Strategy Generation and Evaluation for Meta-Game Playing

Barney Darryl Pell

Trinity College

A dissertation submitted for the degree of Doctor of Philosophy in the University of Cambridge

August 1993

Abstract

Meta-Game Playing (Metagame) is a new paradigm for research in game-playing in which we design programs to take in the rules of unknown games and play those games without human assistance. Strong performance in this new paradigm is evidence that the program, instead of its human designer, has performed the analysis of each specific game.

SCL-Metagame is a concrete Metagame research problem based around the class of symmetric chess-like games. The class includes the games of chess, draughts, noughts and crosses, Chinese-chess, and Shogi. An implemented game generator produces new games in this class, some of which are objects of interest in their own right.

METAGAMER is a program that plays SCL-Metagame. The program takes as input the *rules* of a specific game and analyses those rules to construct for that game an efficient representation and an evaluation function, both for use with a generic search engine. The strategic analysis performed by the program relates a set of general knowledge sources to the details of the particular game. Among other properties, this analysis determines the relative value of the different pieces in a given game. Although METAGAMER does not learn from experience, the values resulting from its analysis are qualitatively similar to values used by experts on known games, and are sufficient to produce competitive performance the first time the program actually plays each game it is given. This appears to be the first program to have derived useful piece values directly from analysis of the rules of different games.

Experiments show that the knowledge implemented in METAGAMER is useful on games unknown to its programmer in advance of the competition and make it seem likely that future programs which incorporate learning and more sophisticated active-analysis techniques will have a demonstrable competitive advantage on this new problem. When playing the known games of chess and checkers against humans and specialised programs, METAGAMER has derived from more general principles some strategies which are familiar to players of those games and which are hard-wired in many game-specific programs.

Related Publications

Selected aspects of the research described in this thesis, as well as some earlier related work have been documented or published elsewhere:

Barney Pell. Exploratory Learning in the Game of GO: Initial Results. In D.N.L. Levy and D.F. Beal, editors, *Heuristic Programming in Artificial Intelligence 2 – The Second Computer Olympiad*. Ellis Horwood, 1991. Also appears as University of Cambridge Computer Laboratory Technical Report No. 275.

Barney Pell. Metagame: A New Challenge for Games and Learning. In H.J. van den Herik and L.V. Allis, editors, *Heuristic Programming in Artificial Intelligence 3 – The Third Computer Olympiad*. Ellis Horwood, 1992. Also appears as University of Cambridge Computer Laboratory Technical Report No. 276.

Barney Pell. Metagame in Symmetric, Chess-Like Games. In H.J. van den Herik and L.V. Allis, editors, *Heuristic Programming in Artificial Intelligence 3 – The Third Computer Olympiad*. Ellis Horwood, 1992. Also appears as University of Cambridge Computer Laboratory Technical Report No. 277.

Barney Pell. Logic Programming for General Game Playing. In *Proceedings of the ML93 Workshop on Knowledge Compilation and Speedup Learning*, Amherst, Mass., June 1993. Machine Learning Conference. Also appears as University of Cambridge Computer Laboratory Technical Report No. 302.

Barney Pell. A Strategic Metagame Player for General Chess-Like Games. In *Proceedings of the AAAI Fall Symposium on Games: Planning and Learning*. AAAI Press, 1993. Also appears as Technical Report FIA-93-32, Artificial Intelligence Research Branch, NASA Ames Research Center.

Acknowledgements

The sustained period of writing leading up to finishing this thesis has given me the opportunity to reflect on all the people who helped me, influenced me, taught me, and supported me in diverse ways throughout this project. I am glad that I get a chance to thank some of them here.

Many thanks to the people who read drafts of this thesis, cover to cover, and provided comments which went to change—and I believe improve—my outlook and my presentation of this work. These were Andrew Moore, Steve Pulman, Manny Rayner, and William Tunstall-Pedoe. Steve and Manny, as my supervisors, read several drafts throughout this process, a task for which I am truly grateful.

I hardly know how to acknowledge the influence Manny Rayner has had on me. He has taken time away from his busy job and his own thesis to talk with me at all hours. He brought me to Sweden for a summer of bridge and chess, taught me the importance of giving examples, and tried to teach me how to moderate my arguments. Most importantly, Manny and the Rayner family (Elizabeth, Jonathan, David, Lotta, and Alison) have been my family in Cambridge.

My work on game-playing started one summer as a project with Peter Cheeseman at NASA Ames Research Center and continued through several subsequent summers. Thank you, Peter, for having been my mentor-in-absentia, and for having taught me to look for big pictures, hidden assumptions, and prior probabilities. Thanks also to RIACS for financing me during those summers.

A few people played a crucial role in the development of Metagame by supporting my ideas when they were just forming, at a time when positive response made all the difference. Victor Allis proofread and edited the first Metagame papers and forced me to get it right the first time. Since then he has been a valuable friend, colleague, and ally. Nick Flann has been extremely supportive of my ideas from start to finish, which has really helped. Christer Samuelsson gave the first talk on Metagame for me when I was too ill to go to Sweden, and Björn Gambäck kept my ego afloat by letting me beat him repeatedly at chess. Matt Hurst kept my program's ego afloat by letting it beat him at a variety of games.

Thanks to my colleagues in the research community for useful discussions and feedback. Here I include Hans Berliner, Jamie Callan, Susan Epstein, Graham Farr, Tom Fawcett, Othar Hansson, Robert Levinson, Andrew Mayer, Shaul Markovitch, Prasad Tadepalli, Dan Pehoushek, William Tunstall-Pedoe, and David Wilkins. Thanks to Nils Nilsson for serving as a mentor in my undergraduate days and as my advocate thereafter.

I have benefitted from interactions with members of the Cambridge University Computer Laboratory, past and present. They have provided an intellectual atmosphere and have helped to make life happy here despite the gloomy architecture of the building. In this list I include Siani Baker, Ralph Beckett, Ted Briscoe, Lynn Cherny, Regis Cridlig, Innes Ferguson, Robin Fairbairns, Ellen Germain, Steve Hodges, Mark Humphrys, Matt Hurst, Karl MacDorman, Andrew Moore, David Padfield, Scarlet Schwiderski, Kish Shen, Malgorzata Stys, Nicko van Someren, and Thomas Vogel. Robin Fairbairns deserves special acknowledgement for TeX-support beyond belief, as do Chris Hadley, Martyn Johnson, and Graham Titmus for much needed systems support.

I thank several researchers for making their systems available to me for my thesis work. Jonathan Schaeffer kindly permitted me to use Chinook, Robert Levinson let me play with MORPH, and Stuart Cracraft must be thanked for contributing GnuChess to the research community. Dan Sahlin gave me a version of MIXTUS-prolog, and Mats Carlsson provided almost instantaneous support with SICSTUS-prolog.

Some people have been my close friends and have believed in me even when I did not believe in myself. In this set I am happy to include Troy Anderson, Michael Frank, Michael Ginn, Rozella Oliver, Josh Preven, Michael Ross, Ferdi Samaria, Kimberly Schumacher, Alex Scott, Mark Torrance, James Wiley, and Carl Witty.

This work would not have been possible without generous financial assistance. For this I thank the British Marshall Scholarship, Trinity College, the Computer Laboratory, the University of Cambridge Lundgren Fund, and the CT Taylor Studentship.

This thesis is dedicated to my mother and my grandparents, who taught me the power of positive thinking.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One reason why game-playing is an exciting activity for humans is that it couples intellectual activity with direct competition: better thinking and learning generally results in winning more games. Thus we can test out and refine our intellectual skills by playing games against opponents, and evaluate our progress based on the results of the competition.

The same motivation accounts for much of the interest in Computer Game-Playing (CGP) as a problem for Artificial Intelligence (AI): programs which think better, should play better, and so win more games. Thus we can test out and refine different theories of intelligence by writing game-playing programs which embody these different theories, and then play the programs against each other, and consider the more intelligent program to be the one which wins the most games. This thesis will refer to that presumed link between winning games and intelligent behaviour as the *competitive performance metric for intelligence.* Such a link would be advantageous for research, because it would mean we would not have to resort to descriptive evaluation measures and could instead use competition to evaluate research success. Many AI researchers working on games have assumed that such a link does exist and have focussed their energies exclusively on building strong game-playing programs.

Unfortunately, the use of such a link has proved problematic: we have been able to produce strong programs for some games through specialized engineering methods, the extreme case being special-purpose hardware, and through analysis of the games by humans instead of by programs themselves. Consequently, it now appears that increased understanding and automation of intelligent processing is neither necessary nor sufficient for strong performance in game-playing. That is, it appears that we can construct strong game-playing programs without doing much of interest from an AI perspective, and conversely, we can make significant advances in AI that do not result in strong game-playing programs.

This is a significant problem for AI researchers working on games, and it raises the following questions:

- How can we evaluate good work in AI applied to games in the absence of performance? Basic AI techniques such as learning, planning, and problem-solving should be useful for something—but if not for improved performance, then for what?

- How can we tell whether the performance of a program on some game is due to the general success of the AI theory it embodies, or merely to the cleverness of the researcher in analysing a specific problem? If the latter, we have little reason to believe the technique will transfer to other problems.

- Is it possible to find some game for which improved performance on that game would be linked necessarily to increased understanding and automation of general intelligent processing?

1.1 Structure of the Thesis

The thesis is structured in three parts. Each part contains its own introduction and summary, as does each chapter. This structure enables the thesis to be read at several levels of abstraction, top-down or in sequence. The first part of the thesis analyses the existing paradigm within which work in CGP is conducted and creates a new paradigm that overcomes the limitations discovered through that analysis. The second part constructs a specific research problem within the new paradigm, and the third part documents progress to date in addressing that research problem.

1.2 Metagame Paradigm

The first part of this thesis addresses the use of games in AI. Most current approaches within CGP–including those employing some forms of machine learning—rely on previous human analysis of particular games. Human researchers do most of the interesting game analysis, which makes it difficult to evaluate the generality and applicability of different approaches. It also makes it difficult to demonstrate empirically the success of research which emphasises generality (like work on learning and planning), as the programs which result from such work are usually incapable of performing well against special-purpose programs. This possibility undermines the use of competition as a metric for evaluating progress in AI and poses methodological problems for the field.

The fact that humans have specific knowledge of each game played by their programs makes it possible for humans, instead of the programs, to do the analysis necessary for competitive performance. This observation is the basis for a new paradigm for research on games called *Meta-Game Playing*, or *Metagame*, in which we design programs to take in the rules of unknown games and play those games without human assistance. Strong performance in this new paradigm is evidence that the program, instead of the human, has performed the analysis of each specific game. Playing Metagame with increasingly general classes of games makes it possible to demonstrate correspondingly general problem-solving ability.

1.3 SCL-Metagame

The second part of the thesis discusses the general issues involved in the construction of concrete Metagame research problems and introduces a specific Metagame research problem, called *Metagame in symmetric chess-like games*, or simply SCL-Metagame. This problem is based around the class of symmetric chess-like games. The class includes the games of chess, draughts, noughts and crosses, Chinese-chess, and Shogi. An implemented game generator produces new games in this class, some of which are objects of interest in their own right.

1.4 Metagamer

The third part of the thesis documents the construction of METAGAMER, a program that plays SCL-Metagame. The program takes as input the *rules* of a specific game and analyses those rules to construct for that game an efficient representation and an evaluation function, both for use with a generic search engine. The strategic analysis performed by the program relates a set of general knowledge sources to the details of the particular game. Among other properties, this analysis determines the relative value of the different pieces in a given game. Although METAGAMER does not learn from experience, the values resulting from its analysis are qualitatively similar to values used by experts on known games, and are sufficient to produce competitive performance the first time the program actually plays each game it is given. This appears to be the first program to have derived useful piece values directly from analysis of the rules of different games.

Experiments show that the knowledge implemented in METAGAMER is useful on games that were unknown to its programmer in advance of the competition and make it seem likely that future programs which incorporate learning and more sophisticated active-analysis techniques will have a demonstrable competitive advantage on this new problem. When playing the known games of chess and checkers against humans and specialised programs, METAGAMER has derived from more general principles some strategies which are familiar to players of those games and which are hard-wired in many game-specific programs.

Part I

Metagame Paradigm

Chapter 2

Introduction to Part I

> In mathematics, if I find a new approach to a problem, another mathematician might claim that he has a better, more elegant solution. In chess, if anybody claims he is better than I, I can checkmate him.
>
> — Emanuel Lasker (quoted in [Hunvald, 1972])

> A human being should be able to change a diaper, plan an invasion, butcher a hog, conn a ship, design a building, write a sonnet, balance accounts, build a wall, set a bone, comfort the dying, take orders, give orders, cooperate, act alone, solve equations, analyze a new problem, pitch manure, program a computer, cook a tasty meal, fight efficiently, die gallantly. Specialization is for insects.
>
> — Robert A. Heinlein

2.1 Introduction

This part of the thesis analyses the existing paradigm within which work in Computer Game-Playing (CGP) is conducted and creates a new paradigm which overcomes the limitations discovered through that analysis.

2.2 The Problem

The introduction to this thesis (Chapter 1) has already motivated a significant problem currently facing the field of CGP: on many of the games we have been using as testbeds it has been possible to achieve strong performance without advancing the scientific goals of Artificial Intelligence (AI). The challenge faced in this part is to explore that problem in detail and, based on a new understanding of the problem, to provide a solution to it. The questions faced in this part include the following:

- How is it possible that programs based on more general and intelligent principles can fail to perform well in competition against programs which use sheer computing power?
- What is interesting about games from an AI perspective anyway?
- Is it possible to find a game for which strong performance could be achieved only by programs which are more interesting in some sense?

2.3 Overview

This part is divided into two chapters. Chapter 3 provides background and motivation about analysing games and gives a survey of past work in CGP as it relates to game-analysis. Chapter 4 then analyses the methodological underpinnings of the field and constructs a new paradigm which overcomes some problems with the current methodology.

Chapter 3

Computer Game Playing

3.1 Introduction

Most current approaches within CGP–including those employing some forms of machine learning—rely on previous human analysis of particular games. Human researchers do most of the interesting game analysis, which makes it difficult to evaluate the generality and applicability of different approaches. It also makes it difficult to demonstrate empirically the success of research which emphasises generality (like work on learning and planning), as the programs which result from such work are usually incapable of performing well against special-purpose programs. This possibility undermines the use of competition as a metric for evaluating progress in AI and poses methodological problems for the field.

Examining and redressing this problem is the central concern of this part of the thesis, and this chapter provides the background and motivation for what will follow. The chapter is broken into two sections. Section 3.2 motivates by means of examples what I mean by *game-analysis*, and argues that understanding and automating game-analysis are two of the major goals behind studying intelligent game-playing in AI. Section 3.3 substantiates the claim that much current work in CGP relies on humans, instead of programs, to perform a good portion of this analysis. Section 3.4 summarises the chapter and points out an important methodological problem confronting the field of CGP which stems, in part, from the very possibility that humans are performing some of the analysis instead of the programs.

3.2 Game-Analysis and Specialisation

Roughly speaking, *game-analysis* comprises the set of processes which operate on an *abstract representation* of a given game, and lead to the development of a set of *specialised* search methods, heuristics, and strategies specialised for that game. The resulting strategies can provide humans and computers with a competitive advantage, relative to a group of other players who may use different strategies.

3.2.1 Fundamentals of Game Analysis

I will not attempt here to draw boundaries between what constitute search methods, heuristics, or strategies. I will instead refer to them all as *conceptual tools* or just *concepts*, which aid a reasoner in some aspect of playing the game. However, the terms *abstract representation* and *specialisation* require further clarification. Also important is a distinction between *active* and *passive* game-analysis. The next sections discuss each of these in turn.

3.2.1.1 Abstract Representation

An *abstract representation* of a game is a set of rules which allow a player to play the game legally, but which is more compact than the *extensive representation* of the game. An extensive representation explicitly lists the transitions and outcomes for each state in the state-space of the game. In contrast, an abstract representation provides a generating function which defines the successors for each state in terms of a set of changes to the state's internal structure. It also defines goals using predicates on states with internal structure. For example, a state-transition rule in an abstract representation of chess dictates that states which contain a knight on a square have as successors states in which that square is empty, a knight is on another square relative to the first square, and all other squares are the same as in the first state. An abstract representation of a game, then, is one which is smaller than the game-tree generated by it.

This concept of abstract representation is basic to all work on games in AI. The reason for stating it explicitly in the definition of game-analysis above is that it is a prerequisite for all such analysis. That is, unless a game has an abstract representation (whether or not a reasoner is aware of it), there is no possibility for the specialised conceptual tools discussed above (strategies, heuristics, and so on) to confer a competitive advantage on that game. In this respect, the approach of CGP differs significantly from pure *game theory* [von Neumann and Morgenstern, 1944], although they both deal with strategies in games.

Game Theory Game theory assumes that games are represented in a *flat* form, as a matrix or tree. It also assumes that players have no resource limitations in terms of time or space, so that they can keep the entire game tree in memory, and can calculate all the possible consequences of each move. Given these assumptions, and as long as we could conceive an algorithm which played the game perfectly in finite time, the game would be effectively *trivial*. This means that the *finite two-player* *perfect-information* games like Chess and Go are considered trivial to this field.[1]

However, when we take into account the resource limitations of the players, it is obvious that a player could never maintain the entire game tree (for a big game) in memory, nor consider all possible consequences of each action. Thus a player must consider possibilities and outcomes selectively, and make decisions based on less-than perfect information.[2] This has also been observed by Botvinnik, who for this reason placed games like chess into the class of *inexact* problems [Botvinnik, 1970]. As the player cannot in general see the exact influence of a move on the final goals of the game, it follows that her reasoning must be *heuristic* [Polya, 1945; Lenat, 1983]. That is, the reasoning must make reference to the structure encoded in an abstract representation of the game.

The preceding discussion thus shows that game analysis, as I conceive of it here, is crucially dependent on the existence of an abstract representation of the game. If an agent had enough resources to manipulate the flat structure, no game-analysis would be necessary: the game would be trivial. And if the game did not have an abstract representation, no game-analysis would be possible.[3]

Having clarified this aspect of game-analysis, I now turn to another aspect, *specialisation*.

3.2.1.2 Specialisation

Specialisation is a matter of degree, and a concept or conceptual tool is *specialised* to the extent that its domain of relevance and utility is restricted to a class of objects.

I distinguish two types (or qualitative regions) of specialised concepts: *game-specific* and *game-assumptive* concepts. In the context of chess, a game-specific search method might be to examine checks (attacks against a king) before other moves. Some game-specific heuristics might be that it is unfavourable to have one's knight on the edge of the board, and favourable to have a king surrounded by pawns. A game-specific strategy might be to build up an attack against enemy pawns which cannot be protected by other pawns.

These concepts are all *game-specific*, in this case specific to the game of chess (*chess-specific*), in that they refer to syntactic elements of this one game, and they

[1] The class of finite two-player games of perfect information is defined formally in Section 9.2.2. Informally, this class contains all games in which all properties of a game situation are known to all players, there are no chance occurrences, and the games cannot continue forever. Backgammon is an example of a game of *imperfect* information, as the outcomes of the dice are unknown. Bridge is an example of a game of *incomplete* information, as players do not know which cards are held by another player. For more on game theory, see [von Neumann and Morgenstern, 1944].

[2] Thus estimating the consequences of a position is like estimating the millionth digit of π: in principle it has just one value, but in practice our estimates vary with increased computation ([Good, 1977]).

[3] The comparison of representations in terms of abstractness or simplicity appears related to the study of *algorithmic information theory* [Chaitin, 1987], which measures the complexity of a concept in terms of the size of the smallest program which can characterise it.

would be meaningless in the context of another game. For example, it does not make any sense to examine checks or attack weak pawns in the context of noughts and crosses (tic tac toe). It is not a question of the concepts being less valid or useful, there simply are no referents for these pieces.

Continuing in the context of chess, there are some concepts which would still be interpretable outside the context of chess, but for which there is no reason to expect them to be *valid*. For example, the strategy: "capture all the enemy pieces" is among the most basic of chess strategies, and in fact is useful in some other games as well, like checkers. However, for the game of lose-chess, in which the goal is to lose all your own pieces, this concept is clearly not useful (it may in fact be counter-productive), although it still is meaningful. This lack of utility, moreover, is not something that needs to be discovered through practice; rather, it follows almost immediately from the goals of lose-chess that it would not be useful. The reason for the failure of this strategy to transfer between the games is that it is based on a set of assumptions about chess. When applied to games in which the same assumptions hold (e.g. checkers), the concept retains its utility, but when applied to games for which they do not hold (e.g. lose-chess), the utility is lost. I call this type of specialised concepts *game-assumptive*, then, because they assume without explicit statement some set of properties of the game in order to be useful.[4]

Game-assumptive strategies are related to *domain-dependent* search control information, as discussed by [Ginsberg and Geddis, 1991]. In both cases, the conceptual tools expressed in the language of the problem can be viewed as the result of a two-step analysis. The first step observes that a problem has a certain structural property. Ginsburg and Geddis call this a *modal* fact. The second step applies some problem-independent conceptual tool to this modal fact. Caching both steps into one results in knowledge which functions as a problem-dependent conceptual tool.

There exists a test to determine the extent to which any given concept is specialised. Firstly, determining whether a concept is game-specific is straightforward: the concept becomes meaningless even in a new game which differs from the original only by a substitution of symbols. So, for example, in chess, changing "knight" to "horse" renders meaningless all heuristics which refer to "knights".

Testing game-assumptive strategies, on the other hand, is less straightforward. The "capture all enemy pieces" strategy above, for instance, still has meaning after relabelling the pieces. Instead, we must determine the assumptions upon which the utility of the strategy depends. The way to do this is to make changes to the rules of a game and observe how these changes affect the specialised concepts. Any time we make meaningful changes to the rules and goals of a game, even to rules which appear

[4] Another example of a game-assumptive concept in both chess and checkers is that of *centre-control*. Current programs are given specific bonuses for controlling the squares in the geometric centre of the board. If the topology of the board were changed to be on a cylinder, the relevant concept should change to *midline-control*. This reveals an assumption about board topology implicit in the implementation of game-specific centre-control heuristics.

insignificant, strategies which were effective for the old game cease to be useful as their assumptions become invalidated. New strategic possibilities arise to exploit the differences, which thus further affect existing strategies for the game. This is why people who play particular games are so concerned when even an innocuous-seeming change to the rules is suggested. A contemporary example of this can be found in the game of soccer.

