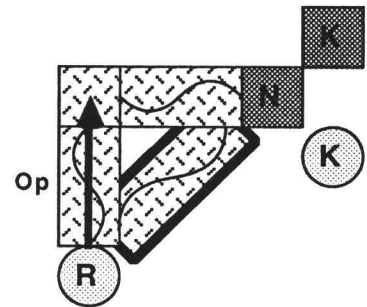
$$Op = \text{move}(\text{nm}, SqWK, SqT, \text{obj}(\text{king}, \text{white}), \text{empty}) \rangle$$


Figure 3.13: Pattern/action pairs that make true the termination pattern *safe-capture-knight2*(S), with white moving

For this section, we will assume that there is some controlling process that generates the proofs for compilation in an appropriate order. Section 7.1 gives full details of this control process, this section covers only the compilation of a given influence proof.

The remainder of this section is in two parts. The first section describes the compilation of influence proofs for the losing side, introduced in Section 2.2.2 and Section 2.4.1. The second section describes the compilation of proofs for the winning side, given in Section 2.4.2. In both cases we assume that previous compilation has produced a set of n goal patterns of lower solution length denoted $\{\nu_1^G(s), \nu_2^G(s), \dots, \nu_n^G(s)\}$. When compiling proofs for the losing side, this set will be used to denote previous *winning* patterns; when compiling proofs for the winning side, this set will be used to denote previous *losing* patterns.

3.2.1 When *LOSS* is to Play

The compilation process when *LOSS* is to play is simplified by noting the three kinds of influence proofs identified in Section 2.2.2: *No-threat*, *One-threat* and *Multiple-threat*. Each proof type has a distinct compilation strategy which is given below.

Compiling *No-threat* Proofs

A *no-threat* proof describes a loss situation where none of the winning goal patterns are true initially (and hence, there are no current threats) but all legal moves available lead to new situations where one or more of the winning goal patterns are true. Logically, these proofs are defined as follows:

$$\begin{aligned} \text{achieve}(\lambda s.G(s), S, \text{LOSS}, i) \Leftarrow \\ \neg\nu_1^G(S) \wedge \neg\nu_2^G(S) \wedge \dots \wedge \neg\nu_n^G(S) \\ \wedge \forall Op, o(S, \text{LOSS}, Op), [\quad \text{make-true}(\lambda s.\nu_1^G(s), Op, S) \\ \quad \vee \text{make-true}(\lambda s.\nu_2^G(s), Op, S) \\ \quad \vee \dots \\ \quad \vee \text{make-true}(\lambda s.\nu_n^G(s), Op, S)] \end{aligned}$$

To compile this proof, given the set of n goal patterns, the first step is to compile the *make-true* relations for each goal pattern as described in Section 3.1.2, Page 47. This leads to a set of *make-true* pattern/action pairs, called MT , where each pair is represented as $\langle \nu_j^{MT}(S), Op_j^{MT} \rangle$. To understand the next step of the compilation process, it is helpful to take a *procedural* view of the logical proof above. This proof states that a situation is a loss if all legal moves available make some subset of winning goal patterns true. Hence, to identify a new loss goal pattern, $\nu^G(S)$ which satisfies this proof, we must first identify some subset of the pattern/action pairs in MT such that the *union* of the actions of each pair *cover* all legal operators that are available in $\nu^G(S)$. Then $\nu^G(S)$ is the *intersection* of the patterns from each pattern/action pair. More precisely, a goal pattern $\nu^G(S)$ is an example of a *no-threat* loss if all the following are true:

1. $\{\langle \nu_1^{MT}(S), Op_1^{MT} \rangle, \langle \nu_2^{MT}(S), Op_2^{MT} \rangle, \dots, \langle \nu_k^{MT}(S), Op_k^{MT} \rangle\} \subset MT$, where MT is defined above,
2. $\nu_1^{MT}(S) \cap \nu_2^{MT}(S) \cap \dots \cap \nu_k^{MT}(S) = \nu^G(S) \neq \emptyset$,
3. $Op_1^{MT} \cup Op_2^{MT} \cup \dots \cup Op_k^{MT} = \forall Op, o(S, LOSS, Op)$.

We find subsets of pattern/action pairs that make true winning goal patterns, such that the intersection of those patterns is non-empty, and the union of the actions *covers* all possible operators that are available in S .

To complete this definition, we must provide procedures for computing both the intersection of patterns with a corresponding test for non-empty pattern, and the union of operators with a corresponding test for coverage of all available operators. The intersection of patterns is easily specified by the procedure *Pattern-Intersection* defined previously in Table 3.3. However, the union operator is more difficult to define, since with our current representation of pattern/action pairs, the set of actions is represented *implicitly*. By this we mean there is no expression that denotes those actions which exactly comprise the actions of the pattern/action pair. Rather, the

pattern implicitly describes the set of actions as a set of constraints on the arguments of the operator. To see this, refer to the example of *make-true* pattern/action pairs given in Figure 3.13. Here, the action set *make true* the winning pattern *safe-capture-knight2* for white. In the top solution, the set of actions is constrained to be by the white rook, with the destination constrained to be a location with an openline to the black knight and the origin to be a location occupied by the white rook, but not inline with the black knight. While these constraints correctly define the action set, they do not represent the set explicitly (such as by a set of legal directions and distances for the piece that is moved). Hence, this representation cannot be used for determining whether the set of all available operators are *covered* by a union of such sets. This means that the current representation of pattern/action pairs is not expressive enough to compile these *no-threat* proofs. In the next Chapter we introduce a new representation that overcomes this problem.

Compiling *One-threat* Proofs

A *One-threat* proof describes a loss situation where one of the winning goal patterns are true initially (and hence, there is one current threat) and all legal moves available lead to new situations where either the initial threat is still true or one or more of the other winning goal patterns are true. There are n such proofs, given n winning goal patterns. Without loss of generality let us assume that the one goal pattern true initially is $\nu_j^G(S), 1 \leq j \leq n$.

Logically, these proofs are defined as follows:

$$\begin{aligned}
 \text{achieve}(\lambda s.G(s), S, \text{LOSS}, i) \Leftarrow & \\
 & \neg\nu_1^G(S) \wedge \neg\nu_2^G(S) \wedge \dots \wedge \nu_j^G(S) \wedge \dots \wedge \neg\nu_n^G(S) \\
 \wedge \forall Op, o(S, \text{LOSS}, Op), [& \text{maintain-true}(\lambda s.\nu_j^G(s), Op, S) \\
 & \vee \text{make-true}(\lambda s.\nu_1^G(s), Op, S) \\
 & \vee \text{make-true}(\lambda s.\nu_2^G(s), Op, S) \\
 & \vee \dots \\
 & \vee \text{make-true}(\lambda s.\nu_{j-1}^G(s), Op, S) \\
 & \vee \text{make-true}(\lambda s.\nu_{j+1}^G(s), Op, S) \\
 & \vee \dots
 \end{aligned}$$

$$\forall \text{ make-true}(\lambda s. \nu_n^G(s), Op, S)]$$

Compiling this proof is similar to the *no-threat* proof. First, given the set of goal patterns, we compile for each goal pattern the *make-true* relations (described in Section 3.1.2, Page 47) and the *maintain-true* relations (described in Section 3.1.2, Page 42). This leads to a set of *make-true* pattern/action pairs, called MT as above and a set of *maintain-true* pattern/action pairs, called M , where each pair is represented as $\langle \nu_k^M(S), Op_k^M \rangle$. Taking the procedural view of the logical proof above leads directly to the compilation algorithm. This proof states that a situation is a loss if all legal moves available either maintain the threat or make some subset of other winning goal patterns true. Hence, to identify a new loss goal pattern, $\nu^G(S)$ which satisfies this proof, we first set $\nu^G(S) = \nu_j^G(S)$ (the threat pattern) then we identify some subset of the pattern/action pairs in MT called MT_C , and some subset of the pattern/action pairs in M which maintain $\nu_j^G(S)$, called M_C , such that the *union* of those actions in M_C and those actions in MT_C cover all legal operators that are available in $\nu^G(S)$. Then $\nu^G(S)$ is the *intersection* of $\nu_j^G(S)$, the patterns from each pattern/action pair in MT_C and the patterns from each pattern/action pair in M_C . More precisely, a goal pattern $\nu^G(S)$ is an example of a *one-threat* loss if all the following are true:

1. $\{\langle \nu_1^M(S), Op_1^M \rangle, \langle \nu_2^M(S), Op_2^M \rangle, \dots, \langle \nu_k^M(S), Op_k^M \rangle\} \subset M$, where each Op^M maintains $\nu_j^G(S)$ (the threat goal pattern),
2. $\{\langle \nu_1^{MT}(S), Op_1^{MT} \rangle, \langle \nu_2^{MT}(S), Op_2^{MT} \rangle, \dots, \langle \nu_m^{MT}(S), Op_m^{MT} \rangle\} \subset MT$, where MT is defined above,
3. $\nu_j^G(S) \cap \nu_1^M(S) \cap \nu_2^M(S) \cap \dots \cap \nu_k^M(S) \cap \nu_1^{MT}(S) \cap \nu_2^{MT}(S) \cap \dots \cap \nu_m^{MT}(S) = \nu^G(S) \neq \emptyset$,
4. $Op_1^M \cup Op_2^M \cup \dots \cup Op_k^M \cup Op_1^{MT} \cup Op_2^{MT} \cup \dots \cup Op_m^{MT} = \forall Op, o(S, LOSS, Op)$.

We find subsets of pattern/action pairs which maintain the threat or make true other winning goal patterns, such that the intersection of those patterns with the threat pattern is non-empty and the union of the actions *covers* all possible operators that are available in S .

Since this compilation step requires us to take the union of the operator sets and determine coverage, compiling *one-threat* proofs suffers from the same problem of unsuitable representation as the *no-threat* proofs. Hence, we will have to wait until the introduction of new representations in Chapter 5 before these proof types can be compiled.

Compiling *Many-threat* Proofs

A *Many-threat* proof describes a loss situation where at least two of the winning goal patterns are true initially (and hence, there are at least two current threats) and all legal moves available lead to new situations where at least one of the the initial threats is still true. There are $2^n - 1 - n$ such proofs, given n winning goal patterns. Without loss of generality let us assume that only two goal patterns are true initially $\nu_j^G(S)$ and $\nu_k^G(S)$, $1 \leq j < k \leq n$. The case when more than two are true can be easily induced from this case.

Logically, these proofs are defined as follows:

$$\begin{aligned} \text{achieve}(\lambda s.G(s), S, \text{LOSS}, i) \Leftarrow & \\ & \neg\nu_1^G(S) \wedge \neg\nu_2^G(S) \wedge \dots \wedge \nu_j^G(S) \wedge \dots \wedge \nu_k^G(S) \wedge \dots \wedge \neg\nu_n^G(S) \\ & \wedge \forall Op, o(S, \text{LOSS}, Op), [\neg\text{make-false}(\lambda s.\nu_j^G(s), Op, S) \\ & \quad \vee \neg\text{make-false}(\lambda s.\nu_k^G(s), Op, S)] \end{aligned}$$

The method of compiling this proof is different from the two previous proofs because the coverage computation over the implicitly represented operator sets can be avoided. This is achieved by noticing that the disjunction of negated influence relations above can be simplified to a negated conjunction by applying DeMorgan's rule, then the negation can be brought outside the scope of the universal

quantification thereby changing the quantification to existential. The result of this manipulation is the following equivalent proof:

$$\begin{aligned} \text{achieve}(\lambda s.G(s), S, \text{LOSS}, i) \Leftarrow \\ \neg\nu_1^G(S) \wedge \neg\nu_2^G(S) \wedge \dots \wedge \nu_j^G(S) \wedge \dots \wedge \nu_k^G(S) \wedge \dots \wedge \neg\nu_n^G(S) \\ \wedge \neg\exists Op, o(S, \text{LOSS}, Op), [\text{make-false}(\lambda s.\nu_j^G(s), Op, S) \\ \wedge \text{make-false}(\lambda s.\nu_k^G(s), Op, S)] \end{aligned}$$

To compile such a proof, the first step is to compile for each goal pattern the *make-false* relations (described in Section 3.1.2, Page 51) leading to a set of *make-false* pattern/action pairs, called *MF*, where each pair is represented as $\langle \nu_k^{MF}(S), Op_k^{MF} \rangle$. The procedural view of the proof leads to a simple compilation algorithm. This proof states that a situation is a loss if there are two winning goal patterns true initially and there does not exist a legal move available that makes both threat patterns false. Hence, to identify a new loss goal pattern, $\nu^G(S)$ which satisfies this proof, we first set $\nu^G(S) = \nu_j^G(S) \cap \nu_k^G(S)$ (the intersection of the threat patterns), then we take the intersection of all those pattern/action pairs which make false $\nu_j^G(S)$ with all those pattern/action pairs which make false $\nu_k^G(S)$. Since we must ensure that this intersection of pattern/action pairs is empty, any resulting patterns from this intersection operation are negated then intersected with the current $\nu^G(S)$ to form the final goal pattern. More precisely, a goal pattern $\nu^G(S)$ is an example of a *many-threat* loss if all the following are true:

1. $\text{make-false}_j^G \Leftrightarrow \forall Op, \text{make-false}(\lambda s.\nu_j^G(s), S, Op)$, (one threat goal pattern),
2. $\text{make-false}_k^G \Leftrightarrow \forall Op, \text{make-false}(\lambda s.\nu_k^G(s), S, Op)$, (the other threat goal pattern),
3. $\nu_j^G(S) \cap \nu_k^G(S) \cap \neg[\text{make-false}_j^G \cap \text{make-false}_k^G] = \nu^G(S) \neq \emptyset$.

To complete this definition we must provide a procedure for computing the intersection of pattern/action pairs. This is simply the procedure *Set-Intersection* previously defined in Table 3.6, which returns a set of pattern/action pairs. For

this compilation step, the actions are not needed and simply ignored. The resulting patterns are negated and incorporated by the function *Pattern-Difference* previously defined in Section 3.1.3 on Page 58.

This form of influence proof can be compiled completely because the required constraint over the influence pattern/action pairs is empty intersection, and as we have previously seen, the implicit representation is adequate for this computation.

We illustrate this process in compiling the influence proof for the rook fork concept, of which Figure 1.1(a) is an example. Here the white rook threatens both the black king and black knight with capture. We employ the two patterns introduced in Section 3.1.4: *rook-takes-king(S)*, which describes the situation when the white rook can capture the black king, illustrated in Figure 3.8; and *safe-capture-knight1(S)*, which describes the situation when the white rook can safely capture the black knight (i.e., with out recapture by the black king), illustrated in Figure 3.9.

The particular proof we are interested in compiling is:

$$\begin{aligned} \text{achieve}(\lambda s.G(s), S, LOSS, i) \Leftarrow \\ \text{rook-takes-king}(S) \wedge \text{safe-capture-knight1}(S) \\ \wedge \neg \exists Op, o(S, \text{black}, Op), [\text{make-false}(\lambda s.\text{rook-takes-king}(s), Op, S) \\ \wedge \text{make-false}(\lambda s.\text{safe-capture-knight1}(s), Op, S)] \end{aligned}$$

Compiling *make-false*($\lambda s.\text{rook-takes-king}(s), Op, S$) for black to move leads to four pattern/action pairs: (a) where the black knight captures the rook, (b) where the black knight blocks the rook attack on the king, and (c) where the black king moves out of danger, and (d) where the black king takes the rook. The result of compiling *make-false*($\lambda s.\text{safe-capture-knight1}(s), Op, S$) for black is illustrated in Figure 3.12. Here we get three pattern/action pairs, (a) where the black knight moves out the way, (b) where the black king captures the rook, and (c) where the black king moves adjacent to the black knight.

The intersection operation is illustrated in Figure 3.14. Note that each box in the matrix represents one call to *Pattern-Intersection*. If we have p *make-false* pattern/action pairs for the first threat pattern and q *make-false* pattern/action

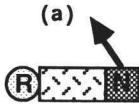
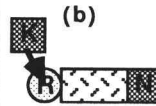
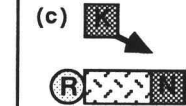






		<i>make-false(safe-capture-knight1 (s),Op,S)</i>		
		(a) 	(b) 	(c) 
<i>make-false(rook-takes-king (s),Op,S)</i>	(a) 	X		
	(b) 	X		
	(c) 		X	(2) 
	(d) 		(1) 	X

Figure 3.14: Compilation of a rook fork losing goal pattern with black to play. We intersect those pattern/action pairs from $make-false(\lambda s.rook-takes-king(s), Op, S)$ (illustrated along the side) with those pattern/action pairs from $make-false(\lambda s.safe-capture-knight1(s), Op, S)$ (illustrated across the top). Crosses denote an empty intersection


```

rook-fork(S) :-
  on(S,SqR,obj(rook,white)),
  legaldirection(rook,DirectR1),
  openline(S,SqR,SqBK,DirectR1),
  on(S,SqBK,obj(king,black)),
  legaldirection(rook,DirectR2),
  openline(S,SqR,SqN,DirectR2),
  on(S,SqN,obj(knight,black)),
  not( on(S,SqBK,obj(king,black)),
      legaldirection(king,DirectK1),
      connected(SqBK,SqN,DirectK1)),
  not( on(S,SqBK,obj(king,black)),
      legaldirection(king,DirectK2),
      connected(SqBK,SqR,DirectK2)),
  not( on(S,SqBK,obj(king,black)),
      legaldirection(king,DirectK3),
      connected(SqBK,SqE,DirectK3),
      on(S,SqE,empty),
      legaldirection(king,DirectK4),
      connected(SqE,SqBN,DirectK4)).

```

Figure 3.15: The final pattern in Prolog for a rook fork in the rook-king, knight-king end-game in chess. Derived from compiling a *Many-threat* proof with two threats, one where the rook captures the king, the other where the rook threatens to safely capture the knight.

pairs for the second threat pattern, we do not necessarily need to perform $p \times q$ pattern intersections. This is because the pattern/action pairs can be partitioned into equivalence classes with respect to the type of object moved. Only like object pairs need be intersected, leading to a block diagonal structure to the intersection matrix and a reduction in calls to *Pattern-Intersection*.

The final goal pattern derived from this compilation step is given in Figure 3.15. Three exceptions are included. The principal conjunction describes the white rook at the beginning of two openlines, both along directions that are legal for a rook. The first openline terminates in a location occupied by the black king.

The second openline terminates in a location occupied by the black knight. The first exception arises from the original **safe-capture-knight1** pattern. The second exception constrains the black king not to be adjacent to the white rook. This arises from the successful intersection marked (1) in Figure 3.14 which demonstrates that a rook capture by the king makes *both* threats false. Since such moves by black must not be available in the goal pattern, the condition is included as an exception. The third exception constrains the black king to be at least two steps away from the black knight. This arises from a second successful intersection marked (2) in Figure 3.14 which demonstrates that a king move to safety which adds protection to the knight makes *both* threats false.

3.2.2 When WIN is to Play

The compilation method when *WIN* is to play follows directly from the definition of optimal win, given in Section 2.4.2 and reproduced below. Recall that $\hat{\nu}_i^{G_{d_i}}(S)$ denotes a previously compiled winning pattern and $\nu^{G_{d_j}}(S)$ denotes a previously compiled losing pattern.

$$\begin{aligned}
 \text{achieve}(\lambda s.G(s), S, \text{WIN}, i, Op) \Leftarrow & \\
 \exists Op, o(S, \text{WIN}, Op), \text{make-true}(\lambda s.\nu_j^{G_{d_j}}(s), Op, S), i = d_j + 1 & \\
 \wedge \neg \hat{\nu}_1^{G_{d_1}}(S), d_1 < d_j & \\
 \wedge \neg \hat{\nu}_2^{G_{d_2}}(S), d_2 < d_j & \\
 \wedge \dots & \\
 \wedge \neg \hat{\nu}_m^{G_{d_m}}(S), d_m < d_j &
 \end{aligned}$$

To compile this proof, the first step is to compile the *make-true* relations for each losing goal pattern as described in Section 3.1.2, Page 47. This leads to a set of *make-true* pattern/action pairs, where each pair is represented as $\langle \nu_j^{MT}(S), Op_j^{MT} \rangle$. The next step is to ensure that none of the previously compiled winning patterns apply. Logically this is expressed as conjoining all $\hat{\nu}_i^{G_{d_i}}(S)$ to each $\nu_j^{MT}(S)$. However, procedurally we would like to produce the simplest possible pattern. This is achieved, as previously discussed, by *incorporating* these negated pattern constraints

through repeated calls to the procedure *Pattern-Difference* defined in Section 3.1.3 on Page 58. If during this process the pattern becomes empty then the process is terminated for that candidate pattern.

Chapter 4

Analysis of the Compiler

In this chapter we analyze the compiler. We first consider whether the compiler is *sound*; is the compiler guaranteed to always produce correct knowledge? We find that the compiler is sound, and therefore any rule produced will be behaviorally equivalent to the initial search process. We next consider whether the compiler is *complete*; will the compiler cover all the problem space with compiled rules? We find that the compiler is not complete: there are rules that cover problem instances which the compiler cannot generate. In addition, the compiler is not guaranteed to terminate.

We finally consider the computational complexity of the compiler and the anticipated complexity of using the compiled knowledge. This analysis identifies a potential problem with the compiler and suggests a solution. We find that the compiler, in the worse case, requires at least an exponential number of calls of a procedure that is NP-complete. We conclude this chapter by identifying the causes of these problems and briefly propose a solution which is fully explored in the next chapter. The solution is based on employing a *geometric representation* of the problem state that allows *constraint incorporation* during computation.

4.1 Soundness

Here we prove the compilation process is sound. Since the focus is on correctness, we ignore issues of optimality. To prove soundness we show that the space of influence proofs is equivalent to the original min/max proof. The soundness proof is inductive on the depth of the influence proof.

When the depth is 0, both the min/max proof and the influence proofs define goal achievement using the user provided termination patterns. Hence, influence proofs are equivalent to the min/max proof when $i = 0$.

To perform the inductive step, we must show that if this equivalence exists at depth $i - 1$, it also exists at depth i . Since we are considering counter-planning where two players alternate, the proof must show equivalence for both cases when *WIN* and *LOSS* is to play.

The proof when *WIN* is to play is trivial and follows directly from the introduction of influence proofs for *WIN* given in Section 2.4.2. The proof for *LOSS* is more complex. Let us assume that at depth $i - 1$ there are n winning goal patterns $\nu_1(S), \nu_2(S), \dots, \nu_n(S)$ (note the subscripts denoting a solution depth and the superscripts denoting the goal G are dropped to simplify the notation). Then the min/max proof for goal achievement at depth i is

$$\begin{aligned} \text{achieve}(G(s), S, \text{LOSS}, n) \Leftarrow \\ \forall Op \in o(S, \text{LOSS}), \nu_1(\text{do}(Op, S)) \vee \nu_2(\text{do}(Op, S)) \vee \dots \vee \nu_n(\text{do}(Op, S)), \end{aligned}$$

which simply states that S is a loss if all available operators lead to any of the win goal patterns. Here we show that this proof is equivalent to the influence proof given below, by a series of equivalence transformations. Recall that *LOSS* influence proofs are partitioned into three types, *no-threat*, *one-threat* and *many-threat*. Below, we apply a series of transformations to each kind of proof, then show that the combined result is equivalent to the min/max proof.

We begin the equivalence proof with the *no-threat* axiom, reproduced below:

$$\begin{aligned}
\text{achieve}(\lambda s.G(s), S, \text{LOSS}, i) \Leftarrow & \\
& \neg\nu_1(S) \wedge \neg\nu_2(S) \wedge \dots \wedge \neg\nu_n(S) \\
\wedge \forall Op, o(S, \text{LOSS}, Op), & [\text{make-true}(\lambda s.\nu_1(s), Op, S) \\
& \vee \text{make-true}(\lambda s.\nu_2(s), Op, S) \\
& \vee \dots \\
& \vee \text{make-true}(\lambda s.\nu_n(s), Op, S)].
\end{aligned}$$

Let us expand the *make-true* influence relations according to their definitions given in Table 2.1(1). Factoring out the operator generator $o(S, \text{LOSS}, Op)$ produces:

$$\begin{aligned}
\text{achieve}(\lambda s.G(s), S, \text{LOSS}, i) \Leftarrow & \\
& \neg\nu_1(S) \wedge \neg\nu_2(S) \wedge \dots \wedge \neg\nu_n(S) \\
\wedge \forall Op, o(S, \text{LOSS}, Op), & [\neg\nu_1(S) \wedge \nu_1(\text{do}(Op, S)) \\
& \vee \neg\nu_2(S) \wedge \nu_2(\text{do}(Op, S)) \\
& \vee \dots \\
& \vee \neg\nu_n(S) \wedge \nu_n(\text{do}(Op, S))].
\end{aligned}$$

Distributing the conjunction over the disjunction and eliminating redundancy produces the simplified *no-threat* axiom:

$$\begin{aligned}
(1) \quad \text{achieve}(\lambda s.G(s), S, \text{LOSS}, i) \Leftarrow & \\
& \neg\nu_1(S) \wedge \neg\nu_2(S) \wedge \dots \wedge \neg\nu_n(S) \\
& \wedge \forall Op, o(S, \text{LOSS}, Op), [\nu_1(\text{do}(Op, S)) \vee \nu_2(\text{do}(Op, S)) \vee \dots \vee \nu_n(\text{do}(Op, S))].
\end{aligned}$$

We now turn our attention to the *one-threat* axioms, reproduced below. Note there are n such axioms, given n winning goal patterns

$$\begin{aligned}
\text{achieve}(\lambda s.G(s), S, \text{LOSS}, i) \Leftarrow & \\
& \neg\nu_1(S) \wedge \neg\nu_2(S) \wedge \dots \wedge \nu_j(S) \wedge \dots \wedge \neg\nu_n(S) \\
\wedge \forall Op, o(S, \text{LOSS}, Op), & [\text{maintain-true}(\lambda s.\nu_j(s), Op, S) \\
& \vee \text{make-true}(\lambda s.\nu_1(s), Op, S) \\
& \vee \text{make-true}(\lambda s.\nu_2(s), Op, S) \\
& \vee \dots \\
& \vee \text{make-true}(\lambda s.\nu_{j-1}(s), Op, S) \\
& \vee \text{make-true}(\lambda s.\nu_{j+1}(s), Op, S) \\
& \vee \dots \\
& \vee \text{make-true}(\lambda s.\nu_n(s), Op, S)].
\end{aligned}$$

Let us expand the *make-true* and *maintain-true* influence relations according to their definitions given in Table 2.1(1) & (3). Factoring out the operator generator $o(S, LOSS, Op)$ produces:

$$\begin{aligned}
\text{achieve}(\lambda s.G(s), S, LOSS, i) \Leftarrow \\
& \neg\nu_1(S) \wedge \neg\nu_2(S) \wedge \dots \wedge \nu_j(S) \wedge \dots \wedge \neg\nu_n(S) \\
& \wedge \forall Op, o(S, LOSS, Op), \left[\begin{array}{l} \nu_j(S) \wedge \nu_j(\text{do}(Op, S)) \\ \vee \neg\nu_1(S) \wedge \nu_1(\text{do}(Op, S)) \\ \vee \neg\nu_2(S) \wedge \nu_2(\text{do}(Op, S)) \\ \vee \dots \\ \vee \neg\nu_{j-1}(S) \wedge \nu_{j-1}(\text{do}(Op, S)) \\ \vee \neg\nu_{j+1}(S) \wedge \nu_{j+1}(\text{do}(Op, S)) \\ \vee \dots \\ \vee \neg\nu_n(S) \wedge \nu_n(\text{do}(Op, S)) \end{array} \right].
\end{aligned}$$

Distributing the conjunction over the disjunction and eliminating redundancy produces the simplified *one-threat* axiom:

$$\begin{aligned}
(2) \quad \text{achieve}(\lambda s.G(s), S, LOSS, i) \Leftarrow \\
& \neg\nu_1(S) \wedge \neg\nu_2(S) \wedge \dots \wedge \nu_j(S) \wedge \dots \wedge \neg\nu_n(S) \\
& \wedge \forall Op, o(S, LOSS, Op), \left[\nu_1(\text{do}(Op, S)) \vee \nu_2(\text{do}(Op, S)) \vee \dots \vee \nu_n(\text{do}(Op, S)) \right]
\end{aligned}$$

Finally, we turn our attention to the last kind of influence axioms, *many-threat*. Here there are a total of $2^n - n - 1$ possible axioms, one for each possible subset of $k, 2 \leq k \leq n$, goal patterns. To simplify the analysis, we expand only the case where $k = 2$. Later we demonstrate how this case is generalized for $k > 2$. The axiom when $k = 2$ has the following form:

$$\begin{aligned}
\text{achieve}(\lambda s.G(s), S, LOSS, i) \Leftarrow \\
& \neg\nu_1(S) \wedge \neg\nu_2(S) \wedge \dots \wedge \nu_j(S) \wedge \dots \wedge \nu_k(S) \wedge \dots \wedge \neg\nu_n(S) \\
& \wedge \forall Op, o(S, LOSS, Op), \left[\begin{array}{l} \neg \text{make-false}(\lambda s.\nu_j(s), Op, S) \\ \vee \neg \text{make-false}(\lambda s.\nu_k(s), Op, S) \end{array} \right].
\end{aligned}$$

Let us expand the *make-false* influence relations according to their definitions given in Table 2.1(2). Factoring out the operator generator $o(S, LOSS, Op)$ produces:

$$\begin{aligned}
\text{achieve}(\lambda s.G(s), S, \text{LOSS}, i) \Leftarrow \\
& \neg\nu_1(S) \wedge \neg\nu_2(S) \wedge \dots \wedge \nu_j(S) \wedge \dots \wedge \nu_k(S) \wedge \dots \wedge \neg\nu_n(S) \\
& \wedge \forall Op, o(S, \text{LOSS}, Op), \left[\begin{array}{l} \neg[\nu_j(S) \wedge \nu_j(\text{do}(Op, S))] \\ \vee \neg[\nu_k(S) \wedge \nu_k(\text{do}(Op, S))] \end{array} \right].
\end{aligned}$$

Applying DeMorgan's rule to the negated conjunctions, distributing the conjunctions and eliminating those expression which contain both $\nu(S) \wedge \neg\nu(S)$ produces:

$$\begin{aligned}
\text{achieve}(\lambda s.G(s), S, \text{LOSS}, i) \Leftarrow \\
& \neg\nu_1(S) \wedge \neg\nu_2(S) \wedge \dots \wedge \nu_j(S) \wedge \dots \wedge \nu_k(S) \wedge \dots \wedge \neg\nu_n(S) \\
& \wedge \forall Op, o(S, \text{LOSS}, Op), \left[\nu_j(\text{do}(Op, S)) \vee \nu_k(\text{do}(Op, S)) \right].
\end{aligned}$$

This proof states that after the application of operators, at least one of the initial threat patterns will be true. However, it says nothing about the truth value of the other goal patterns after the operator has been applied. Since the disjunction is not closed, the operators available may make any of the none threat goal patterns true (i.e., not j or not k). Hence the above axiom can be equivalently written:

$$\begin{aligned}
(3) \quad \text{achieve}(\lambda s.G(s), S, \text{LOSS}, i) \Leftarrow \\
& \neg\nu_1(S) \wedge \neg\nu_2(S) \wedge \dots \wedge \nu_j(S) \wedge \dots \wedge \nu_k(S) \wedge \dots \wedge \neg\nu_n(S) \\
& \wedge \forall Op, o(S, \text{LOSS}, Op), \left[\nu_1(\text{do}(Op, S)) \vee \nu_2(\text{do}(Op, S)) \vee \dots \vee \nu_n(\text{do}(Op, S)) \right].
\end{aligned}$$

From the construction of this equivalent axiom, it is easy to see what will result when there are more than two threat patterns true initially. In the general case, the conjunction before quantification (which defines the initial truth setting of the goal patterns) will remain unchanged, while the expression following the operator quantification will reduce to the same as given above, $[\nu_1(\text{do}(Op, S)) \vee \nu_2(\text{do}(Op, S)) \vee \dots \vee \nu_n(\text{do}(Op, S))]$.