Soccer The players and fans of the game of soccer are presently debating a change which prohibits defenders from passing the ball back to the goal-keeper. This one change will put defenders under more pressure, thus making attack more advantageous, eliminate some of the major power of the goal-keeper, change the strategic positioning of offensive and defensive players, and ultimately lead to a dramatic restructuring of soccer strategy. Among other points, this illustrates how soccer strategy governing the relative effort devoted to attack and defence assumes certain relative levels of difficulty for the two tasks.

3.2.1.3 Active and Passive Analysis

The above example showed how apparently small differences in rules can have major impacts on strategies which are sensitive to them. It is also clear from this example that many of these implications can be conjectured without having yet played or observed a single such game. Clearly any soccer team which modified its strategies passively, by waiting for its existing strategies to be defeated, would be at a great disadvantage to those teams which actively analysed the rules in search of potential for strategic advantage.

With this observation, I shall now distinguish these two types of game analysis. The first type, *active* or *eager* analysis, produces specialised concepts in the absence of experience with situations in the game in which the conditions related to the concept have applied. This type of analysis is also often termed *first-principles* problem solving. An example of active analysis in chess would be realising that it is probably easier to checkmate the enemy king when it is on the edge of the board than when it is in the centre (this is true because on the edge the king has fewer escape squares). This analysis might then lead to a strategy to get the enemy king to the edge of the board in order to checkmate it, or to keep one's own king near the centre of the board to counter such a strategy of the enemy.

The second type of game analysis is *passive* or *lazy* analysis, which produces concepts in response to situations in which significant conditions have already been achieved. This type of analysis is often termed *learning from experience*. Using the example above, the same strategy may be produced in a passive fashion when attempting to learn from a situation in a contest in which a player's king was checkmated when it was on the edge of the board. The player suffering the loss might observe that the shortage of escape squares was a disadvantage, and that this is true for all squares on the edge. As for the active case, this might then produce a strategy to force the enemy king to the edge, or keep one's own king off the edge. Unlike in the active case, however, this analysis could only occur after the fact, and does not explain how this situation (checkmate on the edge) would have come about in the first place.

Another example of a concept which could either be produced through active or passive analysis is a specific type of *fork*, in which a player threatens to achieve two or more goals simultaneously, not all of which can be blocked. As for the king-on-edge strategy, a particular type of fork could be determined actively by construction, or passively when extracted from a position in which it has been achieved.[5]

As both forms of analysis could in principle produce the same specialised concepts, it may seem that this distinction between active and passive analysis is not important. In fact, as will be discussed in Section 3.3, recent research in CGP has tended to focus on passive analysis (learning), with the apparent belief that this is sufficient to explain the development of specialisation and strong performance in game playing. However, there are important advantages to understanding active analysis as well as passive. First, active analysis potentially provides a marked performance advantage to reasoners who use it over those using only learning. As passive analysis only learns after the fact, players who rely on it will develop specialised concepts later than active analysers, generally after losing to the active analyser. In a long-term tournament, then, we would expect players using active analysis to achieve much higher scores than those who do not.[6]

Second, only active analysis is sufficient to explain the achievement of goals (or other learning situations) in the first place. As such goal achievement is sometimes likely to occur only with the aid of specialised concepts, only active analysis can explain the development of such concepts (or *knowledge origins*). Another example from chess may illustrate this issue. It is well known that obtaining a *passed pawn* may increase one's winning chances in some positions. An active analysis approach would have a player discover this concept from first principles. A passive analysis (lazy learning) approach would have a program discover this in response to a given situation, presumably following one player in the game creating a passed pawn, promoting it to a queen, and going on to win the game. But since it requires many careful moves to achieve the promotion of a pawn, this will generally not happen unless one player is actively trying to do so. Now, two *lazy* learning programs competing against each other will each try to gain their knowledge from their opponent. But since neither one

[5]Active and passive analysis leading to the construction of chess strategies has been considered by [Flann, 1992] and [Flann, 1990; Flann and Dietterich, 1989; Collins *et al.*, 1991; Tadepalli, 1989b], respectively (among others). Active and passive analysis leading to construction of forking strategies has been considered by [Epstein, 1991; Allis, 1992; Collins, 1987; de Grey, 1985] and [Epstein, 1990; Minton, 1984; Yee *et al.*, 1990], respectively (among others).

[6]Eventually, a game may become so well analysed that good imitation becomes more important than innovation. In this case a passive analyser would in the limit score equally with a player who also used active analysis. If the rules or the competitive context of the games changed periodically, the advantage to active analysers would be more pronounced.

has this knowledge yet, they are unlikely to observe it in practice, so neither is likely to learn this concept.[7]

Thus, active analysis is an important ability, with some advantages over passive analysis in terms of both the use and development of specialised concepts. This is of course not to say that passive analysis is not also important. Since it is grounded in concrete situations, passive analysis does in fact have advantages over active analysis. First, it does not run the same risk of thinking about possibilities which may never occur, or of neglecting details which render the more general conclusions to be incorrect.[8] Second, the analysis *might* be be simpler in the context of a specific case, rather than for the general case. To the extent that such specific cases tend to recur, the passive approach can focus its efforts on more useful aspects of the game.[9] Finally, there are some details of game-playing which cannot be known in advance, for which a reasoner must adopt a "wait and see" attitude. An example of such a detail involves the preferences and abilities of other players.

3.2.2 Examples of Game Analysis

The preceding discussion has defined and briefly illustrated some important concepts related to game-analysis. To make these concepts even clearer and intuitive, this section presents two more extensive examples.

3.2.2.1 Number Scrabble

Number-scrabble is an example of a game whose extensive representation would be very large, but which has a small abstract representation which can be used to derive a straightforward strategy. The rules are as follows (from [Banerji and Ernst, 1971]):

> **Number Scrabble:** The nine digits, 1, 2, . . . , 9, are used to label a set of nine blocks, which constitute the initial pool as follows:

---

[7] This may not be the best example to prove this point. It is just about imaginable that a lazy program would observe enough random cases of pawns promoting to queens and later winning that it could work out the concept of passed pawns. Some clearer examples of strategies unlikely to be learned passively are those in the version of Knight-Zone Chess with Rule 3 (see Section 3.2.2.2).

[8] [Collins, 1987] discusses an example from American Football, where people develop a strategy for the game at an abstract level before considering the physics of ball movements. They might then use this strategy in practice, only to discover later that it is infeasible.

[9] [Flann, 1990] and [Tadepalli, 1989b] make this point.

1	2	3
4	5	6
7	8	9

> At his turn each player draws a block from the pool. The first player able to make a set of three blocks that sum to 15 is the winner.

On the surface, this might seem to be a difficult game to analyse. Examining the entire game tree would require considering all blocks the first player might choose, then all the opponent might choose, and so on, requiring approximately 9! nodes.[10] However, it also seems clear that we should be able to exploit the compact representation of this game, to play it reasonably well without examining the entire tree.

In fact, some simple considerations on the relationship between the rules and the goal do lead to a simple strategy for this game. First, all the moves (blocks) available in a position, except for the move actually played, remain available to both players in future positions. Second, a goal is achieved when some combination of unique moves has been played. Thus, as a first approximation we can heuristically order alternative moves based on the number of different goals they further for a player and/or block for the opponent, specialising a *maximise options* heuristic, also described as a *multiple-goals* or *least-commitment* principle. Ranking moves for the initial position, then, we have Table 3.1.

Block	*Number of Goals*
5	4
2, 4, 6, 8	3
1, 3, 7, 9	2

Table 3.1: Number of goals involving each block in *Number Scrabble*.

These heuristics thus break the different blocks into three classes, depending on whether they form part of 2, 3, or 4 winning triples, and would thus suggest block 5 as the most valuable, as it participates in the greatest number of winning triples. In effect, the goals here totally determine the relationship between individual moves.[11]

[10] The figure is actually smaller because games will not continue past won positions, and also a game-graph representation is smaller than the tree, due to transpositions.

[11] This is always the case for games where one move does not affect the legality of another.

After re-arranging the blocks so that we can connect winning triples by straight lines, so as to use a more readily accessible visual representation, we arrive at the following reorganisation, in which the blocks are now organised in a magic square:

8	1	6
3	5	7
4	9	2

As might be obvious from this representation, playing *number scrabble* by the strategies of “multiple goals” and forking, combined with simple lookahead to block threats, is equivalent to playing the basic strategy for tic-tac-toe. Although this game is discussed in [Banerji and Ernst, 1971], it is used in that work to illustrate the point that a program which realised that the number scrabble was isomorphic to noughts and crosses could use the strategy from the latter to play the former. It is used here to illustrate a different point, that the same more general analysis can independently derive the specialised strategies for both of these games.

3.2.2.2 Knight-Zone Chess

Section 3.2.1.2 showed that it was possible to determine the assumptions behind game-assumptive concepts by making a series of changes to the rules of the game and observing how this affects the utility of these concepts. The following example applies this test to the game-assumptive concept of piece values in chess. The example also shows the ease with which active game analysis is possible, and the potential competitive advantages such analysis could confer on players which use it.

A standard set of values are used by almost all chess programs, but rarely in the computer chess literature is it explained *why* these values should be used.[12] To discover the assumptions upon which these values depend, we begin with the standard rules for chess, and examine how these values would change under a sequence of rule changes.

As all possible moves in chess are based on the presence of different pieces occupying different squares, and as most pieces remain put after each move, a natural way of evaluating a position is by attaching weights to the existence of certain pieces on certain squares. One strategic concept which bears on this is that of *mobility*:

[12]*The Oxford Companion to Chess* [Hooper and Whyld, 1984, page 369] presents a history of “quasi-scientific attempts” to establish piece values within the human chess community. Several of these attempts were based on the ability of each piece to control squares on an empty board. The discussion in this section is similar in spirit to past attempts, but the details are original to this presentation.

other things being equal (*ceteris paribus*, as in [Wellman and Doyle, 1991]), the more options we have in a position, the better off we are. Note that this is similar to the application of the "multiple goals" strategy discussed in the previous example. Applying just this simple concept to Chess, we might weigh each piece on each square by some combination of the following, averaged over all board squares:

- **immediate mobility:** the number of moves immediately available to that piece from that square, and
- **eventual mobility:** the number of other squares the piece could ever reach from that square.

The second term takes into consideration the fact that a knight can reach every square of the board eventually, while a bishop can reach only half of the squares (those of the same colour the bishop starts on). Using the lowest-scoring piece, the pawn, as reference, this calculation gives us values close to the classical chess material values, which are [Hooper and Whyld, 1984, page 369]: pawn=1, knight=3, bishop=3.25, rook=5, and queen=9. These or similar classical piece values are generally the most strongly weighted components of all serious chess evaluation functions.[13]

Now, we can extend this standard game into *Knight-Zone Chess*, by adding the following rule:

Rule 1 *The board is extended by two on each side, the squares in this region are called the* knight-zone*, as no piece is allowed to move into this zone except for the knight.*

The effect of this new rule is illustrated in Figure 3.1. The central 8 by 8 region in the dotted box is the original chess board, and the outer region is only available to knights. One consequence of this new rule is that the position in in the Figure 3.1 is actually checkmate. The ♘c11 attacks the ♚e10 which has no moves, and the ♝d10 cannot capture it as it is prohibited from moving into the knight-zone.

The same considerations which applied to give us material values in chess should transfer to give us values for knight-zone chess. In fact, as this new rule did nothing but increase the range of the knight (as nothing else could move there), and it increased the potential for that one piece, the only difference *must* be an increased value for the knight, relative to the other pieces. In fact, a more sophisticated analysis of chess would have weighted the pieces based also on their ability to capture other pieces on different squares, and perhaps on the ability to thwart such goals of the opponent by moving a piece to squares inaccessible to such pieces. Thus a knight would increase even more in relative value, as not only would it have more squares to travel to, but it would be able to hide altogether from any attacking pieces except the enemy knights. While this analysis does not provide solid quantitative values for this piece,

[13] Within the field of computer chess, [Hartmann, 1987] found that piece values were significantly correlated with the outcome of grand-master chess games. This provides empirical evidence that the piece values have practical utility as heuristics.

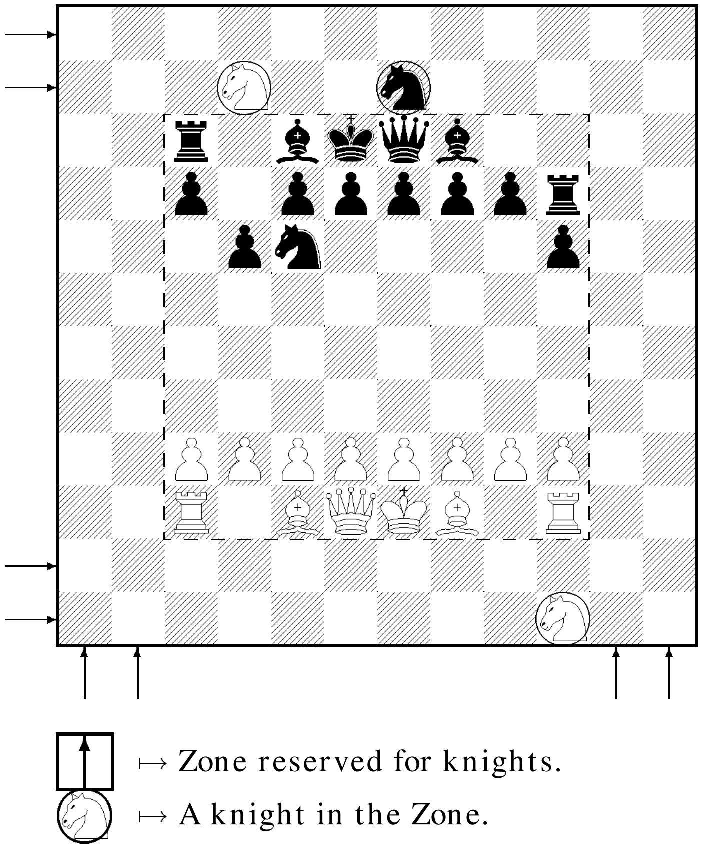


Figure 3.1: A position in *Knight-Zone Chess*.

it clearly suggests a qualitative shift in relative value toward the knight, without having played a single match in this new game. From this one consideration, a set of further strategic consequences also follow (changed priorities on capturing the knight, strategies in which the knight moves to the zone to avoid capture, and so on).

Next, consider the addition of a second rule to the original game:

Rule 2 *If a player moves* any piece *while his opponent has a knight in the knight-zone, he wins the game immediately.*

A little strategic analysis here reveals that the knight-zone is with this rule almost entirely eliminated from importance in the game: no sensible player will ever move his knights into this zone, and so the game effectively reverts back to standard chess.[14] Note again that this conclusion operates on a level much more abstract than the movement of pieces in particular positions, or more concrete still, individual state transitions. Rather, it is almost a kind of theorem-proving on abstract propositions about the rules of the game itself, and relies fundamentally on the abstractness of this representation. If this game were represented as just a game-tree, we could never make conclusions about the utility of achieving certain states without having seen all the terminal positions under them.

Finally, suppose instead that we added the following rule to Knight-Zone Chess, in place of Rule 2:

Rule 3 *If a player moves* a pawn *while his opponent has a knight in the knight-zone, he wins the game immediately.*

An immediate conclusion would be that this has the same effect as the game resulting from Rule 2: as a player can move a pawn in most positions, the knight will be restricted from his region in most positions, and thus worth not much more than in the original game. However, while it is true *in chess* that players usually have pawns to move, the existence of this new rule offers players an incentive to eliminate or block their enemy pawns, in order to make their own knights better. Still without playing a single game, we immediately imagine situations in which a player might sacrifice a piece to block his opponent's last remaining pawn, in order to enable his knight to travel into the knight-zone and become more powerful. In the case where a player's pawn is blocked and his opponent's knight is in its zone, he might exploit this situation by attacking the blocking piece, which is effectively *pinned* to his king, as moving it would allow the enemy to move his pawn, thus winning the game.

From this analysis, we see that the interaction of Rules 1 and 3 creates a change in the values of both knights and pawns, relative to the other pieces, that a complicated pattern of strategies emerges from the introduction of these simple rules, and that

[14] Actually, there is still one situation in which a player would sensibly move a knight into the zone: when it results in checkmate. Readers who spotted this might note that they did so by performing a typical act of active game-analysis.

as a consequence of this the resulting values for all the pieces in general might be radically different from that of standard chess. However, we also observe that much of chess strategy might remain in spite of these changes, as if all the knights were removed from the board in the course of a game, the rules operative in the game would be the same as those in standard chess, in which case pieces should clearly have their original values. As a last note here, it should be clear that a player which did not apply such first-principles reasoning as illustrated here, and instead relied only experience to discover appropriate concepts, would be at a marked competitive disadvantage. In fact, such a player might never discover the strategies which seemed to be so obvious as soon as the rules were presented to us.[15]

3.2.3 Discussion

With these concepts now clarified, game-analysis is then the process by which conceptual tools are produced which are expressed in the language of a specific game, or rely on assumptions about some aspect of the game which may or may not transfer to other problems. Investigation of the process of game-analysis is thus the attempt to provide an answer to the question: where do the strategies, heuristics, features, subgoals, preferences (and so on) come from, when initially there was just the bare definition of the problem? Alternatively, it is the attempt to understand the source of performance strength and competitive advantage across a wide variety of problems.

I summarise this section with a list of observations about the nature of game-analysis and the specialised concepts which it produces, which follow from the examples already provided. In each case, I shall use the term *strategy* to stand in for the set of conceptual tools discussed above.

1. Specialised strategies are dependent on the *rules* of the specific game to which they apply.

2. Useful specialised strategies can be derived through the application of more general principles or analysis to the rules of a given game.

3. This derivation process, and in fact the very possibility for the existence of useful strategies for a given game, requires the game rules to have a compact representation, and that the game be played in a context with constrained resources.

4. An understanding of only passive game-analysis, without an understanding of active analysis, is insufficient to explain the development of competitive strength in game-playing.

[15]This discussion should not be taken to imply that passive analysis would not be useful or necessary for strong performance in this context. It may in fact be the only way to refine strategies or to determine which concepts tend to be more successful empirically. The important point is that passive analysis alone is not *sufficient* for the discovery of these concepts.

3.3 Automating Game Analysis?

Given the generality and ubiquity of the problem, it might seem obvious that CGP would have as a primary focus the increased understanding of game-analysis. On the contrary, this section will show that the focus has been on the use of the *results* of game-analysis, rather than on the process itself.

3.3.1 The Gamer's Dilemma

I begin with a thought experiment which I shall call *The Gamer's Dilemma*:

> Suppose that a researcher is informed that she will soon be given the *rules* of a game, G, played by a group of humans and/or programs. They all are considered to play G at a high performance level. The researcher is given a fixed amount of time to produce a program which will compete against random members of the group, but the researcher is not allowed to communicate with members of the group beforehand. Finally, the researcher will be paid based on the number of games the program managed to win (or draw) out of some pre-specified total amount of games. The researcher's goal is, of course, to maximise the money she expects to receive from her program's play.

As practitioners in the field of Computer Game-Playing, the question now is: what advice and tools might we give to assist this hypothetical researcher? Possible answers to this question reveal the degree of specialisation inherent in existing game-playing methods. Answers which require the researcher to perform game-analysis himself reveal the extent to which the process of game-analysis remains to be investigated.

In what follows, I sketch the advice which might be offered from three main approaches to CGP: knowledge engineering, database enumeration and machine learning.[16]

3.3.2 Knowledge Engineering

Knowledge engineering approaches can be broken into game-specific approaches, game-tree search, and knowledge-based search.

[16]Ideally, we would like to hand over a *general game-playing program* [Williams, 1972], which could study any game for a while on its own (to use its time wisely), and which would become an expert shortly after having encountered this group of players. Although it is unlikely we will ever have a program which is *totally* general, it is useful to bear this ideal goal in mind.

Game-Specific Engineering Approach An example of a game-specific answer would be to enumerate a list of games and current champion programs which play these games. This list might contain the following advice:

> If the game is Chess, then use the latest version of Deep Thought.

If the game given to the researcher happens to be on this list, this advice would be extremely helpful. Unfortunately, if the game falls outside the list, this advice would be of little use. It is becoming a common perception in Artificial Intelligence that such a list of advice is all that many researchers in CGP have to offer.[17]

Game-Tree Search An answer which more accurately reflects the lessons learned by current approaches advocates the use of a *brute-force* search method (e.g., *minimax* with $\alpha\beta$ pruning and singular extensions), combined with extremely fast routines for updating board positions. This technique has proven effective on several games, and some toolkits have been developed to make it easier to apply these techniques to a variety of games (e.g. [Kierulf, 1990; Kierulf *et al.*, 1990]). However, this approach presupposes that the researcher has a good *evaluation function*, which requires specific knowledge of the game.

The burden on game analysis thus shifts to the researcher, who must choose an appropriate set of features and weights for this function. Although there are some general approaches to learning weights (discussed in Section 3.3.4), this approach has offered very little explicit advice about the construction of appropriate features. However, we are now beginning to understand the importance of some features which may be essential in a variety of games, such as *mobility* ([Donninger, 1992; Hartmann, 1987; Lee and Mahajan, 1988; Rosenbloom, 1982]).

Knowledge-Based Search We have learned that exhaustive search may not be appropriate for all games. Therefore we may also advise our researcher as follows: first, find some human who can analyse the game at expert level, then determine an appropriate set of goals and subgoals, and priorities for these goals, and finally write a knowledge-based search program which exploits these ([Wilkins, 1982]). But as in game-tree search, we really have never said much explicitly about how to find useful subgoals, so again the researcher must do the difficult game analysis on his own.

3.3.3 Database Enumeration

While knowledge engineering approaches rely on human analysis by definition, such analysis might seem much less important when programs construct their own database by enumerating a large set of possible positions. On the contrary, the human

[17] The theme of chess-engineering in AI is discussed further in a panel at IJCAI–91 ([Levinson *et al.*, 1991]).

researcher must perform an extensive analysis of a game to determine at least the following ([Allis *et al.*, 1991b; Roycroft, 1990]):

- How to enumerate positions systematically?
- How to avoid generating impossible and symmetric positions?
- Given the above, are there *enough resources* to solve the problem?

By the time the human has answered these questions, it could well be argued that all the game-analysis (if any) has already been done. Perhaps more importantly, this method is applicable only to games that are small enough to be analysed in this way, so the answer to the third question is likely to be negative.

3.3.4 Machine Learning

Machine learning methods for games can be broken into two classes, depending on whether they presuppose the existence of good players.

3.3.4.1 Learning from Experts

Most game-learning methods are designed to enable a program to improve based on watching or playing against *knowledgeable* opponents (e.g. [Samuels, 1967; Tadepalli, 1989a; Epstein, 1991; Collins *et al.*, 1991; Lee and Mahajan, 1988; Levinson and Snyder, 1991; Callan *et al.*, 1991; Tunstall-Pedoe, 1991; van Tiggelen, 1991]). Although it is certainly important to understand how a program (or person) could learn from good players, it is equally important to know *how* those good players became good in the first place. This point was made extensively in Section 3.2.1.3. Until we have an understanding of active game-analysis, progress in this type of machine learning will not save us from having to preface advice to other researchers with the statement: "first, find a human expert."

3.3.4.2 Unsupervised Game-Learning

A much smaller proportion of learning work has considered how programs might become strong players while relying neither on active analysis nor on experience with experts. Most of these approaches can be considered as *self-play* [Samuels, 1959; Angeline and Pollack, 1993; Epstein, 1992; Tesauro, 1993; Axelrod, 1987], in which either a single player or a population of players evolves during competition on large numbers of contests. A related technique, which can also be viewed as a form of self-play, is that of Abramson [Abramson, 1990], who developed basic playing programs which learned to predict the *expected-outcome* of positions if played by random players. This was shown to be effective for constructing evaluation functions for some games.

In principle, approaches based on self-play could give fully satisfactory and general advice to the hypothetical researcher, for example:

> Use an off-the-shelf self-training technique. When given the game rules, choose an appropriate representation for states and strategies in the given game. Have programs play against each other many times, and by the time of real competition, select the strongest evolved program.

It would seem that this advice requires minimal game-analysis on the part of the human. The major issues with this approach are as follows:

1. How much time is necessary to evolve a strong player?
2. How effective is the training method at developing good strategies on different types of games?
3. How much game-analysis must the human perform in order to design an appropriate representation?

With respect to the first and second questions, these are at present unanswered. While the methods appear useful for certain types of problems, Section 3.2 presented several examples of important strategic concepts which appeared difficult to derive without some active analysis.

Unfortunately, these questions are all very hard to answer, because the games, representations, learning methods, and amount of knowledge engineering have varied with each learning system. With respect to the third question in particular, [Flann and Dietterich, 1989] discusses the "fixed representation trick", in which many developers of learning systems spend much of their time finding a representation of the game which will allow their systems to learn how to play it well.[18] [Tesauro, 1993] has produced an extremely strong backgammon program using a training scheme which is claimed to be "knowledge-free", yet so far this method has been demonstrated only for one specific game (backgammon), with which the author was familiar.

3.3.5 Why Specialisation is Bad

To summarise, focussing on particular games can be disadvantageous for the following reasons:

- **Labour:** Much human effort is needed each time we develop a program to play a different game, with limited advice on the real problems.
- **Generalisation:** It is difficult to say what we have learned from our research, beyond performance on particular games.

[18]Note that this point is not restricted to the methods discussed in this section. It is a major problem for evaluating learning systems in general.

- **Evaluation:** It is difficult to evaluate research. Playing a particular game well often means that the researcher, and not the program, has analysed the game well. Conversely, a program which does a more general analysis may not play well against highly-specialised machines.

- **Game Analysis:** We can write successful programs, even learning programs, without understanding the ability actually to analyse games, possibly the most interesting issue in game-playing, from an AI perspective.

Thus, despite our being a field full of experts on getting computers to play games, and having developed world-champion-level game-specific programs, we are forced to leave most of the real game analysis to be done by the human researcher, and not by the computer program.

3.4 Summary

This chapter discussed Computer Game-Playing (CGP) as a subfield of Artificial Intelligence (AI). Section 3.2 discussed the importance of game-analysis, the process by which general and flexible knowledge is applied to a specific new game to produce specialised knowledge which yields competitive advantage on that game. The section suggested that understanding and automating game-analysis are two of the major reasons for studying intelligent game-playing in AI.

Section 3.3 showed that much current work in CGP is centered around engineering the results of human game-analysis into playing programs, and that this has left many aspects of this important process largely un-researched. This aspect of current work in computer game-playing is depicted schematically in Figure 3.2. Here, the human researcher or programmer is aware of the rules and specific knowledge for the game to be programmed, as well as the resource bounds within which the program must play. Given this information, the human then constructs a playing program to play that game (or at least an acceptable encoding of the rules if the program is already fairly general). The program then plays in competition, and is modified based on the outcome of this experience, either by the human, or perhaps by itself in the case of experience-based learning programs.

In all cases, what is significant about this picture is that the human stands in the centre, and mediates the relation between the program and the game it plays. As the human researchers know at the time of program-development which specific game or games the program will be tested on, it is possible that they import the results of their own understanding of the game directly into their program. In this case, it becomes extremely difficult to determine whether the subsequent performance of the program is due to the general theory it implements, or merely to the insightful observations of its developer about the characteristics necessary for strong performance on this particular game. As a consequence of this, success of a game-playing program in

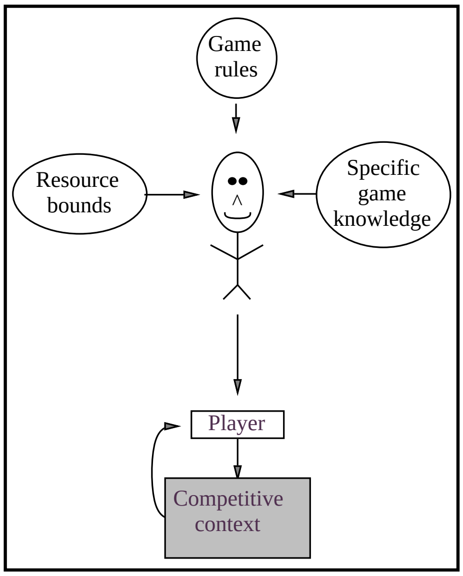


Figure 3.2: Computer Game-Playing with existing games.

competition is no evidence that we have understood or automated any aspect of the important process of game-analysis discussed in this chapter.

With this motivation, Chapter 4 will develop a new methodology for research on games which will make it impossible for humans to mediate the relation between the program and the specific games it plays. The development of competitive programs within this new paradigm will require that future work attempt to understand better the types of game analysis described in this chapter.

Chapter 4

Metagame

> How can we construct mechanisms that will show comparable complexity in their behaviour? They need not play in exactly the same way; close simulation of the human is not the immediate issue. But we do assert that complexity of behaviour is essential to an intelligent performance–that the complexity of a successful chess program will approach the complexity of the thought processes of a successful human chess player. Complexity of response is dictated by the task, not by idiosyncrasies of the human response mechanism.
>
> – Newell, Shaw, and Simon ([Newell *et al.*, 1963], p. 40-41)

4.1 Introduction

Chapter 3 discussed the importance of game-analysis and demonstrated that the current approach to CGP makes it possible to produce competitive game-playing programs without achieving increased understanding or automation of the process of game-analysis itself. This possibility undermines the use of competition as a metric for evaluating progress in AI and poses methodological problems for the field.

This chapter examines the limitations of the current methodology in more detail and presents a new methodology which overcomes these problems. Section 4.2 considers the methodological underpinnings of CGP as a subfield of AI. The section discusses why use of the competitive performance metric is currently problematic and why its abandonment is equally problematic.

Based on this discussion, Section 4.3 presents a new paradigm for conducting research in CGP. Since the source of the difficulty is that humans have foreknowledge of the specific games their programs play, one solution is to eliminate this foreknowledge, and evaluate programs based on competition playing games which were not known in advance to the human. This new paradigm is called *Meta-Game Playing*, or simply *Metagame*. As the human must have some constraints on a problem in order to implement programs to solve it, the human is given a definition of only the *class* of games that the program will play. Playing Metagame with increasingly general classes of games (or problems) provides empirical demonstration of correspondingly general problem-solving ability.

Section 4.4 considers versions of Metagame that do not require full autonomy, which may ease the transition from the current approach (known games) to the new one (unknown games). Section 4.5 briefly summarises the chapter.

4.2 A Question of Values

Given the discussion in the previous chapter, it seems that the field of CGP is at an impasse. The competitive performance metric, one of the most desirable properties of games as a test-bed for AI, no longer seems applicable. In this section, we shall conjecture why this problem has arisen, and in so doing, consider possible ways out of this impasse.

4.2.1 Competitive Performance Metric, Revisited

To some extent, use of this competitive performance metric was motivated by the belief that the complexity of a problem induces a corresponding complexity of any system which can solve it. This view, which is illustrated by the quote from Newell and Simon at start of this chapter, is in some ways liberating for AI: it gives us an objective measure of intelligence, which is methodologically more desirable than many more subjective measures, like similarity to human processing. Along this view, we just have to develop programs which do the right thing, let them compete against each other, and we can conclude that the better program has more relevant complexity, i.e. it is more intelligent.

This methodological assumption, which allows us to direct our energies on developing a solution to the problem which maximises our evaluation criteria, can be schematised by the desired (or assumed) relationship:

$$G \propto E(S, P), \tag{4.1}$$

to be read: G is proportional to E of S and P, where G is our set of research goals, P is a problem we shall try to solve in order to further our goals, and E is the evaluation criteria we will use in determining the extent to which a proposed solution S was successful. In English, this equation says that the extent to which our goal is advanced is proportional to the degree to which our proposed solution measures on our performance criterion, given that we have chosen a particular problem as a test-bed.

Unfortunately, this apparent proportionality, upon which much of Simon's early methodology was based,[1] turned out not to be the case: better programs do not have the required complexity we thought they would have to have, and in fact we are not addressing many of the interesting issues we thought we would be required to address in making a good game-player.

With this realization, the field is in a dilemma: meeting the performance criterion of playing a game well is not necessarily related to approaching the goal of understanding intelligent behaviour. This is a problem for *research* (what should our goal be), *evaluation* (how do we know if we achieved it, and to what extent?), *generalisation* (given some research on a problem, and an evaluation of it, how can we generalise it to other areas?), and *focus* (which game meets our goals the best?).

4.2.2 Changing Priorities

Seeing the breakdown of this critical assumption, we can attempt to restore the desired proportionality by modifying any of the components appropriately, while preserving the other two. In fact, we can place current research in game-playing into three main camps, based on whether they differ from the original approach in terms of their *evaluation criteria*, *goal*,or *problem*, while holding the others constant.

4.2.2.1 Change the Evaluation

The first approach, and the one which has been taken by most games researchers in AI who consider themselves in AI *proper*, is to keep the research goal fixed (build intelligent game-playing programs), keep the problem fixed (like chess), but change evaluation metrics. Proposed solutions are then evaluated along dimensions such as psychological feasibility, psychological similarity, whether the program uses rules for reasoning, does learning, etc.

However, by making this move, this camp encounters problems: by shifting our evaluation metrics, we lose the competitive performance criterion which made games a good domain in the first place. These other dimensions are much harder to measure, more subjective, and purely descriptive. In fact, although such research uses games as a test-bed, it is not strictly proper to say that the resulting systems *play* games. Nobody in AI proper would want to have their psychologically realistic model play a game against Deep Thought: regardless of who might win, they would claim that the psychological program is doing something interesting, while Deep Thought is not. This does not, however, change the fact that the problem was chess, and Deep Thought was the winner.[2]

---

[1] As illustrated in the quote at the start of this chapter.

[2] MORPH ([Levinson and Snyder, 1991]) is a psychological model of pattern learning in chess. While the ideas behind the system are fascinating, at the moment its performance is extremely weak, and one of the major difficulties with this research is to find a convincing way, in the absence of performance, of showing that this system is really doing something intelligent. Similarly, PIONEER [Reznitsky and Chudakoff, 1990] is a chess program which does not actually play chess competitively. The program is

4.2.2.2 Change the Goal

The second approach, and the one which has been taken by the community which calls itself *computer game playing* but not necessarily AI, continues to work on standard problems (like chess), continues to evaluate proposed research on the basis of performance in competition, but no longer holds to the lofty goals of AI. Forget about intelligence *per se*, they argue, and concern ourselves with building good programs to do tasks which normally required intelligence for humans.

With this modified goal, playing better is definitely a sign of having developed a better program to achieve a goal (by definition), so $G \propto E(S, P)$ certainly holds. This camp also stresses a similar approach for other fields (like theorem proving by brute-force, etc).

However, there is still something highly unsatisfactory about this method of restoring the proportion. This is that research in this area has little to do with AI anymore, as the goals of intelligent processing, learning, generality, and flexibility, have been eliminated wholesale from the statement of their research goals. Nor do they appear in their evaluation criteria either.

In fact, the elimination of these goals, coupled with the raw performance criterion, leaves us with a "no holds barred" philosophy in which winning is everything. There is no reason to use general methods, in fact practitioners here do just the opposite. There is, in fact, no reason at all to stop the human researcher from doing, in advance, as much of the analysis of a particular game as he can do, in order that the program may be as stupid but efficient as possible. But now we have really lost something, as this kind of analysis was just the type of analysis much of AI was interested in understanding in the first place.

Researchers in AI have become increasingly concerned that this is the approach taken by games researchers, and that for this reason CGP should not be considered a proper domain for AI research. This was the subject of a recent panel at IJCAI-91 [Levinson *et al.*, 1991], in which Jonathan Schaeffer summarised the problem as follows:

> An entirely new field of "computer chess" has evolved, with the emphasis on chess performance and chess research – not generally of much interest to the AI community. ... The unfortunate correlation between program speed and performance encourages short-term projects (speeding up a move generator) at the cost of long-term research projects (such as chess programs that learn).

evaluated on the extent to which the analysis it provides for a given chess position is similar to that performed by a chess master.

4.2.2.3 Change the Problem

The third approach, and one which often winds up midway between the other two camps, attempts to preserve all that was good about the lofty AI goals, and the rugged performance criterion, by focussing on different problems which, unlike chess (given our current understanding), really do seem to require intelligence.

If we could find a game such that any program to play it well is more likely to be interesting for AI, then we could still evaluate programs to play it based solely on performance, and direct our efforts toward optimising this performance, but rest assured that in so doing we would not be straying from our scientific research goals.

So this approach remains part of mainstream AI, evaluation is easy, and the goal is met if the problem matches the desiderata implicit in the goal. This leaves one question outstanding: *which problem*?

4.2.3 How to Change the Problem

There are two ways to change the problem: we can *find* an *existing* problem or we can *design* a *new* problem.

4.2.3.1 Finding an Existing Problem

The standard solution to this question is to find some other known game which has not received much attention. This new game should be more difficult, with the implicit hope that this will force the performance evaluation to be indicative of goal achievement.[3]

So, now that it appears clear that we can achieve extremely strong performance in chess without addressing many of the important goals of AI, perhaps Go will become the next game to be focussed on as a research testbed ([Berliner, 1978]) (it certainly does look hard).

But this approach still is not fully satisfactory: it requires the hope that strong performance on the game cannot be achieved by other means. This is because this approach still keeps other evaluation metrics implicit, not quantifiable. If someone manages to construct a strong player for our new game, but not in a generally interesting way, what are we prepared to say?

4.2.3.2 Designing a New Problem

This all suggests that we may not necessarily have chosen the problem, and method of evaluation, appropriately for addressing our research goals. Instead of keeping our goals and assumptions implicit, an alternative approach is to make our goals explicit,

[3]Of course, we could also use this new problem to explore other issues of representation, reasoning, etc., but this involves changing evaluation measures, the problems of which have already been discussed.

and *design* a problem with our goals in mind. We could expend significant effort making sure it really has the properties we want, so that solving the problem well has a better chance of being more interesting in terms of our explicit goals.

Of course, we might not be able to design the perfect problem for all our goals, but at least designing it with the relation between G and E given P always in mind might increase the chances that we will make good progress in attempting to solve it. Also, when we see an obvious reason why $G \not\propto E(S, P)$, this also tells us useful information about our goals; i.e. it clarifies our ideas and constraints on what we consider AI to be about.

4.3 A New Methodology: METAGAME

To summarise, what we would really like to have is some kind of game for which we can be more certain that playing it well really was evidence of more intelligent processing. Then we could once again justify the use of performance in competition as a way of evaluating good work.

But rather than attempting to *find* a game for which competitive strength is linked *apparently* to progress in achieving AI goals, we would like to *design* a game for which competitive strength would be linked *necessarily* to progress in achieving AI goals. As suggested in Section 4.2.3.2, the first step in this approach is to state our goals explicitly.

4.3.1 Evidence of General Ability

Thus, the crux of the difficulties with current methodology is the following:

> AI is interested in the *general ability* to solve wide varieties of problems. By definition, agents who can solve diverse problems better than others must be those with greater general ability. However, to the extent that the designer of an agent has information providing constraints on the problems the agent will solve, the agent's subsequent performance on such problems is only evidence of ability to solve *problems for which those constraints are true*.

Now this insight can be applied to game-playing, as follows. Chapter 3 has defined *game-analysis* as the general process that exploits the abstract representation of a specific game to produce competitive advantage on that game. The chapter showed that the problems of CGP stem from the fact that, in all current work done in this field, humans have full information about the rules of the games their programs play. This makes it impossible to infer, even from strong performance on those games, that the theories which the program embodies are applicable to anything but the specific games the program has played. In particular, success on a known game is no evidence that the program has performed game-analysis.

4.3.2 Meta-Game Playing

This motivates the idea of *Meta-Game Playing* (*Metagame*), shown schematically in Figure 4.1. Rather than designing programs to play an existing game known in advance, we design programs (*metagamers*) to play (or themselves produce other programs to play) *new* games, from a well-defined class, taking as input only the rules of the games as output by a *game generator*. The performance criterion is still competition: all programs eventually compete in a *tournament*, at which point they are provided with a set of games output by the generator. The programs then compete against each other through many contests in each of these new games, and the winner is the one which has won the most contests by the end of the tournament.

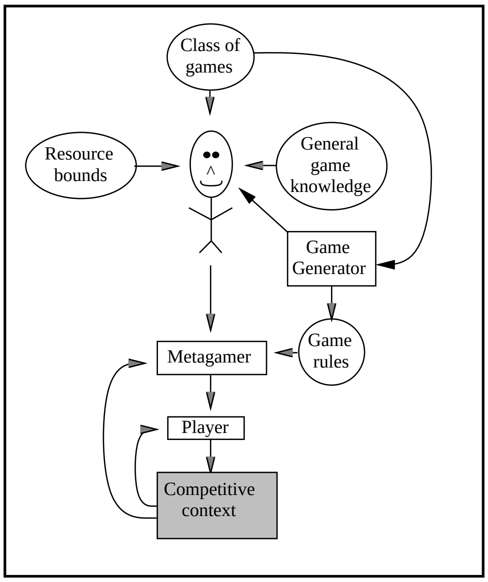


Figure 4.1: Metagame-playing with new games.

As only the *class* of games (as constrained by the generator) is known in advance, a degree of foreknowledge is eliminated, and playing programs are required to perform any game-specific processing without human assistance. In contrast with the role of the human when playing existing games (as discussed in Section 3.4), in this new context the human no longer mediates the relation between the program and the games it plays. Success in competition would then become empirical evidence that a program has performed some of the game specialisation that is potentially performed by humans in the current approach.

4.3.2.1 Quantified Generality

Of course, the human in this new context still mediates the relation between the program and the *class* of games it plays. This means that successful programs will only be evidence of a general ability to solve problems in the class of games. But by making the class explicit, we are able to quantify the level of generality achieved. Moreover, we can begin with classes which represent only moderate generalisations over tasks we have already looked at, and gradually move to more general classes of problems as scientific understanding develops.

4.3.2.2 Evaluating Success in Metagame

It is important to emphasise that the performance criterion is still competition:

> For any given class, strength in competition is the sole metric for comparing different solutions.

This is one of the main benefits derived from designing the problem with an explicit connection to the research goal. Since we only infer general ability relative to the constraints known in advance, competitive advantage subject to these constraints is necessarily correlated with general ability subject to the same constraints. For example, if the class of games is known in advance to contain *only chess*, and the human has no information about the opponents the program will play, it follows trivially that better general chess-players will be stronger programs in this context.[4] If we want to claim that the theories embodied in one program (such as learning or planning) are more generally interesting than those embodied in another in the absence of competitive advantage, this can be tested by applying both theories to a more general class of problems.

4.4 Competitive Contexts

The general idea of Metagame thus formulated would pit computer programs against each other, and present them with a series of games to play. In fact, there are at least three competitive contexts in which researchers can address the challenge of Metagame:

4.4.1 Autonomous Metagame

This is the original problem, schematised in Figure 4.1, where we all make programs which then must read and play the games on their own, without further human intervention.

[4] If there are sufficiently many (and varied) opponents that the human derives no benefit from knowing the opponents in advance, this factor is not important. Otherwise, we would not be justified in inferring general chess-ability from the competition, as the human may have performed some of the *opponent-analysis* for the program.

4.4.2 Mediated Metagame

This version can be viewed as a programming competition. At fixed points throughout the period of the competition, a new game is generated and sent to the competitors. Each competitor then has a fixed amount of time in which to produce a program to play just that game, after which the programs play each other on that game. At the end of the competition, a number of new games and sets of programs to play them will have been produced, and the winner is the human who scored the most total wins.

This is interesting for several reasons. First, it is a much easier problem. There is no longer any necessary reason to have the programs analyse the games, and humans can have fun doing the game analysis, testing their ideas on their programs, etc. There is every reason to make the programs as efficient and specialised as possible. However, there is still the flavour of Metagame: programmers have very limited time to produce each program, so they would be wise to bring as many pre-designed tools as possible. Thus they will in effect be motivated to develop their own *Metagame workbench*. By seeing the tools different players developed and how useful they were, we will gain more of an understanding of the reasoning processes which go into producing a program to play a particular game, and will also move game-playing research to a more general level. Also, it will probably still pay to let programs tune their own parameters while in competition, so there is still a real opportunity to have learning methods result in significantly improved performance (something we are not as likely to see on well-known games like chess or go).

4.4.3 Synergistic Metagame

As in the mediated version, humans will be given the rules and will have time to produce whatever program they would like to bring with them to the competition. Then the humans play against each other (or against Metagame-playing programs), each able to consult with his program in any manner he chooses, or not at all. As in the previous two versions, the goal is to win the most contests across all games and opponents. However, this version allows the human to do even *more* work on the game, playing it entirely on his own if he so chooses. But again, it is possible that having certain tools will allow him to play it better. This again encourages automation of certain types of reasoning, provides a forum in which learning might still help. It seems likely that the resulting tools will again show us which aspects of analysis can be automated, and to what effect.

4.4.4 Incremental Research

These different ways of playing Metagame show the utility of the idea, and provide a way to move more slowly toward the fully-automated problem, also in a way which may motivate more interest from humans. It is also interesting that both of the more interactive approaches still encourage generality, flexibility, and transfer of responsibility for some of the game analysis from human to program, which are after all the main goals of Metagame.