Combining expression (1), (2) and (3) above and factoring out the common expression $[\nu_1(\text{do}(Op, S)) \vee \nu_2(\text{do}(Op, S)) \vee \dots \vee \nu_n(\text{do}(Op, S))]$ leads to the following:

$$\begin{aligned}
\text{achieve}(\lambda s.G(s), S, \text{LOSS}, i) \Leftarrow \\
& \forall Op, o(S, \text{LOSS}, Op), \left[\nu_1(\text{do}(Op, S)) \vee \nu_2(\text{do}(Op, S)) \vee \dots \vee \nu_n(\text{do}(Op, S)) \right] \\
& \wedge \left[\neg\nu_1(S) \wedge \neg\nu_2(S) \wedge \dots \wedge \neg\nu_n(S) \right]
\end{aligned}$$

$$\begin{aligned}
& \vee \nu_1(S) \wedge \neg\nu_2(S) \wedge \dots \wedge \neg\nu_n(S) \\
& \vee \neg\nu_1(S) \wedge \nu_2(S) \wedge \dots \wedge \neg\nu_n(S) \\
& \vee \dots \\
& \vee \neg\nu_1(S) \wedge \neg\nu_2(S) \wedge \dots \wedge \nu_n(S) \\
& \vee \nu_1(S) \wedge \nu_2(S) \wedge \dots \wedge \neg\nu_n(S) \\
& \vee \nu_1(S) \wedge \neg\nu_2(S) \wedge \nu_3(S) \wedge \dots \wedge \neg\nu_n(S) \\
& \vee \nu_1(S) \wedge \neg\nu_2(S) \wedge \neg\nu_3(S) \wedge \dots \wedge \nu_n(S) \\
& \vee \dots \\
& \vee \nu_1(S) \wedge \nu_2(S) \wedge \nu_3(S) \wedge \dots \wedge \nu_n(S)].
\end{aligned}$$

The later bracketed expression represents all possible 2^n truth settings of the initial n goal patterns and can be factored into

$$\begin{aligned}
& [\nu_1(S) \vee \neg\nu_1(S)] \\
& \wedge [\nu_2(S) \vee \neg\nu_2(S)] \\
& \wedge \dots \\
& \wedge [\nu_n(S) \vee \neg\nu_n(S)],
\end{aligned}$$

which is true. Hence, for n given goal patterns, the conjunction of 1 *no-threat* axiom, n *one-threat* axioms and $2^n - 1 - n$ *many-threat* axioms are exactly equivalent to the original min/max proof. Therefore, the abstraction mechanism based on influence is *sound*.

4.2 Completeness

A *complete* compiler constructs a problem solver that can solve all possible problem instances from a give problem class. Here we demonstrate that this compiler is incomplete. There are two problems that lead to incompleteness:

- It is not possible to guarantee termination of the compiler.
- Some of the proof sentences cannot be compiled. Previously we identified three types of proof sentences: *no-threat*, *one-threat* and *many-threat*. It is not possible to compile both the *no-threat* and *one-threat* proofs because of the representation problem described in Section 3.2.1, on Page 75. Hence, the goal patterns may be incomplete. This incompleteness could be considered

minor, since the missing patterns account for only $n + 1$ proof sentence out of the complete set of 2^n . However, these proofs are essential for chess, since they account for many patterns of low depth. If they are not generated, all deeper proofs that need these patterns cannot be compiled.

4.3 Complexity Analysis

This section investigates the computational requirements of the compiler. We are interested in the requirements during compilation and during performance, when the compiled knowledge will be used. The complexity of the compilation process is determined by two factors: the number of proof sentences generated and the complexity of compiling them into patterns. The complexity of performance is determined by two similar factors: the number of patterns that are compiled and the time required to match a problem instance against the patterns.

4.3.1 Number of Proof Sentences Generated

We can determine the number of proof sentences that are generated during compilation by defining and solving simple recurrence relations that arise from the recursive definitions given in Section 2.2.2(6) and Section 2.4.2(9). Let $f(i)$ be the number of proof sentences for a proof depth of i . Since at depth $i + 1$ we can generate a new sentence from each possible subset of the previous sentences, we arrive at the following recurrence relation:

$$\begin{aligned} f(0) &= n \\ f(i) &= 2^{f(i-1)} \end{aligned}$$

Solving this recurrence relation leads to a function that grows much too fast. In fact, when we consider all the proof sentences of at most depth 4, we have 2^{256} , which is considerably more than the complete space of all likely chess positions ($10^{40} = 2^{132}$). On the face of it, the influence proofs do not appear to be providing much abstraction! However, this space is much smaller than it appears, since many

of these sentences will be either empty (i.e., they have no extension) or redundant and need not be generated. A sentence will be empty if, when compiled, it leads to a pattern that has no extension. To understand the effect of these empty sentences, we must review the form of the influence proofs given in Section 2.2.2. Let us assume that we have n winning patterns compiled from previous sentences, which are to be used to generate a new set of losing sentences. From the form of the influence proofs, we know that we have a total of 2^n distinct proof sentences, and hence, we have the potential to generate 2^n new goal patterns. Consider compiling a simple *many-threat* sentence which includes only two goal patterns, $\nu_i(S)$ and $\nu_j(S)$ assumed true initially (i.e., there are two current threats). From Section 3.2.1 (on Page 78) we know that to compile this sentence, we must take the intersection of these two patterns. Consider the effect on the space of proof sentences if this intersection operation returns an empty pattern. Clearly all other *many-threat* sentences that include $\nu_i(S)$ and $\nu_j(S)$ as threats will also compile to empty sentences. Hence, by detecting empty intersections of goal patterns, the large space of *many-threat* proofs can be considerably reduced. In general, if we detect a conjunction of k goal patterns that has an empty extension, we eliminate 2^{n-k} sentences from consideration.

These empty sentences reduce the growth of proofs from hyper-exponential. If we assume, for example, that all conjunctions with 3 or more non-negated patterns are empty, then the recurrence relation is as follows:

$$\begin{aligned} f(0) &= n \\ f(i) &= 1 + f(i-1) + \frac{f(i-1)^2}{2} \end{aligned}$$

Solving this leads to $f(i) = O(n^{2^i})$, which is much smaller than the original function, but still exponential. Empirical results given in Chapter 8 confirm that the space is even sparser than this.

4.3.2 Complexity of Compilation

Given that we know how many proof sentences we need to compile, we now consider the complexity of actually compiling the proof sentences into patterns. To understand the complexity of this process it is useful to review the compiler algorithm given in Section 3.2. Computational resource requirements are dominated by two computations: the intersection of patterns during proof compilation and the compiling of influence relations. In this section we focus on the intersection computation, since it is easily analyzed and provides an accurate bound on resource requirements. We consider a function *Pattern-Intersection*(ν_1, ν_2) defined in Section 3.1.3 (on Page 58), which is called at least $O(f(i)^2)$ times for each depth i .

Given that pattern ν_1 has p exceptions and ν_2 has q exceptions, previous analysis showed that to compute *Pattern-Intersection* we must call the function *Simplify* $O(p+q)$ times, where *Simplify* takes two simple conjunctions of operational literals and returns either empty, if the conjunction has no extension, or a simplified conjunction. This analysis tells us that the number of calls to *Simplify* for each pattern intersection is a linear function of the complexity of both patterns. We also know that in the worst case—when none of the pattern intersections are detected as empty—the number of patterns grows potentially hyper-exponentially with depth i , and the complexity of each pattern grows linearly in i .

We still have not considered the computational complexity of *simplify*(c^1, c^2) itself. Here we are assuming that both c^j are simple conjunctions of operational literals. Hence, the analysis introduced in [Minton 88a] and [Tambe, Newell and Rosenbloom 90] is relevant, for it describes the complexity of matching a conjunctive rule against a conjunctive state description. In this approach the following assumption concerning the form of the operational literals is made:

Literals encode *arbitrary one-to-many relations*, which share variables in the conjunction.

Given this assumption, we can model *simplify* over two conjunctions as a reduc-

tion from graph isomorphism over two graphs, where the nodes in the graphs are variables and the edges are the mappings encoded by the literals. Hence, in the worst case, *Simplify* is NP-complete. In [Tambe, Newell and Rosenbloom 90], the complexity of match is given as $O(b^l)$ where l is the number of literals in the conjunction and b the cardinality of the one-to-many mappings.

4.3.3 Complexity of Performance

The *utility problem* [Minton 88a] concerns the difficulty of ensuring that compilation actually improves problem solving performance. We want the compiled problem solver to return answers quicker than the original search-intensive problem solver. Such improved performance is not guaranteed, because the compiler replaces domain search with pattern matching, and pattern matching can involve expensive search.

The analysis of *Pattern-Intersection* above applies equally well to the problem of matching learned patterns against problem instances. Hence, match time has the potential to grow exponentially with the length of the patterns generated. The principal problem then, is the growth in the length of patterns that occurs in our method as the patterns come to describe goal achievement solutions of longer and longer length. This problem was reported in [Tambe, Newell and Rosenbloom 90], where it was demonstrated that in EBL approaches, the length of the conjunctive rules tends to grow with the depth of the proof, leading to useless rules with “big footprints.”

This trend to patterns with long conjunctions can be seen in the example illustrated in Figure 3.15 which shows the Prolog rule describing situations lost in 2 ply through a fork tactic by a rook. As part of this pattern, the following conjunction is included as an exception:

```
on(S,SqBK,obj(king,black)),
legaldirection(king,DirectK3),
connected(SqBK,SqE,DirectK3),
on(S,SqE,empty),
legaldirection(king,DirectK4),
```

```
connected(SqE,SqBN,DirectK4)),
on(S,SqBN,obj(knight,black)).
```

This conjunction describes a path of length two by the black king to the black knight. To match this conjunction against a state description according to the matching model introduced by [Tambe, Newell and Rosenbloom 90], requires *an exhaustive search from the king location for two steps in all directions*. In other words, we must generate 64 candidate locations and test whether each one is occupied by the black knight. Hence, matching such a short rule is very inefficient. Moreover, as this pattern is used to generate new patterns describing solutions of longer length, the path of the king will be extended, leading to an exponential growth in match time.

To summarize, the performance is limited by the proliferation of patterns that involve long conjunctions. Since the match time grows exponentially with this length, it is critical to the success of the method that the conjunctions be kept small.

4.4 Summary

The compiler was analyzed to determine its computational requirements and to determine whether it is sound or complete. The analysis shows that the approach can compile counter-planning into pattern-action rules that will be behaviorally equivalent to the initial search procedure. Hence, the approach offers a means to learn *correct* knowledge for improving the performance of counter-planning systems.

However, the analysis also uncovered some problems with this approach that make it impractical:

- The compiler is *incomplete* in that it may never compute a rule set that covers every possible problem instance in a domain.
- The compiler computation in the worst case requires solving an NP-complete

problem a hyper-exponential number of times.

In the following chapter the cause of these problems is explored in detail and a solution is presented. Such a solution must provide:

- A representation of pattern/action pairs that allows the coverage of sets of actions to be determined. This will eliminate the problem with incompleteness by enabling the compilation of *no-threat* and *one-threat* proofs.
- An efficient means to exploit the sparseness of the space. In other words, we must incorporate the fact that the intersection of two patterns is empty into the generator of pattern pairs during proof generation and compilation. This will contribute to reducing the hyper-exponential growth in patterns.
- A representation for the patterns that would make *Pattern-Intersection* much more efficient. This will overcome the problem with relying on a provably exponential process during compilation.
- A representation for the patterns that would make matching efficient. This would overcome the problem with producing patterns that could potentially take longer to match than the original state-space search.

Chapter 5

A Theory of Geometry for Efficient Abstractions

Hector Levesque in his 1985 Computers and Thought Award emphasized the effectiveness of spatial representations over simple logical representations [Levesque 86]:

“[One observation] is our obvious success in problem-solving situations where we can rely on *visualization*, such as in the case of *geometry*, or reasoning about sets using Venn diagrams. This is to be contrasted with the great difficulty we have with certain kinds of word puzzles, or solving purely logical tautologies. A first explanation might state that we have been primed by evolution to deal effectively with visual information, as opposed to linguistic information, a relative late-comer. But perhaps a better explanation is that visual information is *inherently* more tractable than unrestricted linguistic information, and all that evolution has done is taught us to exploit this fact.”

In this chapter and the next we present an application that provides support for Levesque’s argument: geometric representations are inherently more tractable than logical representations. This chapter introduces a geometric representation for goal patterns and action sets, and illustrates the compilation algorithms working with the new representations. The next chapter demonstrates that this geometrical view overcomes the previously identified problems with the compilation approach.

In the previous chapters we have been employing a logical representation of the problem space where goal patterns are represented as conjunctions of literals where each literal encodes some predicate or relation. This powerful representation could be used to represent arbitrary problem spaces such as word-puzzles, engineering designs, or planning problems. However, in the domains we are interested in, we do not have an arbitrary problem space. Rather, the problem space describes situations that are composed of *objects* arranged in *two dimensional space* and operators that move the objects between locations in two dimensional space. Hence, we can employ a specialized *geometric* representation which exploits these characteristics of the problem space.

To serve as an introduction to benefits of taking a geometric view of the problem space, let us consider the problem of matching a logical representation of a simple chess goal pattern. Below we give the logical definition of a pattern that arises in chess endings which involve bishops. The pattern describes the situation where a bishop can move to attack the opponent's king:

```

on(S,SqWB,obj(bishop,white)),
legaldirection(bishop,D1),
openline(S,SqWB,SqE,D1),
on(S,SqE,empty),
legaldirection(bishop,D2),
openline(S,SqE,SqBK,D2),
on(S,SqBK,obj(king,black)).

```

This pattern describes a white bishop at location SqWB that can move in direction D1 to location SqE, where there is an openline in a legal direction D2 for the bishop to a location SqBK occupied by the black king. In the logical interpretation, variables such as SqWB and D2 take arbitrary terms as values and literals such as legaldirection(bishop,D1) and openline(S,SqWB,SqE,D1) encode arbitrary relations over unstructured sets. Consider the problem of determining whether a given problem instance is covered by this pattern. This process will involve first binding the locations for the white bishop to SqWB, and then *generating* candidate

location for the black king which lie at the end of all possible compositions of two openlines. Finally, each candidate black king location is *tested* to determine if it is occupied by the black king. This kind of exhaustive generate and test is the only option available with logical representations, where the relations encode arbitrary relations over unstructured sets¹.

When we take into account the fact that the relations are geometric and not arbitrary and that the values of the variables are not unstructured sets but encode locations and spatial vectors, the exhaustive generate and test can be avoided. For example, if the given problem instance included the white bishop in the upper right of the board and the black king in the lower left, then it makes no sense for the matching process to be exploring paths further in the upper left corner, *away* from the king. Neither does it make sense for the matching process to be doing any search at all if the king is on a black square and the bishop is on a white square. These kinds of efficiencies are possible because of the underlying geometric structure of the problem.

By taking a geometric view, we can perform *test incorporation* to eliminate or reduce wasteful generate and test. Test incorporation seeks to improve computation that can be modeled as a generator of candidate solutions followed by a test t , of candidate solutions. Test incorporation improves performance by first *factoring* the test t into a conjunction of sub-tests $t_1 \wedge t_2 \wedge \dots \wedge t_n$, which all must be passed by the candidate solution. Next, the generator is modified to *incorporate* the sub-tests, t_i , such that the new generator only produces candidate solutions which pass t_i . Ideally, all the sub-tests can be incorporated into the generator, and generate and test is eliminated—a satisfactory solution is produced directly. The method has been successfully used in a variety of knowledge compilation applications. In [Smith and Pressburger 88], information from the test of a satisfactory

¹Better orderings of the literals can improve performance some what, but they can not eliminate the generate-and-test behavior (see [Smith and Genesereth 85])

solution is incorporated into the search process that constructs the solution. In [Braudaway and Tong 89], constraints on a suitable design are incorporated into a generator of designs.

The application of test incorporation to the bishop-king pattern matching example first requires additional geometric information about generators and tests, or in this context, the variables and literals in the pattern. Information about variables includes geometric encodings of their values; for example location variables can be represented as $\langle X, Y \rangle$ coordinates, and direction variables can be represented as two dimensional vectors $\langle \Delta X, \Delta Y \rangle$. Information about literals includes linear constraints on the values taken by the relations encoded; for example the *openline* literal encodes a relation over two locations, $\langle X_F, Y_F \rangle$ and $\langle X_T, Y_T \rangle$ and a direction $\langle \Delta X, \Delta Y \rangle$, such that $\langle X_T, Y_T \rangle = \langle X_F, Y_F \rangle + l \cdot \langle \Delta X, \Delta Y \rangle, 1 \leq l \leq 7$. This additional information allows us to produce a set of linear constraints that must hold between the generated and tested values, which can then be simplified. This allows us to *solve* for the tested values rather than have to generate and test them.

For example, solving the resulting linear equations in the bishop-king pattern for the intermediate location SqE results in $X_{SqE} = ((X_{SqWB} \pm X_{SqBK}) + (Y_{SqWB} \pm Y_{SqBK}))/2$ and a similar expression for Y_{SqE} . Since both coordinates of SqE must be integers, these expressions imply that the numerator must be *even*. These expressions then distinguish the black squares from the white squares on a chess board and impose the constraint that the king and bishop must be on squares of the same color. Thus, the geometric view allows us to perform test incorporation such that generate and test is completely eliminated when the two objects are on different color squares.

This simple example of pattern matching has served to demonstrate the value of test incorporation for eliminating search, and illustrated how test incorporation is enabled by exploiting a geometric representation of the problem space. This chapter shows the following:

- Geometry enables test incorporation within compiled influence relations. Here constraints from the pattern are incorporated into the generator of actions in the pattern/action pairs of compiled influence relations. This enables set operations over the set of operators, such as difference and intersection, which are needed to compile the *no-threat* and *one-threat* proofs (as described in Section 3.2.1). Hence, this incorporation eliminates the problem of incompleteness demonstrated in Section 4.2.
- Geometry enables test incorporation during the generation and compilation of new patterns. Here constraints from the set of already compiled loss and win patterns are incorporated into the generator of new influence proofs/patterns. Recall that compilation of both win and loss proofs involves intersecting new patterns with all previously compiled patterns: To generate a new optimal win pattern we must intersect the newly formed non-optimal pattern with all previous win patterns to ensure optimality; to generate a new loss pattern we must consider all possible subsets of previous win patterns. Without incorporation, we must *generate* all candidate patterns (from the set of previously compiled patterns) and *test* whether the intersection with our new pattern is non-empty. Test incorporation allows us to incorporate the test of empty intersection into the generator of candidate patterns. Ideally, we want the generator to produce only those patterns that have a non-empty intersection. This is achieved by employing a geometric representation of the patterns where each object in the pattern is constrained to be in a rectangular region on the board. Patterns are stored in a database indexed by the rectangular regions. Finding all intersecting patterns reduces to *indexing* into this database based on consistency between the regions of the new pattern and the patterns in the database.
- Geometry enables test incorporation during the matching of patterns against problem instances. We have already illustrated a simple example of this kind

ν	$::= C_0 \wedge \neg C_1 \wedge \neg C_2 \wedge \dots \wedge C_l$
C_i	$::= [Objs, RObjs, Emps, Olines, LCons]$
$Objs$	$::= \{\langle Loc_1, Obj_1 \rangle, \langle Loc_2, Obj_2 \rangle, \dots, \langle Loc_n, Obj_n \rangle\}$
$RObjs$	$::= \{\langle Loc_1, R_1 \rangle, \langle Loc_2, R_2 \rangle, \dots, \langle Loc_n, R_n \rangle\}$
$Emps$	$::= \{Loc_{n+1}, Loc_{n+2}, \dots, Loc_m\}$
$Olines$	$::= \{\langle Loc_{m+1}, Loc_{m+2}, D_1 \rangle, \dots, \langle Loc_{m+2.p-1}, Loc_{m+2.p}, D_p \rangle\}$
R_i	$::= \langle Loc_1, Loc_2, Loc_3, Loc_4 \rangle$
Loc_i	$::= \langle Cor_{X_i}, Cor_{Y_i} \rangle$
$LCons$	$::= \{C_{o_1}, C_{o_2}, \dots, C_k\}$
Co_i	$::= Cons Scons Ocons$
$Cons$	$::= Cor_i = C+$
$Scons$	$::= Cor_i = Cor_j + C\pm$
$Ocons$	$::= Cor_i = Cor_j + l_k \times C\pm, C+ \leq l_k \leq C+$
D_i	$::= \langle C\pm, C\pm \rangle$
$C+$	$::= 0 1 2 \dots 8$
$C\pm$	$::= -8 -7 \dots 0 \dots 7 8$

Table 5.1: Representation of geometric patterns

of incorporation.

The rest of the chapter is organized as follows. First, we introduce the geometric representation of patterns and operators. Second, we demonstrate how the previously given algorithms for compiling influence relations are adapted to employ geometric representations. Third and finally, we describe the pattern intersection and difference operations using geometrically represented patterns.

5.1 A Geometric Representation of Patterns

In the previous chapters, a pattern has been represented as a single conjunction of operational literals with a finite number of conjunctive exceptions. This section introduces an equivalent representation for patterns that uses geometric primitives. The representation is equivalent in that a single pattern represented logically can be translated into a disjunction (usually small) of geometrically represented patterns that has exactly the same extension.

$[\{ \langle \langle X_{WR}, Y_{WR} \rangle, \text{obj}(\text{white}, \text{rook}) \rangle, \langle \langle X_{WK}, Y_{WK} \rangle, \text{obj}(\text{white}, \text{king}) \rangle, \langle \langle X_{BK}, Y_{BK} \rangle, \text{obj}(\text{black}, \text{king}) \rangle \} ,$	}	Objects
$\{ \langle \langle X_{WR}, Y_{WR} \rangle, \langle \langle 1, 1 \rangle, \langle 1, 8 \rangle, \langle 7, 1 \rangle, \langle 7, 7 \rangle \rangle \rangle, \langle \langle X_{WK}, Y_{WK} \rangle, \langle \langle 1, 1 \rangle, \langle 1, 8 \rangle, \langle 8, 1 \rangle, \langle 8, 8 \rangle \rangle \rangle, \langle \langle X_{BK}, Y_{BK} \rangle, \langle \langle 2, 1 \rangle, \langle 2, 8 \rangle, \langle 8, 1 \rangle, \langle 8, 8 \rangle \rangle \rangle \} ,$	}	Rectangular object regions
$\{ \}$	}	Empty locations
$\{ \langle \langle X_{WR}, Y_{WR} \rangle, \langle X_{BK}, Y_{BK} \rangle, \langle 0, 1 \rangle \rangle \}$	}	Openlines,
$\{ X_{BK} = (X_{WR} + l, 1 \leq l \leq 7), Y_{BK} = Y_{WR} \}$	}	Constraints

Figure 5.1: A geometric representation of the pattern **rook-takes-king**. See Figure 3.8 for the logical and graphical representation. Note the components of the pattern are marked on the right

The geometric representation is based on a simplification of one introduced in [Brooks 81], where two and three dimensional physical scenes are represented as sets of linear and non-linear constraints among points. In this case the representations are simplified to model only in two dimensions and involve only linear constraints, because the objects are always aligned with a reference grid (in this case the squares of the playing board).

Patterns are represented by a principal conjunction and a finite number of conjunctive exceptions as before, only this time the conjunctions are not of operational literals. Where before a conjunction of relations such as *openline* and *connected* was used to describe the constraints among the locations of objects, *openlines* and *empty squares*, here we use a set of linear constraints. More precisely, a pattern, ν , is defined in Table 5.1. Each conjunctive expression comprises *Objs*, a list of the n objects and their location variables in the pattern; *RObjs*, a list of rectangular regions, one for each object; *Emps*, a list of all the empty locations in the pattern; *Olines* a list of all the openlines in the pattern; and finally, *LCons* a list of linear constraints among the locations of the objects, corners of the regions, empty squares and openlines. Since there are many equivalent sets of constraints among variables,

$$\begin{aligned}
& [\{ \langle \langle X_{WR}, Y_{WR} \rangle, \text{obj}(\text{white}, \text{rook}) \rangle, \\
& \quad \langle \langle X_{WK}, Y_{WK} \rangle, \text{obj}(\text{white}, \text{king}) \rangle, \\
& \quad \langle \langle X_{BN}, Y_{BN} \rangle, \text{obj}(\text{black}, \text{knight}) \rangle \}, \\
& \{ \langle \langle X_{WR}, Y_{WR} \rangle, \langle \langle 1, 1 \rangle, \langle 1, 8 \rangle, \langle 7, 1 \rangle, \langle 7, 7 \rangle \rangle \rangle, \\
& \quad \langle \langle X_{WK}, Y_{WK} \rangle, \langle \langle 1, 1 \rangle, \langle 1, 8 \rangle, \langle 8, 1 \rangle, \langle 8, 8 \rangle \rangle \rangle, \\
& \quad \langle \langle X_{BN}, Y_{BN} \rangle, \langle \langle 2, 1 \rangle, \langle 2, 8 \rangle, \langle 8, 1 \rangle, \langle 8, 8 \rangle \rangle \rangle \}, \\
& \{ \} \\
& \{ \langle \langle X_{WR}, Y_{WR} \rangle, \langle X_{BN}, Y_{BN} \rangle, \langle 0, 1 \rangle \rangle \} \\
& \{ X_{BN} = (X_{WR} + l, 1 \leq l \leq 7), \\
& \quad Y_{BN} = Y_{WK} \}] \\
& \neg [\{ \langle \langle X_{BK}, Y_{BK} \rangle, \text{obj}(\text{black}, \text{king}) \rangle, \\
& \quad \langle \langle X_{BN}, Y_{BN} \rangle, \text{obj}(\text{black}, \text{knight}) \rangle \}, \\
& \{ \langle \langle X_{BK}, Y_{BK} \rangle, \langle \langle X_{ll}, Y_{ll} \rangle, \langle X_{ul}, Y_{ul} \rangle, \langle X_{lr}, Y_{lr} \rangle, \langle X_{ur}, Y_{ur} \rangle \rangle \rangle \}, \\
& \{ \} \\
& \{ \} \\
& \{ X_{ll} = X_{BN} - 1, Y_{ll} = Y_{BN} - 1 \\
& \quad X_{ul} = X_{BN} - 1, Y_{ul} = Y_{BN} + 1 \\
& \quad X_{lr} = X_{BN} + 1, Y_{lr} = Y_{BN} - 1 \\
& \quad X_{ur} = X_{BN} + 1, Y_{ur} = Y_{BN} + 1 \}]
\end{aligned}$$

Figure 5.2: A geometric representation of the pattern `rook-takes-knight1`, illustrated in graphical form in Figure 3.9

the method attempts to maintain the pattern constraints in a canonical form. A total order is chosen for the objects and constraints for an object higher in the order are always expressed in terms of variables lower in the order. In addition, redundant internal variables—those whose value has been determined—are eliminated. There are three kinds of constraints: *Cons*, the simplest, where a coordinate of a location equals a positive constant; *Scons*, where a coordinate of a location equals another coordinate plus some constant; and *Ocons*, where a coordinate of a location equals another coordinate plus some multiple of a positive constant, where the multiplier (referred to as an internal variable) is constrained to be positive.

The three kinds of linear constraints among locations arise from the geometric relations of the original logical representation: `connected` and `openline`. Constraints of the form *Scons* are implied by the `connected` relation which encodes a relation over two locations, $\langle X_F, Y_F \rangle$ and $\langle X_T, Y_T \rangle$, and a direction $\langle \Delta X, \Delta Y \rangle$ such that

$X_T = X_F + \Delta X$, and $Y_T = Y_F + \Delta Y$. Constraints of the form *Ocons* are implied by the *openline* relation which encodes a relation over two locations, $\langle X_F, Y_F \rangle$ and $\langle X_T, Y_T \rangle$, and a direction $\langle \Delta X, \Delta Y \rangle$ such that $X_T = X_F + l \times \Delta X$, and $Y_T = Y_F + l \times \Delta Y$, $1 \leq l \leq 7$. Note that the openlines and corresponding constraints in a pattern always have directions that are instantiated with constants. This specialization simplifies the geometrical reasoning but results in one-to-many mapping between logically represented patterns and geometrically represented patterns.

A geometric representation allows symmetries to be exploited. Given a chess board without pawns, there are 4 degrees of rotational symmetry and one degree of reflexive symmetry. These symmetries were exploited in constructing the extensional databases, referred to in Section 1.1.2, to cut the size of the database by a factor of eight. With an abstract geometric representation as we have here, reflexive symmetry is difficult to exploit. However, rotational symmetry is exploited to cut the size of the abstract database by a factor of four. Two simple examples of chess patterns described geometrically are given for the pattern *rook-takes-king* in Figure 5.1 (see Figure 3.8 for the original logical representation) and *rook-takes-knight1* in Figure 5.2 (see Figure 3.9 for the original logical representation).

5.2 Geometric Influence Relations

In this section we define new influence compilation algorithms that employ the geometric representations. We first define a new representation for operator sets that overcomes the problems with incompleteness previously described. We then define the compilation algorithms as an adaptation of the algorithms for logic defined in Chapter 3.