4.5 Summary

This chapter has considered a set of methodological problems currently facing the field of computer game-playing (CGP), which in part explain why much of the important process of game-analysis has yet to be understood. As a solution to these difficulties, this chapter has introduced Metagame, a new paradigm for computer game-playing in which humans can no longer mediate the relation between a program and the game it plays. Within this new framework it becomes possible and necessary to address many interesting and important questions about game analysis, which until now has largely been performed by humans and built directly into programs. Meeting the challenge of Metagame will shift the field of computer game-playing back from an engineering to a scientific discipline, wherein winning a game would again be an indication that the program, and not simply its programmer, is doing something intelligent.

Chapter 5

Summary of Part I

This part of the thesis analysed the existing paradigm within which work in Computer Game-Playing (CGP) is conducted, and created a new paradigm which overcomes the limitations discovered through this analysis.

Chapter 3 defined *game-analysis* as the general process that exploits the abstract representation of a specific game to produce competitive advantage on that game. The chapter showed that many of the problems of CGP stem from the fact that, in all current work done in this field, humans have full information about the rules of the games their programs play. This makes it impossible to infer whether the performance of a program playing a known game is due to the AI theories which the program embodies, or due to human analysis and engineering of that specific game.

Chapter 4 introduced Meta-Game Playing (Metagame), a new paradigm for research in game-playing in which we design programs to take in the rules of unknown games and play those games without human assistance. Strong performance in this new paradigm is evidence that the program, instead of the human, has performed the analysis of each specific game.

It should be emphasised that Metagame itself is not a concrete research problem, but rather a paradigm and a methodology within which concrete research problems can be formulated and investigated. In order to construct a concrete Metagame research problem, it is necessary to define a specific *class* of games to be played, to develop a *game generator* to produce new instances from this class, and to *analyse* the resulting problem with respect to a set of stated goals.

Part II of this thesis will construct a specific Metagame research problem, called *Metagame in symmetric chess-like games*, or simply SCL-Metagame. This problem is a useful area for research in its own right. Moreover, the construction of the problem serves as an example of a practical application of the general idea of Metagame. This should be useful for future research in the construction of different Metagame research problems. Part III will then document progress to date in addressing the new concrete research problem, SCL-Metagame, developed in Part II.

Part II

SCL-Metagame

Chapter 6

Introduction to Part II

6.1 Introduction

Part I of the thesis has introduced a new paradigm for research on games, called *Meta-Game Playing*, or simply Metagame. This part of the thesis discusses both the general issues involved in the construction of concrete Metagame research problems and the construction of one specific Metagame research problem, called *Metagame in symmetric chess-like games*, or simply SCL-Metagame.

6.2 The Problem

Although the abstract idea of Metagame is straightforward, a fair amount of work must be done to develop a concrete problem that can actually be addressed. First, we need to define a *class* of games for which performance will be linked explicitly to our goals. Second, we need a *game generator* which produces new instances in this class. Finally, we must *analyse* the resulting problem to ensure that performance on this problem will be linked to progress on our stated goals. This section provides some general ideas for addressing the issues of class definition and game generation, and the remaining chapters in this part of the thesis provide an example of how the various components have been addressed concretely.

6.2.1 Class Definition

Many different variants of Metagame can be played, depending on what class of games it is based upon. Here we state a few desiderata for classes of games.

- **Coverage:** A good class should be large enough to include several games actually played by humans. This encourages us to generalise the lessons we learned from working on the specific games included. Also, existing games have a known standard of performance and existing bodies of theory. These can be used to assess the extent to which a metagamer's game-analysis yields competitive performance on specific games. If a metagamer is unable to derive important strategies or is otherwise weak on a known game, this reveals areas for future research. Note that the opposite of this does not necessarily hold. That is, a metagamer that plays a set of known games well is not by virtue of that to be evaluated as a strong Metagame-player. As the games are known in advance, it cannot be proven that this strength does not derive from human analysis of those specific games.[1]

- **Diversity:** In addition to known games, a class should be diverse enough to include arbitrarily many possibly different games, to discourage researchers from exhaustively analysing each possible game and still building their own analysis into the program.

- **Structure:** A class should still be small enough to represent the structure which makes the individual games appear similar. While chess-like games and trick-taking card games seem like appropriate generalisations of existing games, the class of arbitrary theorem-proving games appears to be too general.[2]

- **Varying Complexity:** The generated games should be of varying degrees and dimensions of complexity, such as *decision complexity* and *search complexity* ([Allis *et al.*, 1991a]), so that different games afford different analysis methods. This also enables interesting experiments to test the utility of alternative methods with respect to varying degrees of complexity.

- **Extensibility:** It should be easy to generalise the class to increase the coverage of known and unknown games. This makes it easy to conduct incremental research, and reminds us that achieving strong performance in a fixed class is not a stopping point, but rather an indicator that it may be time to move to a more general problem. The knowledge that the next problem will be a small generalisation of the current one encourages us to approach the current problem in the most general manner still consistent with strong performance.[3]

One type of game which has been studied extensively is *positional games*, in which pieces are never moved or removed once they are played. This class of games has been the domain for several general game-learning systems ([Epstein, 1991; Koffman, 1968]), and could easily serve as a Metagame class definition.

[1] This point is crucial. One way to build a very strong game-player for a set of known games is to combine specialised programs for each game into a monolithic program with switches to recognise which game is being played, and thus which component program should be used to play it.

[2] The class of *mathematical games* ([Berlekamp *et al.*, 1982]) is also fully general, which suggests that this class might be inappropriate for Metagame in the near future.

[3] That is, if unnecessarily general techniques do not much hurt performance on the current class, it pays to develop them, as this will prove advantageous when the class is later generalised.

However, this class is both simple and regular (thus falling short on several desiderata), and there are well-researched games which fall outside this restricted class. In particular, it would be interesting to play Metagame based on a class complex enough to include the chess-like games which have received much of the attention thus far in CGP. The choice taken in this part of the thesis is to develop the necessary components to play Metagame in the class of *symmetric chess-like games*.

6.2.2 Game Generator

Given a class of games, there are two ways to produce new instances within this class: we can either use human-generated games, or program-generated games.

6.2.2.1 Human Game Generators

The easiest way of obtaining games would be to have some humans design a set of games within the class. They must provide the games to the programs once the competition has begun, i.e., the developers of these programs can no longer help in the game-specific analysis. This procedure has the advantage that the game designers could try out their games in advance, and make sure that they are interesting and playable. However, this also has a major disadvantage, in that it forces playing-program developers to *predict* which types of games these humans will actually produce. Since human game designers may be very creative, they are an unknown variable from the perspective of scientific experiments. Thus, while we would hope that our programs could play human-generated games within a class, and we may even test our own programs against games we design, the unknown human element may cloud research issues, at least in the short term.

6.2.2.2 Programmed Game Generators

The alternative is to develop a program to generate new games within a class. If the program is transparent and available to all researchers, this has the advantage that everyone will know what kind of games to expect, which is more desirable from a research perspective.[4] However, this also has a potential disadvantage, in that the generated games may not actually be interesting. Three points may be made in connection with this concern.

Intelligent Game Design First, this concern introduces an important and general issue, for which there is no simple answer. This is that *intelligent game design* is an interesting and difficult research issue in its own right. Games which survive ([Allis

[4] Another way of stating this is that writing programs to play any games generated by humans may result in researchers attempting to hit a moving target. A transparent generator is needed to make the problem well-defined.

et al., 1991a]) do so because they provide an intellectual challenge at a level which is neither too simple to be solvable, nor too complex to be incomprehensible. Understanding the processes by which games are created and evolve, from a computational perspective, would be a valuable complement to the analysis of strategies for playing games which has been the focus of research in computer game-playing thus far.[5]

Interestingness requires intelligence Second, although this will matter to human observers, it will not make much difference to the programs whether they are playing interesting or boring games. In fact, if we could develop a program which, upon consideration of a particular game, declared the game to be uninteresting, this would seem to be a true sign of intelligence! So when this becomes an issue, we will know that the field has certainly matured.[6]

Fairness is simpler Third, it should be possible to develop a class and generator in a manner which increases the chances that generated games within this class will at least be *fair* for both players, so that the games will be more interesting to human researchers. Chapter 7 addresses this goal by defining a class in which all the rules are *symmetric* between both players.

6.3 Overview

With this motivation, Chapter 7 presents a definition for a class of games called *symmetric chess-like games*. Chapter 8 presents a generator for this class and discusses an interesting new game produced by the generator, called *Turncoat-Chess*. Chapter 9 then analyses the class as constrained by generator to ensure that performance on this new problem is linked necessarily to progress on our stated goals.

[5] The link between game-evolution and game-playing may be even tighter than we imagine. Games often change when strategic analysis has rendered them either too difficult or too boring, and each change to a game introduces new opportunities for strategic analysis, so that games and their strategies evolve together.

[6] It would be interesting to allow programs to negotiate over draws, to avoid them playing out games that neither player can win. However, this ability complicates the rules of competition, and so will be left as an idea for the future.

Chapter 7

A Class of Games

> A player can see the board and the pieces and comprehend the pattern. But if a pawn – or, in fairy chess, a night-rider – could see the board from the viewpoint of a player – what would be his reaction?
>
> – Henry Kuttner, *Chessboard Planet* ([Kuttner, 1983])

7.1 Introduction

This chapter addresses the construction of a class to be used as a basis for Metagame-playing. Section 6.2.1 provided several desiderata for a class for Metagame. Two of these, *coverage* and *structure*, desire a class which generalises the problems which have been the focus of current research, in a way that preserves enough of their structure to render this research still relevant to the generalised class. To this end, we have defined a class based around the chess-like games, like Chess, Checkers, Chinese Chess, and Shogi.[1] In addition to having served as research test-beds, these games share a perceivable structure. Moreover, there already exists another field of research, under the name of Fairy Chess ([Dickins, 1971]), which has as a goal the generalisation and analysis of precisely these types of games.

The *varying complexity* desideratum was that a class have a decent proportion of problems which are different and difficult enough that they can serve as a basis for comparing and evaluating different approaches. If too many of the games which the programs play are heavily one-sided, to the extent that one of the players has enough of an advantage that he can always and easily win (and such that no particular insight is necessary to realize this), we would lose the precious performance criterion which formed part of the motivation for Metagame, and which allows us to say that a program really is better because it wins more games.

[1] This chapter mentions several games which may be unfamiliar. Descriptions of these games can be found in ([Bell, 1969]).

This can be seen as placing a constraint either on the class itself (to ensure that class instances are not too lopsided), or on the generator (to ensure that selected instances are not too lopsided). As intelligent game generation is a difficult area of research in its own right (see Section 6.2.2.2), a natural decision was to constrain the class in a way which increased the chances that the games would be competitive, so that the generator would need as little intelligence as possible. To this end, a straightforward way to constrain a problem so that two aspects are comparable, is to make those aspects in some way *symmetric* in the problem. This can be achieved in games by ensuring that all the rules are somehow symmetric between the two players.

With this, then, we arrive at the basis for an appropriate class definition: symmetric chess-like games. Section 7.2 provides an overview of the class at varying levels of detail. This detail may be skipped over now, but will be referred to throughout the rest of the thesis. A full grammar and formal semantics for this class of games are provided in Appendix A.1 and Appendix B, respectively.

7.2 Symmetric Chess-Like Games

Informally, a *symmetric chess-like game* is a two-player game of perfect information, in which the two players move pieces along specified directions, across rectangular boards. Different pieces have different powers of movement, capture, and promotion, and interact with other pieces based on ownership and piece type. Goals involve eliminating certain types of pieces, driving a player out of moves, or getting certain pieces to occupy specific squares. Most importantly, the games are *symmetric* between the two players, in that all the rules can be presented from the perspective of one player only, and the differences in goals and movements are solely determined by the direction from which the different players view the board.

At the highest level, a game consists of a *board*, a set of *piece definitions*, a method for determining an *initial setup*, and a set of *goals* or termination conditions. Each of these components is defined from the perspective of the *white* player, who initially places his pieces on the half of the board containing ranks with the lower numbers (call this *white's half* of the board). Unless they can move both forward and backward, white's pieces are generally forced to move toward black's side of the board, and vice-versa, which implies that there must be an inevitable point in the game when these opposing forces come into contact.

7.2.1 Definitions

Before we can describe the rules for games in this class, we need a few definitions.

A *board* B is a finite two-dimensional rectangular array $[1..X_{max}, 1..Y_{max}]$, where X_{max} and Y_{max} are the number of *files* and *ranks*, respectively.[2]

[2]Ranks and files correspond to rows and columns, respectively.

Each element of B is a *square*, an ordered pair which is denoted by its position in this array: (x, y).

A *direction-vector* (*d-v*), $\langle dX, dY\rangle$ is a function which maps a square (X, Y) into a square $(X + dX, Y + dY)$. If $dY > 0$, this is a *forward d-v*, and if $dX > 0$, this is a *rightward d-v*.

A *directional symmetry* is a function which maps one *d-v* to another *d-v*. We define three special symmetries:

$$\begin{aligned} forward: \ & \langle dX, dY\rangle & \mapsto \langle dX, -dY\rangle \\ side: \ & \langle dX, dY\rangle & \mapsto \langle -dX, dY\rangle \\ rotate: \ & \langle dX, dY\rangle & \mapsto \langle dY, dX\rangle \end{aligned}$$

A *symmetry set*, SS, is a subset of $\{forward, side, rotate\}$.

An *inversion* is a function which maps one square to another square, and maps one *d-v* to another *d-v*. We define two distinct inversions:

$$\begin{aligned} diagonal: \ & (x, y) & \mapsto (X_{max} - x + 1, Y_{max} - y + 1) \\ & \langle x, y\rangle & \mapsto \langle -x, -y\rangle \\ forward: \ & (x, y) & \mapsto (x, Y_{max} - y + 1) \\ & \langle x, y\rangle & \mapsto \langle x, -y\rangle \end{aligned}$$

Applying one of these inversions to a square or *d-v* thus produces the corresponding square or *d-v* from the perspective of the other player.

A *symmetric closure*, $SC(SS, D)$ of a *d-v* D under a symmetry set SS, is defined inductively as follows:

1. $D \in SC(SS, D)$
2. $S \in SS \wedge D_1 \in SC(SS, D) \longrightarrow S(D_1) \in SC(SS, D)$
3. Nothing else is in $SC(SS, D)$

Thus, a symmetric closure of a direction vector under a set of symmetries is the set closure obtained by applying these symmetries to the direction vector. If the symmetry set contained all three symmetries, then applying this set to direction $\langle 1, 2\rangle$ would yield all eight possible directions of a knight move in Chess. Keeping only the side symmetry would yield the two possible moves of a Shogi knight ($\langle 1, 2\rangle$ and $\langle -1, 2\rangle$).

7.2.2 Global Symmetry

As *symmetric chess-like games* are totally symmetric, it is possible to present the entire set of rules for a given game (movements, capturing, initial setup, and goals) from the perspective of one player only. The rules also select one of the two *inversions* defined in Section 7.2.1, and the rules for the other player can be determined by applying this inversion to every square and *d-v* mentioned in the rules for the first player. In what follows, then, we define the rules only for the forward-moving player, illustrated by the *white* player in Chess,[3] and refer to the inversion operative in a given game by the symbol $\mathcal{I}$, which we shall assume by default to be the *diagonal* inversion.

Thus, if the movement of a white piece of a certain type involves a direction vector D, then the corresponding movement of the black piece of that type will involve instead the vector $\mathcal{I}(D)$. Similarly, if white's goal is to have his knight arrive at a given square SQ, then black's goal will be to have his knight arrive at $\mathcal{I}(SQ)$. The same holds for promotion ranks, as will be discussed in the next section.

An interesting effect of this global symmetry is that it allows a generator to produce rules from the perspective of only the white (forward moving) player, and the global symmetry automatically implies that white pieces travel forward and to the right, and black pieces travel backward and to the left (from white's perspective), so that by default, opposing forces tend to move toward each other. If a piece movement has both a forward and a side symmetry, however, then the piece will travel along the same direction vectors for both players, because the inversion of a direction vector is precisely the same as the result of applying a forward and then a side symmetry to this vector.[4]

7.2.3 Board

The dimensions of a board are declared by the statement: SIZE X_{max}BY Y_{max}.

A board has one of two *types*: *planar* and *vertical-cylinder*. The planar board is the standard one used in almost all board games. The vertical-cylinder board is like the planar, but the left and right sides of the board are connected to each other so that pieces can *wrap-around* the side of the board.[5] Formally, for vertical-cylinders,

$$d\text{-}v\langle dx, dy\rangle : \quad (x, y) \quad \mapsto ((x + dx - 1) \bmod (X_{max}) + 1, y)$$

For example, in Figure 7.1, the ♘b4 can capture the ♜h5.

A board also has a privileged rank, called the *promotion rank*, such that any piece which, as a result of movement, arrives at or past this rank *at the end of a turn* must then exercise its *promotion power*, if it has one (see Section 7.2.4.4). If this rank had value 6 on a board consisting of 8 ranks, then white pieces would promote on reaching any rank numbered 6 or greater, while by inversion, black pieces would promote on

[3]Epstein ([Epstein, 1989b]) uses the terms *player* and *opponent* to refer to the two players in a game. We shall here use these terms indexically, such that *player* refers to whichever player is to move.

[4]It is thus possible to characterise the degree of symmetry in a particular game by the extent to which movements are invariant under inversion. For example, Chess is almost totally symmetric (every piece except pawns has all three symmetries), while Shogi is much less so.

[5]The rules for vertical-cylinder boards are the same as for normal boards except for modular addition. It is thus legal for a piece to wrap around a vertical-cylinder board back to its original square, effectively passing the move to the next player.

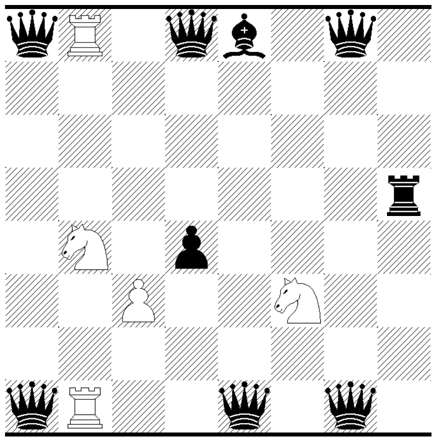


Figure 7.1: A *vertical-cylinder* board and capture movements

reaching rank 3 or less ($Y_{max}+1-Y$). The set of squares at which a player can promote pieces is that player's *promotion territory*.

7.2.4 Pieces

A *piece* is defined by a power of *moving*, *capturing*, and *promoting*, and by an optional set of *constraints* on the use of these powers.

7.2.4.1 Movements

A *basic movement* consists of a *movement type*, which may have associated *movement restrictions*, a *direction vector*, D, and a *symmetry set*, SS. A piece with a given movement can move to any square reachable from its current square, according to its movement type, along D or any *d-v* in the symmetric closure $SC(SS, D)$.

Movement Types The simplest type of movement, called a *leap*, is that of taking a piece from a square S *directly* to the next square along a particular direction vector D, without regard for intervening squares. A piece which moves in this way is called a *leaper*. Thus, a movement which takes a piece only one square forward (for white) would be a $\langle 0,1\rangle$-*leap*, which is the basic movement of pawns in Chess. Similarly, if a Chess knight were restricted to moving one square to the right, and two squares forward, this would be a $\langle 1,2\rangle$-*leap*.

The next type of movement, called a *ride*, allows a piece to continue for some number of leaps along the same direction vector, as long as the squares on intermediate leaps are empty. So a pawn which is allowed to continue indefinitely forward through a line of empty squares (a *pawn rider*) would be a $\langle 0,1\rangle$-*rider*. This is the basic movement of a *lance* in Shogi. This piece can be converted to a Chess *rook* by adding *rotation* and one of *side* and *forward* symmetries.[6]

The final type of movement, called a *hop*, is that in which we relax the constraint on a rider that intervening (leap) squares must be empty, and insist instead that some of these squares must be occupied by pieces.This type of movement is exemplified by the capturing power of a *man* in Draughts or a *cannon* in Chinese Chess.

Movement Restrictions Since a leap is a direct movement from an initial square to a final square, no other squares are considered. However, rides and hops pass through a set of intermediate squares, and additional restrictions may apply to those squares, as part of the rules for a particular piece's movement. For example, the *cannon* in Chinese Chess hops over any one piece *owned by either player*, with any number of empty squares before and after it, and captures the first enemy piece it lands on thereafter. However it is possible to restrict this piece further, by allowing hops over certain pieces only (e.g., black knights), constraining the number of empty squares before or after the hopped-over piece (called the *cannon-support*) to be within some interval (e.g., less than 3, at least 2), and requiring a piece to hop over a specified number of pieces matching a certain description (e.g., 2 pawns of either player).

To illustrate these restrictions, a constrained hopping movement might be defined as follows:

```
MOVEMENT
  HOP BEFORE [X >= 0]
      OVER   [X = 2]
      AFTER  [X <= 2]
  HOP_OVER [opponent any_piece]
  ⟨1,2⟩ SYMMETRY {side}
END MOVEMENT
```

A piece with this movement would move in one of the directions $\langle 1, 2\rangle$ or $\langle -1, 2\rangle$ (by side symmetry). In a given direction, the piece would first leap zero or more times, so long as each leap lands on an empty square. Then the piece would make two more leaps along the same direction, with the condition that each square be occupied by the opponent's pieces. Then the piece would make 0, 1 or 2 further leaps (still along the same direction), through empty squares. Finally the piece would make one last leap along the same direction, landing on its final square. If any of these conditions fail, the move is not legal. For example, in Figure 7.2, if ♘d1 were a piece with a capturing power whose movement was as a hopper so defined, it could move along direction $\langle 1, 2\rangle$ to leap through 0 empty squares, then hop over the two enemy pieces ♞e3 and ♜f5, then leap through 0 empty squares, and then make a final leap to land

[6] Note that a $\langle 0, 2\rangle$-*rider* could move from $(4, 1)$ to $(4, 5)$, as long as $(4, 3)$ was empty, without regard for squares $(4, 2)$ and $(4, 4)$, as each leap along the way moves directly to the second square forward.

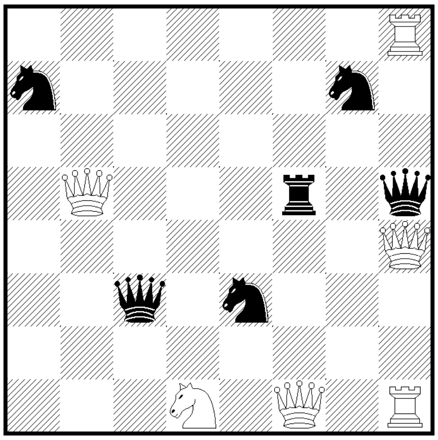


Figure 7.2: Example piece movements.

on ♞g7. However, ♘d1 could not make a similar movement along direction $\langle -1, 2\rangle$ to land on ♞a7, as it cannot hop over the friendly piece ♕b5.

In a similar manner, a ride can be restricted to *at least* L leaps and/or *at most* M leaps. A ride can also be restricted to the maximum number of available leaps (by the presence of the keyword LONGEST in the riding movement definition). For example, in Figure 7.2, if ♖h8 were a rook constrained to the longest ride in each direction for purposes of moving, it could only move to a8 or h6. This constraint thus limits a piece to one move in each legal direction. This restriction to *longest* ride is not applied on vertical-cylinder boards, as it is not well defined for this case.

Disjunctive Movements In addition to the basic movements, we also allow *disjunctive movements*, which are the union of several basic movements. Thus, if we have movements corresponding to a bishop and rook in Chess, then we can define the movement of a queen as the disjunction of these two simpler movements. This is also the method of definition of certain promoted pieces in Shogi.

7.2.4.2 Capturing

Capturing Movements The movements discussed above are used in defining both the *moving* and *capturing* powers of pieces. While a normal movement is used simply to move a piece from one square to another, with no effects on other pieces, a capturing movement always results in some change to the status of other pieces. It is possible (and even common) that pieces in chess-like games *move* in one way, and *capture* in another. Examples of this are pawns in Chess, cannons in Chinese Chess, and all pieces in Draughts.[7]

[7]As in the case of movements, a piece can also have multiple (disjunctive) capture powers.

How To Capture In addition to special capturing movements, different pieces have different methods of capture. The most common capturing method, called *clobbering*, is when a piece ends its movement on a square occupied by another piece, and thus captures it. A second method, applicable only to pieces which *hop* as part of their capture movement, allows certain hopped-over pieces (see below) to be captured. The final capturing method is *retrieval*, in which a piece moves directly away from another piece, and thus captures it. A particular capture definition may allow different types of capture at once, so a piece might hop over one piece and land on another, capturing both.[8] Examples of these capturing methods are presented shortly.

What To Capture As in the case of restrictions on movements, there can be restrictions on what a piece can capture (e.g., any piece, opponent rook or queen), using a particular capturing power. This allows some pieces to be capable of capturing *anything but* a particular piece, for example. To be a legal use of a particular movement for a capturing power, at least one piece must actually wind up being captured.[9]

Effects of Capture Now, given that a piece can move in a certain way to capture a piece (or set of pieces), and that this piece is of the kind that it can legally capture, there are a number of possible *capture effects*, all of which remove the captured piece from its present square. The possible effects are:

- *Remove* a piece from the game altogether.
- *Player Possesses* the piece, converts it to his own side (if necessary) and can place it on any empty square, instead of making a normal piece movement, on one of his turns later in the game (i.e., starting with his next turn). This is the capture effect used in Shogi.[10]
- *Opponent Possesses* the piece, converts it to his own side (if necessary), and can place it on any empty square later in the game.

Here *player* and *opponent* are relative to whichever player has performed the capture. So if *white* captures a black piece, the *opponent possesses* effect means that *black* is then free to place this piece (still black) on any empty square later in the game, while the *player possesses* effect means that *white* would be able to place a white piece of that type later in the game. Examples of each type of capture effect, with the corresponding notation, are presented in Section A.2.

[8] A fourth common method of capture is *coordination*, in which some relationship between two or more pieces determines an additional set of pieces to be captured. Examples are bracketing in *Hasami Shogi* and *Othello*. This would be an interesting extension to the class presented here.

[9] Note that a piece can be restricted to capturing only friendly pieces.

[10] Although Shogi restricts placement squares for some pieces (pawns cannot be placed on files where the player has a pawn already), the class defined here makes no such restrictions.

Examples of Capturing We can illustrate the capture methods and restrictions using Figure 7.1. First, if pieces captured by clobbering, as in normal Chess, then ♖b1 could capture ♛a1 or ♛e1 by landing on them. Second, if pieces captured by hopping, then ♙c3 could capture ♟d4 by hopping over it to the empty square e5, and ♖b1 could capture ♛e1 by hopping over it to f1. Third, if pieces moved as in Chess but captured enemy pieces by *retrieval*, then ♖b1 could capture ♛a1 by moving away from it to c1 or d1, and ♘f3 could capture ♛g1 by moving directly away from it to the square e5.

Finally, suppose ♖b8 is a piece with a *capture movement* of hopping on straight lines over any number and type of pieces, a *capture restriction* that it can capture only enemy ♛'s, and all three *capture methods* (i.e., clobbering, retrieval, and hopping capture). Then in one move, it could move directly away from ♛a8, hopping over ♛d8 and ♝e8, to land on ♛g8. All three enemy queens (♛a8, ♛d8, and ♛g8) would be captured and removed from the board, but ♝e8 would not be captured as this piece can only capture enemy ♛s.[11]

7.2.4.3 Compulsory and Continued Captures

The above sections describe the conditions and effects of capturing moves. In addition, there are two additional types of rules affecting the use of capture movements. The first type of rule *requires* a capture move to be made in preference to an ordinary piece movement. This is indicated by the presence of a `must_capture` constraint, which can appear as both a *global* and a *local* constraint (attached to the game definition or to the piece definition, respectively). As a *global* constraint, this indicates that if a player is to move *any* piece (as opposed to *placing* a piece from his hand), and *some piece* has a capture move available, he must play it. As a *local* constraint, this indicates that if a player is to move a *particular* piece, and *this piece* has a capture move available, he must play it. If the global version is present, any local versions are irrelevant. In both cases, when multiple such captures are available, the player is free to choose any one of them.

The second type of rule *allows* a player to make multiple capture movements within a single logical move, and is indicated as a `continue_captures` constraint. This occurs only in a local version, which allows multiple capture movements with the *same* piece. Unlike the game 10x10 Draughts, captured pieces are removed immediately, not at the end of a turn. Thus, a continued capture sequence is logically equivalent to one player making a sequence of capture movements with a particular piece, while the other player passes.

Finally, these two rules interact as follows: if at any point, both the `must-capture` and `continue-captures` rules are in effect, then the player *must* continue capturing if

[11]Note that if there had been a ♝ at g8 instead of a ♛, this move would not have been legal, as a piece can never land on an occupied square unless it does so using a clobbering capture power which is restricted to pieces of a type consistent with the occupant.

it is legal to do so. As the continue captures is only a local rule, only the piece which just captured is constrained to continue capturing.

The game of Checkers (Figure 7.3) illustrates the use of these rules. In this game, the `must_capture` rule is *global*, meaning that a player must make a capturing move if any of his pieces can capture, and the `continue_captures` rule is local to each piece, meaning that a player is allowed to continue capturing with a piece which has just made a capture movement. The interaction of these two rules means that a player must capture if he can, and once he has done so, he must continue capturing with the same piece until it cannot make any more captures.

7.2.4.4 Promotion

In addition to the normal moving and capturing powers attached to a piece, there is a special power, called *promotion*, which allows the piece to be changed while remaining on its final square. The rule applies when a player has *moved* a piece (possibly several times if this piece made a sequence of captures), which finishes its movements on a square which is in *promotion territory* for the player who moved it (see Section 7.2.3).[12]

In this case, one of the players (as specified in the definition of the piece) gets to replace the promoting piece with any piece of his choice matching a certain *description*. Thus, while in Draughts and Shogi pieces promote to a specific piece of the same colour, pawns in Chess promote to any of a set of pieces of the same colour, as chosen by *player*. Under the generalisation here, this choice could also be made by *opponent*. If so, the opponent performs the promotion at the start of his next turn, before proceeding to make his ordinary placement or transfer move.

7.2.5 Initial Setup

Given a board and a set of pieces, it is necessary to determine a method for setting up an initial configuration of pieces. While some chess-like games begin with an arbitrary, fixed initial state, others have the players alternate assigning either their own piece, or their opponent's piece, to any of a set of squares.[13] A final possibility is that each contest of a particular game could begin with a randomised assignment of a known set of pieces to a known set of squares. Since these games are symmetric (subject to a specified inversion), both the fixed and random configurations are guaranteed to be symmetric. When players place their own pieces, however, there is no constraint that

[12] Note that moving from one square in promotion territory to another still qualifies a piece for promotion, and also that promotion applies only to a piece which actually used a moving or capturing power, as opposed to one which was placed on the board by one of the players.

[13] This corresponds to the clause for `assignment_decision` in the grammar in Appendix A.1. Note that `piece_names` may contain duplicates, as a player may have multiple pieces of the same type (e.g., players have two knights each in Chess).

such placement be symmetric.[14] Finally, it should be noted that not all piece-types are necessarily present at the start of the game, as some can only be obtained through promotion (as in Checkers and Shogi).

7.2.6 Goals

So far we have described the method of determining the initial state, and the set of operators, which characterise this class of games. The final component necessary to describe any problem is the *goal*. As these games are symmetric, the goals, like the initial setup and piece movements, are defined from the perspective of the forward player. Thus, a goal definition simply defines those positions in which a player has achieved a goal, and we define a position as a *win* for *player* if only *player* has achieved a goal, a *draw* if *both* players have achieved a goal, and a *loss* if only *opponent* has achieved a goal. Goals are evaluated, from the perspectives of *both players*, at the *start* of each turn, when control is transferred from one player to another.[15] Thus either player might win at the start of each turn, if a goal is true from his perspective.

This class of games has three types of goals. First, a player achieves a *stalemate goal* in a position in which a specified player (player or opponent) cannot legally make any complete moves at the start of his own turn.[16] Thus, *white* is stale-mated if he begins a turn with no legal moves, and a player is not stale-mated if it is the other player's turn to move. Every game in this class must have a defined stalemate goal, as the rules must cover positions in which a player cannot move. However each game decides whether such an outcome is a win, draw, or loss for the stale-mated player.

Second, a player achieves an *eradicate goal* if, at the start of any turn after the initial assignment stage, there are no pieces on the board which match a certain description. Examples are goals to eliminate the opponent king (Chess, Chinese Chess, Shogi), to eliminate all the opponent's pieces (Checkers), or to eliminate all your own pieces (Giveaway Chess). Note that a description might be complex, allowing goals to eliminate the opponent's knights and pawns. Note also that the description might be of the form: `[any_player king]`. Since *any-player* is symmetric for both players, this implies that both players achieve a goal if there are no more kings on the board. In other words, this outcome would be a draw.

Third, a player achieves an *arrival goal* if, at the start of a turn, a piece matching a certain description occupies a certain square on the board. This allows goals such as player getting his own knight to the square $(4, 5)$, or player getting opponent's queen

[14] White places the first piece, and players alternate thereafter. During this phase there are no captures or promotions. Also, the initial squares upon which pieces are placed typically comprise the first R ranks for each player, so that the pieces always wind up assembled facing each other across the board.

[15] Note that under the *opponent-promotes* promotion method, a player begins a turn by promoting his opponent's piece (see Section 7.2.4.4), so goals are evaluated before he does this.

[16] Note again that promoting an opponent's piece does not constitute a complete move. In order to be legal the player must also be able to move or place a piece on the board.

to the square $(2,2)$.[17]

7.2.6.1 Disjunctive Goals

An additional source of complexity in the rules of games in this class is that players can have disjunctive, or multiple, goals, in which a player achieves a goal if any of a number of conditions arise. For example, white may win the game if *either* someone eradicates black's knights, *or* white loses all his pieces. Such complex goals are especially interesting when the separate goals interfere with each other.[18]

7.2.6.2 General Termination

The goals of a game define the primary ways in which a game can end. However, it is possible that a game reaches a state in which neither player can (or knows how to) win. To stop such games from continuing forever, two additional rules are assumed for all games within this class. The first is an N-move rule, which says that the game automatically ends in a draw after some number of moves have been played. Since it is difficult to determine just how many moves any game in this class may require, the choice of N is rather arbitrary. For now we shall leave N at 200 moves (i.e., after black plays his 100th move, if neither player has won, the game is a draw).

A second rule is included to disallow endless cycles. Although this is not strictly necessary (since games terminate after N moves anyway), we adopt a rule similar to the *triple repetition rule* in Chess, which says that a game is a draw if the same position has been reached a third time with the same player to move. By *position*, we mean the contents of the board and *hands* of the players (i.e., they possess the same set of pieces).[19]

7.2.7 Coverage of Existing Games

As an illustration of how chess-like games are defined in this class, Figure 7.3 presents a grammatical representation of the complete rules for American Checkers as a *symmetric chess-like game*.

The definition of the class presented here has made continuous reference to existing games which this class attempts to generalise. Chapter 9.2 discusses the extent to which this class does in fact cover many existing games, and also provides a proof that

[17] In the absence of certain compulsions, like *must capture* rules, this effectively means that a rational player will never move his piece to such a square, thus effectively adding a constraint instead of a goal to the game. However, it is certainly *legal* for a program to play such a move, thus losing the game instantly.

[18] It is difficult to imagine a naive evaluation function which could automatically handle these disjunctive goals.

[19] As no rules in games in this class make use of history, there is no need to discuss history in determining repetition of position, as is done in Chess.

```
GAME            american_checkers
GOALS           stalemate opponent
BOARD_SIZE      8 BY 8
BOARD_TYPE      planar
PROMOTE_RANK    8
SETUP           man AT {(1,1) (3,1) (5,1) (7,1) (2,2) (4,2)
                        (6,2) (8,2) (1,3) (3,3) (5,3) (7,3)}
CONSTRAINTS     must_capture

DEFINE man
  MOVING
    MOVEMENT
      LEAP
      ⟨1,1⟩ SYMMETRY {side}
    END MOVEMENT
  END MOVING
  CAPTURING
    CAPTURE
      BY {hop}
      TYPE [{opponent} any_piece]
      EFFECT remove
      MOVEMENT
        HOP BEFORE [X = 0]
            OVER   [X = 1]
            AFTER  [X = 0]
        HOP_OVER [{opponent} any_piece]
        ⟨1,1⟩ SYMMETRY {side}
      END MOVEMENT
    END CAPTURE
  END CAPTURING
  PROMOTING
    PROMOTE_TO king
  END PROMOTING
  CONSTRAINTS continue_captures
END DEFINE

DEFINE king
  MOVING
    MOVEMENT
      LEAP
      ⟨1,1⟩ SYMMETRY {forward side}
    END MOVEMENT
  END MOVING
  CAPTURING
    CAPTURE
      BY {hop}
      TYPE [{opponent} any_piece]
      EFFECT remove
      MOVEMENT
        HOP BEFORE [X = 0]
            OVER   [X = 1]
            AFTER  [X = 0]
        HOP_OVER [{opponent} any_piece]
        ⟨1,1⟩ SYMMETRY {forward side}
      END MOVEMENT
    END CAPTURE
  END CAPTURING
  CONSTRAINTS continue_captures
END DEFINE

END GAME.
```

Figure 7.3: Definition of *American Checkers* as a symmetric chess-like game.

the class is, in fact, extremely general, in that any finite two-player game of perfect information can be represented as a *symmetric chess-like game*.

7.3 Summary

This chapter has defined a class of games to serve as a concrete basis for Metagame-playing. The class generalises many features of the chess-like games, instances of which have received much of the attention in CGP, and represents games in a manner which preserves the compact structure which makes them appear similar. In order to increase the chances that arbitrary generated games within this class would be *fair* to both players, all games in this class are constrained to be symmetric. This is achieved by requiring the rules of any instance game to be defined from the perspective of one player only. A global inversion function then produces the symmetric set of rules from the perspective of the other player.

The discussion in this chapter has described the motivation and definition of the class at varying levels of detail. While the detail in some cases may seem excessive, it serves three main purposes. First, it highlights the considerations and details which are involved in the process of constructing a class of games for Metagame-playing. In this respect serves as an example for future research on Metagame problems centred around different classes of games. Second, it provides the detail necessary to enable this specific problem, Metagame in symmetric chess-like games, to be used as a testbed by other researchers. To aid this effort further, a formal semantics of this class of games is provided in Appendix B. Third, it provides background for later chapters, which focus on the class of games presented here.

With the class of games fully defined, Chapter 8 will now proceed to develop a method for generating new instances of this class of games without human assistance.

Chapter 8

Game Generation

> Variable rules ... You may invent your own men and assign them arbitrary powers. You may design your own boards. And you can have rule games.
>
> – Henry Kuttner, *Chessboard Planet* ([Kuttner, 1983])

8.1 Introduction

This chapter develops a general method for automatically generating problem definitions, and applies this to the definition of symmetric chess-like games to produce a game generator for this class. Section 8.2 presents the theory and implementation behind the game generator. By means of example generated games, it is shown in Section 8.3 that the generator produces a variety of interesting games. Section 8.4 summarises the discussion.

8.2 Game Generator

The goal of game generation, as shown in Figure 8.1, is to produce a wide variety of games, all of which fall in the same class of games as described by a *grammar*. We also want to be able to modify distribution of games generated by changing *parameters*, often in the form of *probabilities*. Finally, the combination of grammar and probabilities may not be enough to constrain the class according to our needs, in which case the generator should be able to handle a set of *constraints*.

8.2.1 Stochastic Context-Free Generation

Generation can in general be done either at a syntactic level or a semantic level. Semantic-level generation corresponds to producing English sentences from intended

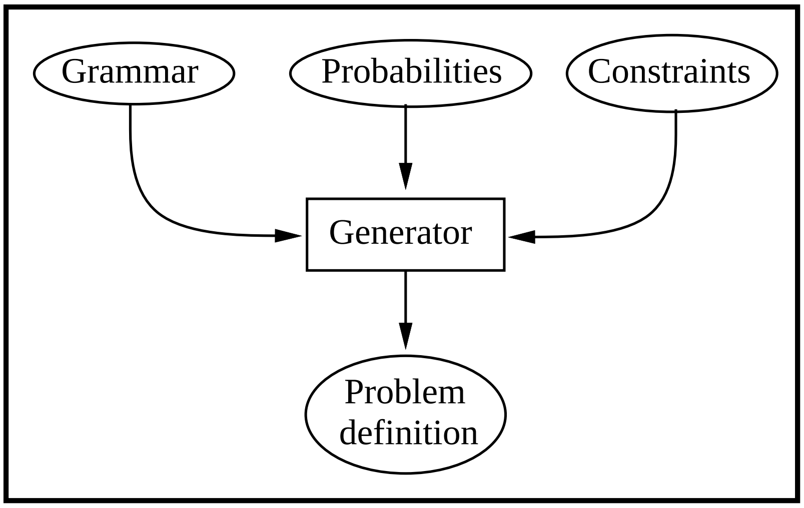


Figure 8.1: Components of a Problem Generator.

meanings, in conjunction with a grammar which relates sentences to meanings, while syntactic-level generation corresponds to producing English sentences either directly from a grammar, or by filtering random strings of words through a grammar.

The approach which is most direct, in that it operates directly on the specification to which it is to conform, but which requires the least understanding of the use to which it will be made, is to generate directly from the grammar. This also has the advantage of clarity, in that it is obvious where the grammar comes from (i.e. it is provided), but it is not at all obvious where intended game rules might come from. Thus, this is the approach I have taken here.

The method I have used to generate from the grammar is to attach a probability to each nondeterministic choice-point, which corresponds to the probability that each of the possible choices will be taken. This corresponds to generating from a stochastic context free grammar ($SCFG$).

For example, the clause in the grammar defining *movement types* is as follows:

```
movement_type --> leaper | rider | hopper
```

This states that a movement type can be either a leaper, a rider, or a hopper. The probability distribution corresponding to these possibilities is defined as follows:

```
parameter(movement_type,
    distribution([leaper=0.4, rider=0.4, hopper=0.2])).
```

This states the probability that the generator will choose each of these options, when they are available.

8.2.2 Constrained *SCFG*

A grammar also has associated a set of pre-terminals (pre-terms), in our case names of pieces. If the generator selected a new symbol each time a piece was needed, we would be unlikely to have an interesting, or even meaningful game produced. This is because there are additional extra-grammatical *constraints*, which mediate between the syntax and the semantics of the game. This is just the traditional problem that syntax is always looser than semantics (not every grammatical English sentence has a meaning, nor does every parse-able Pascal program).

To meet these constraints, the *SCFG* generator is augmented in two ways. First, it generates in advance the set of pre-terminals (piece-names) which will be used as the supply of pre-terms of this type throughout generation. Whenever a preterminal is required by the grammar, it is chosen randomly (or subject to some constraint) from this set.

8.2.2.1 Interactivity

This method of *sharing* a set of pre-terminals among all the productions in a grammar creates an automatic method for building complex patterns of interaction between the different generated structures in a game. For example, the movement generator might decide to generate a hopping movement. These movements have a restriction component, which describes the set of pieces over which a particular hopper can hop. The restriction component then chooses a subset from the set of piece names. Although the specific details of the pieces in this subset are inaccessible to the generator, the inclusion of the name of a particular piece in this restriction component causes the two pieces to be closely related at a semantic level (in particular, as the hopper and the piece it can hop over).

8.2.2.2 Constraints

Second, the generator allows the user to attach *constraints* on some of the choice points. The constraints reject possible daughters, and generation is repeated until an acceptable choice is generated. One constraint used in the current generator rejects a generated piece movement definition if the direction vectors on the movement imply that it could never be applicable on the current board. Another constraint only accepts types of goals which do not make the game drawn or easily won. This constraint rejects the following types of goals:

- Arrival goals predicated on pieces which are in the initial setup. Thus the only generated arrival goals are those which require promotions in order to be achieved.

- Eradicate goals predicated on pieces which are *not* in the initial setup. Such goals are always achieved by both players at the start of the game, and would always result in draws if not rejected.

- Goals subsumed by previously generated goals. These add no information to the game and would cause the number of distinct goals to be reduced.
- Goals to achieve the opposite of previously generated goals. These goals would be uninteresting because their achievement would always result in a draw.[1]

The constraints used by the current generator may exclude some games which are nevertheless interesting. However, their addition increased the likelihood that a generated game was not automatically drawn. Also, it would be possible in principle to place constraints only on the game as a whole. However, it is computationally much more efficient to attach constraints directly to the points in the grammar at which they can first be checked.