5.2.1 Representation of Operator Sets

In Table 3.1 we defined the form of a compiled influence relation to be a set of pattern/action pairs where the action is represented as a term, whose variables, such

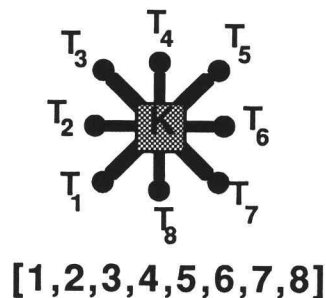
$$\langle [\{ \{ \langle X_{WR}, Y_{WR} \rangle, \text{obj}(\text{white}, \text{rook}) \rangle, \\
\langle X_{WK}, Y_{WK} \rangle, \text{obj}(\text{white}, \text{king}) \rangle, \\
\langle X_{BK}, Y_{BK} \rangle, \text{obj}(\text{black}, \text{king}) \rangle \}, \\
\{ \langle X_{WR}, Y_{WR} \rangle, \langle \langle 1, 1 \rangle, \langle 1, 5 \rangle, \langle 7, 1 \rangle, \langle 7, 5 \rangle \rangle \}, \\
\langle \langle X_{WK}, Y_{WK} \rangle, \langle \langle 1, 1 \rangle, \langle 1, 8 \rangle, \langle 8, 1 \rangle, \langle 8, 8 \rangle \rangle \}, \\
\langle \langle X_{BK}, Y_{BK} \rangle, \langle \langle 3, 1 \rangle, \langle 3, 8 \rangle, \langle 7, 1 \rangle, \langle 7, 8 \rangle \rangle \}, \\
\{ \langle X_1, Y_1 \rangle \} \\
\{ \langle \langle X_{WR}, Y_{WR} \rangle, \langle X_{BK}, Y_{BK} \rangle, \langle 0, 1 \rangle \rangle \} \\
\{ X_{BK} = X_{WR} + l, 1 \leq 2 \leq 6, Y_{BK} = Y_{WR}, \\
X_1 = X_{BK}, Y_1 = Y_{BK} \}], \\
[1, 0, 1, 1, 1, 0, 1, 1] \rangle$$


Figure 5.3: Geometric representation of $make\text{-}false(\text{rook-takes-king}(s), Op, S)$

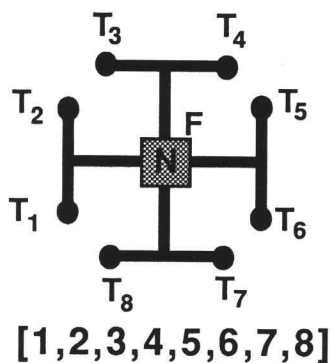
as TSq (the destination square) and FSq (the originating square), are constrained by the corresponding pattern. This representation of operators was shown to be inadequate for compiling *no-threat* and *one-threat* proofs in Section 3.2.1 (page 75). The inadequacy arises because the term representation describes the set of operators *intentionally*, as a set of constraints in the pattern, which prevent the intersection and difference operations over the operator set that are needed by the compilation process. This section introduces a new representation for compiled influence relations where the patterns are described geometrically and the operator set is made *explicit*, thereby enabling the required set operations for the compilation of *no-threat* and *one-threat* proofs.

An example of the new representation of compiled influence relations is given in Figure 5.3 for the king moves that *make-false* the simple termination pattern *rook-takes-king*. Note that the pattern is represented geometrically. The operator set that make this pattern false is represented as the bit vector following the pattern. This vector is an example of an *operator subspace*. An operator subspace is a representation designed to describe generic operators that move objects in quantized space from one location to a finite number of other locations. Figure 5.4 illustrates 3 examples of operator subspaces for chess. In the case of a king, the vector is eight units long, one for each possible direction the king can move. A 1 in the vector

$$\begin{aligned}
& \{ \{ X_{T1} = X_F - 1, Y_{T1} = Y_F - 1 \}, \\
& \{ X_{T2} = X_F - 1, Y_{T2} = Y_F \}, \\
& \{ X_{T3} = X_F - 1, Y_{T3} = Y_F + 1 \}, \\
& \{ X_{T4} = X_F, Y_{T4} = Y_F + 1 \}, \\
& \{ X_{T5} = X_F + 1, Y_{T5} = Y_F + 1 \}, \\
& \{ X_{T6} = X_F + 1, Y_{T6} = Y_F \}, \\
& \{ X_{T7} = X_F + 1, Y_{T7} = Y_F - 1 \}, \\
& \{ X_{T8} = X_F, Y_{T8} = Y_F - 1 \} \}
\end{aligned}$$



$$\begin{aligned}
& \{ \{ X_{T1} = X_F - 2, Y_{T1} = Y_F - 1 \} \\
& \{ X_{T2} = X_F - 2, Y_{T2} = Y_F + 1 \} \\
& \{ X_{T3} = X_F - 1, Y_{T3} = Y_F + 2 \} \\
& \{ X_{T4} = X_F + 1, Y_{T4} = Y_F + 2 \} \\
& \{ X_{T5} = X_F + 2, Y_{T5} = Y_F + 1 \} \\
& \{ X_{T6} = X_F + 2, Y_{T6} = Y_F - 1 \} \\
& \{ X_{T7} = X_F + 1, Y_{T7} = Y_F - 2 \} \\
& \{ X_{T8} = X_F - 1, Y_{T8} = Y_F - 2 \} \}
\end{aligned}$$



$$\begin{aligned}
& \{ \{ X_{T1} = X_F - l_1, Y_{T1} = Y_F, 1 \leq l_1 \leq 7 \} \\
& \{ X_{T2} = X_F, Y_{T2} = Y_F + l_2, 1 \leq l_2 \leq 7 \} \\
& \{ X_{T3} = X_F + l_3, Y_{T3} = Y_F, 1 \leq l_3 \leq 7 \} \\
& \{ X_{T4} = X_F, Y_{T4} = Y_F - l_4, 1 \leq l_4 \leq 7 \} \}
\end{aligned}$$

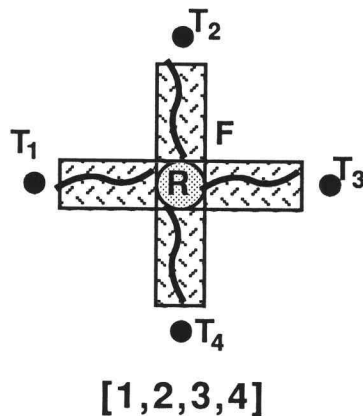


Figure 5.4: Operator *Sub-spaces* and linear constraint sets for the king, knight and rook

signifies a king move in that direction. Moves by pieces like the rook, which move along lines, require move information to be stored in the vector. Here each unit in the vector holds a sequence of one dimensional intervals, where each interval denotes the destination locations as an offset from the originating location. This generic operator representation also includes a set of linear constraints for each object, relating the originating location of the move $\langle X_F, Y_F \rangle$ to the destination locations $\langle X_{Ti}, Y_{Ti} \rangle$. These constraints are illustrated on the left in Figure 5.4. Note that there is one constraint for each possible direction. The constraints arise from the geometric view of the primitives **connected** and **openline** previously discussed.

The operator subspace defines all the operators that are possible for a given object. However, an object may be near the edges of the board and all the moves may not be available, due to the destination location being outside the legal region. The system automatically generates specializations of the subspaces to take into account these edge effects. Figure 5.5 illustrates the result of this analysis for the king subspaces. Here we find 3 mutually exclusive subspaces, one for the king in the middle of the legal region, one on the side and the other when the king is in a corner. Each specialized subspace includes the bit vector representing the operator directions that are available and a simple pattern that constrains the location of the object moved. For example, the middle subspace in Figure 5.5 includes bit vector representing the 5 moves available and a pattern that constrains the black king to the bottom edge of the board ($Y_{BK} = 1$). Note that only one side and one corner subspace are needed because of rotational symmetry. Similar analysis for the knight produces seven subspaces.

This section has introduced operator subspaces as an explicit representation of the operators available for each object. The next section describes how these operator subspaces are used to compile influence relations and produce pattern/operator set pairs such as the one illustrated in Figure 5.3.

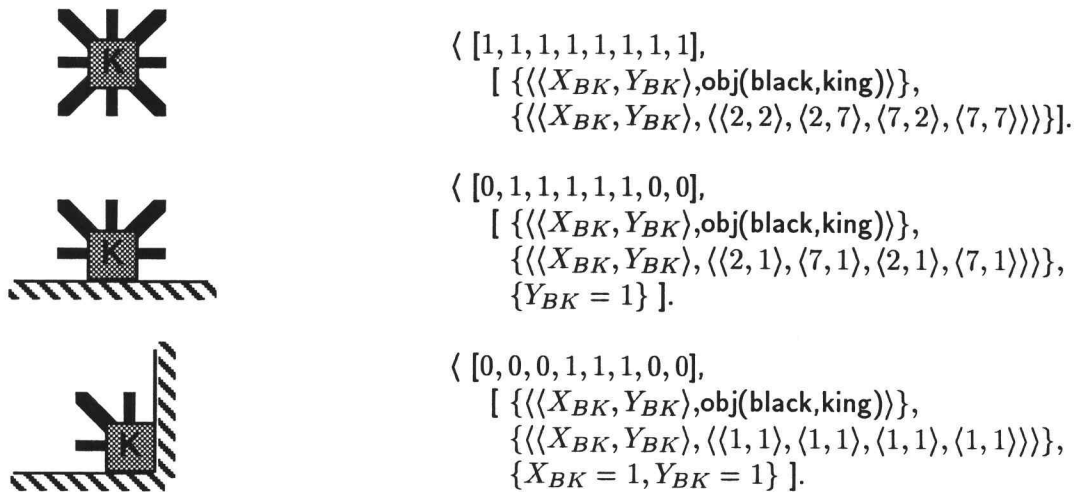


Figure 5.5: Geometric representation of the king operator subspaces taking into account edge affects

5.2.2 Compiling Geometric Influence Relations

The algorithms for compiling influence relations with geometrically represented patterns and operator subspaces are similar to the previously presented algorithms for the logic based representation. With a logic based representation it was demonstrated in Section 3.1.3 that compiling influence relations for complex patterns is decomposable into compiling influence relations for simple conjunctions. This decomposition is unaltered when using geometrically represented patterns. However, the algorithms for compiling the individual conjunctions is changed. Hence, this section describes algorithms for compiling geometrically represented conjunctions.

The algorithms for compiling a conjunctive logical expression, described in Section 3.1.2, apply partial evaluation to the logical definitions of the influence relations when instantiated with the given conjunction. There are two principal steps involved: *unfolding*, where the frame axioms, represented as horn clauses, are unfolded; and *simplification*, where the resulting conjunction is simplified by eliminating redundancy and detecting inconsistency. The new algorithm employs

the same logical definitions of the influence relations, but implements the two steps differently. First, rather than unfold horn clauses to implement an operator application, the algorithm employs specialized procedures that exploit the operator subspace representations. Second, rather than simplifying a conjunction of logical relations, the algorithm simplifies a set of linear constraints using a combination of Gaussian elimination and the SUP-INF method ([Shostak 77]). An additional step is involved when working with geometric representations, which is applied after the above steps have produced a set of pattern/opset pairs. This final step combines solutions, which apply only to individual moves, so the resulting operator subspaces cover multiple operators (as illustrated in Figure 5.3, where all 6 operators that make the pattern false are included). All three steps are described in detail below.

Applying Operator Subspaces

The frame axioms given in Figure 3.7 define the effects of operators on the logical primitives $\text{on}(S, Sq, Obj)$ and $\text{openline}(S, Start, End, Dir)$. In the geometric representation we do not use these primitives, rather we define objects at a location as a list of $\langle Loc, Obj \rangle$ and openlines as a list relating the start location, the end location and a direction vector $\langle LocS, LocE, \Delta \rangle$. Adapting the frame axioms to apply to the alternative representations is straightforward. The method applies the axioms to the geometrical primitives and explores the disjunctive cases for each primitive as before. The algorithm collects consistent solutions.

In the logical representation, these solutions would be sufficient and the compiler would terminate. However, in the geometric representation, we must further expand each solution so that the individual operators in the subspaces are explored. Hence, we further enumerate each solution with the disjunctive cases defined for each operator subspace. In Figure 5.4 each direction that can be moved is described by a set of linear constraints between the originating and destination locations. Determining if that direction satisfies the influence relations involves composing the

subspace constraints with the constraints of the solution and simplifying.

Simplifying Linear Constraints

The *Simplify* procedure attempts to eliminate any redundancy and determine if the set of constraints is inconsistent. The method repeatedly applies the following operations until either a contradiction is found or no more operators apply and the expression is in canonical form.

- *Apply functional dependencies.* If the *Objs* list in the pattern includes $\langle Loc_1, Obj_a \rangle$ and $\langle Loc_2, Obj_a \rangle$, then set $Loc_1 = Loc_2$ and propagate. Conversely, if the *Objs* list includes $\langle Loc, Obj_a \rangle$ and $\langle Loc, Obj_b \rangle$, and $Obj_a \neq Obj_b$, then *fail*.
- *Eliminate Variables.* Apply Gaussian elimination to pairs of equations to derive constraints on object variables and among object variables. Prefer to eliminate intermediate variables (denoting empty locations) and internal variables (which arise from the openlines in the pattern).
- *Propagate Values.* If a value is determined for a variable, then substitute the value in all constraints that involve that variable. Attempt to further simplify those affected constraints.
- *Tighten bounds.* A pattern includes interval bounds on both internal variables (from the openline bounds) and object variables (from the rectangular region constraint). These bounds are tightened by propagating high and low values through the linear constraints using the SUP-INF method [Shostak 77]. The general method for Presburger formulas is considerably simplified because the equations only involve unit coefficients of the variables. In [Davis 87] an $O(n)$ algorithm for n variables and linear constraints is given. The algorithm used here takes only $2n$ steps. If an interval can be tightened to a single value, then that value is assigned and propagated.

- *Detect inconsistencies.* Inconsistent variable assignments are detected as early as possible. Inconsistencies detected include: $X = C_1, X = C_2$, where $C_1 \neq C_2$; $X \leq C_0, X = C_1$, where $C_1 > C_0$; $X \leq C_0, X \geq C_1$, where $C_1 > C_0$.

Combining Cases

The above steps produce influence relations where the operator subspace includes only a single operator. It is desirable (for reasons that will become clear in Chapter 7) that the operator subspaces of compiled influence relations include as many operators as possible. This final stage of the algorithm clusters the single solutions together so as to find a partitioning of the solution set, where the patterns of each partition have consistent intersections. The process is illustrated in Figure 5.10.

5.3 Example of Compiling *maintain-true*

We use a simple example to illustrate the compilation process for *maintain-true* for the pattern *rook-takes-king* illustrated in Figure 5.1. Here we are interested in finding those cases where the black king is to move and maintain true the condition that the king can still be captured by the rook. These cases consist of a set of pattern/operator set pairs where *rook-takes-king* is true in the initial pattern and each operator in the set, when applied leads to a new pattern where *rook-takes-king* is still true. The final two cases are illustrated at the bottom of Figure 5.10. In the first case, the king is adjacent to the rook and the three operators illustrated maintain the *rook-takes-king* condition. The second case describes the situation where the king is at least one square away from the rook and the two operators, along the line of the rook attack maintain the threat.

The first step of the compilation algorithm is illustrated in Figure 5.6. In this step we enumerate the possible operator subspace cases, which describe possible directions, to see if the pattern is maintained. These operator subspace cases are illustrated along the side of the figure. Recall that each case is described by a set

of linear constraints relating the originating and destination locations of the king. The cases along the top describe the possible alternative orientations of new rook-takes-king pattern following the king move. The first column assumes that following the king move, the rook will be below the king. Each of these columns is described by a set of linear constraints relating the current locations of the king and the rook, and the new location of the king, denoted $T1?$. Each box in the matrix represents a composition of the two sets of linear constraints, the row constraints relating the current king location to the destination location $T1?$, and the column constraints relating the current king and rook locations to the new destination location $T1?$. For each set of constraints in the matrix *Simplify* is called. Those boxes with a cross denote an empty solution, where an inconsistency was detected. The other boxes illustrate the solution found. For three of the cases marked (a), (b) and (c), we illustrate the operation of *Simplify* in more detail.

The case marked (a) in Figure 5.6 is illustrated in Figure 5.7. Here, because we are in the first column, we are assuming that the new location for the king will be directly above the rook. In terms of the constraints, we are assuming that Xt (the X coordinate $T1?$) is equal to Xr (the X coordinate of the rook). The full set of constraints from the column is illustrated on the left in Figure 5.7, while the constraints from the row (from the king move) are illustrated on the right of Figure 5.7. In this case the king move is to the north west. Applying *Simplify* to the set of constraints first involves Gaussian elimination to determine a value for both internal variables la (the length of the openline between the rook and the old king location) and lb (the length of the openline between the rook and the new king location). Here we determine that both internal variables have value 1, producing the specialized solution illustrated.

An inconsistent case, marked (b) in Figure 5.6 is illustrated in Figure 5.8. This case is also in the first column and so has the same set of constraints on the left. The king move constraints on the right are different and describe a move of

the king vertically. It is clear that these two constraints are inconsistent, since the king will not be attacked by the rook following the move. This inconsistency is determined by Gaussian elimination that determines $la = 0$, which is inconsistent with the constraint $la > 0$.

Finally, the case marked (c) in Figure 5.6 is illustrated in Figure 5.9. Here we are in the third column, where it is assumed that the king is attacked in the same direction before and after moving. The king move constraints describe a move to the left. As we would expect, the system determines that this case is consistent. What is interesting about this case is that it demonstrates the use of the SUP-INF method as it tightens the bounds on the object regions. We illustrate this process for the X dimension in Figure 5.9. The upper boxes on the left and right of the figure give the initial bounds on X_{BK} and X_{WR} respectively. We illustrate computing the new lower bound for X_{BK} on the left. Here the equation giving X_{BK} in terms of X_{WR} and the internal variable lb is used. Because we want the lower bound, we substitute the lower bound of X_{WR} and the lower bound of lb , producing a tighter bound of 3. Computing a new upper bound on X_{WR} is illustrated on the right. Here the transformed equation giving X_{WR} in terms of X_{BK} and the internal variable lb is used. Since we want the upper bound, we substitute the upper bound of X_{BK} , but because the variable lb is *negated* in the equation, we use its *lowerbound*, producing a tighter bound of 5.

The final clustering stage is illustrated in Figure 5.10, where two final solutions are determined. This stage clusters the four cases found (those non crosses in Figure 5.6) into consistent sets, where each set becomes a final solution. To determine the pattern for a solution we intersect the patterns in the set and to determine the operator set for a solution we form the union of each operator in the set.




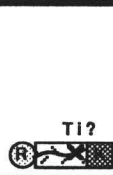
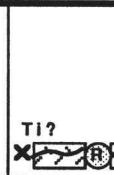

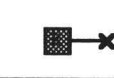





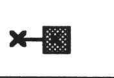
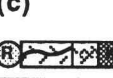



				
	X	X	X	X
	X	X		X
	X	X	X	X
	X	X	X	X
	X		X	X
	X	X	(c) 	X
	(a) 	X	X	X
	(b) X	X	X	X

Figure 5.6: Determining consistent moves of the black king that *maintain-true* rook-takes-king

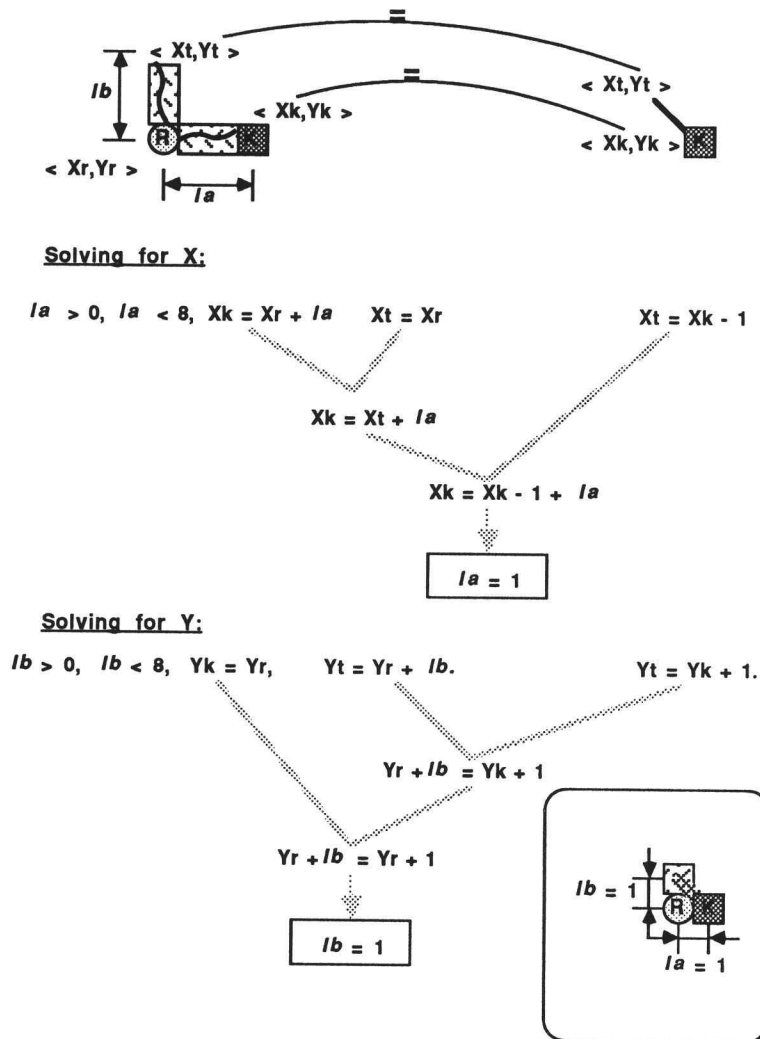


Figure 5.7: Detail of *Simplify* determining the consistent solution to *maintain-true* rook-takes-king marked (a) in Figure 5.6

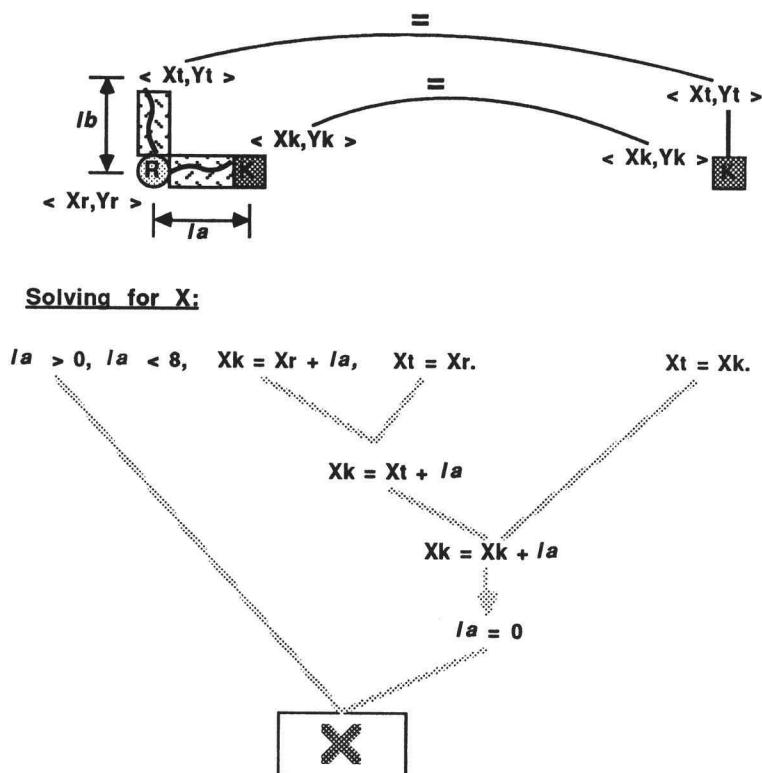


Figure 5.8: Detail of *Simplify* determining an inconsistent case during compilation of *maintain-true* rook-takes-king marked (b) in Figure 5.6

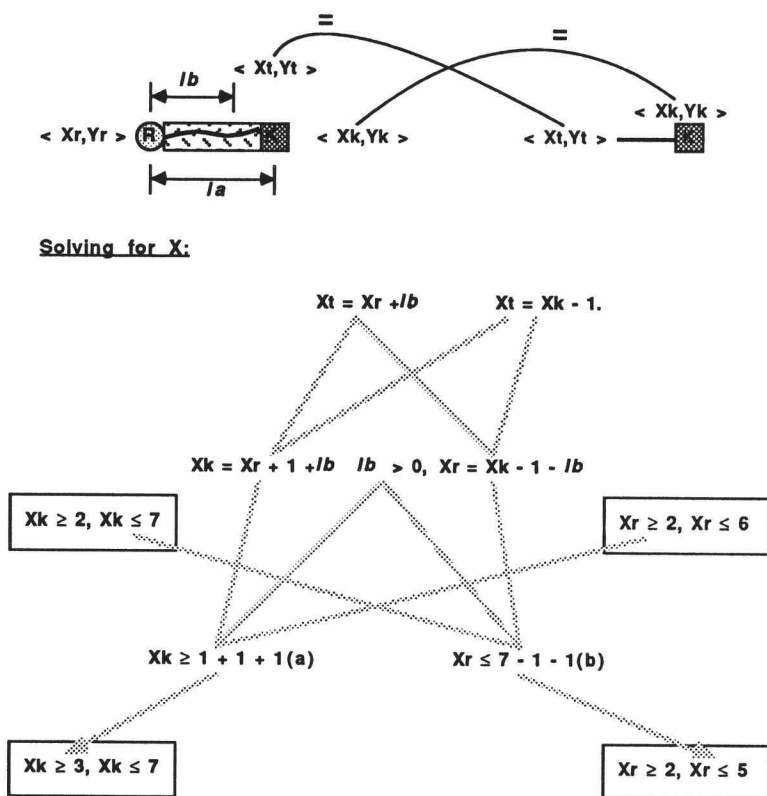


Figure 5.9: Detail of *Simplify* tightening the region bounds in the X dimension during compilation of *maintain-true* rook-takes-king marked (c) in Figure 5.6

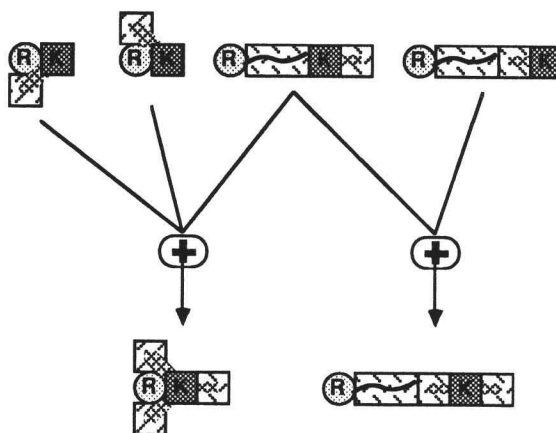


Figure 5.10: The final clustering stage combining individual solutions to compiling *maintain-true* rook-takes-king

5.4 Geometric Intersection

The original *Pattern-Intersection* algorithm defined on Page 58 demonstrated how the intersection of two patterns reduces to many intersections of their component conjunctions. This decomposition is retained with a geometric representation of the patterns, but in this case the reduction is to intersections of simple conjunctive constraints. To intersect two conjunctive constraints, the constraints are composed and then *Simplify* (defined on Page 111) is called.

The geometric representation of patterns enables a significant inefficiency with logically represented patterns to be eliminated. As discussed in the introduction to this chapter, recall that the computation of both win and loss patterns involves intersecting new patterns with all previously generated patterns. Given a new pattern p and a set of previous patterns PP , the simplest implementation of this process is by generate and test: generate candidate patterns p_p from PP then test whether p intersects with p_p . Geometry eliminates this inefficient generate and test behavior by incorporating tests of p into the generator of previous patterns. The tests are incorporated by storing all of PP in a database and using properties of p as an index into this database to find only those patterns that intersect with p . In fact, we index using a *necessary condition* of p and hence, the set returned from indexing represents a superset of those patterns that intersect with p . In otherwords, not all the tests of P are incorporated into the generator of candidate patterns.

The database and indexing scheme exploit the rectangular region constraints for each object in the principal conjunction of a pattern (see Section 5.1). The database is a multi-dimensional *rectangle tree* [Edelsbrunner 83], a specially designed structure to enable efficient queries based on rectangular constraints. Given a pattern with d objects, lookup complexity is bound by $O(\log^{2d}(2d) + N)$. where N is the number of patterns returned. Figure 5.11 gives some empirical results demonstrating the effectiveness of this indexing scheme. The rectangle tree indexing is

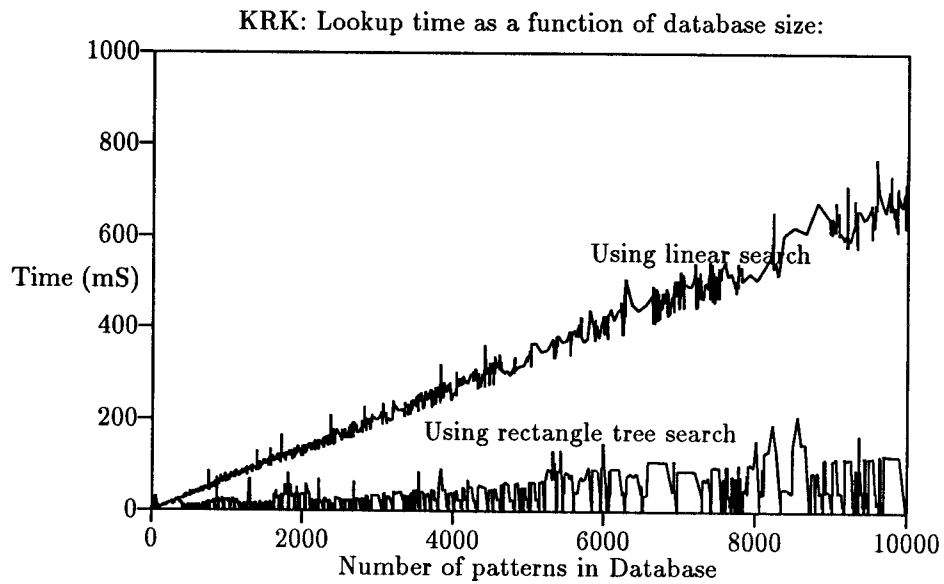


Figure 5.11: A graph showing the time in mS needed to lookup all intersecting patterns as a function of the size of the database. The top line is where the database is represented as a list and lookup is linear search. The lower line is where the database is indexed using a rectangle tree. Execution times for all calls to $Lookup-Intersection(\nu(s), P_{Loss})$ are illustrated during compilation of the KRK chess ending.

compared to a simple linear search.

The indexing represents only a necessary condition because only the consistency between the rectangular constraints of the principal conjunctions are tested. In order to determine whether the two patterns intersect, the other constraints of the principal conjunctions and the exceptions must be checked by repeated calls to *Simplify*.

Chapter 6

Analysis of Geometric Abstractions

In this chapter we analyze the compiler that incorporates the geometric representations introduced in Chapter 5 and contrast it with the previous analysis given in Chapter 4 when the compiler employed logic based representations. We first consider whether the new compiler is *complete*. We find that, in contrast to the previous compiler that employed a logical representation, the new compiler is complete. We next consider the computational complexity of the compiler and the anticipated complexity of using the compiled knowledge. This analysis demonstrates that the problems of intractability identified previously (in Section 4.3) have been overcome through the use of geometric representations.

6.1 Completeness

There are two aspects to proving completeness. First we must show that the method will terminate within a finite time and using a finite amount of space. Second we must show that all the influence proof types identified can be compiled into patterns by the compilation algorithm.

6.1.1 Proof of Termination

To prove termination, we show that there will only be a finite number of patterns generated, and once these patterns have been produced, the compiler terminates.

The argument relies on two important characteristics of the *extensional database* of a give problem class. First, it is clear that the extensional database is finite, because we have a fixed number of objects arranged in a fixed number of locations. Second, given a problem class (defined by the objects) we know that there is a maximum depth, n that represents the longest possible *optimal* solution to win or loss, beyond which there are no instances. This can easily be shown by induction on the depth: we know by definition that there are instances lost (won) in depth 0, and we know that it is impossible for there to exist lost(won)-in- $i + 2$ instances without there existing lost(won)-in- i instances. Hence, because there are only a finite number of instances, there must be a finite depth at which there are no instances.