8.2.3 Generator Parameters

The probabilities attached to the choice points, and the parameters which are used by the subtree constraint testers, are thus the parameters of the generator. Interestingly, they can be used (directly) to affect the distributions of syntactic structures generated (like larger or smaller piece definitions), and also (indirectly) to influence semantic aspects of generated games (like more or less constrained goals).

Rule Complexity One property of interest is the complexity (length) of the rules in a game. This can be controlled by means of a small set of parameters in the generator which are consulted in order to choose between making a game component more complex, or leaving it as it is.[2] This allows components to be generated with arbitrarily long descriptions, though longer descriptions are exponentially less probable than shorter ones. By varying these parameters, we can thus change the overall expected complexity of the components to which they are associated. Examples of such parameters are those attached to the `movement_def` and `capture_def` clauses, which control the probability of adding another disjunct to these definitions.

Decision Complexity Another statistical property of a game which can be determined in this way is the degree to which a game allows players to make choices, instead of assigning arbitrary values to these choices as part of the game definition. For example, pieces in Shogi promote to exactly one type of new piece each, whereas pawns in Chess promote to any one of a set of choices, to be decided by the player at the time of promotion. This property can easily be varied to produce different types of

[1] Recall that a draw results when both players have achieved a goal in a position (Section 7.2.6).

[2] More precisely, several rules choose between two possibilities, one of which is tail-recursive. Assigning a probability p to choosing the non-recursive case means that the recursion will continue with probability $1 - p$.

games, by modifying the distribution attached to the rule which decides, for example, whether a promotion or initial-setup decision should be arbitrary or not.

Search Complexity A related statistical property of generated games is that of search complexity, essentially the size of the search space in a particular game. This can be adjusted, without affecting the rule or decision complexities discussed above, by altering the probability distribution on *board size*, as larger board sizes will tend to allow more possible movements for each piece, and thus more possible moves in each position in the game. Of course, the parameters mentioned above (such as capture complexity) also affect the size of the search space, such as increasing increasing the probability that a piece has different types of movement available.

Certain parameters, in fact, have dramatic consequences on the search space. For example, the presence of an `opponent_promotes` rule, which allows the opponent to make a promotion decision before starting his move (see section 7.2.4.4), multiplies the number of possible moves available to him in such a position: if a player had n ordinary move movements in a position, but he first has to promote an opponent's piece to one of p other pieces, then the total branching factor for that position is pn. If he had also to promote whichever piece he moved to one of M pieces, the branching factor would rise to Mpn.

At the opposite extreme of affecting search complexity, the presence of *must_capture* constraints has the effect of dramatically *reducing* the size of the search space.

Locality A final property of interest is *locality*, which determines the fraction of a board which can be traversed by a piece in one leap, without regard for the other squares on the board. The less locality, the more pieces on one side of the board can directly affect the status of pieces on another side of the board. It is possible that this affects the degree to which a program could reason about separate aspects of the board individually. Locality is affected by the modules constraining the restrictions on riders and hoppers, the module which generates direction vectors, and the `board_type` parameter, as a cylindrical board allows pieces to move from one side to the other with a direct leap.

Game Complexity Metrics The preceding paragraphs discussed the manner in which qualitative properties of a game are influenced by specific combinations of parameters. Our understanding of the generator could be improved by a more systematic study of the relation between the low-level parameters and high-level properties. This effort has not yet been undertaken, in part because of the difficulty of quantifying the complexity of large games. For example, the length of the rules increase the potential for obscure interactions between rules to influence strategies, but it is also possible for a game with a long description nevertheless to be strategically simple. Even the obvious metric on the size of the search-space of the game (e.g. the game tree) cannot necessarily be measured or even estimated reliably for large games. The development of a useful set of game complexity metrics is an area for future work.

8.2.4 Consistency Checking

Deciding whether a generated game can possibly be won generally requires a level of analysis beyond that implemented in the generator (in fact, the general problem is NP-Hard, as proved in Section 9.3.1). However, the current generator does perform a simple analysis to avoid some of the common problems which would otherwise produce a high proportion of trivial games (see Section 8.2.2.2). Ultimately, it is up to the programs to decide whether or not a game is trivial or even winnable, which is indeed an aspect of game analysis traditionally left to humans.

8.2.5 Implementation Details

Including extensive comments, the code which implements the generator for symmetric chess-like games amounts to 2500 lines of Prolog. Under Sicstus Prolog 2.1 patch 8 on a SUN4, generating a random game takes under one second.

8.3 A Worked Example

A recurrent point in this thesis has been that existing methods of computer game-playing have left much of the interesting game analysis to the human researcher, and that existing methods like minimax do not offer much advice on developing programs to play a new game. Thus, I developed a class of new games, and a generator for it, to highlight these issues and provide a test bed for addressing them. This section provides an example game actually produced by the generator and discusses some strategies which humans have discovered for this game.[3] Section 8.3.1 explains the rules of the game. Section 8.3.2 provides my own initial analysis of this game. This analysis was performed when the generator was first developed, before any programs existed that could play Metagame. After I published this game and my analysis of it [Pell, 1992], some other researchers extended my initial analysis to produce more sophisticated strategies. Section 8.3.3 discusses one such strategy discovered by another researcher. Section 8.3.4 draws two conclusions from this example. First, although generated games often look silly at first, the complexity of the rules and symmetric structure offer chances for interesting strategic analysis. Second, the kind of game-analysis used to analyse these games does not appear easily amenable to a naive and general-purpose evaluation function.

[3] Appendix D.1 contains more examples of generated games, which were used in a Metagame tournament discussed in Chapter 16.

8.3.1 Turncoat-Chess

I generated a random game using the generator with parameters set to prefer small boards and moderate complexity of movements, captures, and goals.

8.3.1.1 Generated Rules

The resulting game, as actually output from the generator, is presented in Figure 8.2 and Figure 8.3. I have replaced some internal symbols with more mnemonic names, and named this game *turncoat-chess*.

As the rules of this game are fairly complex, I shall attempt to summarise them in a more comprehensible form. For a full explanation of the meaning of particular rules, such as movement powers, see Section 7.2.

8.3.1.2 Summary of Rules

Turncoat-Chess is played on a 5-by-5 planar board. There are three types of pieces: *slug*, *termite*, and *firefly*. The initial setup is fixed, with pieces placed on the first rank of each player, symmetrically across the board. Each player starts with one slug, two termites, and two fireflies. Figure 8.4 shows a representation of the initial position for *turncoat-chess*. Fireflies are represented by the symbols ♕and ♛, termites by ♘and ♞, and slugs by ♖and ♜, for white and black pieces, respectively.

The pieces move and capture in different ways, discussed below, but all pieces can capture *any* type or colour of piece, by landing on it, and the captured piece is then permanently removed from the game. All pieces *promote* upon reaching the last rank, at which point the player who owns the piece can replace it with any type of piece, although for two of the pieces he must transfer ownership of it to the enemy after promoting.[4] A player wins if he has no legal moves at the start of his turn.

The descriptions of pieces are broken into powers of moving, capturing, and promoting.[5]

Slug The first type of piece is a *slug*. The slug moves by continually leaping (i.e. riding) to every second square along a particular rank or file, with the constraint that for each direction, it must ride as far as it can.[6] A slug's power to *capture* is very restricted: if there are two consecutive *fireflies* (of any colour) along a file, it can hop over any number of empty squares, then over the two fireflies, then over any number of empty squares, and finally capture *any* piece it lands on. So in Figure 7.2 (Page 53), if ♖h1 were a slug and ♕h4 and ♛h5 were fireflies, then ♖h1 could capture ♖h8,

[4]Hence the name, *turncoat-chess*.

[5]It should be remembered that a capturing power can only be used if it results in a piece being captured.

[6]A way to think of this is that it *can't stop* riding along a line, until it is blocked.

```
GAME           turncoat_chess
GOALS          stalemate player
BOARD_SIZE     5 BY 5
BOARD_TYPE     planar
PROMOTE_RANK   5
SETUP          termite AT { (1,1) (2,1) }
               slug AT { (3,1) }
               firefly AT { (4,1) (5,1) }
```

```
DEFINE slug
  MOVING
    MOVEMENT
      RIDE LONGEST
      ⟨2,0⟩ SYMMETRY all_symmetry
    END MOVEMENT
  END MOVING
  CAPTURING
    CAPTURE
      BY {clobber}
      TYPE [any_player any_piece]
      EFFECT remove
      MOVEMENT
        HOP BEFORE [X >= 0]
            OVER   [X = 2]
            AFTER  [X >= 0]
        HOP_OVER [any_player {firefly}]
        ⟨0,1⟩ SYMMETRY {forward side}
      END MOVEMENT
    END CAPTURE
  END CAPTURING
  PROMOTING
    DECISION player
    OPTIONS [{player} any_piece]
  END PROMOTING
END DEFINE
```

```
DEFINE termite
  MOVING
    MOVEMENT
      HOP BEFORE [X >= 0]
          OVER   [X = 1]
          AFTER  [X >= 0]
      HOP_OVER [any_player {termite}]
      ⟨0,1⟩ SYMMETRY {side rotation}
    END MOVEMENT
    MOVEMENT
      RIDE LONGEST
      ⟨0,1⟩ SYMMETRY all_symmetry
    END MOVEMENT
  END MOVING
  CAPTURING
    CAPTURE
      BY {clobber}
      TYPE [any_player any_piece]
      EFFECT remove
      MOVEMENT
        LEAP
        ⟨2,3⟩ SYMMETRY {forward side}
      END MOVEMENT
    END CAPTURE
  END CAPTURING
  PROMOTING
    DECISION player
    OPTIONS [{opponent} any_piece]
  END PROMOTING
END DEFINE
```

Figure 8.2: *Turncoat-Chess*, a new game produced by the game generator.

```
DEFINE firefly
  MOVING
    MOVEMENT
      LEAP
      ⟨1,2⟩ SYMMETRY all_symmetry
    END MOVEMENT
    MOVEMENT
      HOP BEFORE [X >= 0]
            OVER    [X = 1]
            AFTER   [X >= 0]
      HOP_OVER [any_player {termite}]
      ⟨2,1⟩ SYMMETRY {side rotation}
    END MOVEMENT
    MOVEMENT
      LEAP
      ⟨2,3⟩ SYMMETRY all_symmetry
    END MOVEMENT
    MOVEMENT
      LEAP
      ⟨0,1⟩ SYMMETRY all_symmetry
    END MOVEMENT
  END MOVING
  CAPTURING
    CAPTURE
      BY {clobber}
      TYPE [any_player any_piece]
      EFFECT remove
      MOVEMENT
        LEAP
        ⟨0,1⟩ SYMMETRY all_symmetry
      END MOVEMENT
      MOVEMENT
        RIDE
        ⟨2,3⟩ SYMMETRY {forward side}
      END MOVEMENT
    END CAPTURE
  END CAPTURING
  PROMOTING
    DECISION player
    OPTIONS [{opponent} any_piece]
  END PROMOTING
END DEFINE
END GAME.
```

Figure 8.3: *Turncoat-Chess* (continued).

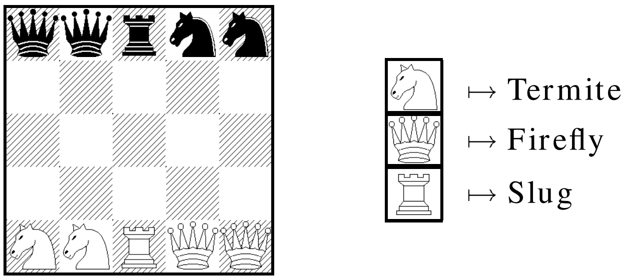


Figure 8.4: Initial board for *turncoat chess*.

which is the first piece beyond the fireflies on the h-file. Finally, a slug can promote to any other piece, and does not change colour on promotion.

Termite The second type of piece is a *termite*, which moves in one of two ways. First, it can *hop* along a line forward, backward, or to either side, but must hop over a single *termite* of any colour, though it can pass over any number of empty squares before and after it. Second, it can move like a chess rook, in which case it makes the *longest ride* in a given direction until it is blocked. A termite captures any piece at relative coordinates $\langle 2, 3\rangle$, forward and backward, left and right (but not $\langle 3, 2\rangle$, which requires *rotational* symmetry). Finally, a termite promotes to any type of piece, though it then changes ownership (so a white termite promotes to any type of black piece).

Firefly The third type of piece is a *firefly*, which has many forms of movement and capture (see Figure 8.3). Its simple forms of movement are leaping as a chess knight, leaping 1 square orthogonally, or leaping to any square at relative coordinates $\langle 2, 3\rangle$ or $\langle 3, 2\rangle$, in any directions. Its more complicated form of movement is as a knight-hopper, in which case it must hop over a single termite. For example, in Figure 7.2, a firefly ♕f1 could hop over a termite ♞e3, and then land on either of the empty squares d5 and c7.

A firefly captures either by leaping to an orthogonally adjacent square, or leaping to a square at relative coordinates $\langle 2, 3\rangle$, $\langle -2, 3\rangle$, $\langle 2, -3\rangle$, or $\langle -2, -3\rangle$.[7] Finally, a firefly promotes the same way as a termite.

8.3.2 A Quick Analysis

As the rules look extremely complex, it can be difficult for a human to remember them, much less play a game using them. However, to illustrate the kind of simple analysis which is typical of humans analysing games, I will give an example of my own analysis of this game.

8.3.2.1 Envisioning a Win

In order to win, a player must begin a turn having no legal moves. Thus either he must have no remaining pieces, or they must have no moves. The first case seems easier to achieve. A player can remove his own pieces either by capturing them, or, in the case of fireflies and termites, by giving them to the opponent via promotion. As a player cannot give away a slug, he must either capture it with one of his own pieces, or first promote it into a termite or firefly, and then promote that piece to give it to the opponent. As the latter takes more moves, capturing a slug to start with seems the simplest option.

[7] According to the piece definition, it rides along these vectors, but on a 5 by 5 board there is enough room for only 1 leap.

8.3.2.2 A Naive Winning Plan

Thus, the simplest plan to win, ignoring opposition, is as follows: first, capture the rest of one's own pieces using one of the fireflies, then promote the final firefly, which will take at least 2 more moves.

8.3.2.3 Two Counter Plans

However, this plan can easily be defeated with any opposition. First, it is not enough for a player to get rid of his last piece, as the opponent might be able to give him a piece back, and it is only stalemate if a player *begins* his turn without any moves. Second, while a player captures all his pieces with a firefly, the opponent can advance his own pieces to promotion range (after capturing his own slug first). Then when the first player has only 1 firefly left, the opponent can promote each piece to give away several slugs. Slugs are hard to promote, and have limited mobility, so the first player should be so busy trying to promote the slugs back to fireflies, that the opponent can capture or give his pieces away by promotion.

8.3.2.4 Initial Conjecture

So, this simple analysis reveals that it is at least possible to win this game, and there are a set of straightforward plans and counter plans which must be traded off. In the end, it is likely that one player will be overloaded with slugs, giving the other player time to win, but the means by which this happens are far from trivial. Thus, while the rules are strange and complex, the game could prove to be interesting, and certainly does present some elements of strategic complexity.

8.3.3 Turncoat Chess Revisited

The preceding section was first drafted shortly after the generator was implemented. Thus, I generated an arbitrary game and analysed it for 20 minutes, and at that point had not yet played against an opponent.

Since that time, many actual Turncoat games have been played, by the author, other researchers, and several programs. This has resulted in the development of many interesting new strategies and insights for this game, which has proven itself to be both interesting and challenging. As an illustration of the potential for strategic analysis afforded by generated games, I will now discuss one of the most interesting strategies for this game, discovered by Victor Allis. Allis' analysis begins with my initial conjecture above: that giving away pieces as quickly as possible might not be the strongest strategy. In giving away pieces, I conjectured, a player would lose control of the game, which could make it easier for the opponent to go on to win (see also the discussion on mobility in Section 14.2.1, page 122). This was determined to hold for

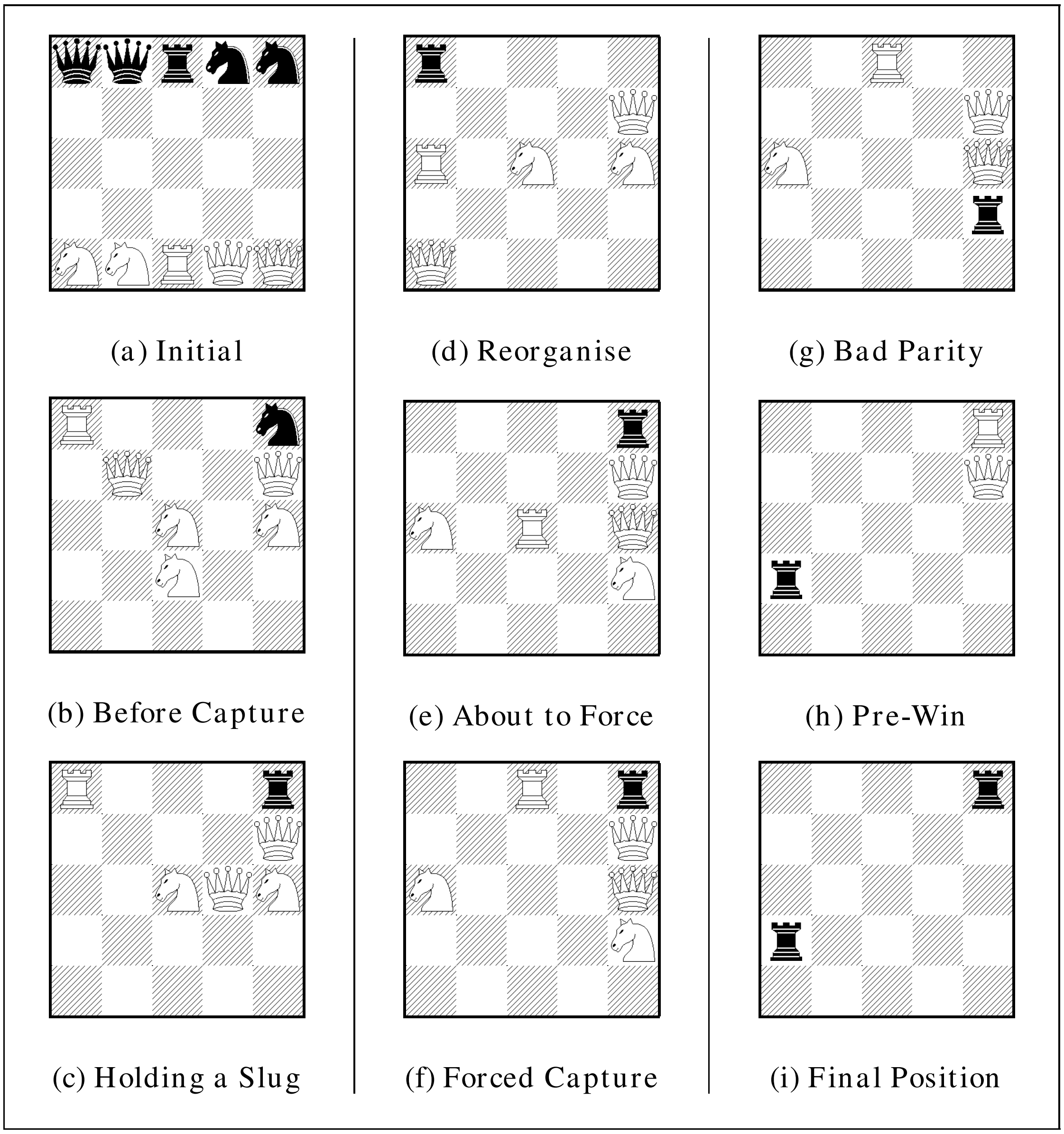


Figure 8.5: An advanced strategy for Turncoat Chess.

Turncoat Chess by experience playing it. As this was the case, it appeared for a while that there was no way to force a win in the game.

However, Allis discovered that under certain conditions it was possible to force the opponent's slug onto a square of the wrong parity, so that it could never thereafter go on to promote. Once this situation has been set up, the player can then proceed to capture all of his own pieces (with a firefly), and finally promote the firefly into a piece of the opponent to win the game.

The stages in this strategy are shown in Figure 8.5. The actual positions are taken from a game in which I used this strategy against RAD, a random-aggressive Metagame-player discussed in Chapter 13. Diagram (a) is the initial position for Turncoat Chess. Diagram (b) shows a position after White has captured most of Black's pieces (remember that a player wins this game by having no moves, not by dominating the opponent). Although Black has only to eliminate his last termite (♞e5), it is confined to the fifth rank. Since termites must ride as far as possible, Black has no choice but to move the ♞e5 to b5 and back. While Black has moved back and forth, White has now maneuvered his own termite to c2, which now captures the ♞e5 and promotes to a black slug.

Diagrams (c) and (d) show White reorganising his pieces, all the while confining Black's slug to a few squares (remember that slugs must also ride as far as possible). Diagram (e) is the key position for this strategy. White has set up a row of pieces on the e-file, ♕e4, ♕e3, and ♘e2. He now plays ♖c5, moving his slug to take away the last moving square for Black's slug ♜e5 (see Diagram (f)). This is also interesting because it demonstrated a function for the slug which was not considered in my initial analysis of the game. I had assumed that slugs were the least valuable piece because they could not promote directly to an opponent's piece, but it turns out here that the slug is valuable because it is the only piece which can move freely on the promotion rank.

Diagram (f) is the resulting position. In this position, Black has no choice but to use the restricted capturing power of a slug (to hop over two fireflies and capture a piece), and is thus forced to capture the ♘e2 (see Diagram (g)). After making this capture, however, the ♜e2 is on a square of the wrong parity for promotion. That is, since a slug always moves $\langle 0, 2\rangle$ (with symmetries), and it is now on the second rank, it can never reach the first rank in order to promote (unless White happened to line its fireflies up again to facilitate this). As Black can no longer interfere by giving away a piece, White now captures all the rest of his own pieces with one of his fireflies. Diagram (h) shows the position after most of these captures have been made. White now plays ♕e4x♖e5, and promotes into a black slug. Diagram (i) shows the pre-final position. Black has no moves which promote a piece into a white piece, so White will start the next move with no legal moves, thus achieving his goal to stalemate himself and winning the game.

8.3.4 Discussion

This section presented a typical game produced by the game generator. The game was not selected because it was in some sense interesting; rather, it was the first game generated at the time the generator was written up. Like most games produced by the generator, the rules are more complex and idiosyncratic than those of conventional board games. Also like most generated games, the rules at first appeared complicated, arbitrary, and uninteresting in general. However, the game nevertheless has a compact representation which shares a degree of structure with chess-like games. The game is also symmetric, so the balance of power is relatively fair. Most importantly, the interaction between the pieces in terms of moving, capturing, and promoting, presents ample opportunities for game-analysis.