Given the assumption that the pattern compiler is correct, it will generate no patterns at depth i if $i > n$ (i.e., there are no instances at depth i). This is because the geometric representation enables pattern intersections that have no extension to be efficiently detected. Hence, to prove termination of the pattern compiler, we need only show that it terminates the compilation for some arbitrary depth j , where $j \leq n$. This is straightforward because as we have previously shown (Section 4.3.1) that only a finite number of patterns can possibly be generated for each depth. Hence, the compiler will eventually complete a given depth and therefore will eventually terminate.

6.1.2 Proof of Compiler Completeness

The problems with incompleteness arose because of the inability of the logic representation of operator sets to support set intersection and set difference (see Section 3.2.1, page 75). The new representation of operators as operator subspaces, introduced in Section 5.2.1, employs two representations for the space of all operators. The first involves representing the operator set as a bitvector. Clearly, this representation supports both set intersection and set difference operations. The second involves representing the operator sets as a set of ordered intervals. Both

set intersection and set difference can be implemented over this representation by employing interval arithmetic. Hence, this new representation supports the needed set operations and enables compilation of all three kinds of proofs.

Since the compilation algorithm has been shown to terminate, and is capable of compiling the three kinds of proofs that exist, the new compiler is complete.

6.2 Complexity Analysis

This section analyses the previously stated geometric algorithms to determine their computational complexity.

6.2.1 Number of Proof Sentences

The number of proof sentences has grown because of the one-to-many mapping between logically represented patterns and geometrically represented patterns described on Page 105. However, many of these sentences have no extension because this reduction in individual coverage of patterns implies fewer consistent subsets. Chapter 7 presents an efficient algorithm that avoids generating all empty sentences. Chapter 8 presents empirical results demonstrating a manageable growth in patterns.

6.2.2 Number of Pattern Computations

The number of calls to *Pattern-Intersection* is reduced by using geometry because the need for a new pattern to intersect with all previously generated patterns is eliminated. Geometry enables the previously generated patterns to be stored in a database and indexing used to obtain those that intersect with the new pattern as described in Section 5.4.

6.2.3 Complexity of Pattern Computations

It was demonstrated in Section 3.1.3 that the principal operators of the compiler, *Pattern-Intersection* and *Pattern-Difference*, both decompose to calls to the subroutine *Simplify*. When using geometric representations this routine is defined in Section 5.2.2. There are two main operations performed by this routine: First, we have gaussian elimination, which requires $2n$ operations for n variables and n equations. Second, we have the SUP-INF method, which also requires only $2n$ operations [Davis 87] (when we have n linear constraints with unit coefficients). Hence, by employing geometric representations we have substituted a process with linear complexity for one provably NP-complete.

6.3 Summary

This chapter has demonstrated that the problems with intractability have been eliminated by exploiting a geometric problem space representation. The principal result is that the argument quoted by Hector Levesque in the introduction to Chapter 5 has been shown to be true: geometric representations are *inherently* more tractable than logical representations. The core procedures of the compilation method, which manipulate geometrically represented patterns, have only linear complexity.

Chapter 7

Techniques for Searching the Space of Abstractions

In the previous chapters we have described in detail how individual influence proofs are compiled into goal patterns. This chapter describes the overall control structure of the compiler and details the selection and ordering of influence proofs to be compiled.

In speedup learning there have been two principal methods for controlling the generation of learned knowledge. The first method is the example based approach, of which EBL is the prototypical technique ([Mitchell Keller and Kedar-Cabelli 86]; [Minton 88a]). In this approach, problem instances are provided by a teacher or the environment. These examples are first explained and then compiled into some form of rule. This new rule is intended to improve problem solving for the given problem instance and other similar instances. This approach could be characterized as *lazy*, since improvement is initiated only when new problem instances are provided. In contrast, the second method is the knowledge compilation approach. Knowledge compilation employs no training examples and generates compiled knowledge from analysis of the initially specified problem space. There are many examples of this approach including [Etzioni 91], where control rules which avoid backtracking are generated from the initial specification of a problem solving domain; [Braudaway and Tong 89], where efficient design rules are generated from

a declaratively written specification of a correct design; and [Schoppers 89], where a complete “reaction plan” is generated from a specification of a planning domain. This approach is *eager*, since the compiler actively generates knowledge independent of problem instances.

The methods introduced in this thesis can be incorporated into either approach. [Flann 90] describes an application of the method when the influence proofs are generated from explaining user-provided training examples. That work demonstrates how the method can be used to learn correct patterns for low ply problems from the King-rook king-knight ending, originally explored in [Quinlan 83]. In this chapter we describe an eager approach to generating and compiling influence proofs.

There are two principal considerations when determining the best approach to organizing the compilation process. First, we wish to organize the compilation so that it is *efficient* and no redundant work is performed. Second, we wish to organize the compilation so that the *compile-time/coverage tradeoff* discussed in Section 1.2.2 is maximally exploited.

Since the previous chapters describing geometric representations have demonstrated efficient compilation algorithms, the emphasis here is on ordering the compilation so as to avoid redundancy. Redundancy can occur in many ways during compilation. One obvious redundancy arises from the use of influence proofs of depth $i - 1$ to define proofs of depth i . To avoid the potential redundancy of having to construct lower depth proofs multiple times, we order the compilation so that the lower depth proofs are compiled before higher depth proofs. In this way, the resulting goal patterns are compiled only once and shared among higher depth proofs. Other redundancies can arise from influence proofs of the same depth that employ common subsets of goal patterns. Here we want to compile that common subset, then share the partial result among those proofs that include it.

A well established method to avoid redundancy in computation is *dynamic programming* [Bellman 57], [Larson and Casti 78]. Dynamic programming is a pow-

erful method that has been applied to many different problems in domains such as scheduling, resource management, control, and game theory. What characterizes problems that are suitable for dynamic programming is the sequential nature of the computation and the reliance of the computation on previously computed results. Recently, the method has been advocated as a technique for *anytime problem solving* [Boddy 91]. Dynamic programming has been used successfully in a chess end game application described in Section 1.1.2. There an *extensional* database is constructed for a chess endgame by performing backwards search from a user provided set of termination *positions*. In this chapter we describe a similar approach that constructs an *abstract* database by performing backwards search from a user provided set of termination *patterns*.

The tradeoff between the time spent compiling a domain and the coverage achieved over that domain arises because, even with abstraction, we may not be able to run the compiler to completion. Hence, we want to organize the compilation so that the coverage over a domain is maximally accumulated, given the limited compilation time available. This implies that we should order the compilation so that the patterns are generated *most general first*. Recall that in Section 6.1.1 we demonstrated that the coverage of patterns tends to be a *monotonically decreasing* function of their depth. Hence, the backwards search strategy of compiling shorter depth proofs before longer depth proofs—developed to avoid redundancy—also allows us to exploit the compile-time/coverage tradeoff.

We are still left with the decision of how to order proof/pattern generation within the framework of backwards search. Two options are explored within this thesis. The first option is *breadth-first search*, where we work backwards from depth 0, each time increasing the depth by 1 and completely compiling each depth before moving to the next depth. This is the most straight forward approach, and it is described first. The second option is *best-first search*, where we still work backwards from depth 0. However in this case, we do not necessarily wait until we have com-

- *Loss* Contains a set of $\langle \nu(s), Op, i \rangle$, where $\nu(s)$ is a *LOSS* pattern (in *i* ply) and Op the optimal action to take when $\nu(s)$ is true.
- *Win* Contains a set of $\langle \hat{\nu}(s), Op, i \rangle$, where $\hat{\nu}(s)$ is a *WIN* pattern (in *i* ply) and Op is the optimal action to take when $\hat{\nu}(s)$ is true.
- *PLoss* Contains a set of $\langle \nu(s), \overline{Op}, i \rangle$ which are partially completed loss proofs. In this case the \overline{Op} represent those actions available in $\nu(s)$ which are not known to lead to a loss.

Table 7.1: The databases used by the dynamic program

pletely finished one depth before starting compilation on the next depth. Rather, the compiler always tries to compile the proof that will have the most coverage, irrespective of its depth. This best-first search is worthwhile, because even with the monotonic relation between coverage and depth, the patterns within each depth display a diversity of coverages. Hence, by having a more flexible control strategy, the compiler can focus attention on the best patterns from each depth.

The remainder of this chapter is divided into two sections. The first section describes in detail the breadth-first search algorithm and illustrates it with examples from compiling chess endgames. The second section describes the best-first search algorithm as a modification of the breadth-first search algorithm.

7.1 Breadth-first Search

To understand the dynamic programming algorithm for constructing *abstract* databases described in this section, it is helpful to first review the compilation algorithm for *extensional* databases introduced in Section 1.1.2, which is also a dynamic program. That algorithm represents the extensional database as a large array, where each cell in the array represents a single position. Indexing into the array is achieved efficiently by employing a Gödel function to map directly from the locations of each of piece in a given position to the cell representing that position (see [Thompson 86]

for more details). Initially, all the known white-win termination positions are tagged as wins in the array. All other positions are initialized with a count of all the legal operators for black that are available. Let us focus on how the method demonstrates that some position p is a loss. To do this, the system must demonstrate that all of the legal actions that can be taken by the losing player from p lead to a win for the opponent.

The backwards search is initiated by computing all possible *predecessors* of the termination positions by applying the standard moves of chess backwards (called unmoves) for black. Each time such an unmove generates the position p , its count of black moves is decremented by 1. When the count reaches 0, we have shown that all available black moves lead to *successors* that are white wins—and hence p is a loss for black. Each new loss position can be similarly “unmoved” for white to create new won positions. The process continues until there are no new won or lost positions. All remaining unclassified positions are labeled as draws.

The algorithm for constructing abstract databases described in this section works analogously. Backwards search is initiated from the win patterns of depth 1, provided by the user. First, loss patterns of depth 2 are compiled, then new win patterns of depth 3. The process continues, alternating between loss and win, until either the time or space allotted runs out or no new patterns are derived.

There are some important differences between the extensional algorithm and the abstract algorithm. First, rather than employing an array to hold positions, the abstract algorithm employs geometrically indexed databases described in Section 5.4 to hold patterns. Looking up a pattern requires a search through a multi-dimensional rectangle tree, which is more costly than the simple array indexing of the extensional algorithm. Second, rather than apply unmoves to positions, we compute influence relations which either *make-true* or *maintain-true* the patterns. Third, rather than represent all the legal operators available as a count, which is reduced to 0 to prove loss, the abstract compiler represents all legal operators as a *set* which is reduced

to \emptyset to prove loss. This reduction in the operator set is achieved by incrementally removing sets of operators from compiled influence relations which *make-true* or *maintain-true* known win patterns.

We now describe the abstract breadth-first algorithm in more detail. There are three global databases employed by the algorithm to store completed goal patterns and intermediate results during compilation. Each database is indexed to facilitate efficient intersection queries as described in Section 5.4. The databases are listed in Table 7.1 and initialized to be empty. The function $Store(Datum, Database)$ is used to store information in each database. Let Win_0 be a set of $\langle \hat{\nu}(s), Op, 0 \rangle$, where each $\hat{\nu}(s)$ is a winning termination pattern and Op is the optimal winning operator. This set is initially specified by the user. The main loop of the program is given below:

```

Done  $\leftarrow$  false,
Forall  $p \in Win_0$ ,  $Store(p, Win)$ ,
NewWin  $\leftarrow Win_0$ ,
i  $\leftarrow$  1,
While Not(Done) Do:
  NewLoss  $\leftarrow Compile-all-losses(i, NewWin)$ ,  $i \leftarrow i + 1$ ,
  Forall  $p \in NewLoss$ ,  $Store(p, Loss)$ ,
  Done  $\leftarrow NewLoss = \emptyset$ ,
  If Not(Done)
  Then NewWin  $\leftarrow Compile-all-wins(i, NewLoss)$ ,  $i \leftarrow i + 1$ ,
    Forall  $p \in NewWin$ ,  $Store(p, Win)$ ,
  Done  $\leftarrow NewWin = \emptyset$ .

```

The top level loop simply compiles *LOSS* patterns and *WIN* patterns for each depth. The flag *Done* is set true when no more new patterns are derived. Both algorithms use the 3 global databases in Table 7.1. In the remainder of this section we define both compilation algorithms in detail.

7.1.1 The $Compile-all-losses(Depth, NewWin)$ Algorithm

The algorithm for $Compile-all-losses(Depth, NewWin)$ comprises two sequential sub-routines. The first routine, $Initialize-partial-proofs(NewWin)$ compiles influence re-

lations for each of the new win patterns in *NewWin* and then generates new *partial proofs* using the compiled influence relations. A partial proof represents a potential loss pattern and includes an *undetermined Op set* which describes all those operators that are available in the pattern for which the outcome is *not known to be a loss*. The second routine, *Complete-partial-proofs* generates new loss patterns by attempting to *complete* the partial proofs by reducing their sets of undetermined operators to empty. A partial proof with an empty undetermined operator set becomes a new loss pattern, since it implies that all possible operators available in the pattern lead to a win for the opponent. The two subroutines are described in detail below.

The *Initialize-partial-proofs(NewWin)* Algorithm

This routine has two parts. First, both *maintain-true* and *make-true* influence relations are compiled for each new winning pattern in *NewWin*. Second, each pattern/action pair from the compiled influence relations is used to generate new partial proofs.

The first routine for compiling influence relations simply loops through all new win patterns compiling both *make-true* and *maintain-true* influence relations as defined in Section 5.2:

$$\begin{aligned}
 &NewLoss \leftarrow \emptyset, \\
 &MTWin \leftarrow \emptyset, \\
 &MWin \leftarrow \emptyset, \\
 &\forall \langle \hat{\nu}(s), Op_{WIN}, i \rangle \in NewWin, \\
 &\quad Push(CompileIR(make-true(\lambda s.\hat{\nu}(s), Op_{LOSS}, S)), MTWin), \\
 &\quad Push(CompileIR(maintain-true(\lambda s.\hat{\nu}(s), Op_{LOSS}, S)), MWin).
 \end{aligned}$$

Partial proofs are represented as $\langle \nu(s), \overline{Op} \rangle$, where $\nu(s)$ is some pattern and the undetermined set \overline{Op} is represented as an operator subspace (introduced in Section 5.2.1, Page 106). We generate initial partial proofs for each $\langle \nu(s), Op_{LOSS} \rangle$ in *MTWin* and *MWin*. The patterns of the new partial proofs are simply each $\nu(s)$,

while the undetermined \overline{Op} sets are initially computed by taking the set difference of *all* those operators that are available in $\nu(s)$ minus Op_{LOSS} , those operators which *maintain-true* or *make-true* some win. More precisely, the initial partial proofs are generated by the following algorithm:

$$\begin{aligned}
& MPLoss \leftarrow \emptyset, \\
& \forall \langle \nu_{MWin}(s), Op \rangle \in MWin, \\
& \quad \overline{Op} \leftarrow \{Op_1 \mid o(S, LOSS, Op_1) \wedge \nu_{MWin}(S)\} \\
& \quad Push(\langle \nu_{MWin}(s), \overline{Op} - Op \rangle, MPLoss), \\
& MTPLoss \leftarrow \emptyset, \\
& \forall \langle \nu_{MTWin}(s), Op \rangle \in MTWin, \\
& \quad \overline{Op} \leftarrow \{Op_1 \mid o(S, LOSS, Op_1) \wedge \nu_{MTWin}(S)\} \\
& \quad Push(\langle \nu_{MTWin}(s), \overline{Op} - Op \rangle, MTPLoss).
\end{aligned}$$

The Complete-Pproofs Algorithm

This routine is a sequential, dynamic programming version of the algorithms given in Section 3.2.1 (on Page 74 and Page 76), adapted to compile all three kinds of proofs, *no-threat*, *one-threat* and *many-threat*. New loss patterns are derived by reducing to empty the undetermined \overline{Op} sets of the existing partial proofs using the newly compiled influence relations. First, the compiled *maintain-true* relations are used to complete both the new partial proofs in $MPLoss$ and previously generated partial proofs stored in the database $PLOSS$. Next, the compiled *make-true* relations are used similarly, this time to complete the new partial proofs in $MTPLoss$ and all the previously generated partial proofs. The two sets of new partial proofs are handled differently to prevent redundancy caused by combining the same partial proofs in different orders. More precisely, the algorithm is defined as follows:

$$\begin{aligned}
& \forall \langle \nu(s), \overline{Op} \rangle \in MPLoss, \\
& \quad Push(\langle \nu(s), \overline{Op} \rangle, PLOSS), \\
& \forall \langle \nu_{MW}(s), Op_{MW} \rangle \in MWin, \\
& \quad PP_0 \leftarrow Lookup-Intersection(\nu_{MW}(s), PLOSS), \\
& \quad Cover-Pproofs(\langle \nu_{MW}(s), Op_{MW} \rangle, PP_0, i), \\
& \forall \langle \nu(s), \overline{Op} \rangle \in MTPLoss, \\
& \quad Push(\langle \nu(s), \overline{Op} \rangle, PLOSS), \\
& \forall \langle \nu_{MTW}(s), Op_{MTW} \rangle \in MTWin,
\end{aligned}$$

<pre> Cover-Pproofs($\langle \nu_W(s), Op_W \rangle, PP, i) \leftarrow$ If $PP \neq \square$ Then $PP = [\langle \nu_{PP}(s), \overline{Op_{PP}} \rangle, PP_R]$, If $Op_W \cap \overline{Op_{PP}} = \emptyset$ Then $Cover-Pproofs(\langle \nu_W(s), Op_W \rangle, PP_R, i)$ Else $\nu_0(s) \leftarrow Pattern-Intersection(\nu_W(s), \nu_{PP}(s))$, $\overline{Op_0} \leftarrow \overline{Op_{PP}} - Op_W$, If $\overline{Op_0} = \emptyset$ Then $Store(\langle \nu_0(s), Op_o, i \rangle, Loss)$, $Push(\langle \nu_0(s), Op_o, i \rangle, NewLoss)$ Else $Store(\langle \nu_0(s), \overline{Op_0} \rangle, PLoss)$, $Cover-Pproofs(\langle \nu_{MTW}(s), Op_{MTW} \rangle, PP_R, i)$. </pre>

Table 7.2: A definition of *Cover-Pproofs*.

$PP_1 \leftarrow Lookup-Intersection(\nu_{MTW}(s), PLoss)$,
 $Cover-Pproofs(\langle \nu_{MTW}(s), Op_{MTW} \rangle, PP_1, i)$,

The first iteration simply stores all the new partial proofs generated from *maintain-true*'s into the global *PLoss* database. The next iteration then works through the new *maintain-true*'s using each to complete as many partial proofs as possible using the function *Cover-Pproofs* defined in Table 7.2. The remainder of the algorithm performs similarly, only this time we use the partial proofs and influence relations generated from *make-true*'s. Note the use of indexing, through the function *Lookup-Intersection* (defined in Section 5.4), to obtain only those partial proofs which could potentially be completed by each *make-true* or *maintain-true*.

The function *Cover-Pproofs* is defined in Table 7.2. It takes as input a single *make-true* or *maintain-true* pattern/action pair $\langle \nu_W(s), Op_W \rangle$, a list of relevant partial loss proofs, *PP*, and the current depth *i*. The first test directs termination when there are no more partial proofs. The second test determines if there is any intersection between the *make-true* or *maintain-true* operator set and the undetermined set of this partial proof. If there is no intersection, then this partial proof cannot

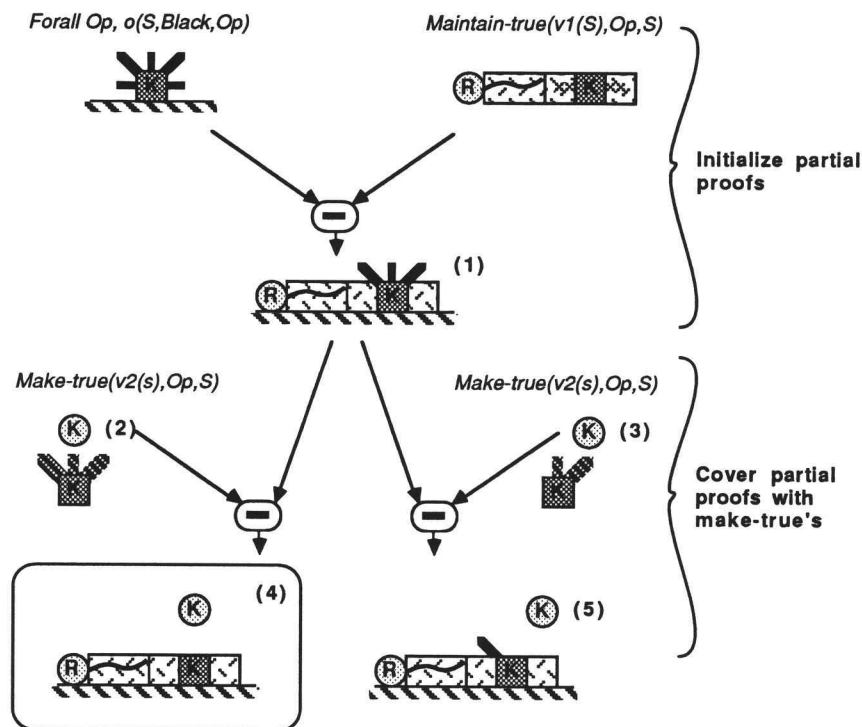


Figure 7.1: Dynamic program deriving a loss pattern from a *one-threat* proof for the king-rook-king chess endgame

be completed by the influence relation pattern/action pair and processing continues with the rest of the partial proofs. Otherwise we form a new partial proof. The new pattern is found by intersecting the old proof pattern with the $\nu_W(s)$ pattern. The new undetermined set is found by subtracting the *make-true* or *maintain-true* operator set from the old undetermined set. We then test if the new partial proof is complete, i.e., the undetermined set is empty. If the set is empty then the partial proof pattern is stored in the database *Loss* and the variable *NewLoss* along with the current depth and the optimal operator. If the partial proof is not complete, then it is stored in the database *PLoss*.

We illustrate the algorithm deriving examples of the three kinds of proofs. A simple *one-threat* derivation is illustrated in Figure 7.1 from the king-rook vs. king chess endgame. The top of the figure illustrates the initialization of partial

proofs. All operators available to the black king, when located on the lower side of the board, are combined with the pattern/action pair which *maintain-true* the *rook-captures-king* terminating win pattern. The result is the partial proof labeled (1) in the figure. Note the undetermined *Op* set is illustrated on the partial proof by the 3 possible moves for the king—those moves whose outcome is not known. The next relevant step of the algorithm is when the new *make-true* pattern/action pairs are used to try to complete this partial proof. The indexing procedure finds (1) when using those *make-true*'s labeled (2) and (3) in the figure along with others not illustrated. For each *make-true*, the *Cover-Pproof* function is then called with pattern (1) as one of the partial proofs. Each *make-true* is intersected with (1) and the set difference of the operator sets is computed. With *make-true* (2), the set difference operation results in an empty set, so a new loss pattern is derived. This pattern, labeled (4), describes one of the check-mate patterns for the king-rook-king ending. With *make-true* (3), the set difference operation leaves the undetermined set containing a single operator. This partial proof remains in the global database *PLoss* for potential completion by *make-true*'s or *maintain-true*'s derived later in the compilation.

Figure 7.2 illustrates a *no-threat* derivation from the king-rook-king chess endgame. The initialization of partial proofs is illustrated at the top of the figure. Here all moves of the king, when located on the side of the board are combined with the *make-true* pattern/action pair illustrated. The partial proof, labeled (1) illustrates the remaining three undetermined operators. During the next stage, when the new *make-true* from this level are used to complete proofs, two of the undetermined operators are eliminated by the *make-true* pattern/action pair labeled (2) resulting in the partial proof labeled (3). This partial proof is not completed during the compilation of this depth, so it remains in the database *PLoss*. The compiler next compiles winning patterns for white (not shown) and then begins compilation for loss again, this time for patterns of depth 3. At this stage, new

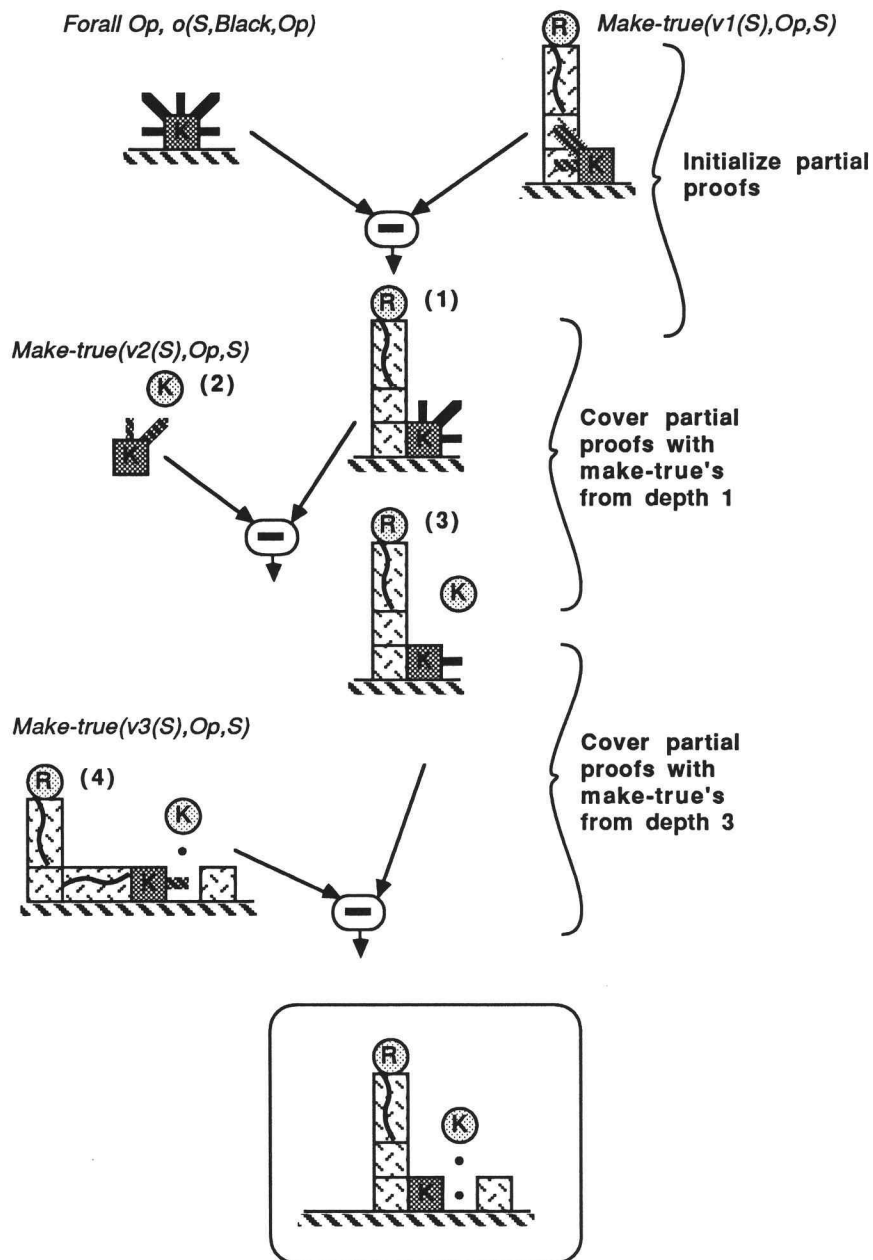


Figure 7.2: Dynamic program deriving a loss pattern from a *no-threat* proof for the king-rook-king chess endgame

make-true pattern/action pairs are derived. One such pair is illustrated in the figure and labeled (4). This *make-true* is derived from a win pattern which was in turn derived from the loss pattern found in Figure 7.1. This pattern states that if the king moves to the right then the resulting position is lost for black, because the rook can move down to the side and create a check-mate position. The two patterns (3) and (4) intersect and the single *make-true* operator eliminates the remaining undetermined operator of (3), thereby deriving a new loss pattern.

The *many-threat* proofs are compiled differently from the method described in Section 3.2.1. Previously, we exploited an equivalent encoding for the *many-threat* proofs where it was necessary to prove an empty intersection over operator sets that *make-false* the threat patterns. Here we exploit an alternative encoding where proving loss involves demonstrating that all available operators *maintain-true* at least one of the threat patterns. The advantage of this new encoding is that the algorithms available for compiling the *no-threat* and *one-threat* proofs, which exploit set coverage over operator sets, can be easily adapted to compile *many-threat* proofs. Note that during partial proof initialization, proofs are generated from *maintain-true* pattern/action sets, where the undetermined *Op* set is initialized to those operators that do not maintain the threat pattern. Also note that these same *maintain-true*'s are used to *complete* all partial proofs by intersecting new threat patterns and subtracting those operators which maintain the new threat. Hence, during this proof completion stage, many-threat loss patterns can be derived by accumulating threat patterns and showing that all available operators maintain at least one of the threats.

Figure 7.3 illustrates two different *many-threat* derivations from the king-rook-king-knight chess endgame. Note that the figure, because it involves more than 3 pieces, employs a special notation to denote geometric exceptions. The rook symbols with the bold lines indicate where the rook must *not* be, while the rectangles indicate regions that exclude the black king. At the top of the figure

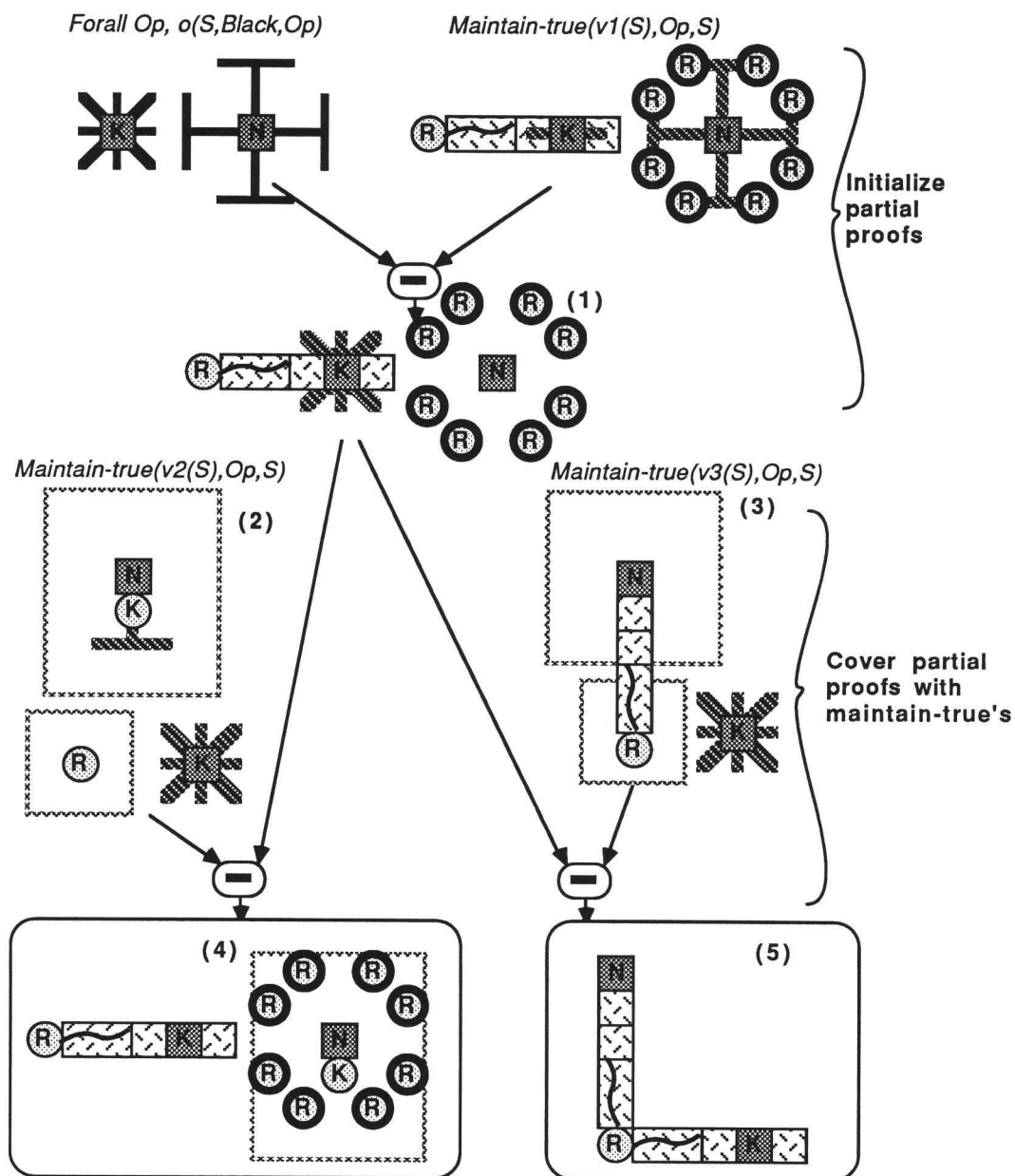


Figure 7.3: Dynamic program deriving two loss patterns (4) and (5) from *many-threat* proofs for the king-rook-knight chess endgame.

we illustrate the generation of a partial proof using a *maintain-true* pattern/action pair. In this case we use the set of operators which *maintain-true* the capture of the king by the white rook. Here king moves along the line of the attack, and all knight moves, so long as they do not capture the rook, are included in this set. Note that this pattern does not constrain the knight, which can be anywhere on the board. The resulting partial proof is denoted (1).

The next step of the compilation algorithm illustrated is when other *maintain-true*'s are used to complete this partial proof. One such *maintain-true* is illustrated as (2). This *maintain-true* describes the case when the black knight is attacked by the white king. Two moves of the knight and all moves of the black king, so long as the king is outside the noted region around the knight and the region around the rook, *maintain-true* the knight loss. The result of intersecting the two patterns is shown in (4). Because the set difference returns the empty set, this pattern is a new loss. This pattern describes the case where there are two simultaneous threats by two different pieces: the white rook is attacking the black king, while the white king is attacking the black knight. The exceptions ensure that the black king cannot move to protect the knight, or that the black knight cannot capture the rook.

Another *maintain-true*, which can complete this partial proof, is illustrated as (3). This pattern describes the case when the rook attacks the knight. No moves of the knight and all moves of the black king (when outside the excluded regions) *maintain-true* the knight loss (since the black king cannot capture the rook or move to protect the knight). The result of intersecting the two patterns is illustrated as (5). Here, because the rook is constrained to be at least 2 squares away from the knight (from (3)), the multiple exceptions on the rook are eliminated. In addition, because the black king is constrained to be at least one square away from the rook (from (1)), the exception region for the king around the rook is eliminated. Both of these simplifications are described in Section 5.2.2.

<pre> NewWin ← ∅, ∀⟨ν(s), Op, i⟩ ∈ NewLoss, ∀⟨ν_{MTL}(s), Op_{MTL}⟩ ∈ (CompileIR(make-true(λs.ν(s), Op_{WIN}, S)), OWins ← Lookup-Intersection(ν_{MTL}(s), Wins), ν_{OW}(s) ← Make-Optimal(ν_{MTL}(s), OWins), If ν_{OW}(s) ≠ ∅ Then Store(⟨ν_{OW}(s), Op_{MTL}, i), Wins), Push(⟨ν_{OW}(s), Op_{MTL}, i), NewWin). Make-Optimal(ν_{MTL}(s), OWins) ← If ν_{MTL}(s) = ∅ ∨ OWins = [] Then Return(ν_{MTL}(s)) Else OWins = [ν_W(s) OWins_R], Make-Optimal(Pattern-Difference(ν_{MTL}(s), ν_W(s)), OWins) </pre>

Table 7.3: The *Compile-all-wins* algorithm

This concludes our description of the *Compile-all-losses* algorithm. The next section describes the other main routine of the abstract compiler, the routine which compiles new win patterns.

7.1.2 The *Compile-all-wins*(Depth, NewLoss) Algorithm

The algorithm for *Compile-all-wins* is straightforward and follows directly from the algorithm given in Section 3.2.2. The algorithm works through each new loss pattern stored on *NewLoss* and generates those operators for white which *make-true* the pattern. To ensure that the resulting pattern/action pair is optimal, none of the previously compiled winning patterns of lower depth must intersect with this new pattern. Thus, optimality is enforced by looking up (using geometric indexing) those previous win patterns that may intersect with the new pattern, then subtracting them from the new pattern using the function *Pattern-Difference* defined in Section 3.1.3, but employing geometric representations. More precisely, the *Compile-all-wins* algorithm is defined in Table 7.3.

Figure 7.4 illustrates two simple examples of the *Compile-all-wins* algorithm

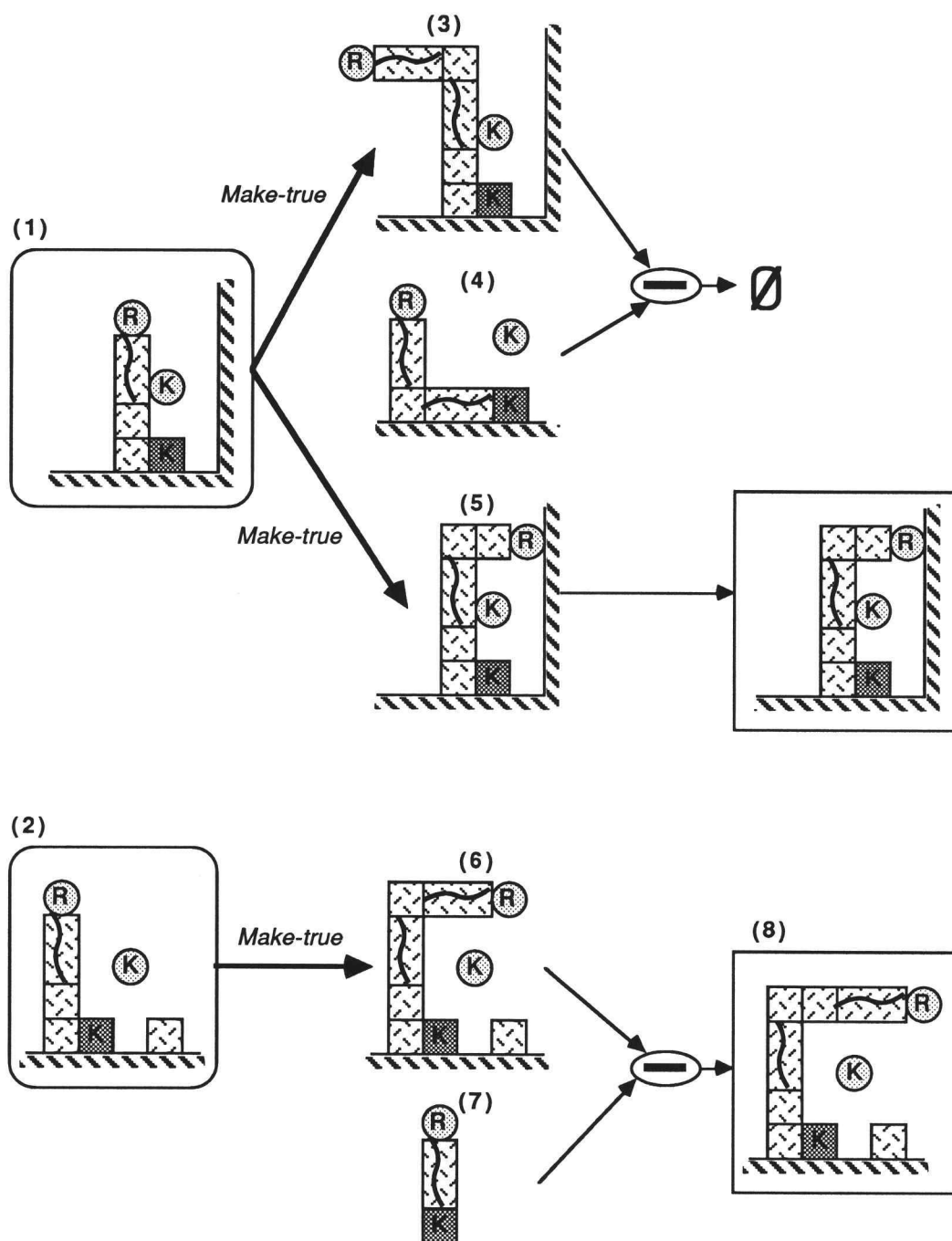


Figure 7.4: Dynamic program deriving two optimal win patterns for the king-rook-king chess endgame. Patterns (1) and (2) are new black-to-play loss patterns that are used to generate the new white-to-play win patterns (3), (5) and (6) through the influence relations shown. The patterns (4) and (7) are previously determined white-to-play wins.

deriving optimal win patterns in the king-rook-king endgame. Here the recently derived loss patterns are illustrated on the left side of the figure. The derivation of the pattern marked (2) is illustrated in Figure 7.2. The first step of the algorithm is to derive new win patterns by computing *make-true* influence relations for white over these loss patterns. Two new patterns are shown derived for (1), while one is shown derived from (2). The next step of the algorithm is to make these patterns optimal. A lookup for win patterns that intersect with win pattern (3) returns pattern (4), a pattern which wins in 3 ply (derived from the loss pattern shown in Figure 7.1). Here the call to *Pattern-Difference* returns empty, since pattern (3) is subsumed by pattern (4). A lookup for win patterns that intersect with (5) returns null, so this pattern is an optimal win. An example when the intersection narrows the coverage of a win pattern is shown in the bottom of the figure where optimal win pattern (7) is subtracted from win pattern (6). Here the position of the rook is restricted so as not to attack the king directly.

This concludes our description of the breadth-first compilation algorithm. The next section adapts these algorithms so as to compile the patterns *best-first*.

7.2 Best-first Search

This section describes the modifications made to the breadth-first search algorithms to implement *best-first search* of the abstract search space. The main difference in searching best-first is that the compilation of patterns for depth i is not necessarily delayed until all patterns of depth $i - 1$ have been compiled. Rather, a more flexible control structure is employed, where the best patterns are compiled irrespective of their depth.

The goal of employing best-first search is to improve the effectiveness of the compiler. Two issues that influence the effectiveness of the compiler are considered here. First, as discussed in the introduction to this chapter, we want the compiler to optimize the compile-time/coverage tradeoff by compiling the most general patterns

first. Second, we want the compiler to utilize the limited resources available by avoiding unnecessary work. An important component of unnecessary work is the generation of partial proofs that will never be completed during compilation (i.e., patterns that represent drawn positions).

To implement such a best-first search, a flexible, agenda-based compiler architecture is employed. The sequential compilation algorithms previously defined are partitioned into a series of independent tasks, whose order of execution can be determined dynamically. Best-first search then reduces to controlling this execution order, by picking appropriate tasks off an agenda, so as best to exploit the coverage/compile time tradeoff and avoid unnecessary work. Given the previously defined compilation algorithms, there are many ways to partition them into independent tasks. A fine grained partition, such as considering each recursive call to *Cover-Pproofs* as an individual task, provides great flexibility in control but introduces unwanted complexity. It was decided that a partition of the algorithms into four tasks provided the appropriate balance between flexibility of control and complexity. The four tasks are identified below:

- *Task1*: Compile *make-true* and *maintain-true* for *LOSS* for a new win pattern. This produces new *Task2*'s.
- *Task2*: Use a single new *make-true* or *maintain-true* of win patterns to generate new partial proofs and cover existing partial proofs. This task can potentially generate new loss patterns and thus new *Task3*'s.
- *Task3*: Generate non-optimal win patterns by compiling *make-true* for *WIN* for a new loss pattern. This produces new *Task4*'s.
- *Task4*: Optimize a win pattern and make it available for *Task1*'s.

The four tasks represent a simple partitioning of the previously defined algorithms for breadth first search. *Task1* is the first part of the algorithm *Initialize-Partial-proofs* (defined on Page 132), which compiles influence relations for the new

win patterns. *Task2* is a combination of the second part of *Initialize-Partial-proofs*, where new partial proofs are generated from a compiled influence relation, and *Complete-Pproofs* (defined on Page 133), where compiled influence relations are used to complete existing partial proofs. *Task3* and *Task4* partition the algorithm *Compile-all-wins* defined in Table 7.3.

To simplify the implementation, each set of like tasks is stored on its own agenda, rather than storing them all on the same agenda. The overall control procedure cycles through the four agendas, 1 through 4, choosing tasks and running them. There are two decisions that must be made with each agenda: (a) Which task to pick to execute, and (b) when to move to the next agenda. Both these decisions are made by applying a set of *heuristic preference rules*. Below we give the rules for choosing tasks off the agendas:

- *Task1*: Prefer compiling *make-true* and *maintain-true* for the win pattern that is most general (i.e., one that has the greatest extension) and simplest (i.e., one with a low count of exceptions). The generality of a pattern is determined by the function *Estimate-coverage* which is defined below.
- *Task2*: Prefer using the influence relation pattern/operator-set pair with the most general pattern and the largest operator-set.
- *Task3*: Prefer compiling *make-true* for the most general and simplest win patterns.
- *Task4*: Prefer optimizing the most general non-optimal win patterns.

These preference rules implement best-first search by addressing three issues which influence the effectiveness of the compiler introduced above:

Compile/time coverage tradeoff. The preference rules exploit the compile-time/coverage tradeoff in two ways. First, by consistently preferring the most general patterns at all stages of compilation, more general loss and win patterns

are identified earlier. Second, by preferring *make-true* and *maintain-true* influence relations with the largest operator sets at *task2*, new loss proofs are *completed* earlier. An early completion for a pattern means that the pattern was derived from an intersection of fewer *make-true* and *maintain-true* patterns. Since each pattern intersection tends to reduce the generality of the resulting pattern (i.e., given any two patterns p_1 and p_2 , then $p_1 \cap p_2 \subseteq p_1$ and $p_1 \cap p_2 \subseteq p_2$ is always true) fewer intersections lead to more general patterns.

Generating uncompletable patterns. It is very difficult to determine before hand if a partial proof generated is ever going to be completed and thus form a new loss pattern. However, it is true that more general partial proofs are more likely to intersect with *make-true* or *maintain-true* patterns during future proof completion operations. Hence, general partial proofs are more likely to be completed. In addition, by preferring *make-true* or *maintain-true* with large operator sets, partial proofs with smaller undetermined sets will be generated, which are more likely to be completed.

We have described the preference heuristics used to choose tasks off the agendas, but we have not yet described the heuristics which decide to move on to the next agenda. This decision for moving among tasks *1*, *2* or *3* is based on a simple evaluation function that estimates the generality and simplicity of the pattern or partial proof produced. A different agenda is chosen when the best task of the current agenda has a lower evaluation than a task on a different agenda. The only difficult decision to be made is when to move to agenda *4* and run a task. Each task on agenda *4* will generate a new optimized win pattern and a new *task1*. If the previous depths have not been completely compiled—a likely event in best-first search—then this process may produce only sub-optimal win patterns. This problem arises because without knowing all previous win patterns, it is impossible to determine for sure that the current win pattern does not include some win pattern for a shorter depth solution. Since in performing best-first search,

we are not prepared to wait for all lower win patterns to be compiled, there is a tradeoff between optimality and the eagerness of the best-first search. More eager best-first search processes produce less optimal databases.

All that remains to be done is to define the function *Estimate-Coverage* used to assess the generality of a pattern. This can be accurately computed by exploiting the geometric representation of patterns. Recall that in Section 5.1 we introduced the geometric representation of a pattern which included a set of rectangular regional constraints for each object in the pattern. One simple way to estimate the extension of a pattern is to form the product of the extensions of each object's rectangular region (computed as height \times width). However, this would lead to a gross over estimation, since often objects are mutually constrained. In this case, the regional constraints are enumerated into individual coordinates and a count of successful instantiations is made.

7.2.1 Summary

This chapter has introduced a dynamic programming approach to compiling abstract databases. First we presented a breadth-first search approach that works backwards from the initial termination patterns, completely compiling each depth before moving on to the next depth. We then introduced a best-first approach that again performs a backwards search from the termination patterns, only this time in a more flexible way in order to improve the effectiveness of the compilation process. We showed how best-first search can be implemented by restructuring the algorithms presented for breadth-first search into an agenda-based organization. We also demonstrated that the benefits for best-first search do come at the cost of optimality.

Chapter 8

Experimental Evaluation

This section reports on experiments on an implemented compiler¹.

8.1 Evaluation Criteria

To evaluate the effectiveness of the method introduced in this work, the following criteria are considered:

Abstraction Effectiveness: The principal goal of abstraction is to reduce complexity. In this context, abstraction is employed to reduce the size of the search space explored during compilation and hence the size of the resulting database. In this evaluation we consider the effectiveness of the abstraction by comparing the abstracted search space with the original extensional search space.

Compiler Effectiveness: For the compiler to be effective it must use a “reasonable” amount of time and space when generating a database. In particular, the additional complexity of working with abstractions must not negate any benefits. In this evaluation we consider how coverage over a given domain is accumulated with compile time for the two control strategies introduced in Chapter 7, breadth-first and best-first search.

¹The compiler is implemented in common lisp, and all experiments were run on a Sun SPARC2 station with 56M of real memory.

Utility of Compiled Knowledge: The principal goal of knowledge compilation is to improve problem-solving performance. Hence, it is important to evaluate the efficiency of using the generated databases to solve problems. Previous work has noted that due to inefficient matching and an overproduction of useless patterns, performance can actually degrade with learning. In this evaluation we test the cost of using the database as it grows.

Generality of Approach: It is important to assess the generality of the method, and in particular, to understand how the characteristics of a domain affect the method's effectiveness. In this evaluation we demonstrate the compiler in two different counter-planning domains.

To perform this evaluation we have chosen two similar counter-planning domains: chess and checkers. In particular, we have performed much of the evaluation with two sub-domains of chess: KRKN, where the pieces are restricted to a white king and rook against a black king and knight, and KRK, where the pieces are a white king and rook against a black king. These two endings are difficult to play well, even for experts, who have been shown to rarely play optimally or even correctly with problems from KRKN [Kopec and Niblett 80]. A further advantage with working with these endings is that they have been extensively studied for abstractions [Quinlan 83], [Bratko and Michie 80], [Bratko 84] and extensional databases are available for checking the correctness of the compiler.

8.2 Abstraction Effectiveness

There are two characteristics of a search space that measure its complexity: the number of states and the branching factor. In this evaluation we compare both quantities in the extensional space and abstracted space for KRK and KRKN chess endings. Figure 8.1 and Figure 8.2 illustrate the number of individual positions and the number of patterns generated by the breadth-first compiler for the KRK

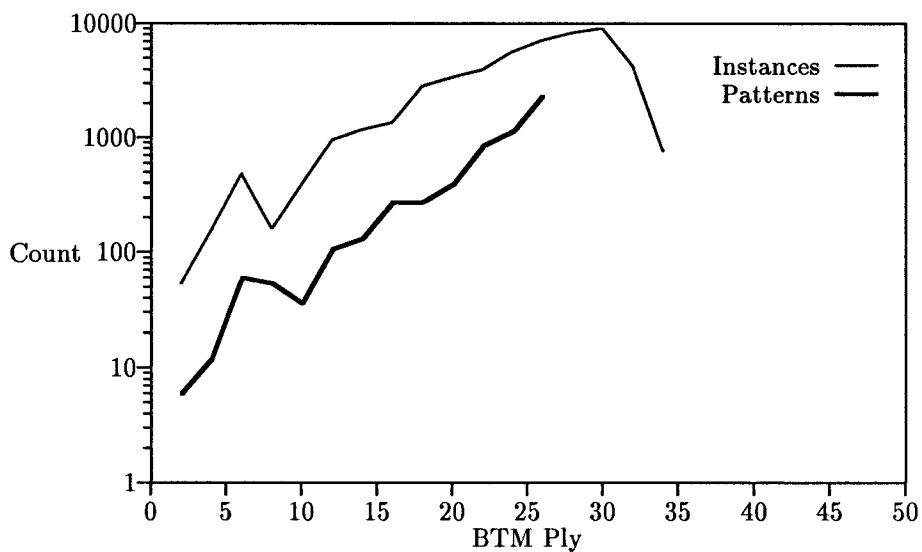


Figure 8.1: The number of *instances* as a function of ply for the problem KRK with black-to-move and lose compared to the number of *patterns* generated by the compile. Note the log vertical scale.

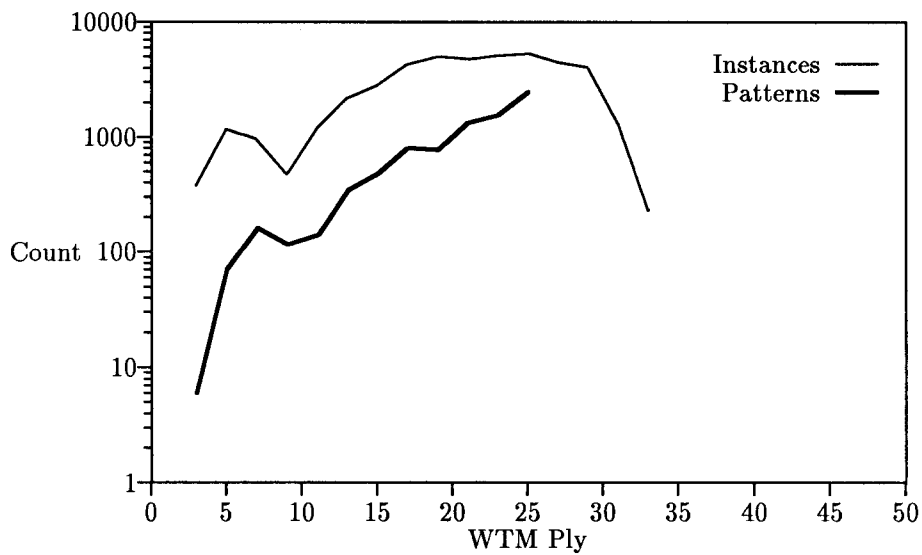


Figure 8.2: The number of *instances* as a function of ply for the problem KRK with white-to-move and lose compared to the number of *patterns* generated by the compile. Note the log vertical scale.

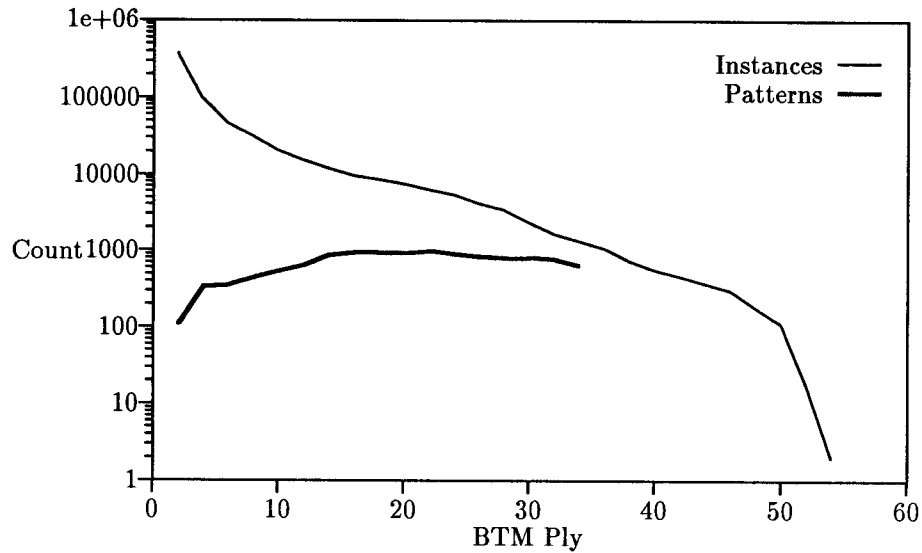


Figure 8.3: The number of *instances* as a function of ply for the problem KRKN with black-to-move and lose compared to the number of *patterns* generated by the compile. Note the log vertical scale.

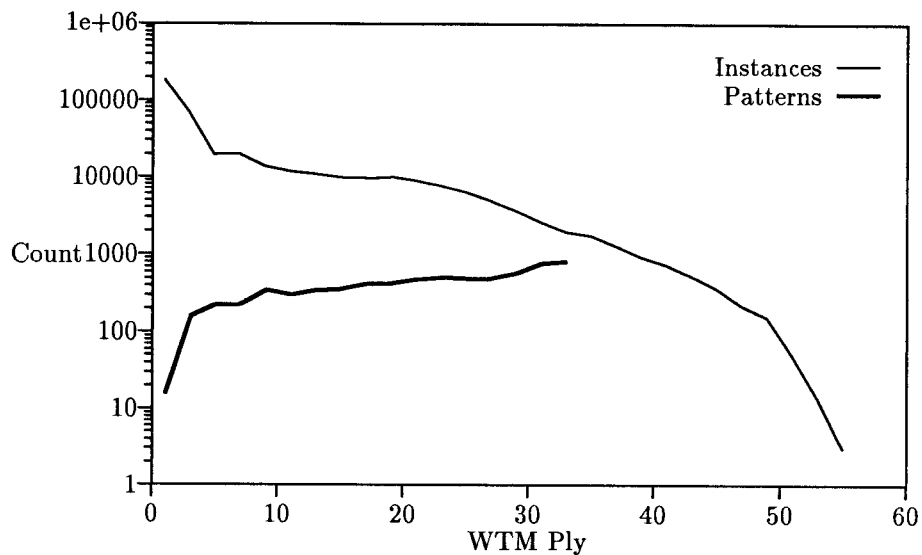


Figure 8.4: The number of *instances* as a function of ply for the problem KRKN with white-to-move and lose compared to the number of *patterns* generated by the compile. Note the log vertical scale.

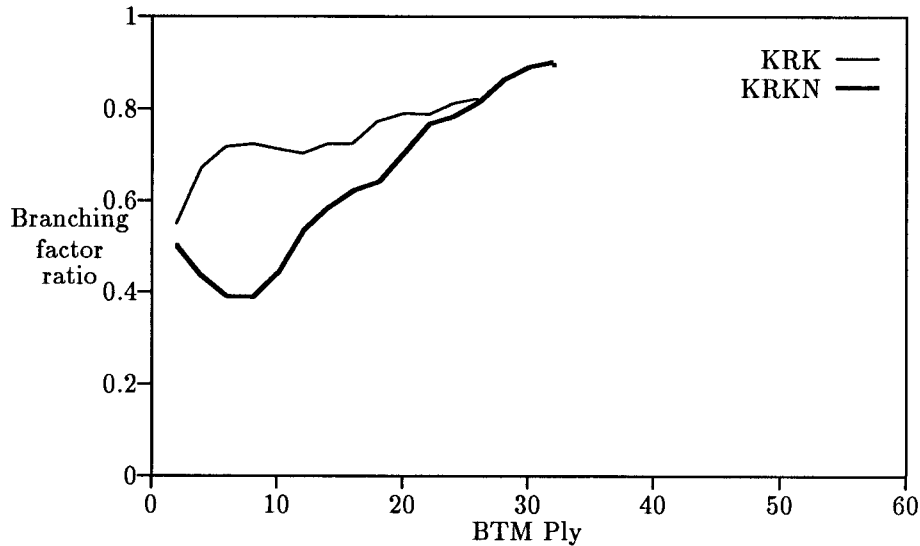


Figure 8.5: The average ratio between the branching factors of the abstract search space and the extensional search space as a function of ply for black-to-move. Both KRK and KRKN endings are shown.

ending. Figure 8.3 and Figure 8.4 illustrate the same information for the KRKN ending. Both graphs give the number of positions and patterns as a function of the ply of the position to a win or a loss.

There are a few significant features to notice in these graphs. First, the vertical scale is logarithmic, and hence, the difference between the two lines is a measure of the *ratio* between the size of the two spaces. Second, the line for the abstracted space does not extend to as deep a ply as the extensional space, which is given to the maximum possible ply. This is because of the large memory requirements of the abstract compiler and is discussed in Section 8.3. Third, the extensional data shown reflects a four-fold reduction obtained by taking into account rotational symmetry. Finally, note the distribution of instances with ply for both domains. In KRK, most of the examples are of larger ply, while the opposite is true for the KRKN ending. This characteristic of the domain is significant, since it influences the effectiveness of the compiler, and will be discussed in Section 8.3.

Figure 8.5 illustrates the effect of abstraction on the branching factor of the space for both chess endings. The vertical axis gives the average ratio between the extensional branching factor and the abstract branching factor as a function of ply. Only the branching factor for black—the losing side—is given. For a position in the KRKN ending when both black pieces are in the center of the board, the extensional branching factor is 16. This is reduced in some cases in the abstract database to 3 or 4, for example, where there is a threat on one piece and all moves of the other piece maintain the threat. This is the reason why in the KRKN ending, the effectiveness of the abstraction in reducing the branching factor initially improves with ply. At the start, the majority of the low ply patterns involve pieces trapped on the side, but as the ply increases, more patterns involve pieces in the center of the board, leading to a larger extensional branching factor and an improvement in the effectiveness of the abstraction.

8.3 Compiler Effectiveness

Figure 8.6 illustrates the compiler generating an abstract database for the KRK ending. The percentage of extensional instances covered in the domain is given as a function of elapsed time². Results when using the best-first and breadth-first control structure are shown. The steps in the graph for breadth first search, not present in the best-first graph, are caused by the compiler sequentially stepping through each ply. The four principal stages of the compiler described in Section 7.1 can be identified in the graph. The horizontal period of no growth in coverage at 23 signals that the compiler is analyzing new *Won-in-23-ply* patterns and producing new *make-true* and *maintain-true* influence relations with black-to-move. The following period of slow growth corresponds to the partial proof coverage and generation stage,

²For times less than 15000 seconds this time is approximately 100% of cpu time, since no other significant processes were running on the machine. After 15000 seconds real memory was exhausted causing cpu utilization to drop to less than 10%.