After presenting the rules of the game, this section discussed some examples of game-analysis applied to it, both by the author and by other researchers. While the game appeared strange at first, it has proven itself to be interesting and challenging. This may be the first case of a computer generating a game which proved to be of interest to human players and researchers.

Finally, the game-analysis examples illustrate the kind of analysis humans perform when they are presented with a new game. Similar analyses for other games can be found in specialist books on these games. Examples can also be found in many papers on computer game-playing, in which the human begins by analysing the game for significant features which could form the basis of an evaluation function. Thus far, though, this type of analysis has been considered a prerequisite for computer game-playing, and not a subject of research in its own right. Designing programs which can compete well on arbitrary generated games without human assistance may require researchers to transfer some of the responsibility for this analysis onto the programs which actually will play the games.

8.4 Summary

This chapter developed a general method for automatically generating problem definitions for instances of a class of problems. The method is called *Constrained Stochastic Context-Free Generation*. It operates by making statistical choices at each decision point in the grammar defining the class of problems. The probabilities of making different choices are controlled by parameters, and the structures produced at each choice point are rejected if they fail to pass user-imposed constraints. This method of generation is primarily syntactic, in that the only knowledge about the meaning of the structures is that input into the constraints.

This method of problem generation was used to implement a generator for symmetric chess-like games. This chapter discussed the details of this implementation, which involved the specific parameters and constraints used for this class of games. The chapter then illustrated the results of the implemented generator by means of an example game it has produced, which was later named Turncoat Chess. The discussion showed that the generator is capable of producing games which are interesting to humans, despite the apparent complexity and unfamiliarity of the rules. The discussion also presented examples of strategies for playing Turncoat Chess which were discovered by humans. The extent to which current techniques in CGP must be extended to enable programs to perform the type of sophisticated game-analysis demonstrated here is at present unknown.

Chapter 9

Analysis of Class of Games

> The idea ... is that Azad is so complex, so subtle, so flexible and so demanding that it is as precise and comprehensive a model of life as it is possible to construct. Whoever succeeds at the game succeeds in life; the same qualities are required in each to ensure dominance.
>
> – Ian Banks, *The Player of Games* ([Banks, 1988])

9.1 Introduction

This chapter analyses the class of symmetric chess-like games in detail to assess its usefulness as a basis for Metagame-playing. Section 9.2 discusses the properties of the real and theoretical games which this class contains, and provides practical examples of the expressive power by means of representing some commonly-known games in this class. Section 9.3 provides some results on the complexity of reasoning about games in the class. The most important result of this section is that the class contains games for which answering simple questions (such as whether a particular goal could ever be achieved) can be combinatorially difficult. This suggests that strong playing programs will have to analyse games individually, as the class is too general to be fully analysed in advance. Section 9.4 reviews the analysis from this chapter and discusses the extent to which this class fulfills the desiderata on classes for Metagame-playing presented in Section 6.2.1. Section 9.5 summarises the chapter.

9.2 Coverage

Now that we have described the class of games in detail, we can discuss the general coverage of this class.

9.2.1 Empirical

As is discussed in Section 7.1, the class of symmetric chess-like games was deliberately designed to be a generalisation which was restricted enough to preserve the structure of some real games, while general enough to allow complex interactions and a variety of games. This goal was assisted by drawing on research from the field of *Fairy Chess*, as developed by T.R. Dawson ([Dickins, 1971]). This field specialises in developing new variants and generalisations of Chess. Dawson's *Theory of Movements* formed the basis for the movement types (leap, ride, and hop) discussed in Section 7.2.4.1. This allows the class defined here to capture the basic forms of movement encountered not only in existing standard chess-like games, like Chess, Shogi, and Checkers, but also to handle many of the variants developed in Fairy Chess.

Although this allows most of the basic movements to be represented in this class, there are several aspects of common games which seemed too idiosyncratic to generalise. For example, it is difficult to find a natural generalisation of the *en passant* or *castling* rules in Chess, or of the rule in Shogi which prohibits a player from placing a pawn on a file on which he already has a pawn. Thus, these rules cannot easily be represented in the class defined here.

Another point about empirical coverage of this class is that players are allowed to make moves which would lose the game instantly, since piece movements are separate from goal criteria. For example, in Chess it is illegal to leave your king in check, and the game ends if a player can make no legal moves. In the class defined here, it is legal to move into check, but doing so would cause a loss of the game against any opponent capable of finding a one-ply win.[1]

9.2.1.1 Examples of Known Games

By using various representational tricks, it is possible to implement most of the rules of many games, including the following:

- Chess
- Giveaway Chess
- Replacement Chess
- Chinese Chess
- Shogi
- Checkers
- Lose Checkers

[1]Thus the distinction between *checkmate* and *stalemate* in Chess cannot easily be fully represented.

- Tic Tac Toe
- Go-Moku

The encodings of several of these games as symmetric chess-like games are presented in Appendix D.

9.2.1.2 Challenges of Representation

While it seems natural that the various chess variations can be represented as symmetric chess-like games, it may seem odd that Tic Tac Toe can be so represented also. The reason this is strange is that the goal of Tic Tac Toe, to have three of one's pieces in a row, does not correspond to any of the goals in this class. As discussed in Section 7.2.6, a stalemate goal is true when a player has no legal moves at the start of his turn, an arrival goal is true if a player has a piece matching a certain description on a certain square, and an eradicate goal is true if a player has no pieces on the board matching a certain description. The closest of these goals to the goal of Tic Tac Toe is the arrival goal, but as goals in symmetric chess-like games are disjunctive, there is no natural way to represent a conjunctive arrival goal.

The method by which Tic Tac Toe can be represented, then, illustrates that this class of games is actually more flexible than one might have imagined.

The initial board of Tic Tac Toe, when represented as a symmetric chess-like game, is displayed in Figure 9.1, Diagram (a). The central 3 by 3 region is the main playing board, and the top and bottom regions contain two types of pieces, marked ♙and ♖, for *man* and *dummy*, respective. A ♙moves by leaping exactly four squares forward (backward) for player (opponent), which corresponds to placing a piece on the board in the original game. Now, for reasons explained above we cannot easily represent the concept of a player winning *when* he has three in a row. However, we can use the capturing movements to define a situation where a player can make a winning move (arriving a piece to a certain square) exactly in those positions in which he *could achieve* three pieces in a row.

To this end, Figure 9.2 displays the definition of a piece which has been moved onto the board, which is marked in Figure 9.1 as a ♔. The capturing definitions ensure that whenever a player has two pieces in a line with the third point on the line empty (Diagram (b)), there exists a capture movement which allows that player on his next turn to claim a win (Diagram (c)). Thus, this game would seem to be strategically isomorphic (in the sense of [Banerji and Ernst, 1971]) to tic-tac-toe, although it superficially looks very much different. Developing a program which could perform the reasoning necessary to realize that this game was equivalent to the standard representation of Tic Tac Toe, would be an interesting area for future research.[2]

[2]While the *ride* capture movements in this representation apply only to the case where two pieces are in a line, a similar technique can be used for games with goals to have *at least N* pieces in a row. The method makes use of *hopping* movements and *continue-capture* constraints, and is straightforward.

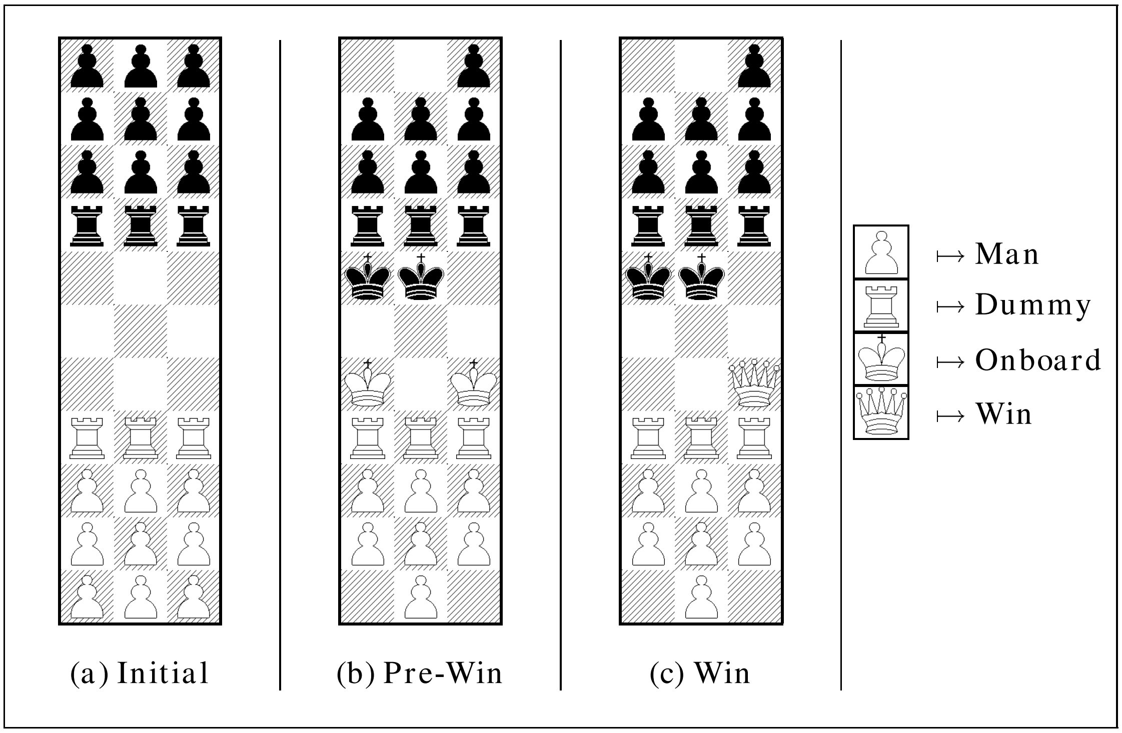


Figure 9.1: Boards for Tic Tac Toe as a symmetric chess-like game. A ♙ hops 4 forward (backward for black) and promotes into a ♔. A ♔ can capture another ♔ to promote into a ♕ (and win the game) exactly when the player could have made 3-in-a-row. The ♖s stop the ♔s from hopping off the main board. The full encoding of this game is in Appendix D.2.1.

```
DEFINE on_board
 capturing
  capture by {clobber}
   type [{player} {on_board}]
   effect remove
     movement
      ride min 2 max 2
       <1,0> symmetry {rotation}
     end movement
     movement
      ride min 2 max 2
       <1,1> symmetry {side}
     end movement
  end capture

  capture by {retrieve}
   type [{player} {on_board}]
   effect remove
     movement
      leap
       <1,0> symmetry all_symmetry
     end movement
     movement
      leap
       <1,1> symmetry all_symmetry
     end movement
  end capture
 end capturing
 promoting promote_to win
 end promoting
end define
```

Figure 9.2: Piece definition for Tic-Tac-Toe as a symmetric chess-like game.

9.2.2 Theoretical

The Tic Tac Toe example above showed that the class of symmetric chess-like games covers more games than might have appeared from its (intentionally) restricted syntax. Given that some games are more naturally represented, and some less so, an obvious question is: ultimately, what can be represented as a symmetric chess-like game? In this section we prove that at the level of game trees, the class of symmetric chess-like games contains all the finite two player games of perfect information.

We begin with a few definitions:

9.2.2.1 Definitions

Following Banerji and Ernst ([Banerji and Ernst, 1971]), we shall define a game as follows:

A two-person game is a 5-tuple $< S, R, P, W, L >$. S is the set of game situations. R is the legal move relation: sRt if and only if t is the result of making a legal move from situation s. P is the set of situations in which the first player (henceforth called $player$) is to move. W is the set of situations in which $player$ has achieved a goal, and L is the set in which $opponent$ has achieved a goal. We define D to be $W \cap L$, the set of drawing positions, and shall say that $W - D$ and $L - D$ are winning, and losing positions, respectively.

Finally, two-person games must satisfy the following postulates:

- G0: $P, W, L \subset S$.

- G1: Dom$(R) \cap W \cup L = \emptyset$
- G2: $sRt \Rightarrow (s \in P \Leftrightarrow t \notin P)$

G0 indicates that the terminal positions are in fact positions in the game.
G1 requires that there are no legal moves from a position in which at least one player has achieved a goal.
G2 ensures that there is strict alternation between players.

A game is a *perfect information game* if both players can distinguish all situations in S (they know perfectly which state they are in), and if R is deterministic (if sRt and a player chooses t, the resulting position will be the current state of the game).

A game is *finite* if S is finite (thus a finite game might never end, but it is finite because the players will visit only a finite set of positions).

9.2.2.2 Coverage Theorem

With these definitions, we now present the following theorem:

Theorem 1 *For every finite two player game of perfect information G there exists a corresponding symmetric chess-like game S such that G and S have the same game tree.*

Proof:

Suppose G is a finite two player game of perfect information. Then G can be represented as a finite graph, where the nodes represent positions, and a directed edge exists between two nodes $n1$ and $n2$ just in case the player to move in $n1$ can choose to move to $n2$.

We shall now represent G as a symmetric chess-like game S. For each position g_i in G we define a corresponding piece s_i in S, such that the piece s_i can promote to `[opponent` s_j`]` exactly when there is a legal move from position g_i to position g_j, and s_j is the piece corresponding to g_j.

For each position g_w in win(player) we make a corresponding goal to arrive `[player` s_w`]` on the board, and conversely for `opponent`.

There is now a 1-1 correspondence between positions. The same player is to move in any pair of corresponding positions, and a player wins in a position in the finite two-player game of perfect information just in case the same player would win in the corresponding position in the symmetric chess-like game corresponding to it. Thus the two games have the same game tree.[3] □

[3] Actually, they have the same game graph, which implies they have the same tree.

9.2.2.3 Discussion

This result is provided more for completeness than actual utility: while theoretically possible, it is clearly impractical to represent a game of reasonable size by explicitly encoding its game tree.

A more interesting result would be one characterising the class of games which can be represented as symmetric chess-like games using an *abstract representation*, in the sense of Section 3.2.1.1. For example, one conjecture might be as follows:

Conjecture 1 *Any game with rules representable as a Prolog program P has a representation as a symmetric chess-like game whose size is polynomial in the size of P.*

This particular conjecture does not seem likely at present. For example, the game of Go has a small representation as a Prolog program, but it is not clear how any game isomorphic to Go could be represented as a symmetric chess-like game. One problem is that the capture method used in Go, that of surrounding a set of connected pieces, is different from any of the capture methods in this class. Although there might be a symmetric chess-like game in which the players are forced to make a series of moves which have the net effect of removing a captured group, such a construction has yet to be suggested. Also useful, of course, would be to find a counter-example which refutes a conjecture like the above. This would help to place tighter constraints on the games compactly representable in this class.

A second type of useful result would characterise the games which are contained in this class under different assumptions about the strategic abilities of the players. For example, the version of Tic Tac Toe discussed above was not strictly isomorphic to the original game. Our version forces players to play immediately winning moves whenever possible.[4] However, this means that there are positions reachable in Tic Tac Toe proper that are not reachable in our encoding of it.[5] The same consideration applies to our representation of chess (see Appendix D.2.2), which does not legally prohibit a player from leaving his king in check. However, if a player is doing one full ply of lookahead, he will certainly never leave his king so threatened. Thus, he will play effectively the same game as he would have with the check rule in operation.

9.3 Complexity of Reasoning about the Games

The previous section discussed the generality of the class of symmetric chess-like games, in terms of which types of games could be represented in it. In this section we consider the computational complexity of determining certain properties of games in this class.

[4] If players are not forced to take wins when available, there are positions in which a player constructs three-in-a-row without winning by the game definition.

[5] The interested reader is invited to construct such a position.

The first question we are interested in is the complexity of deciding whether a player could *possibly achieve*, as opposed to force, a particular goal. This question came up in the design of the generator, as it would have been desirable to have the generator or a playing program simplify game definitions by removing goals which could not possibly be achieved (see Section 8.2.4). One type of goal is that of eradicating a piece of a certain type T. It happens that generated games contain such eradicate goals, even though it is clear that they could never be achieved. For example, if there are no pieces which are defined to be able to capture pieces of type T, and pieces of type T promote to themselves, then it is obvious that once there is a piece T on the board, it will remain there forever (i.e. it is *invariant*).

9.3.1 Possible Wins is NP-Hard

It turns out, as again might have been expected, that this question is not easy to answer in the general case:

Theorem 2 *The question: "Given a symmetric chess-like game* **G***, can a player possibly win* **G***?" is NP-hard.*

Proof: (by reduction from 3SAT).

We show that an arbitrary instance of the domain 3SAT [Garey and Johnson, 1979, page 46] can be transformed into an instance of this question in a polynomial number of steps. 3SAT is a restricted version of the Satisfiability problem which is often used to prove NP-Completeness. For full definitions of 3SAT and NP-Completeness, see [Garey and Johnson, 1979].

The basic idea of the proof is that we can represent assignments to propositional variables by choice of promotion for the corresponding piece. Thus, for each proposition P_i in a 3SAT problem, we define a corresponding piece Val_i, which moves onto the main board (as in our representation of Tic Tac Toe in section 9.2) and promotes to either $true_i$ or $false_i$.

This gives us a way of generating any possible total truth assignment (tta) on this set of propositions. Now we will be done if we can make a goal in our language such that we can achieve this goal if and only if some tta satisfies the set of constraints.

Let the constraints in an instance of 3SAT be $c_1, \ldots, c_n$. For each constraint $c_j = p \vee \neg q \vee \ldots$, define a corresponding piece $test_i$, which can move left or right as it likes (thus choosing a proposition which will satisfy it), and only moves forward by hopping over the piece representing one of its disjuncts instantiated correctly, like [{player} $\{true_P, false_Q \ldots\}$]. After hopping, let it promote to a piece called *constrained*, which can move around the region, to clear room for other constraints to hop through also.

We also start the game with a *satisfier* piece, who sits leftmost on the row where the propositions move to. The satisfier's only move is to hop *forward* over n *constrained* pieces (with no empty squares before or after) where n is the number of constraints. After hopping, it promotes to a piece called *satisfied*.

Now let the goal of a symmetric chess-like game be to arrive a *satisfied* on that final square. In order to achieve the goal (i.e. win the game), we must do the following:

1. Find a fixed assignment of propositions such that each constraint will be able to hop over one of its component propositions.

2. For each constraint: (a) Hop over a proposition which satisfies it. Finding such a proposition takes at worst O(P) time where P is number of propositions. (b) Move the constraint to clear room for others. The simplest method is to require that the nth constraint move to square(1,N+1).

3. When all n constraints are lined up in front of the satisfier, it can then hop over them to win.

As this is our only goal to win the game, the game can be won if and only if we have a *satisfied* on the board. This happens only when n *constrained* are lined up in front of a satisfier. The only way we increase the number of constrained pieces anywhere is when we promote a *constraint* into a *constrained*. Since this number is monotonically increasing (*constrained* pieces do not promote into anything else), we will have n of them only when we have promoted all n *constraint* pieces.

We can only promote a *constraint* piece when the total assignment at that time satisfies it. Thus we can only promote *all* of them when, at successive states, the *tta* at that state satisfies each constraint in order.

Since pieces corresponding to *propositions* never move once they are on the board, the *tta* is also monotonic, in that earlier *tta*s are always generalisations of later *tta*s, so that any later *tta* would satisfy any constraints that all earlier ones did. Thus, the same (and final) *tta* satisfies all the constraints.

Thus, the set of constraints is satisfiable if and only if the game can be won.

Therefore, if we had a program to determine, in polynomial time, if an arbitrary symmetric chess-like game could be won, we could use this program to solve, in polynomial time, any 3SAT problem, so it is at least as hard as 3SAT. Therefore, the symmetric chess-like game winnability problem is NP-hard. □

Discussion This theorem showed that determining whether a symmetric chess-like game game could possibly end in a win for one player is NP-hard. An open question is whether this problem is also in NP, and thus NP-Complete. At present we do not have an answer to this question.

The difficulty in answering stems from the fact that the problem is not necessarily in NP, because solution sequences to arbitrary symmetric chess-like games may be of length exponential in the size of the game description. This means we could not verify a proposed sequence in time polynomial in just the game description.[6] For example, consider a game on a million by million board, where a player wins if a piece can reach

[6] I am grateful to Carl Witty for this observation.

a square on the opposite side of the board. As the board size is represented in decimal notation in the game description, the board size component of the game description is only 7 digits long, while the maximum solution length is an exponential function of this.

We note in passing that the questions (a) is a given move legal from a given position, and (b) is a goal is achieved in a given position, can both be answered in time polynomial in the length of the game description. These observations can be determined by inspection from the domain theory for symmetric chess-like games provided as Appendix B. From this it follows that the question would be in NP if it were possible to determine a polynomial bound on the length of the shortest move sequence to achieve a goal from a given position. At present the existence of such a bound does not appear likely.

9.3.2 Forced Wins is NP-Hard

Before we complete this section, we present two more results about the complexity of determining if a player could *force* a win in an arbitrary game:

Theorem 3 *The question: "Can a player force a win in a symmetric chess-like game from a given position?" is NP-hard.*

Proof: this problem must be at least as hard as determining whether a game could possibly be won at all.

9.3.3 Forced Wins is PSPACE-Hard

Finally, this problem is also PSPACE-hard:

Theorem 4 *The question: "Can a player force a win in a symmetric chess-like game from a given position?" is PSPACE-hard.*

Proof: this follows from the similar result on NxN checkers ([Fraenkel *et al.*, 1978]), and the fact that checkers can be represented as a symmetric chess-like game (Figure 7.3).

9.4 Desiderata Revisited

Section 6.2.1 stated some desiderata for good classes of games to be used in a given application of the general idea of Metagame. Section 4.2.3.2 suggested that a significant effort could be devoted to ensuring that a given research problem is appropriate for the goals it is supposed to further. In this section we restate these desiderata and discuss the extent to which we have observed the class of symmetric chess-like games to fulfill them. The desiderata were as follows:

- **Coverage:** A good class should be large enough to include several games actually played by humans.
- **Diversity:** A class should be diverse enough to include arbitrarily many possibly different games, to discourage researchers from exhaustively analysing each possible game and still building their own analysis into the program.
- **Structure:** A class should still be small enough to represent the structure which at first blush makes the individual games appear similar.
- **Varying Complexity:** The generated games should be of varying degrees and dimensions of complexity, so that different games afford different analysis methods.
- **Extensibility:** It should be easy to generalise the class to increase the coverage of known and unknown games.