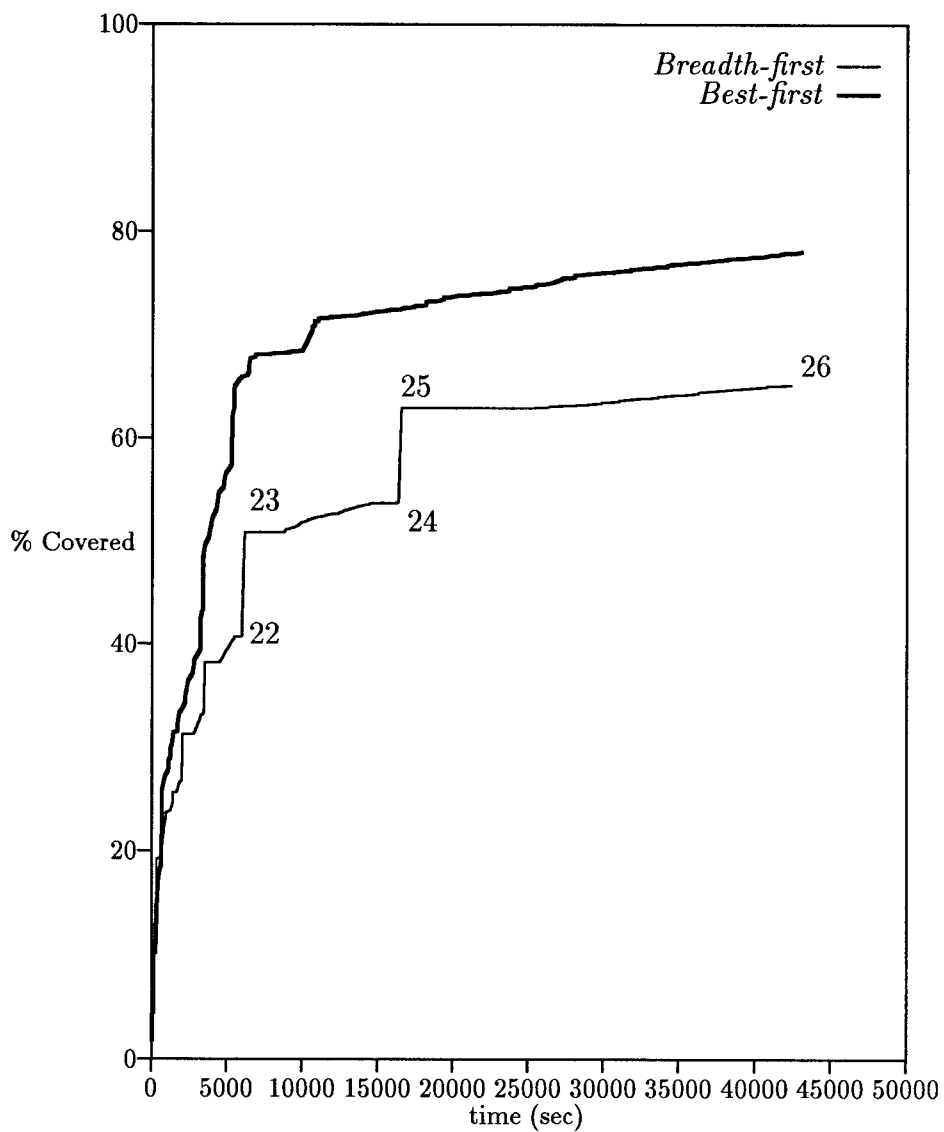


Figure 8.6: The percentage of extensional instances covered as a function of the compile time for the KRK problem when using *Breadth-first search* and *Best-first search*. The numbers marked on *Breadth-first search* are the Ply achieved.

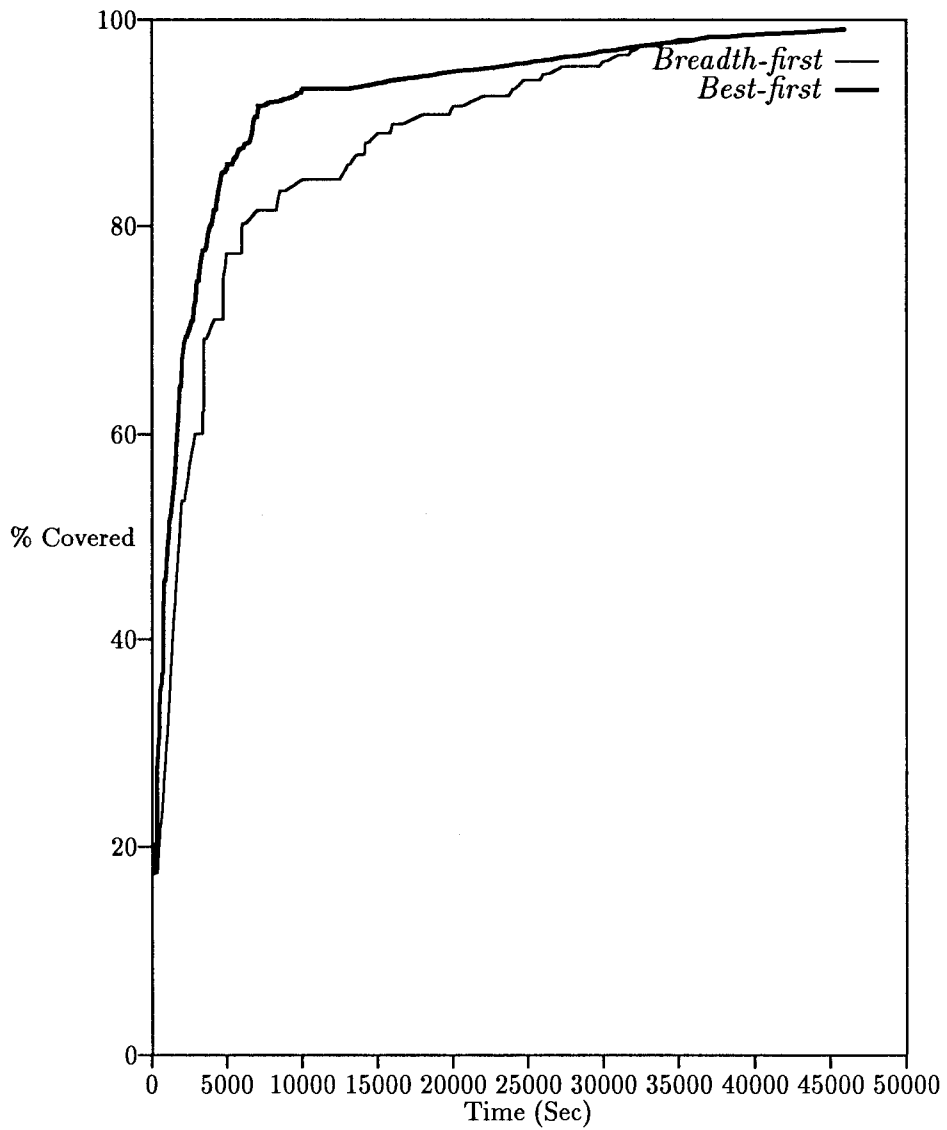


Figure 8.7: The percentage of extensional instances covered as a function of the compile time for the KRKN problem when using *Breadth-first search* and *Best-first search*.

where new loss patterns are derived, in this case we generate new *Loss-in-24-ply* patterns. The next period of no growth corresponds to the other analysis stage, where influence relations with white-to-move are compiled for the *Loss-in-24-ply* patterns. This stage produces a list of new non-optimal *Win-in-25-ply* patterns. The final period where coverage is rapidly accumulated is when the compiler optimizes the non-optimal wins by removing intersecting wins of lower ply. The new optimal wins are now available for a repeat of the first analysis stage that produces new *make-true* and *maintain-true* influence relations with black-to-play.

Figure 8.7 illustrates the compiler generating an abstract database for the KRKN ending. Here the steps in coverage accumulation during breadth-first control are not as pronounced due to the smaller number of patterns generated at each ply.

In both graphs and under both control strategies, the rate of coverage accumulation declines considerably after approximately 15000 seconds (≈ 4 hours) due to the compiler exhausting the real memory available. Swapping behavior is particularly inefficient because the dynamic program regularly sweeps through memory.

8.4 Analysis

An important characteristic of the compiler under either control strategy is the rapidity with which coverage is accumulated initially. This is more pronounced in the KRKN problem, where 80% coverage over the domain is achieved within only 50 minutes of run time. This rapid growth in coverage is an important advantage of the abstract approach compared with to the extensional approach. With an extensional database generator given the same patterns, no coverage is accumulated initially, since the compiler must first enumerate the patterns into instances in order to initialize the database. Following this stage, coverage is then accumulated linearly. While it is difficult to compare the two methods directly due to differences in implementation, these results show that the abstract database approach is most effective at accumulating coverage initially, while the extensional approach is least

Coverage Goal	KRK		KRKN	
	Breadth	Best	Breadth	Best
50	102	62	31	18
60	275	88	48	27
70	-	176	65	42
80	-	-	100	66
90	-	-	280	122
100	-	-	-	-

Table 8.1: Time in minutes needed to reach a coverage goal when using best-first or breadth-first search

effective initially.

In both cases the best-first search strategy accumulates coverage faster than breadth-first search. In the KRK ending, it enabled more of the domain to be covered in the time available. In the KRKN ending, the advantage was less pronounced. Table 8.1 gives the run times needed to achieve a given coverage for both search strategies.

In comparing the performance between the two sub-domains, it is interesting to notice that the compiler performs better in the KRKN ending, although that ending is larger than the KRK ending. This can be explained two ways. The first reason is that the distribution of the instances in the two endings is different. In KRK, the majority of the instances are of higher ply, while in KRKN the majority of the instances are of lower ply. Since the compiler works from low ply to high ply and must work within resource limitations, it does not have the opportunity to compile patterns for higher plys. Hence, only low coverage was achieved in the KRK ending because the compiler did not reach the higher ply patterns where much of the potential coverage exists. The second reason is that with more pieces, there

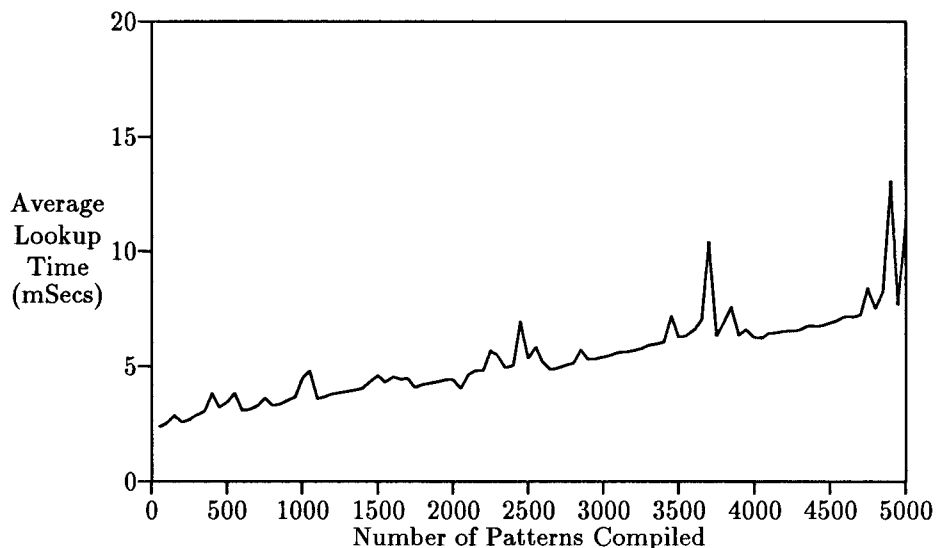


Figure 8.8: The average lookup time (in mS) as a function of the number of patterns produced by the compiler for the KRK ending

is more opportunity for generality in the patterns. Given a three piece pattern with coverage q , an additional piece has the potential to produce a pattern with coverage $61 \times q$. Indeed, this is exactly what has occurred when we compare the patterns produced in the KRK ending to those for the KRKN ending, especially in low ply patterns. For example, the KRK pattern describing a check-mate (such as in Figure 7.2) describes only 25 instances, while the equivalent KRKN loss pattern describes 1450 instances.

8.5 Utility of Compiled Knowledge

The compiled database is used to solve a given problem or play against an opponent by looking up the current instance in the appropriate database whenever the compiler must make a move. Hence, the principal cost during performance is lookup time in the database. Figure 8.8 gives the average lookup time for a random sample of instances as a function of the number of patterns stored in the database.

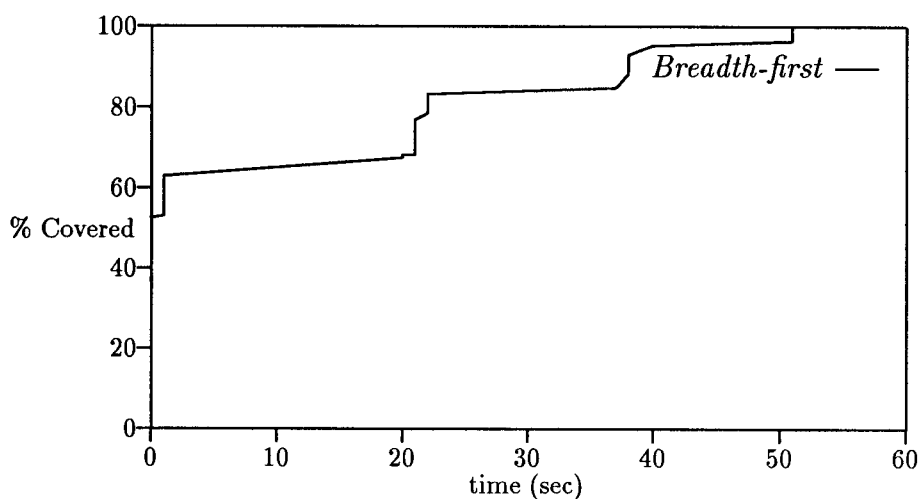


Figure 8.9: The coverage achieved as a function of run time for a king-man ending in checkers.

As would be expected, the lookup time grows with the size of the database. The observed linear rate is due to using a combination of the geometric indexing described in Section 5.4 and linear search. Only the locations of the first two objects in the pattern are indexed using the rectangle tree approach, the remaining test is performed by linear search. The lookup time of 5 to 10 mS means that it takes approximately 230mS to produce a 34 ply solution sequence for a difficult problem in KRKN. Solution time is linear in search depth using the abstract database, rather than exponential with a brute-force problem solver.

8.6 Generality of Approach

It is difficult to accurately assess the generality of the approach presented here. The method only works in counter-planning domains and due to the restrictions of geometric representation, it appears suitable only for chess-like domains that are played on a quantized 2D playing surface. Other restrictions include the need for the operators to move objects in space and the requirement that the operators

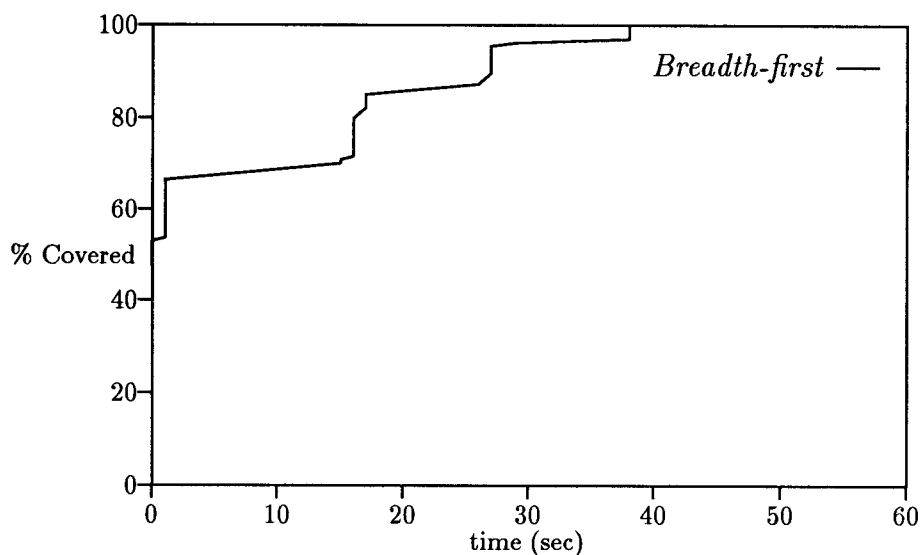


Figure 8.10: The coverage achieved as a function of run time for a king-king ending in checkers.

be describable as geometric constraints between the originating and destination locations. Although restrictive, most popular games satisfy these constraints.

Previous work in abstraction in counter-planning has been very problem specific, with abstraction methods developed for one ending unusable in other endings [Seidel 86]. Abstractions have even been limited to a given ply, with an abstraction developed for n ply failing to help in developing abstraction for $n + 1$ ply [Quinlan 83]. The abstraction mechanism presented here must be evaluated in this context, more fully described in Section 9.4. The fact that the abstraction mechanism works effectively in two different endings and across ply is therefore significant. Moreover, the initial encoding of the domain theories is simple and requires no special engineering, since it employs a generic geometric representation for describing objects in space.

To further demonstrate the generality of the method, the compiler was applied to some simple endings in checkers. This game is similar to chess in that it is played on the same board, but uses only 32 of the squares. The moves of the

pieces can be described as linear constraints relating the originating and destination locations, as in diagonal king moves in chess. The results are given in Figure 8.9 for the ending where a king wins against an opponent's man and Figure 8.10 for the ending where a king wins against an opponent's king.

Chapter 9

Related Work

This chapter describes related work in knowledge compilation and abstraction in counter-planning. The first section discusses knowledge compilation approaches to improving search-based problem solvers. Emphasis is on techniques like the one introduced in this thesis that aim to completely eliminate search during performance. The second section discusses abstraction methods in counter planning. Emphasis is on applications in chess or related games.

For reasons of brevity and focus, this chapter discusses only topics that are directly related to this thesis. Thus, example-driven improvement of problem solvers, such as explanation-based learning approaches [Minton 88a], are not discussed. Also excluded is work in automatically generating abstractions for problem solvers in single agent domains, since none of these approaches are successful in counter-planning domains. See [Tenenbergs 87] and [Knoblock 90] for a review of these approaches.

9.1 Knowledge Compilation

Knowledge compilation is a broad field concerned with improving the performance of intelligent systems. This review considers only knowledge compilation approaches applied to problem solvers that either employ methods similar to those described in the thesis or have the same goal of eliminating problem solving search at run time.

[Braudaway and Tong 89] presents a compilation method that improves a generate and test problem solver in the domain of floor plan design. Initially, good designs are simply stated as constraints that can be used as a test to reject or pass candidate designs. Knowledge compilation seeks to improve performance by incorporating some of the constraints from the test into the generator. This work is related to the abstract enumerator in that both systems employ geometric representations to encode constraints. In [Braudaway and Tong 89] a floor plan is represented as a set of linear inequalities among the coordinates of the rooms, while in the enumerator patterns are represented as sets of linear equalities among the coordinates of the playing pieces. The computational advantages of geometric representations (discussed in Chapter 6) are demonstrated in this work, since test incorporation becomes enumerating and simplifying sets of linear equations—the same process described in Section 5.2 to compute the influence relations.

[Schoppers 87] present a compilation method that eliminates costly planning for a robot trying to achieve goals in an unpredictable environment. The compiler constructs a “reactive plan,” which is a database of problem-situation/best-action pairs, that quickly recommends the best action to take in any situation that the robot may encounter. This reactive plan serves the same function as the abstract database constructed in this thesis and has the same principal advantage: fast performance time. The planning domain differs from the chess domain since in planning there are many possible goals that could be achieved and the reactive plan needs to consider these goals in addition to the current situation when deciding which action to take. The database is constructed using a simplified reverse enumeration method suitable for single agent domains. Given some goal to achieve, situation patterns are created by back-chaining with actions that could achieve the goal, and enumerating cases depending upon whether the preconditions are assumed true or false. The method has been applied successfully to simplified blocks world problems, but suffers from the same exponential growth in database size discussed in Chapter 1.

[Ginsberg 89] presents a criticism of [Schoppers 87] and the reactive plan approach in general. This criticism applies equally to the approach in this thesis, since the abstract databases constructed are equivalent to reactive plans. The principal problem with the approach according to Ginsberg is the potential size of the reactive plan, since it is at least an exponential function of the number of features with which the situations are described. He presents a simple counting argument similar to the one made in Chapter 1 to show this. In companion articles defending their positions and refuting Ginsburgs arguments, [Schoppers 89] and [Chapman 89] argue that the assumptions underlying the counting arguments are flawed and present evidence that in some cases simple reactive plans can be constructed to solve complex problems. [Schoppers 89] argues that the mapping represented in a reactive plan is not arbitrary and any regularity present can be exploited to considerably reduce its size. This argument is true in chess endings as the results in Chapter 8 demonstrate. However, the exponential growth cannot be completely eliminated and some combination of search or planning with reactive plans may be necessary as advocated by Ginsberg.

[Chapman 89] presents an implementation of a reactive system that effectively solves a toy problem (the fruitcake problem) which Ginsberg argued (above) could not be solve by reactive plans. The system employs a specialized visual representation of situations that overcomes the problem with the size of universal plans. The abstraction achieved through visual representations is very powerful since it employs visual markers that denote functionally significant objects in the situation and avoids the intractability of variable binding needed in logical representations (see [Agre and Chapman 88] for more details). The method has been applied successfully to activities such as playing video games [Agre and Chapman 87] where abstraction is achieved by employing visual markers on objects such as “the-bee-that-is-chasing-me.” Similar abstractions are suggested in chess where functionally significant objects such as “the-piece-that-is-threatening-my-king” or “the-piece-

that-is-blocking-my-pawn” are relevant. Although the approach appears to avoid the problem with exponential growth, these abstractions to date have been manually engineered. This suggests that an important research problem is to construct compilers that can automatically derive such visual representations directly and avoid the construction of universal plans or abstract databases.

9.2 Abstraction in Counter-planning

The importance of abstraction in counter-planning was recognized in [de Groot 65] and [Chase and Simon 72], where it was determined that human experts represent their knowledge in *chunks*—patterns that describe significant arrangements of playing pieces. Much of the work in abstraction in games has focused on the role of these patterns in problem solving. Two issues have been emphasized: (a) How are the abstractions effectively exploited during problem solving? and (b) What do the abstractions represent and how are they derived? Although these two issues are difficult to view in isolation, the following review is partitioned by considering each question in turn. The first section considers work that emphasizes the use of abstractions in problem solving, the second section considers work that emphasizes the form and derivation of abstractions.

9.2.1 Abstraction in Problem-Solving

[Wilkins 80] presents a system called PARADISE that solves complex mid-game positions in chess using high-level abstractions and flexible planning. The emphasis was on solving tactical problems where the losing player was strongly constrained by multiple threats. All the abstractions were hand-engineered and involved terms such as “can-safely-move-a-piece” and “safely-capture-a-piece.” The principal achievement of this work was to demonstrate the effectiveness of abstractions in reducing the search space needed to solve complex problems by 4 to 5 orders of magnitude, to around 150 positions—a number comparable to that searched by

human experts.

[Campbell 88] presents a system for solving difficult problems in pawn endings through an abstraction mechanism based on chunking. In pawn endings, because of the limited freedom of movement, a problem can often be decomposed into subproblems based on the proximity of pawns. This work presents a technique that groups nearby pawns and kings into chunks and then solves the problem by considering the interactions among pieces in a chunk independent of the interactions among chunks. The behavior of a chunk is characterized by computing abstract features such as the number of pawns in the chunk, whether one side has a tempo move, whether a pawn can queen or whether the king can capture enemy pawns without allowing a pawn to queen. These properties of the chunks present in a given problem determine the space of specialized plans that is searched. Although the grouping mechanism employed to form the chunks and the abstract features computed were carefully engineered, this work demonstrates how abstraction can effectively reduce the search space in chess pawn king endings enabling errors to be identified in published play.

[Tadepalli 89] presents an integrated problem-solving and learning system that is capable of automatically learning useful abstractions from problem solving experience and effectively using that knowledge to solve problems in king pawn endings. The learning method analyzes training examples that are carefully chosen to exercise gaps in the system's knowledge. The main focus of this work was to overcome the inherent intractability of problem solving and learning in counter-planning domains, due to the need to consider all possible actions by the opponent. Two ideas are explored: *lazy explanation-based learning*, which learns from incomplete explanations and then incrementally corrects the errors that are necessarily introduced, and *optimistic counter-planning*, which flexibly combines chunks or macros learned from small problems to solve larger problems. A difficulty with the approach is that the system does not know when its knowledge is incorrect and depends upon an

external agent to identify and correct (through supplementary training) any errors detected. Since these errors are caused by an incomplete search, this implies that the external agent must have performed a more thorough search than the system. The principal achievement of this work is that it introduces a mechanism to automatically learn useful abstractions, such as removing protection and freeing a lane for queening, with the minimum of initial domain theory engineering.

[Bratko and Michie 80] present two carefully engineered systems that solve most problems in the KRK and KRKN endings correctly but non-optimally. The emphasis is on encoding sufficient knowledge in the form of advice that can be combined with some limited search at performance time to effectively solve most problems. Advice is a set of patterns that when true recommend goals that should be achieved and maintained during look ahead search. An example of advice from the KRK ending is the following: when the area where the opponent's king is safe is decreased and the opponent's king cannot attack our rook and the rook divides both kings and stalemate is not possible, then look for a way to further constrain the safe area of the opponent's king. The advice language incorporates abstraction in two powerful ways. First, the patterns that determine which advice is relevant partition the problem space into large equivalence classes. Second, the right hand sides of the advice provide only constraints on the moves rather than particular moves to make. For these reasons, only 5 rules incorporating 9 features were needed to solve problems in the KRK ending, while only 12 rules incorporating 14 features were needed for the KRKN ending. Although it was not clear how much of the two domains each advice table covered or by how much the resulting play extended the optimal play, this work illustrates the power of manually designed abstractions in concisely capturing complex problem solving knowledge.

9.2.2 Abstraction Form and Derivation

[Seidel 86] presents a method for generating an abstract database for the KRK ending that is correct but non-optimal. The method is similar to the one presented in this thesis, since it generates the abstract database by reverse enumeration from initial patterns. However, it is less general since it requires a specialized encoding of the board in terms of a ring structure, suitable only for the KRK ending. Losing patterns are derived by demonstrating that all moves of the king lead to known winning patterns, while winning patterns are derived by applying one of a hold transition (a waiting move), a rook production, or a king production to known loss patterns. The method developed iterative schemes for patterns that are lost or won in greater than 8 ply. For non-iterative patterns, the count of patterns derived for each ply was approximately half that reported in Chapter 8 (due to the use of reflective symmetry).

In Chapter 6 of [Michie, 82], a review of manually engineered abstractions in the KRK ending is presented as well as a method that automatically generates an abstract database for the ending when played on an infinite board with only one corner. This method, although limited to a simplified problem with only three pieces, formed the basis for the one presented in this thesis. The method performs backward abstract enumeration by generating new patterns from preimages of existing patterns. Patterns are represented geometrically, by x and y offsets from the black king enabling efficient indexing. Iterative schemes are produced for problems that are won or lost in greater than 6 ply. The procedure produced a small table of 11 patterns that completely describe an optimal and correct strategy to win the ending from any opening position. The principal disadvantage of the method is its limitation to the one simplified ending.

[Beal and Clark 1980] presents a semi-automated method for generating an abstract database in the KPK ending that can correctly determine whether a given position is a win or a draw. The first step in the method was to manually identify

non-stalemate termination patterns and repeated patterns that enable the pawn to advance one square safely. The second step in the method was to semi-automatically derive patterns that led to these known patterns via a backing-up procedure. This backing-up procedure manipulated patterns consisting of many specially chosen geometric features such as “the maximum of the row distance or the column distance,” and “the number of king moves needed to reach the target square taking squares blocked by the pawn into account, but not squares blocked by the other king.” The complexity of geometric features made the backing-up procedure impossible to fully automate, but led to patterns much more general than were derived by the method presented in this thesis. The authors comment that “... these descriptions were generated *ad hoc* as the need for them arose. No systematic deductive method of obtaining them is apparent.” This thesis presents such a method for determining appropriate descriptions using a generic and much simplified geometric vocabulary.

[Quinlan 83] presents a method for learning the concept *lost-in-n-ply* for small n for the KRKN ending. The concept was learned by applying the ID3 inductive learning algorithm to examples described by a set of manually engineered abstract features. The bulk of the effort in achieving correct performance was in the engineering of these features. Quinlan estimates that 2 man-weeks of work was required to produce the *lost-in-2-ply* features, 3 man-months of work was required to produce the *lost-in-3-ply* features while efforts were abandoned for *lost-in-4-ply*, since the problem was considered too difficult. The difficulty of engineering suitable abstractions in chess is well illustrated by this work, especially considering that during this development process, the complete database of all example positions was available to evaluate features that were developed. [Quinlan 83] concludes with some approaches to automatically deriving useful features through clustering of training examples based on geometric constraints over the coordinates of the pieces involved.

[Utgoff 86] presents an approach for generating concepts in chess using a method from single-agent domains known as constraint-back propagation. The

method uses a sequence of moves that terminates in a recognized goal such as safely capturing a piece. To derive new concepts, this terminating goal is “back-propagated” through a generalized form of the original move sequence. In this way, concepts that indicate goal achievement in the future can be derived. The method is not easily extended to counter-planning, where it is difficult to back-propagate goals through the losing player’s move, since to ensure correctness, all possible such moves must be considered. To avoid these problems, the only conditions where the method can be applied is when the losing player is restricted to only one move—a forced move. Even then, moves that could exist in the generalized concept, but were not present in the example, must be considered if they could affect the remaining move sequence. For example, if in the given move sequence the losing player was forced to move away, allowing the winning player to capture, the back-propagation mechanism must consider the option for the losing player to take the capturing piece, even though these moves were not observed in the move sequence. [Minton 84] reported on a similar method that required additional “forcing conditions” to be conjoined with the derived patterns to ensure that only one losing move was possible.

[Muggleton 88] presents a method for automatically deriving chess strategies by grammar induction over optimal and correct move sequences. To demonstrate the approach, a sub-problem was chosen from the extremely complicated KBBKN ending where a trapped bishop must be freed from a corner in less than 12 moves under any opponent responses. A suitable vocabulary of attributes and actions with which to describe the strategy was first developed with the help of A. Roycroft—a chess expert. Only four relatively simple attributes were needed, including whether white was free to take the black knight and whether the white king is on the same diagonal as the release position (where the bishop can escape). Four actions were needed including white taking the knight and white moving the king towards the release position. Following careful choice of training sequences, the induction algorithm was able to derive a small set of rules that could solve the problem and

were easily understood by the chess expert. While this work demonstrated that sequence induction algorithms can succeed in deriving generalized strategies from correct move sequences, the method still requires considerable vocabulary engineering by human experts who understand the target strategy. Thus, the generality of the approach cannot be determined from this single case.

[Gould and Levinson 1991] introduces a method called *experience-based learning* applied to improving a chess playing system which looks ahead only one ply. The method combines a variety of traditional machine learning approaches including genetic algorithms, evaluation function learning, temporal difference learning and simulated annealing. What is most interesting about this work is the lack of careful vocabulary engineering needed to represent chess patterns, in contrast to almost all previous work reviewed. Chess patterns are represented as subgraphs where nodes represent pieces and edges represent simple attacking or defending relationships. Little other chess-specific knowledge is build in. To improve problem-solving performance, new patterns are created from analysis of played games and more importantly, patterns are assigned an evaluation depending on whether they took part in successful or unsuccessful play. Fundamental concepts such as the disadvantage of being a piece down were automatically discovered. The principal limitation of the approach is the simplicity of the problem solver being improved (only one ply look ahead) and the reliance on an external agent who currently must be a better player.

[Fawcett and Utgoff 1991] presents a method that aims to overcome the principal weakness of much of the work in abstraction in games—the need to carefully engineer an appropriate vocabulary. The method has been applied to the game of Othello and automatically generated features such as semi-stable square and frontier that had been manually engineered in previous systems [Rosenbloom 82]. The method takes a transformational approach to feature generation where new features are derived by applying transformations (such as decomposition, abstraction,

regression and specialization) to existing features. Initial features are derived from the defined goals of the problem domain. The generality of the approach has been well demonstrated by applying the method to two distinct domains, Othello and telecommunications network management. However, it is not clear how successful the method would be in a complex game like chess or how dependent the method is on the initial encoding of the operators and the goals of the domain. For example, it is difficult to see how many interesting features could be derived when the goal in chess is defined as simply capturing the opponent's king.

9.3 Summary

This chapter has reviewed previous work in knowledge compilation methods that aim to eliminate problem-solving search and abstraction methods applied to counter-planning domains with emphasis on chess. This review of knowledge compilation approaches has reinforced the need to consider more flexible target problem-solvers, rather than the reactive plan approach explored in the thesis. Our review of abstraction methods has demonstrated the extreme difficulty in successfully designing abstractions manually, yet illustrated the importance of abstraction in simplifying problem-solving in counter-planning domains.

Chapter 10

Conclusions and Future Work

10.1 Summary

The reverse enumeration approach has proved useful in generating efficient and correct problem solvers in counter-planning, but at a cost exponential in problem size. This thesis has investigated modifying the approach to alleviate this exponential growth. In particular, this thesis has introduced the following:

- *An abstraction mechanism* that enables the reverse enumerator to use *patterns*, rather than instances. The patterns are defined by the influence theory, where each pattern is an equivalence class of instances with respect to the goal achieved and the tactic used. A geometric representation of patterns is employed to ensure that the computation involved is polynomial.
- *A best-first control structure* that optimizes the resources used by the reverse enumerator. The compiler attempts to find the most general and simplest patterns first thereby maximally exploiting the tradeoff between the time spent compiling a domain and the coverage achieved over the domain.

10.2 Contributions

The following is a list of major contributions of this thesis:

1. A method for automatically determining correct abstractions in counter-planning. This is significant since in the past counter-planning abstractions have either been laboriously hand engineered [Quinlan 83] or engineered semi-automatically with the aid of a computer [Michie, 82], [Muggleton 88]. Moreover, weak abstraction approaches, which have proved successful in single agent planning domains [Knoblock 90], are ineffective in counter-planning.
2. A method that demonstrates the effectiveness of a general purpose geometric representation for capturing and efficiently processing abstractions. Without the use of geometric representations, the compiler would be hopelessly intractable and the performance of the compiled database would degrade severely as its size grew to hundreds of patterns. This result strengthens the arguments made in [Braudaway and Tong 89] and [Levesque 86].
3. A method for optimizing the performance of a knowledge compiler by heuristically controlling backwards search to prefer generating the simplest and most general patterns. This approach is effective in domains that are so large that it is impractical to run the compiler to completion.
4. A demonstration of the 80/20 phenomenon in chess endgames and its exploitation in an abstract database compiler. This phenomenon enabled the compiler to quickly construct a small database of patterns that covered most of a domain. The 80/20 rule has been demonstrated in other domains where the goal has been to construct abstract problem-solvers [Agre and Chapman 87] and [Kaelbling 90]. It is not clear whether the phenomenon can always be exploited, nor is it clear what characteristics of a domain lead to this phenomenon.

10.3 Limitations of the Thesis

A principal weakness of the method is the ineffectiveness of the abstraction mechanism when the depth of the pattern grows. In all examples it was found that the coverage of the patterns tended to decrease with depth; eventually the abstraction mechanism became completely ineffective as the patterns became instances. This effect was moderated by the trend for the count of instances at each depth to diminish in the KRKN ending and others.

Another weakness of the method is its sensitivity to the initial representation of operators and patterns. In the cases where the terminal patterns were specific (as in the KPK ending) the abstraction produced only specific patterns. A specific representation of operators compounds the problem. Notice that in the domains that involve sliding pieces (such as the rook in the KRKN ending) the compiler produced effective abstractions, while in domains that involve only non-sliding pieces (such as the KPK ending) the compiler produced only specific patterns. The sliding operators are more abstract (4 cases describe 14 moves for the rook) than the non-sliding operators (8 cases are needed to describe the 8 moves of the king) and hence contribute to more abstract patterns.

The growth in complexity is a further weakness. In particular, some domains lead to low-utility patterns that exhibit rapid growth in the number of exceptions, as in branches of the KRKN ending. This complexity arises because of the need to project negated constraints among the objects: the black knight must not be able to take the white rook, or the black king must not be adjacent to the black knight when it is captured by the rook.¹ Applying best-first search in place of breadth-first search did not solve this problem, rather it avoided the problem by not generating these complex patterns.

There are a number of other shortcomings with the current approach. First, only problem solvers that involve goal achievement, such as check-mate or safe-

¹Except when the white king is adjacent to the black knight!

knight-capture, can be compiled. This kind of goal is unrealistic, even for simple games. Most useful is the general goal of improving the evaluation of the current position. However, this would require major modification to the influence theory where operators increment or decrement an evaluation rather than make-true or make-false a propositional goal. Finally, a significant shortcoming is that the method still will not scale to problems that involve many pieces such as the opening and midgame phases of chess. This problem, and others are considered in the following future work section.

10.4 Future Work

This section considers future work from two perspectives. First, we cover pragmatic issues that modify the method or current system so it could contribute to a complete game playing system. Second, we address interesting research issues that are suggested by our research.

10.4.1 Pragmatic Issues

Improving the Current Program

The current program was developed incrementally to demonstrate the viability of the approach, and it is inefficient. In particular, the program is memory intensive and runs out of real memory quickly. Moreover, the program utilizes virtual memory poorly because of the behavior of the dynamic program sweeping through memory. A more efficient, less memory-intensive implementation would enable further investigation.

The current program employs a simple geometric representation that cannot *concisely* represent the complex constraints that arise. Complexities are avoided by giving up generality: complex cases are enumerated into a sufficient number of simple cases. For example, the preimage of a king move is represented by 8

separate cases, rather than a single constraint describing the region from which the king could have moved. A more powerful geometric representation and reasoning system that could manage the resulting regions would improve the effectiveness of the abstraction. Such a representation would also address the problem of too many exceptions by enabling many of the exceptions to be *incorporated* into the regions, thereby eliminating the need to represent them explicitly. Quad trees and Box trees [Omohudro 90] both provide both efficient reasoning and a versatile, yet concise representation of irregular shaped regions.

Reimplementing the Method in Parallel

The limit imposed on the extensional enumeration algorithm (due to its exponential time and space complexity) has been advanced in the last few years through parallel processing. In 1986, [Thompson 86] used 12 IBM machines in parallel to successfully generate the complete database for some 5 piece endings. Each database took approximately 2 months (24 cpu months) and occupied 120M of memory. More recently, Scientific American² reports that 6 piece endings have been compiled by implementing the algorithm on a connection machine and exploiting the massive parallelism available. This work identified a 446-ply forced win, the longest ever discovered. The enumeration algorithm lends itself to easy parallelism due to the simple decomposition of problems by position index. The abstract algorithm described here has many of the same characteristics and could be similarly decomposed. The abstract geometric indexing poses more of a problem than in the extensional algorithm where the address structure of the machine can be used directly.

²November 1991, Page 38. The program was implemented by Lewis Stiller from John Hopkins University.

Extending the Domains

This thesis has reported applying the method to two quite similar domains only. To demonstrate the generality of the approach, more domains are needed. Of particular interest is the recent work by [Pell 92a] and [Pell 92b] that describes a meta-game framework, where thousands of different “chess like” games can be automatically produced. Each game is defined by a set of rules generated by setting parameters of the meta game.

10.4.2 Research Issues

Problem Solver/ Learning Tradeoff

This work has employed a very simple problem solver that performs no problem solving search during performance. Such a simple problem-solver is very efficient at run time, but leads to the need to store large databases. This tradeoff can be illustrated with an example from the KRKN ending illustrated in Figure 10.1. In this position white can play and win in 5 ply (**d1-e1, d7-e6, f3-h3, e6-f5, h3-h1**). This instance is an example of a pattern illustrated in Figure 10.2, where all but the black king are constrained to be in fixed locations. The region of the black king was calculated to prevent it from interfering with any of the other pieces. This pattern has low generality, because of the complexity of this interference. The compiler derives a unique pattern for each position of the white king moving to **e1** (**d1, d2**), because each leads to a different region constraint for the black king. Generally, where there is any interference between kings (by threatening the rook or protecting the knight), or other pieces, the effectiveness of the abstraction is diminished. The compiler is forced to enumerate each possible case of the interactions leading to an explosion of specific cases.

To avoid this explosion, a new kind of abstraction is needed: *computational abstraction*, where the enumeration of interactions is postponed until problem solv-

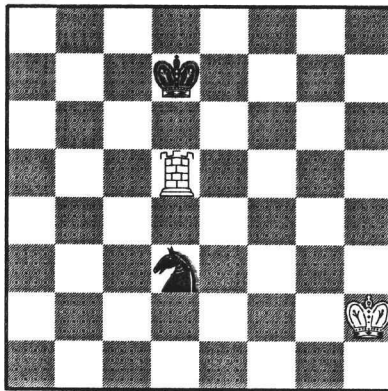


Figure 10.1: Example of a white-to-move-and-win position from the KRKN ending.

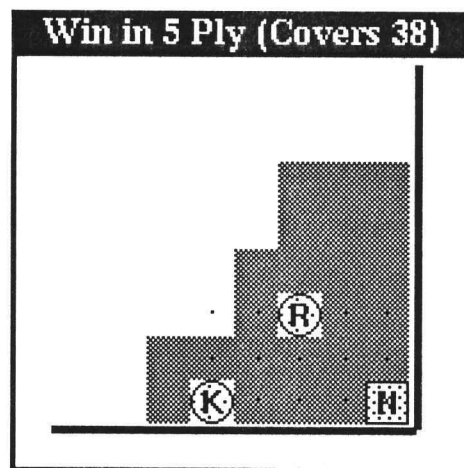


Figure 10.2: A pattern that is generated by the compiler that describes the position illustrated in Figure 10.1. Note that the white king, black knight and white rook are in fixed positions and the black king is prohibited from the gray region.

ing time. In this way, only a few general abstractions such as “can move safely to square e1” are recorded. This postponement of the computation complicates the problem solver and reduces its efficiency, but it leads to a much more concise description of the problem solving knowledge. In [Bratko and Michie 80], for example, a non-optimal problem solver for KRK was constructed using only 12 rules, but the problem solver needed to look at least one move ahead. This trade-off between the complexity of the problem solver and the simplicity of the learned knowledge has been explored by chunking [Campbell 88] and optimistic counter-planning [Tadepalli 89]. In [Campbell 88] a problem instance was first partitioned into chunks based on pawn structure. The problem was solved efficiently by considering the interactions within chunks separately from those interactions among chunks. In [Tadepalli 89] the chunks are learned from problem solving experience and dynamically combined using a plan language during problem solving.

Computational abstraction is an important method, since it offers a way for problem solving knowledge concerning only a few pieces to be effectively used to solve problems involving many more pieces. Hence, the method may provide a way to overcome the fundamental problem with intractability that the abstract database approach cannot address directly.

The particular question of interest in this thesis is the following: given a problem instance P containing a set S of objects, how can we solve P using a collection of n databases d_i , each covering some subset of S where $d_1 \cup d_2 \cup \dots \cup d_n = S$? The influence theory introduced in Chapter 2 may be of use. Recall that the theory provides a language for describing goal achievement that is unfolded and evaluated during compilation. Hence, the database approach takes an extreme of the space/time tradeoff where all search (time) is compiled out. This view suggests that the dynamic combining of small databases to solve large problems may be modeled as the same abstract search performed by the compiler, strongly constrained by the particular patterns in the given problem instance.

A further contribution that this work may make to the computational abstraction approach is in the use of geometric representations for efficiency. Dynamically combining chunks has proved to be computationally expensive [Tadepalli 89] and geometry may provide a means of reducing its cost.

Bibliography

- [Agre and Chapman 87] Agre, P. & Chapman, D. (1987). Pengi: an implementation of a theory of activity. *Proceedings of the Sixth National Conference on Artificial Intelligence* (pp. 268-272). Seattle, WA: Morgan Kaufmann.
- [Agre and Chapman 88] Agre, P. & Chapman, D. (1988). Indexicality and the binding problem. *Proceedings of the Spring Symposium on Parallel Models of Intelligence: how can slow components think so fast?* Stanford University.
- [Amarel 80] Amarel, S., (1981). On representation of problems of reasoning about actions. *Readings in Artificial Intelligence*, Palo Alto, CA: Tioga Publishing Company.
- [Anantharaman Campbell and Hsu 88] Anantharaman, T., Campbell, M. and Hsu, F. (1988). Singular extensions: adding selectivity to brute-force searching. *Proceeding of the Spring Symposium on Computer Game Playing*, Stanford University.
- [Beal and Clark 1980] Beal, D. F. and Clark, M. R. B. (1980). Economical and correct algorithms for King and Pawn against King. M. R. Clarke (Ed), *Advances in Computer Chess*.
- [Bellman 57] Bellman, R. E. (1957) *Dynamic Programming*. Princeton University Press, Princeton, NJ.
- [Berliner 74] Berliner, H., J. (1974). *Chess as problem solving: the development of a tactics analyzer*. Ph.D. thesis, Carnegie-Mellon University.
- [Boddy 91] Boddy, M. (1991). Anytime problem solving using dynamic programming. *Proceedings of the Ninth International Conference on Artificial Intelligence*, CA.
- [Bramer 84] Bramer, M. A. (1980). An optimal algorithm for King and Pawn against King using pattern knowledge. M. R. Clarke (Ed), *Advances in Computer Chess 2*.

- [Bratko and Michie 80] Bratko, I. and Michie, D. (1980). A Representation for pattern-knowledge in chess endgames. M. R. Clarke (Ed), *Advances in Computer Chess 2*.
- [Bratko 84] Bratko, I. (1984). Advice and planning in chess end-games. *Artificial and Human Intelligence*. Elsevier Science Publishers.
- [Braudaway and Tong 89] Braudaway, W. and Tong, C. (1989). Automated synthesis of constrained generators. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, Michigan.
- [Brooks 81] Brooks, R. A. (1981). Symbolic reasoning among 3-D models and 2-D images. *Artificial Intelligence*, 17 (1-3), 285-348.
- [Campbell 88] Campbell, M. (1988). *Chunking as an Abstraction Mechanism*, Ph.D. Thesis, Carnegie Mellon University.
- [Chapman 89] Chapman, D. (1989) Penguins can make cake. *Artificial Intelligence Magazine* 10 (4), 45-50.
- [Chapman 87] Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence*, 32 (3), 333-378.
- [Chase and Simon 72] Chase, W. G. and Simon, H. A. (1972). The mind's eye in chess. *Visual Information Processing*, Ed. W. Chase.
- [Chien 89] Chien, S. (1989). Using and refining simplifications: Explanation-based learning of plans in intractable domains. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 590-595, Detroit, MI.
- [Clark 1977] Clark M. R. B. (1977). A quantitative study of king and pawn against king. M. R. B. Clark (Ed), *Advances in Computer Chess 1*, 30-59.
- [Cohen 90] Cohen, W. (1990). *Concept Learning Using Explanation-Based Generalization as an Abstraction Mechanism*. Ph. D. Thesis, Rutgers University.
- [Collins *et. al*] Collins, G., Birnbaum, L. and Krulwich, B. (1989) An adaptive model of decision-making in planning. *Proceedings of the Eleventh International Conference on Artificial Intelligence* (pp. 511-516). Detroit: MI.
- [Davis 87] Davis, E. (1987). Constraint propagation with interval labels. *Artificial Intelligence*, 32 (3), 281-332.
- [de Groot 65] de Groot, A. (1965) *Thought and Choice in Chess*. The Hague: Mouton, 1965.

- [DeJong and Mooney 86] DeJong, G., and Mooney, R. (1986). Explanation-based learning: an alternative view. *Machine Learning* 1(2), 145–176.
- [Doyle 86] Doyle, R., J. (1986). Constructing and reining causal explanations from an inconsistent domain theory. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 538-544). Philadelphia, PA: Morgan Kaufmann.
- [Ebeling 86] Ebeling, C. (1986). *All the right moves: A VLSI architecture for chess*. Ph.D. thesis, Carnegie-Mellon University, 1986.
- [Edelsbrunner 83] Edelsbrunner, H. (1983). A new approach to rectangle intersections. *International Journal of Computer Mathematics*, 13 pp. 209-219.
- [Egan and Schwartz 79] Egan, D. E., and Schwartz, B. J. (1979). Chunking in recall of symbolic drawings. *Memory and Cognition*, 7, 149-158.
- [Ellman 88] Ellman, T. (1988). Approximate theory formation: An explanation-based approach. *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 570-574, St. Paul, MI.
- [Epstein 90] Epstein, S. (1990). Learning plans for competitive domains. *Proceeding of the Seventh International Conference on Machine Learning*, pp. 190-197, Austin, Texas.
- [Etzioni 91] Etzioni, O. (1990). *A Structural Theory of Explanation-Based Learning*. Ph.D. thesis, Carnegie Mellon University, 1990.
- [Fawcett and Utgoff 1991] Fawcett, T. E. and Utgoff P. E. (1991) A hybrid method for feature generation. *Machine Learning: Proceedings of the Eighth International Workshop* (pp. 137-141). Evanston, IL: Morgan Kaufmann.
- [Flann 88] Flann, N. S. (1988). Improving problem solving performance by example guided reformulation of knowledge. *Proceedings of the First International Workshop in Change of Representation and Inductive Bias* (pp. 14–34).
- [Flann and Dietterich 89] Flann, N. S. and Dietterich, T. G. (1989). A study of explanation-based methods for inductive learning. *Machine Learning*, 4, 187-226.
- [Flann 90] Flann, N. S. (1990) Applying Abstraction and Simplification to Learn in Intractable Domains. *Proceeding of the Seventh International Conference on Machine Learning*, pp. 277-285, Palo Alto, CA: Morgan Kaufmann.
- [Freuder 82] Freuder, E. C. (1982). A sufficient condition of backtrack-free search. *JACM* 29(1), pp. 24-32.

- [Genesereth and Nilsson 87] Genesereth, M. R. and Nilsson, N. J. (1987) *Logical Foundations of Artificial Intelligence*, Los Altos, CA: Morgan Kaufmann.
- [Ginsberg 89] Ginsberg, M. L. (1989). Universal planning: An almost universally bad idea. *Artificial Intelligence Magazine*, 10 (4) pp. 40-44.
- [Ginsberg and Smith 88] Ginsberg, M. L. and Smith, D. E. (1988) Reasoning about action II: The qualification problem. *Artificial Intelligence*, 35 (3), 311-342.
- [Goetsch and Campbell 90] Goetsch, G. and Campbell, M. S. (1990). Experiments with the null-move heuristic. T. A. Marsland and J. Schaeffer (Eds.) *Computers, Chess and Cognition*.
- [Gould and Levinson 1991] Gould, J. and Levinson R. (1991) Method integration for experience-based learning. *Proceedings of the Ninth National Conference on Artificial Intelligence* (pp. 378-393) Morgan Kaufmann.
- [Hirsh 87] Hirsh, H. (1987). Explanation-based generalization in a logic programming environment. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pp. 221-227, Milan.
- [Kaelbling 90] Kaelbling, L. P. (1990). Learning functions in k -DNF from reinforcement. *Machine Learning: Proceedings of the Seventh International Conference*, (pp. 162-169). Austin, TX: Morgan Kaufmann.
- [Kahn 84] Kahn, K. M. (1984). Partial evaluation, programming methodology, and artificial intelligence. *Artificial Intelligence Magazine*, Spring 1984, pp. 53-57.
- [Kahn and Carlsson 84] Kahn, K. M. and Carlsson, M. (1984). The compilation of prolog programs without the use of a prolog compiler. *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp. 348-355.
- [Knoblock 90] Knoblock, C. A. (1991). *Automatically Generating Abstractions for Problem Solving*, Ph.D. Thesis, Computer Science Department, Tech. Report CMU-CS-91-120, Carnegie Mellon University, May 1991.
- [Kopec and Niblett 80] Kopec, D. and Niblett, T. (1980). How hard is it to play the king-rook king-knight ending? M. R. Clarke (Ed), *Advances in Computer Chess*.
- [Larkin *et al.* 80] Larkin, J., McDermott, Simon D. P. & Simon H. A. (1980). Expert and novice performance in solving physics problems. In *Science*, 206, 1335-1342.

- [Larson and Casti 78] Larson, R. E. and Casti, J. L. (1978). *Principles of Dynamic Programming, Part 1*. Marcell Dekker, Inc., New York, New York, 1978.
- [Levesque 86] Levesque, H. J. (1986). Making believers out of computers. *Artificial Intelligence*, 30, (1) 81-108.
- [McCarthy and Hayes, 69] McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the stand point of artificial intelligence. B. Meltzer and D. Michie (editors), *Machine Intelligence 6*. Edinburgh University Press, Edinburgh, 1969.
- [Michie, 82] Michie, D. (1982). *Machine Intelligence and related topics*. Gordon and Breach Science Pub.
- [Minton 84] Minton, S. (1984). Constraint-based generalization: Learning game-playing plans from single examples. *Proceedings of the Third International Conference on Artificial Intelligence*, pp. 251-254, Austin, TX.
- [Minton 88a] Minton, S. (1988). *Learning effective search control knowledge: An explanation-based approach*. Ph.D. Thesis, Computer Science Department, Carnegie Mellon University, March 1988.
- [Minton 88b] Minton, S. (1988b). Quantitative results concerning the utility of explanation-based learning. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 564-569). Saint Paul, MI: Morgan Kaufmann.
- [Mitchell Keller and Kedar-Cabelli 86] Mitchell, T., Keller, R. and Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. In *Machine Learning*, 1, (1) 47-80.
- [Mooney and Bennett 86] Mooney, R. J., and Bennett, S. W. (1986). A domain independent explanation-based generalizer. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 551-555). Philadelphia, PA: Morgan Kaufmann.
- [Montanari 74] Montanari, U. (1974). Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences* 7, pp. 95-132.
- [Mostow and Prieditis 89] Mostow, J. and Prieditis, A. E. (1989) Discovering admissible heuristics by abstraction and optimizing: A transformational Approach. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 701-707, Detroit, MI, USA.

- [Muggleton 88] Muggleton, S. H. (1988). Inductive acquisition of chess strategies. D. Michie, J. E. Hayes and J. Richards, (Eds), *Machine Intelligence 11*. Oxford University Press, Oxford, 1988.
- [Nievergelt 77] Nievergelt, J. (1977). Information content of chess positions. *SIGART newsletter*, (62) 13-14.
- [Omohudro 90] Omohudro, S. M. (1990). Geometric learning algorithms. *Physica D*, 42:307-321.
- [Pell 92a] Pell, B. (1992). Metagame: A new challenge for games and learning, H.J. van den Herik and L.V. Allis (Eds). *Heuristic Programming in Artificial Intelligence 3—The Third Computer Olympiad*, H. J. van den Herik and L. V. Allis (Eds). Ellis Horwood, 1992.
- [Pell 92b] Pell, B. (1992). Metagame in symmetric, chess-like games, *Heuristic Programming in Artificial Intelligence 3—Third Computer Olympiad*, H. J. van den Herik and L. V. Allis (Eds). Ellis Horwood, 1992.
- [Quinlan 83] Quinlan, J., R. (1983). Learning efficient classification procedures and their application to chess end games. R. S. Michalski, J. G. Carbonell and T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* San Mateo, CA: Morgan Kaufmann.
- [Rosenbloom 82] Rosenbloom, P. (1982). A world-championship Othello program. *Artificial Intelligence*, 19 279-320.
- [Schaeffer 91] Schaeffer, J. (1991) Checkers program earns the right to play for world title. *Computing Research News*, pp. 12, January.
- [Schoppers 87] Schoppers, M. J. (1987) Universal plans for reactive robots in unpredictable environments. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pp. 1039-1046, Milan, Italy.
- [Schoppers 89] Schoppers, M. J. (1989) In defense of reaction plans as caches. *Artificial Intelligence Magazine* 10 (4) pp. 51-60.
- [Seidel 86] Seidel, R. (1986). Deriving correct pattern descriptions and rules for the KRK endgame by deductive methods. *Advances in Computer Chess 4*. D. F. Beal, Ed. Pergamon Press.
- [Shostak 77] Shostak, R. E. (1977). On the SUP-INF method for proving preburger formulas. *Journal of the Association for Computer Machinery* 24 (4) pp. 529-543.

- [Smith and Pressburger 88] Smith, D. R. and Pressburger, T. T. (1988). Knowledge-based software development tools. *Software Engineering Environments*, P. Brereton, (Ed)., Ellis Horwood Ltd., Chichester, pp. 79-103.
- [Smith and Genesereth 85] Smith, D. E. and Genesereth, M. R. (1985). Ordering Conjunctive Queries. *Artificial Intelligence*, **24**, 171-215.
- [Subramanian and Genesereth 87] Subramanian, D. & Genesereth, M., R., (1987). The relevance of irrelevance. *Proceedings of the Sixth National Conference on Artificial Intelligence* (pp. 416-422). Seattle, WA: Morgan Kaufmann.
- [Subramanian 89] Subramanian, D. (1989). *A Theory of Justified Reformulations*, Ph.D. Thesis, Computer Science Department, Stanford University, March 1989.
- [Sterling and Shapiro 86] Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*, MIT press, Cambridge: MA.
- [Tadepalli 89] Tadepalli, P. (1989). Lazy explanation-based learning: A solution to the intractable theory problem. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 694-700, 299-348, Detroit, MI.
- [Tambe and Newell 88] Tambe, M. and Newell, A. (1988). Some chunks are expensive. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 451-458). Ann Arbor, MI: Morgan Kaufmann.
- [Tambe, Newell and Rosenbloom 90] Tambe, M., Newell, A. and Rosenbloom, P. S. (1990). The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning*, 5, (3).
- [Tenenbergs 87] Tenenbergs, J. D. (1987) Perserving consistency across abstract mappings. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pp. 1011-1014, Milan, Italy.
- [Thompson 86] Thompson, K. (1986) Retrograde analysis of certain endgames. *ICCA Journal*, September 1986.
- [Ullman 83] Ullman, S. (1983). Visual routines. A. I. Memo No. 723, Massachusetts Institute of Technology.
- [Utgoff 88] Utgoff, P., E. (1988). ID5: An incremental ID3. *Proceedings of the Fifth International Conference on Machine Learning*, pp. 107-120, Ann Arbor, MI.

- [Utgoff 86] Utgoff, P., E. (1986). Shift of bias for inductive concept learning. *Machine Learning: An Artificial Intelligence Approach*, Vol 2. R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.). Morgan Kaufmann, Los Altos; CA.
- [Wilkins 80] Wilkins, D. (1980). Patterns and plans in chess. *Artificial Intelligence*, 14, 165-203.

Appendix A

Chess Domain Theory

```
; This file contains the definition of the chess domain theory
; The domain is described in a standard syntax for interpretation
; by the compiler. The definitions of the operators employ
; a vocabulary of spatial primitives that are known to the system
; on, directions, connected, openline
```

```
(in-package "USER")
```

```
; OPERATORS
```

```
; *****
```

```
; Operators are given as (op name parameters precondition addlist deletelist)
```

```
(setq *ops* '(
```

```
(op normal-ops
```

```
  (?side)
```

```
  (?Obj ?F ?T ?vector ?symmetry)
```

```
  ?Obj
```

```
  ?F
```

```
  ?T
```

```
  ((on ?F ?Obj)
```

```

(= ?Obj (obj ?side ?type))
(single ?type)
(directions ?Obj ?vector ?symmetry)
(connected ?F ?T ?vector ?symmetry)
(on ?T empty))
((on ?T ?Obj)
 (on ?F empty))
((on ?T empty)))

```

```

(op normal-ops
 (?side)
 (?Obj ?F ?T ?vector ?symmetry)
 ?Obj
 ?F
 ?T
 ((on ?F ?Obj)
 (= ?Obj (obj ?side ?type))
 (multiple ?type)
 (directions ?Obj ?vector ?symmetry)
 (openline ?F ?T ?vector ?symmetry ?length)
 (on ?T empty))
 ((on ?T ?Obj)
 (on ?F empty))
 ((on ?T empty)))

```

```

(op take-ops
 (?side)
 (?Obj ?Obj0 ?F ?T ?vector ?symmetry)

```

```

?Obj
?F
?T
((on ?F ?Obj)
 (= ?Obj (obj ?side ?type))
 (single ?type)
 (directions ?Obj ?vector ?symmetry)
 (connected ?F ?T ?vector ?symmetry)
 (opside ?side ?side0)
 (= ?Obj0 (obj ?side0 ?type1))
 (on ?T ?Obj0))
((on ?T ?Obj)
 (on ?F empty))
((on ?T ?Obj0)))

```

```

(op take-ops
 (?side)
 (?Obj ?Obj0 ?F ?T ?vector ?symmetry)
?Obj
?F
?T
((on ?F ?Obj)
 (= ?Obj (obj ?side ?type))
 (multiple ?type)
 (directions ?Obj ?vector ?symmetry)
 (openline ?F ?T ?vector ?symmetry ?length)
 (opside ?side ?side0)
 (= ?Obj0 (obj ?side0 ?type1))

```

```

      (on ?T ?Obj0))
    ((on ?T ?Obj)
      (on ?F empty))
    ((on ?T ?Obj0)))
  ))

```

; Definitions of the directions for the various objects. Note that directions i
; predicate.

; Type information: (directions objectdes direction symmetry)

```

(setq *database* '(

(directions (obj ? king) (1 1) ((rotation 4)))
(directions (obj ? king) (0 1) ((rotation 4)))

(directions (obj ? bman) (1 1) ((rotation 4)))

(directions (obj ? wman) (1 1) ((rotation 4)))

(directions (obj ? knight) (2 -1) ((rotation 4)))
(directions (obj ? knight) (2 1) ((rotation 4)))

(directions (obj ? rook) (1 0) ((rotation 4)))

(directions (obj ? bishop) (1 1) ((rotation 4)))

(directions (obj ? queen) (1 1) ((rotation 4)))
(directions (obj ? queen) (0 1) ((rotation 4)))

```



```
; Definitions are given for user predicates
```

```
(user-predicate (single piece-type))
```

```
(single king)
```

```
(single knight)
```

```
(user-predicate (multiple piece-type))
```

```
(multiple rook)
```

```
(multiple bishop)
```

```
(multiple queen)
```

```
(user-predicate (opside piece-side piece-side))
```

```
(opside white black)
```

```
(opside black white)
```

```
))
```

```
(setq *object-graphics*
```

```
  '((obj black king) . black-K)
```

```
    ((obj black bman) . black-K)
```

```
    ((obj black knight) . black-N)
```

```
    ((obj black queen) . black-Q)
```

```
    ((obj black rook) . black-R)
```

```
    ((obj black bishop) . black-B)
```

((obj black pawn) . black-P)

((obj white king) . white-K)

((obj white wman) . white-K)

((obj white rook) . white-R)

((obj white queen) . white-Q)

((obj white knight) . white-N)

((obj white bishop) . white-B)

((obj white pawn) . white-P)))

Appendix B

King-Rook-King Problem Specification

```
;;; the objects involved
(setq *objects* '(

(obj black king)    ; 0

(obj white king)   ; 1
(obj white rook)   ; 2

))

(setq *location-variables* '(

((obj black king) . (?Xbk ?Ybk))

((obj white king) . (?Xwk ?Ywk))
((obj white rook) . (?Xwr ?Ywr))

))
```

```

(flag-all-vars '(?Xbk ?Ybk ?Xwk ?Ywk ?Xwr ?Ywr ?L))
(setf (get '?L 'vartype) 'internal)

;;; the initial won patterns
;;; 2 describe the king-king attack
;;; 1 describes the rook king attack

(setq *initial-won-patterns* '(

,(make-pattern
  :name 'king-attacks-king-1
  :object-definition-order '(0 1)
  :object-constraints
    (make-array 3
      :initial-contents '(, (make-location-constraint
                            :xvar '?Xbk
                            :yvar '?Ybk
                            :xhl (cons 8 1)
                            :yhl (cons 8 1)
                            :x '?Xbk
                            :y '?Ybk)
                          ,(make-location-constraint
                            :xvar '?Xwk
                            :yvar '?Ywk
                            :xhl (cons 8 1)
                            :yhl (cons 8 1)
                            :x (make-linear-constraint
                                :variable '?Xwk

```

```

:equation '((1 . 1)(1 . ?Xbk))
:defined-using '(?Xbk)
:y (make-linear-constraint
:variable '?Ywk
:equation '((1 . ?Ybk))
:defined-using '(?Ybk))
nil))
:rotation-symmetry 4)

,(make-pattern
:name 'king-attacks-king-d
:object-definition-order '(0 1)
:object-constraints
(make-array 3
:initial-contents '(,(make-location-constraint
:xvar '?Xbk
:yvar '?Ybk
:xhl (cons 8 1)
:yhl (cons 8 1)
:x '?Xbk
:y '?Ybk)
,(make-location-constraint
:xvar '?Xwk
:yvar '?Ywk
:xhl (cons 8 1)
:yhl (cons 8 1)
:x (make-linear-constraint
:variable '?Xwk

```

```

:equation '((1 . 1)(1 . ?Xbk))
:defined-using '(?Xbk)
:y (make-linear-constraint
:variable '?Ywk
:equation '((1 . 1)(1 . ?Ybk))
:defined-using '(?Ybk)))
nil))
:rotation-symmetry 4)

,(make-pattern
:name 'rook-attacks-king
:object-definition-order '(0 2)
:object-constraints
(make-array 3
:initial-contents '(,(make-location-constraint
:xvar '?Xbk
:yvar '?Ybk
:xhl (cons 8 1)
:yhl (cons 8 1)
:x '?Xbk
:y '?Ybk)
nil
,(make-location-constraint
:xvar '?Xwr
:yvar '?Ywr
:xhl (cons 8 1)
:yhl (cons 8 1)
:x (make-linear-constraint

```

```

:variable '?Xwr
:equation '((1 . ?L)(1 . ?Xbk))
:defined-using '(?Xbk ?L)
:internal-variables '(?L)
:y (make-linear-constraint
:variable '?Ywr
:equation '((1 . ?Ybk))
:defined-using '(?Ybk))
:internal '(?L)
:internal-line-constraints
(list (make-line-constraint
:variable '?L
:low 1
:high 7))))
:openlines '(,(make-openline :from (cons '?Xwr '?Ywr) :to (cons '?Xbk '?Ybk)
:rotation-symmetry 4)

))

```

;; a black to move pattern must not subsume these patterns

```

(setq *illegal-btm-patterns* '(
,(make-pattern
:name 'king-attacks-king
:object-definition-order '(0 1)
:object-constraints
(make-array 3
:initial-contents '(,(make-location-constraint

```

```

:xvar '?Xbk
:yvar '?Ybk
:xhl (cons 8 1)
:yhl (cons 8 1)
:x '?Xbk
:y '?Ybk)
,(make-location-constraint
:xvar '?Xwk
:yvar '?Ywk
:xhl (cons 8 1)
:yhl (cons 8 1)
:x (make-linear-constraint
:variable '?Xwk
:equation '((1 . 1)(1 . ?Xbk))
:defined-using '(?Xbk))
:y (make-linear-constraint
:variable '?Ywk
:equation '((1 . ?Ybk))
:defined-using '(?Ybk)))
nil))

:rotation-symmetry 8)

))

;; a while to move pattern must not subsume these patterns
(setq *illegal-wtm-patterns* '(
,(make-pattern
:name 'king-attacks-king

```



```

:object-definition-order '(0 1)
:object-constraints
  (make-array 3
    :initial-contents '(, (make-location-constraint
      :xvar '?Xbk
      :yvar '?Ybk
      :xhl (cons 8 1)
      :yhl (cons 8 1)
      :x '?Xbk
      :y '?Ybk)
      ,(make-location-constraint
        :xvar '?Xwk
        :yvar '?Ywk
        :xhl (cons 8 1)
        :yhl (cons 8 1)
        :x (make-linear-constraint
          :variable '?Xwk
          :equation '((1 . 1)(1 . ?Xbk))
          :defined-using '(?Xbk))
        :y (make-linear-constraint
          :variable '?Ywk
          :equation '((1 . ?Ybk))
          :defined-using '(?Ybk)))
      nil))
    :rotation-symmetry 8)
  ))

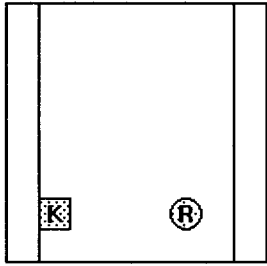
```

Appendix C

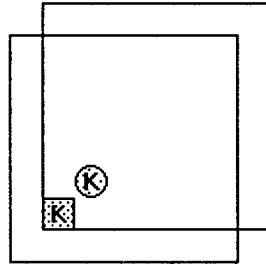
Example run for KRK Problem

In this ending, we illustrate the first 120 patterns derived by the compiler when employing a breadth-first search control strategy. The patterns are numbered consecutively in the order that they are generated. The first three patterns were provided to the compiler and describe situations where white can win in one ply by capturing black's king. The square pieces represent black, the losing side, the round pieces represent white, the winning side. If a piece is not constrained to be in a single location, it is enclosed within a rectangle that represents the region that it can occupy.

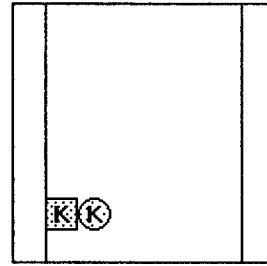
Included with each pattern is the estimated number of instances that it covers. Also included is a brief explanation of how the pattern was derived. For the loss patterns, the set of successor won patterns is included along with the relevant influence relations. For example pattern 26, which includes the set $\{ mt15, m1 \}$, was derived by intersecting a *make-true* of pattern 15 and a *maintain-true* of pattern 1.



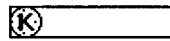
1: Won in 1



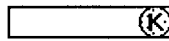
2: Won in 1



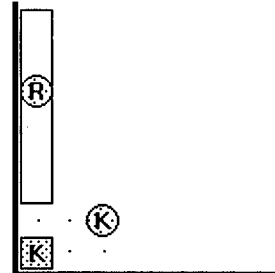
3: Won in 1



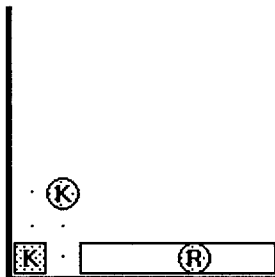
4: Loss in 2 (Covers 25)
{*mt2, mt3, m1*}



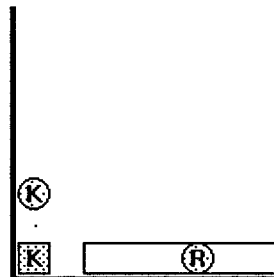
5: Loss in 2 (Covers 25)
{*mt2, mt3, m1*}



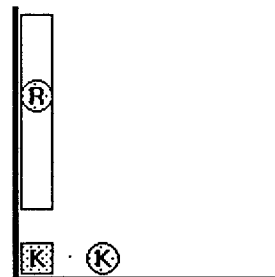
6: Loss in 2 (Covers 6)
{*mt2, mt3, m1*}



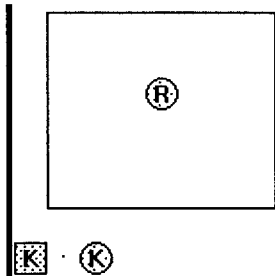
7: Loss in 2 (Covers 6)
{*mt2, mt3, m1*}



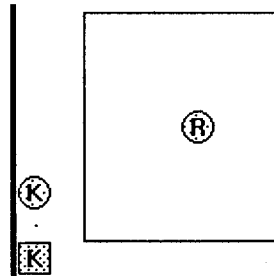
8: Loss in 2 (Covers 6)
{*mt2, mt3, m1*}



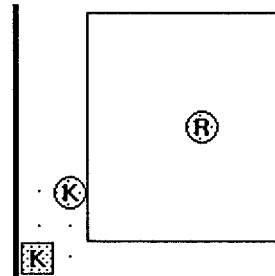
9: Loss in 2 (Covers 6)
{*mt2, mt3, m1*}



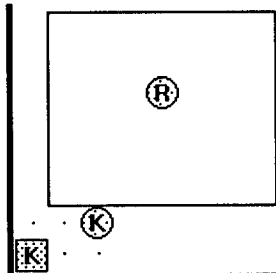
10: Won in 3 (Covers 42)
{*mt9*}



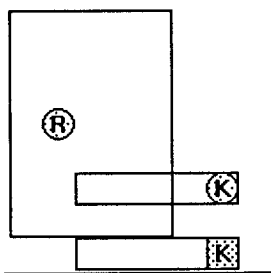
11: Won in 3 (Covers 42)
{*mt8*}



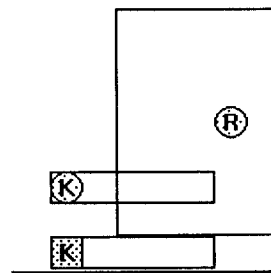
12: Won in 3 (Covers 42)
{*mt7*}



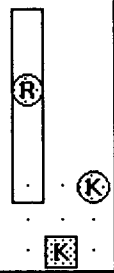
13: Won in 3 (Covers 42)
{*mt6*}



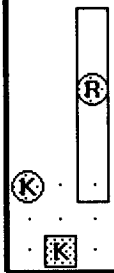
14: Won in 3 (Covers 175)
{*mt5*}



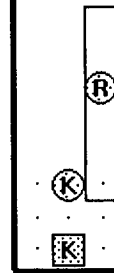
15: Won in 3 (Covers 175)
{*mt4*}



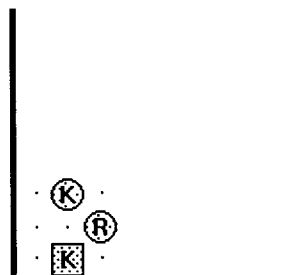
16: Loss in 4 (Covers 6)
{*mt10, mt1, mt2, mt3*}



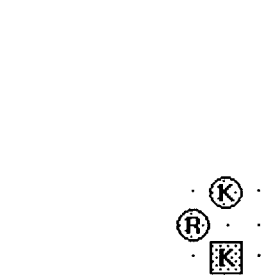
17: Loss in 4 (Covers 6)
{*mt11, mt1, mt2, mt3*}



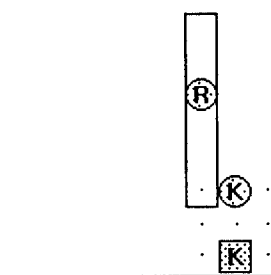
18: Loss in 4 (Covers 6)
{*mt12, mt1, mt2, mt3*}



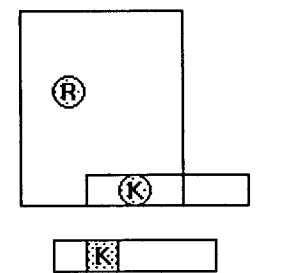
19: Loss in 4 (Covers 1)
{*mt12, mt1, mt2, mt3*}



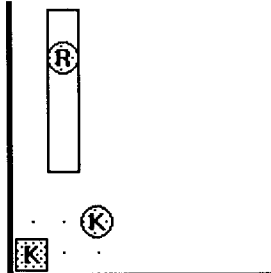
20: Loss in 4 (Covers 1)
{*mt13, mt1, mt2, mt3*}



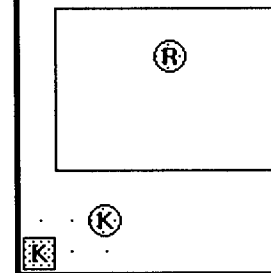
21: Loss in 4 (Covers 6)
{*mt13, mt1, mt2, mt3*}



22: Loss in 4 (Covers 30)
{*mt14, mt1, mt2, mt3*}



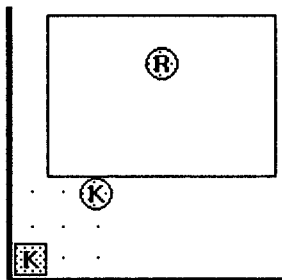
23: Loss in 4 (Covers 5)
{*mt14, mt1*}



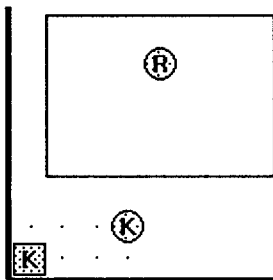
24: Loss in 4 (Covers 35)
{*mt14, mt2, mt3*}

<p>25: Loss in 4 (Covers 30) {<i>mt15,mt1,mt2,mt3</i>}</p>	<p>26: Loss in 4 (Covers 5) {<i>mt15,mt1</i>}</p>	<p>27: Loss in 4 (Covers 35) {<i>mt15,mt2,mt3</i>}</p>
<p>28: Won in 5 (Covers 35) {<i>mt27</i>}</p>	<p>29: Won in 5 (Covers 5) {<i>mt27,-13</i>}</p>	<p>30: Won in 5 (Covers 35) {<i>mt27</i>}</p>
<p>31: Won in 5 (Covers 35) {<i>mt27</i>}</p>	<p>32: Won in 5 (Covers 35) {<i>mt27</i>}</p>	<p>33: Won in 5 (Covers 5) {<i>mt26</i>}</p>
<p>34: Won in 5 (Covers 5) {<i>mt26</i>}</p>	<p>35: Won in 5 (Covers 5) {<i>mt26</i>}</p>	<p>36: Won in 5 (Covers 5) {<i>mt26</i>}</p>

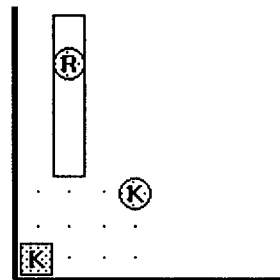
<p>37: Won in 5 (Covers 30) {mt25}</p>	<p>38: Won in 5 (Covers 30) {mt25}</p>	<p>39: Won in 5 (Covers 30) {mt25}</p>
<p>40: Won in 5 (Covers 30) {mt25}</p>	<p>41: Won in 5 (Covers 30) {mt25}</p>	<p>42: Won in 5 (Covers 30) {mt25}</p>
<p>43: Won in 5 (Covers 210) {mt25, -1}</p>	<p>44: Won in 5 (Covers 96) {mt25}</p>	<p>45: Won in 5 (Covers 30) {m25}</p>
<p>46: Won in 5 (Covers 35) {mt24}</p>	<p>47: Won in 5 (Covers 35) {mt24}</p>	<p>48: Won in 5 (Covers 5) {mt24, -12}</p>



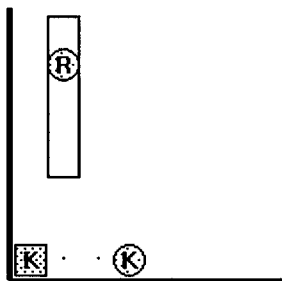
49: Won in 5 (Covers 35)
{mt24}



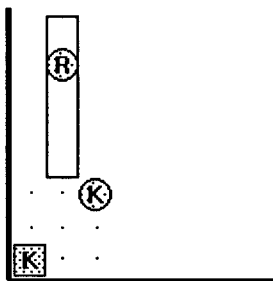
50: Won in 5 (Covers 35)
{mt24}



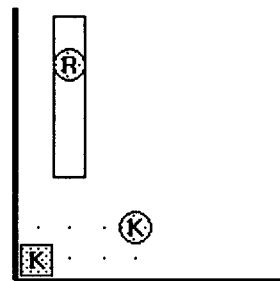
51: Won in 5 (Covers 5)
{mt23}



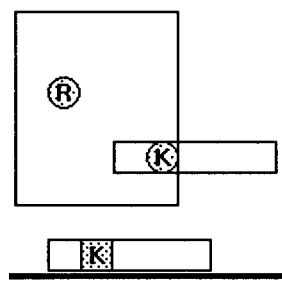
52: Won in 5 (Covers 5)
{mt23}



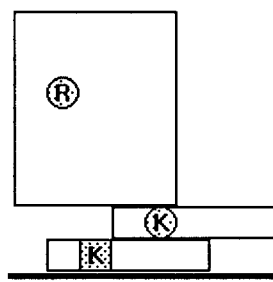
53: Won in 5 (Covers 5)
{mt23}



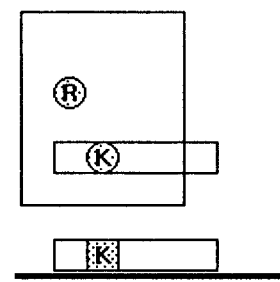
54: Won in 5 (Covers 5)
{mt23}



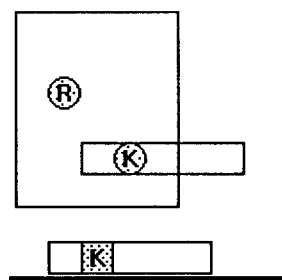
55: Won in 5 (Covers 30)
{mt22}



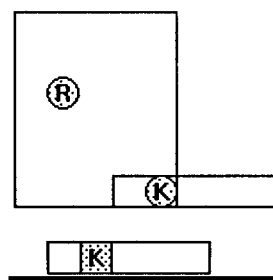
56: Won in 5 (Covers 30)
{mt22}



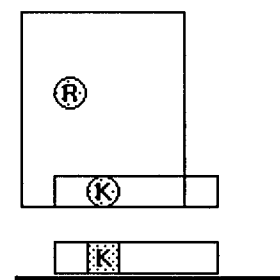
57: Won in 5 (Covers 30)
{mt22}



58: Won in 5 (Covers 30)
{mt22}

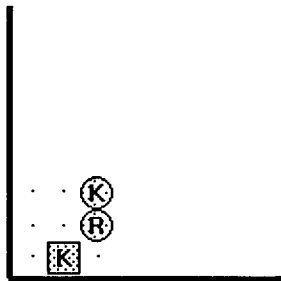


59: Won in 5 (Covers 30)
{mt22}

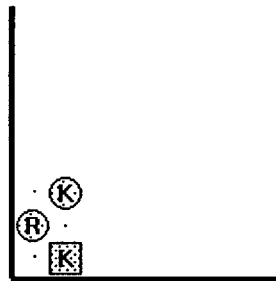


60: Won in 5 (Covers 30)
{mt22}

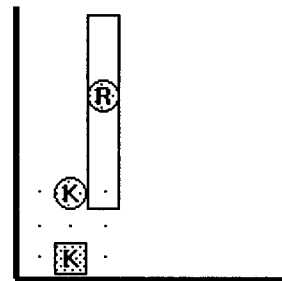
<p>61: Won in 5 (Covers 96) {<i>mt22</i>}</p>	<p>62: Won in 5 (Covers 210) {<i>mt22, -1</i>}</p>	<p>63: Won in 5 (Covers 30) {<i>mt22</i>}</p>
<p>64: Won in 5 (Covers 6) {<i>mt21</i>}</p>	<p>65: Won in 5 (Covers 6) {<i>mt21</i>}</p>	<p>66: Won in 5 (Covers 6) {<i>mt21</i>}</p>
<p>67: Won in 5 (Covers 6) {<i>mt21</i>}</p>	<p>68: Won in 5 (Covers 6) {<i>mt21</i>}</p>	<p>69: Won in 5 (Covers 6) {<i>mt21, -1</i>}</p>
<p>70: Won in 5 (Covers 6) {<i>m21</i>}</p>	<p>71: Won in 5 (Covers 1) {<i>mt20</i>}</p>	<p>72: Won in 5 (Covers 1) {<i>mt20, -1</i>}</p>



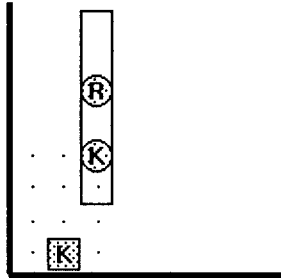
73: Won in 5 (Covers 1)
{*mt19*}



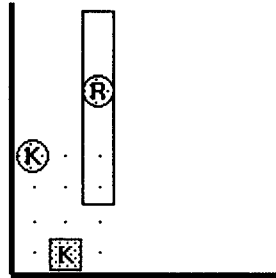
74: Won in 5 (Covers 1)
{*mt19, -1*}



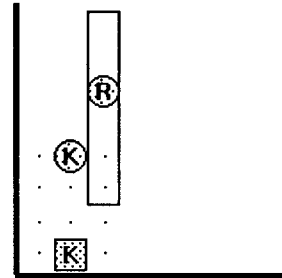
75: Won in 5 (Covers 6)
{*mt19*}



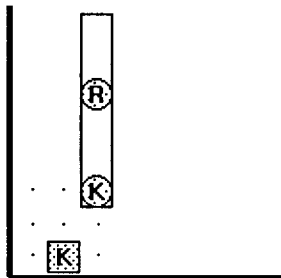
76: Won in 5 (Covers 6)
{*mt18*}



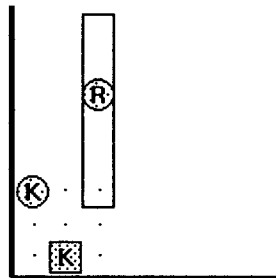
77: Won in 5 (Covers 6)
{*mt18*}



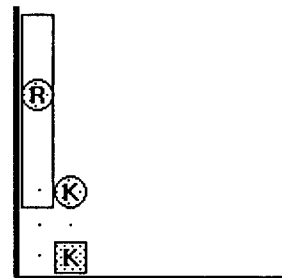
78: Won in 5 (Covers 6)
{*mt18*}



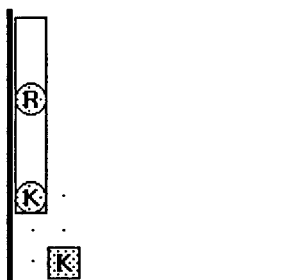
79: Won in 5 (Covers 6)
{*mt18*}



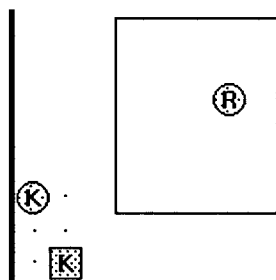
80: Won in 5 (Covers 6)
{*mt18*}



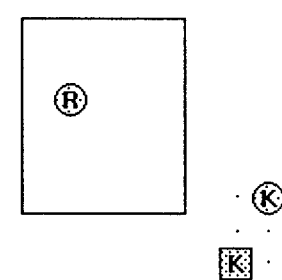
81: Won in 5 (Covers 6)
{*mt18, -1*}



82: Won in 5 (Covers 6)
{*mt17, -1*}

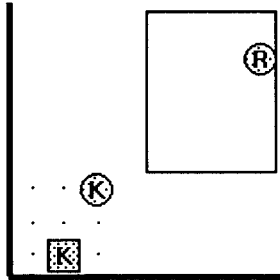


83: Won in 5 (Covers 30)
{*mt17*}

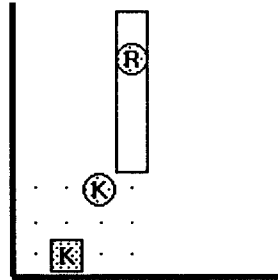


84: Won in 5 (Covers 30)
{*mt16*}

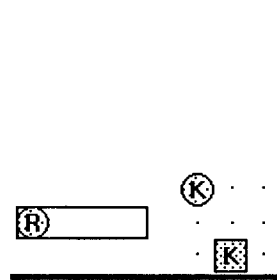
<p>85: Won in 5 (Covers 6) {<i>mt16, -1</i>}</p>	<p>86: Loss in 6 (Covers 4) {<i>mt32, mt15, mt1, mt2</i>}</p>	<p>87: Loss in 6 (Covers 6) {<i>mt41, mt2, mt3</i>}</p>
<p>88: Loss in 6 (Covers 30) {<i>mt43, mt1, mt2, mt3</i>}</p>	<p>89: Loss in 6 (Covers 6) {<i>mt43, mt41, mt32, mt2, mt3</i>}</p>	<p>90: Loss in 6 (Covers 28) {<i>mt43, mt32, mt15, mt2, mt3</i>}</p>
<p>91: Loss in 6 (Covers 24) {<i>mt44, mt12, mt2, mt3</i>}</p>	<p>92: Loss in 6 (Covers 6) {<i>mt45, mt12, mt2, mt3</i>}</p>	<p>93: Loss in 6 (Covers 5) {<i>mt45, mt2, mt3, m1</i>}</p>
<p>94: Loss in 6 (Covers 6) {<i>mt45, mt2, m1</i>}</p>	<p>95: Loss in 6 (Covers 5) {<i>mt48, mt43, mt2, mt3</i>}</p>	<p>96: Loss in 6 (Covers 5) {<i>mt49, mt43, mt1, mt2</i>}</p>



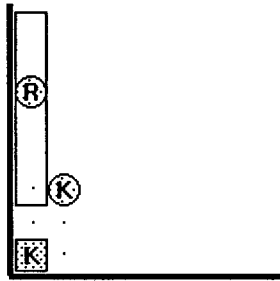
97: Loss in 6 (Covers 20)
{mt49,mt43,mt15,mt2,mt3}



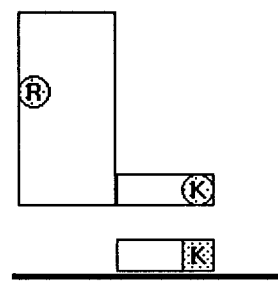
98: Loss in 6 (Covers 5)
{mt49,mt43,mt41,mt2,mt3}



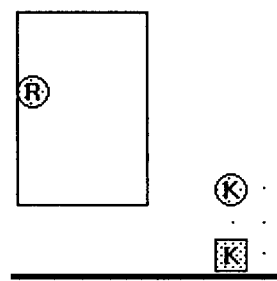
99: Loss in 6 (Covers 4)
{mt49,mt14,mt1,mt2}



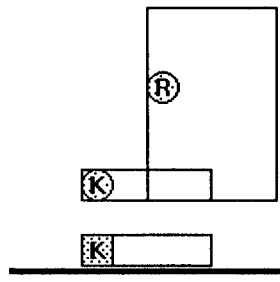
100: Loss in 6 (Covers 6)
{mt60,mt2,mt3}



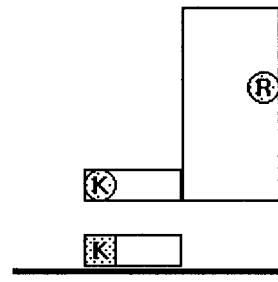
101: Loss in 6 (Covers 54)
{mt61,mt43,mt2,mt3}



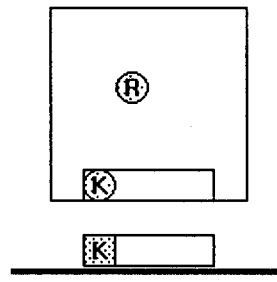
102: Loss in 6 (Covers 24)
{mt61,mt13,mt2,mt3}



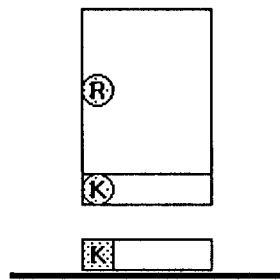
103: Loss in 6 (Covers 24)
{mt62,mt45,mt2,mt3}



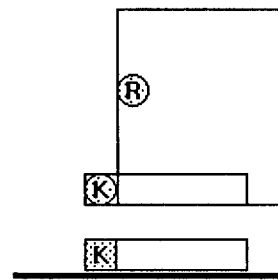
104: Loss in 6 (Covers 54)
{mt62,mt44,mt2,mt3}



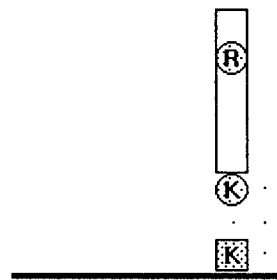
105: Loss in 6 (Covers 144)
{mt62,mt43,mt2,mt3}



106: Loss in 6 (Covers 24)
{mt62,mt43,mt2,mt1}



107: Loss in 6 (Covers 30)
{mt62,mt1,mt2,mt3}



108: Loss in 6 (Covers 5)
{mt62,mt29,mt2,mt3}

<p>109: Loss in 6 (Covers 5) <i>{mt62, mt60, mt32, mt2, mt3}</i></p>	<p>110: Loss in 6 (Covers 6) <i>{mt62, mt60, mt49, mt2, mt3}</i></p>	<p>111: Loss in 6 (Covers 28) <i>{mt62, mt49, mt14, mt2, mt3}</i></p>
<p>112: Loss in 6 (Covers 5) <i>{mt62, mt32, mt1, mt2}</i></p>	<p>113: Loss in 6 (Covers 20) <i>{mt62, mt32, mt14, mt2, mt3}</i></p>	<p>114: Loss in 6 (Covers 36) <i>{mt62, mt43, mt2}</i></p>
<p>115: Loss in 6 (Covers 24) <i>{mt63, mt43, mt2, mt3}</i></p>	<p>116: Loss in 6 (Covers 6) <i>{mt63, mt13, mt2, mt3}</i></p>	<p>117: Loss in 6 (Covers 5) <i>{mt63, mt2, mt3, m1}</i></p>
<p>118: Loss in 6 (Covers 6) <i>{mt63, mt2, m1}</i></p>	<p>119: Loss in 6 (Covers 6) <i>{mt67, mt2, mt3}</i></p>	<p>120: Loss in 6 (Covers 6) <i>{mt68, mt62, mt2, mt3}</i></p>