These desiderata are generally informal, and serve more for motivation than as rigid constraints. However, following the discussion in this chapter and in Chapters 7 and 8, it should now be clear that the class of symmetric chess-like games measures favourably on each criterion.

9.4.1 Coverage

In terms of coverage, Section 9.2.1 has shown empirically that the class does include at least most of the rules of many games actually played by humans. Some rules, which appear to be details of these games, are not included in our current representations of them. Section 9.2.2 has also shown that the entire class of finite, two-player games of perfect information are included in the class, although this does leave open the question of whether all such games could be compactly represented. Types of games that cannot be compactly represented are exponentially unlikely to be produced by a generator.

9.4.2 Diversity

As for diversity of this class, Section 9.2.1 has shown empirically that this class contains a variety of qualitatively different games, including many which do not on the surface seem even chess-like (like Tic Tac Toe). Section 9.3 has also shown that the class contains games corresponding to all logic satisfiability problems, for which answering even simple questions can be combinatorially difficult. As a consequence, humans could not in advance analyse all possible games unless they could do the same for all NP-complete problems.

It should be pointed out that the discussion of the *class* in this chapter does not demonstrate conclusively that the current game generator with a particular setting of parameters does in fact generate such diverse problems with high likelihood. Although practical experience with the generator does indicate that the generated games are diverse in some sense, a more thorough analysis of the game generator could make this more convincing.

9.4.3 Structure

This class of games was explicitly designed to represent games in a manner preserving their structure. The success in this respect is shown by the compactness with which real games have been encoded as instances of this class. For example, the definitions of checkers and Tic Tac Toe as instances require one page each, and chess requires only two. This means that we can expect to benefit from previous work on known games since we have maintained the structure which past work exploited. We can also expect that a small generalisation of current game-specific methods will enable programs to play automatically a large set of games with similar structure to the games for which the original methods were developed.

9.4.4 Varying Complexity

As discussed in Section 8.2.3, the generator parameters can be modified to change the distribution of generated games having various degrees and types of complexity. For example, modifying the mean board size corresponds to increasing the average branching factor, while modifying the mean number of pieces increases the complexity of the rules and piece interactions. Thus the class and generator can be used to explore the effectiveness of different analysis and search methods on different distributions of problems.

9.4.5 Extensibility

The current class of games is itself the generalisation of several earlier, more restricted classes. It is easy to generalise the class by adding new types of movements, captures, promotion, capture effects, goals, and so on.

For example, the class could be extended to cover (compactly) games like Othello[7] by adding a *bracket* capture-method, a *conversion* capture-effect, and a goal type based on the relative number of pieces owned by both players at the end of the game.[8]

[7]It is possible that a game isomorphic to Othello could be compactly represented in the current class. Because all captures in the present class have the effect of removing the pieces from the board, it is hard to see how the effect of in-place piece conversion used in Othello would be represented.

[8]The changes just mentioned will make sense to readers familiar with Othello. Describing them formally here would be unnecessarily detailed.

9.5 Summary

This chapter has analysed the class of symmetric chess-like games in detail to assess its usefulness as a basis for Metagame-playing. Section 9.2 characterised the games which are instances of this class. The section showed that most of the rules of many games which have been studied in the literature can be represented compactly as instances of this class. For some rules which are highly idiosyncratic, it is not clear that they can be represented compactly in this class. The section also proved a theorem which stated that all finite two-player games of perfect information can be represented in this class, but it is an open question precisely which of these could be represented *compactly*. Section 9.3 addressed the computational properties of games which can be represented compactly in this class. One interesting result proved in the section is that deciding whether an arbitrary instance game could end in a win for one player (even if both players cooperated) was combinatorially hard (i.e. NP-hard). Finally, Section 9.4 assessed this class of games with respect to each the desiderata for Metagame-classes layed out in Section 6.2.1. The section showed that the class measures reasonably well in terms of coverage, diversity, structure, varying complexity, and extensibility. This implies that the problem of SCL-Metagame is a good instance of a Metagame research problem, and that competitive performance on this new problem will be evidence of increased general ability in game-playing.

Chapter 10

Summary of Part II

This part of the thesis has discussed both the general issues involved in the construction of concrete Metagame research problems and the construction of one specific Metagame research problem, called *Metagame in symmetric chess-like games*, or SCL-Metagame.

Chapter 7 defined a class of games to serve as a concrete basis for Metagame-playing. The class generalises many features of the chess-like games, instances of which have received much of the attention in CGP, and represents games in a manner which preserves the compact structure which makes them appear similar. In order to increase the chances that arbitrary generated games within this class would be *fair* to both players, all games in this class are constrained to be symmetric. This is achieved by requiring the rules of any instance game to be defined from the perspective of one player only. A global inversion function then produces the symmetric set of rules from the perspective of the other player. The chapter provided the detail necessary to enable this specific problem, SCL-Metagame, to be used as a testbed by other researchers. The definition consisted of a formal syntax in which game rules will be encoded and a formal semantics in which the rules will be interpreted.

Chapter 8 developed a general method for automatically generating problem definitions for instances of a class of problems. The method is called *Constrained Stochastic Context-Free Generation*. It operates by making statistical choices at each decision point in the grammar defining the class of problems. The probabilities of making different choices are controlled by parameters, and the structures produced at each choice point are rejected if they fail to pass user-imposed constraints. This method of problem generation was used to implement a generator for symmetric chess-like games. The generator has produced games which are objects of interest in their own right, despite the apparent complexity and unfamiliarity of the rules.

With the class and generator fully instantiated, Chapter 9 analysed the class of symmetric chess-like games in detail to assess its usefulness as a basis for Metagame-playing. The chapter analysed the coverage and computational properties of the class and then assessed the class of games as constrained by the generator with respect to each the desiderata for Metagame-classes layed out in the introduction to this part of the thesis (Chapter 6). The chapter showed that the class measures reasonably well in terms of coverage, diversity, structure, varying complexity, and extensibility. The conclusion from this is that the problem of SCL-Metagame is a good instance of a Metagame research problem, and that competitive performance on this problem will be evidence of increased general ability in game-playing.

Thus, the work in this part of the thesis offers a new research challenge, SCL-Metagame. This problem was designed with the explicit intention that strong performance on the problem should correlate with increased generality, flexibility, and autonomy. Having spent substantial effort assuring ourselves that the class of symmetric chess-like games satisfies our desiderata, one logical next step is to construct programs to play SCL-Metagame. Part III will now document progress to date in addressing this new research problem.

Part III

Metagamer

Chapter 11

Introduction to Part III

11.1 Introduction

Part II of the thesis has introduced a new research problem, called *Metagame in symmetric chess-like games*, or simply SCL-Metagame. The problem is to design programs to play unknown games in a large but well-defined class, as output by a game generator. This part of the thesis discusses both the general issues involved in the construction of Metagame-playing programs and the specific directions which have been taken in the development and implementation of the first programs to address this new problem.

11.2 The Problem

As shown in Figure 11.1, repeated from Chapter 4 (page 96), the rules of specific games will only be generated after the program is entered in competition with other programs, at which point the human is no longer able to modify the program in any way.

Because of this, any game-specific optimisations and analysis must be performed by the program, although the human is still free (and even encouraged) to exploit all information available in advance of competition. As indicated in the figure, the following sources of information can be exploited when constructing a playing program:

- the definition of the class of games
- the details of a game generator
- general game knowledge
- resource bounds

Now, given the above sources of information, the following questions must be addressed as we try to develop a playing program:

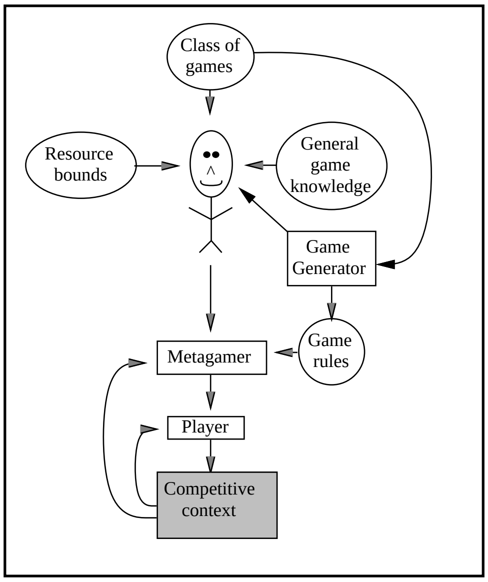


Figure 11.1: Metagame-playing with new games (repeated from Figure 4.1).

Generality: How do we make a program general enough to play legally all the games in the class without human assistance (regardless of playing well)?

Efficiency: Does generality necessarily imply inefficiency, or can the program make itself more efficient once given the rules of a particular game?

Search: Is it easy to build a strong player for this whole class using just game-tree-search and a naive evaluation function, or are more sophisticated techniques necessary?

Knowledge Acquisition: How do we find general knowledge and strategies that might be useful for many games in this class, when at present we have only knowledge and strategies that are useful for specific games within the class?

Knowledge Representation: How can we represent general knowledge to a playing program without knowing the details of specific game rules?

Competitive Advantage: Does the knowledge and search provided to the program actually provide it with a competitive advantage on games unknown to its programmer in advance of the competition?

11.3 Overview

These questions are discussed in turn in the following chapters. Chapter 12 addresses the linked issues of generality and efficiency, and shows that it is possible to achieve both of these goals to some extent by shifting some of the work of building special-purpose programs onto the program itself. Chapter 13 observes that several aspects of game-tree-search need special consideration in this more general context, but that standard methods can still be used to construct a search engine which enables a program to search as deeply as resource bounds allow, on any game in the class. However, the chapter does point out that the search engine requires a good evaluation function for each game the program plays. In order to provide such a function, Chapter 14 considers the issue of knowledge acquisition. The chapter presents several methods which have been useful in generalising existing game-specific knowledge and in analysing by hand the details of the class in order to isolate important general strategies. Chapter 15 then discusses how some general strategic processing motivated by the preceding analysis has been implemented in a program, called METAGAMER. When presented with the rules of a specific game, METAGAMER analyses the game to convert the general strategies into a game-specific evaluation function which can then be used by the search engine to play the specific game without human assistance. Chapter 16 discusses experiments which assess the extent to which the knowledge encoded in METAGAMER provides it with a competitive advantage across games unseen before the time of competition. Chapter 17 provides examples of METAGAMER playing the known games of chess and checkers against humans and specialised programs in order to identify areas for improvement. Finally, Chapter 18 concludes this part of the thesis.

Chapter 12

Representation and Efficiency

12.1 Introduction

Of particular importance in developing more general problem solving systems, such as a Metagame-playing program, are the linked issues of *representation* and *efficiency*. As discussed in [Russell and Wefald, 1992], AI is concerned with making reasonable decisions with limited resources. If we neglect time concerns, then all the games in this class can be seen, from the pure game-theoretic perspective, as "trivial", in that their value can be calculated perfectly [von Neumann and Morgenstern, 1944]. If we take time resources into consideration, however, it is clear that programs which perform their basic computations (such as move generation) more efficiently than others have a marked competitive advantage, other factors being equal. This explains why much current research in CGP focusses on extremely efficient implementations of the basic computations, to the extent that each idiosyncrasy of the rules of a particular game are optimised in advance by the designers of the playing program (for example, [Ebeling, 1986]).

However, an approach relying on a highly-efficient but special-purpose representation encounters difficulties when applied to developing a SCL-Metagame-player. First, as the class itself is fairly general (see Section 9.2), it is difficult to see a way of hand-optimising the entire class of games in advance. Second, we would like to represent the rules in a general and declarative fashion, so that the program can explicitly reason about them for a variety of purposes.

A natural way to reconcile these two opposing goals to some extent is to automate the process by which a general program is specialised to handle specific sets of problems. Two well-understood techniques for doing this are *partial evaluation* [Sahlin, 1991; van Harmelen and Bundy, 1988] and *abstract interpretation* [Cousot and Cousot, 1977; Cousot and Cousot, 1992]. Using these techniques, we can represent the semantics of the class of games in a general and declarative way, but then have the program transform this representation into a more efficient version once it is presented with the rules of a new game. Consistent with the philosophy of Metagame, this process can be viewed as moving some of the responsibility for game analysis (that concerned with efficiency) from the researcher to the program itself.

The rest of this chapter elaborates on the issues of representation and efficiency in our construction of a Metagame player. Section 12.2 discusses our representation of the semantics, and Section 12.3 illustrates the methods used to specialise this general representation into one more optimised for particular games. Section 12.4 summarises the chapter.

12.2 Declarative Representation

The input to a playing program is a *game definition* in the grammar of Appendix A.1 . The program must interpret this game as an instance of symmetric chess-like games, and then play it according to its interpretation of these rules. There are essentially two straightforward ways to bring this about.

One approach would be to write a program to convert the grammatical game definition directly into a program in some language which plays according to the interpretation of that definition. Another approach, and that taken here, is to view the class itself as a meta-game: the class represents the total set of possible moves which could be made in any position, in any game which is an instance of it. A legal move (or any other property) in a position in a *particular* game G, then, can be seen as an instance of all the possible moves (or other properties) such that G satisfies the conditions on the game which validate that move (or property). This has the advantage over the first approach, in that the relationship between all particular games in the class is here specified declaratively, in addition to the relationship between positions in a particular game. This might facilitate analogical reasoning across games (see [Collins and Birnbaum, 1988]).

12.2.1 Game description language

To this end, we have represented the rules for the entire class of games in a *game description language* (*gdl*). The syntax and semantics of this language is very similar to Prolog, with the addition of constructs which access an implicit *current state*, *current player*, and *current game*. These constructs are as follows:

- `true(P)`: P is a state-dependent property, which is true in the current state of the game.
- `add(P)`: add state-dependent property P to the current state.
- `del(P)`: delete state-dependent property P from the current state.
- `control(Player)`: true if Player is the player on move in the current state of the game.

- `game:Pred`: *Pred* is a game-dependent property, which is true of the current game, from the perspective of the current player.[1]
- `transfer_control`: transfers control from the player currently in control to the opponent.

As an example, Figure 12.1 displays a portion of the domain theory for symmetric chess-like games, as represented in *gdl*. Our encoding of this theory represents a legal move as a sequence of legal sub-moves, or `pseudo_operators`, and the rules in the figure are a subset of those defining a *moving* portion of a move (see the discussion on moves in Section A.2). Thus, these rules say that if the current game has a global `must_capture` rule, the current player must make a capturing movement if he has one available, or a non-capturing movement if not. Otherwise, he can make a locally-constrained move, which allows him to move a piece of his choice, and then capture or move normally based on local constraints on the piece. A full semantics of the class of games using *gdl* is presented in Appendix B.

The important point to note about this representation is that all the rules are expressed without *explicit* reference to the current game or to the current state, i.e. the game and state do not appear as arguments in any of the rules. In some ways, this is syntactically cleaner than representations which have state as an explicit argument in their domain axioms (for example, the *situation calculus* [Flann and Dietterich, 1989; Genesereth and Nilsson, 1987; Hölldobler, 1992]), as we can transform from implicit to explicit representation quite easily, whereas the opposite direction requires giving a particular argument of certain predicates (the state predicates) a special status throughout any routines which operate on a theory so expressed.

In our representation, the meanings of these indexical predicates are expressed in the interpreter for this language. This interpreter has explicit arguments for these predicates and interprets indexical expressions in *gdl* by binding these variables appropriately.

The *gdl* meta-interpreter is displayed in Figure 12.2. The first six clauses are standard for implementing a Prolog interpreter in Prolog, and the rest (starting with the case for `true(GIn)`) define the special constructs used for *gdl*, as discussed above.

12.2.2 Flexibility

One advantage to this declarative representation of the interpreter and indexical predicates, as well as the rules defining the class, is that they can all be processed and modified by a program in a variety of ways, and it is easy to use different *state representations* for different purposes. For example, implementing `true_in`, `add_in`, and `del_in` as relations between bags of properties corresponds to a STRIPS-like

[1] Recall that the rules for the opponent are the result of applying a symmetric inversion to the rules from **player**'s perspective (Section 7.2.2).

```
% MOVE Operator:
% If global must_capture rule, and player can capture, then he must.
% Otherwise, player can move any piece, subject to
% local must_capture constraints.

pseudo_op(move(Piece,Player,SqF,SqT)) ==>
        control(Player),
        if( game:global_must_capture,
            global_prefer_capture(Piece,Player,SqF,SqT),
            local_move(Piece,Player,SqF,SqT)).

global_prefer_capture(Piece,Player,SqF,SqT) ==>
        if( capturing(Piece,Player,SqF,SqT),
            true,
            moving(Piece,Player,SqF,SqT)).

local_move(Piece,Player,SqF,SqT) ==>
        true(on(Piece,Player,SqF)),
        if( game:piece_must_capture(Piece),
            local_prefer_capture(Piece,Player,SqF,SqT),
            general_moving(Piece,Player,SqF,SqT)).

local_prefer_capture(Piece,Player,SqF,SqT) ==>
        if( capturing(Piece,Player,SqF,SqT),
            true,
            moving(Piece,Player,SqF,SqT)).

general_moving(Piece,Player,SqF,SqT) ==>
        capturing(Piece,Player,SqF,SqT).
general_moving(Piece,Player,SqF,SqT) ==>
        moving(Piece,Player,SqF,SqT).
```

Figure 12.1: Some rules expressed in the Game Description Language

```
gseval((A,B),SIn,SOut,Game) :- !,
    gseval(A,SIn,S1,Game),
    gseval(B,S1,SOut,Game).

gseval((A;B),SIn,SOut,Game) :- !,
    ( gseval(A,SIn,SOut,Game)
    ; gseval(B,SIn,SOut,Game)).

gseval(if(Cond,Then,Else),SIn,SOut,Game) :- !,
    if(gseval(Cond,SIn,S1,Game),
      gseval(Then,S1,SOut,Game),
      gseval(Else,SIn,SOut,Game)).

gseval((not Goal),SIn,SOut,Game) :- !, SIn=SOut,
    not gseval(Goal,SIn,_,Game).

gseval(setof(X,Test,Xs),SIn,SOut) :- !, SIn=SOut,
    setof(X,
        S1^seval(Test,SIn,S1),
        Xs).

gseval(X^Test,SIn,SOut,Game) :- !,
    X^gseval(Test,SIn,SOut,Game).

gseval(true(GIn),SIn,SOut,_) :- !, SIn=SOut, true_in(GIn,SIn).

gseval(add(GIn),SIn,SOut,_) :- !, add_in(GIn,SIn,SOut).

gseval(del(GIn),SIn,SOut,_) :- !, del_in(GIn,SIn,SOut).

gseval(control(P),SIn,SOut,_) :- !, SIn=SOut, in_control(P,SIn).

gseval(transfer_control,SIn,SOut,_) :- !, transfer_control(SIn,SOut).

gseval(game:Pred,SIn,SOut,Game) :- !,
    in_control(Player,SIn),
    player_game(Player,Game,GameP),
    true_for_game(Pred,GameP),
    SIn=SOut.

gseval(H,SIn,SOut,Game) :- H ==> B,
    gseval(B,SIn,SOut,Game).

gseval(GIn,SIn,SIn) :- operational(GIn), call(GIn).
```

Figure 12.2: A Meta-Interpreter for the Game Description Language

representation, whereas implementing them as situational fluents corresponds to a situation-calculus representation [Genesereth and Nilsson, 1987].

12.2.3 Bidirectionality

In addition, these predicates are all *logical*, in that state is represented as a relation between two variables, `StateIn` and `StateOut`, instead of a global structure which is changed by side-effects (as in a *current board* array used in many traditional playing programs). This enables a program to use the predicates in the domain theory in both directions. For example, by constraining `SOut` in Figure 12.2 instead of `SIn`, a program can determine possible predecessor states, thus using the rules "in reverse" to find all the positions which would have been legal before a given move.

12.3 Automated Efficiency Optimisation

Given this declarative representation of the domain theory for symmetric chess-like games, we created a program which can thus take as input the grammatical specification of a particular game, and play the game by interpreting the rules for legal moves and goal achievement with respect to this particular game. This program is the initial Metagame-playing program.

Unfortunately, this generality of representation does have its costs in terms of efficiency. First, there is a high overhead to interpreting a theory instead of using a compiled version of a theory. Second, it is inefficient to consider possibilities which have no connection to the definition of a given game. For example, in interpreting the game of chess, which has no `must_capture` rules at all, there is no need for a program to spend any time checking for the presence of these rules when playing a game: it would be preferable if they could somehow be eliminated altogether when the program is playing chess.

Fortunately, both of these problems can be overcome by standard techniques from logic programming. Since both the interpreter and the domain rules are declaratively expressed, it is possible to write a program to transform them automatically into a much more efficient version of the same theory, specialised to the particular game.

12.3.1 Partially-Evaluating Game-Specific Properties

Partial evaluation is a technique for specialising a logic program to run more efficiently on a class of queries which is a constrained subset of those on which the program is defined. As this technique is well described in the literature (for example [Sahlin, 1991; van Harmelen and Bundy, 1988]), we will here only illustrate its application to specialising a playing program for a particular game.

As noted in Section 12.2.3 above, our meta-interpreter is defined in a general manner to handle any modes of instantiation in its variables (e.g. , `SIn` could be

```
chess_pseudo_op(move(Piece,Player,SqF,SqT)) ==>
        control(Player),
        true(on(Piece,Player,SqF)),
        chess_general_moving(Piece,Player,SqF,SqT).

chess_general_moving(Piece,Player,SqF,SqT) ==>
        chess_capturing(Piece,Player,SqF,SqT).
chess_general_moving(Piece,Player,SqF,SqT) ==>
        chess_moving(Piece,Player,SqF,SqT).
```

Figure 12.3: Specialised chess domain theory after partial evaluation.

ground, SOut un-instantiated, or vice versa). However, when playing a particular game, we know that the Game variable, for example, will always be instantiated to a particular value upon invocation, so we can propagate this information by symbolic execution. By this method, each rule in the domain theory which depends only on the current game (i.e. not on any properties of the current state) can be executed at compile-time, after which we replace (a copy of) the existing rule with the results of that execution. In the simplest case, if the rule R fails to apply to the particular game, we can be sure that any other rules R', which are conditional on R succeeding, will not be called. Thus, this entire conditional branch can be eliminated from the theory (for this particular game).

Figure 12.3 shows the results of applying this simple example of partial evaluation to the part of the domain theory in Figure 12.1. As the global and local must_capture predicates both fail when applied to the definition of chess, these predicates, and those which are only called after their success, can be entirely removed from the specialised theory. As the number of goal reductions is dramatically reduced, the result is a much more efficient program.

12.3.2 Folding the Interpreter into the Domain Theory

Although this partial evaluation has greatly simplified the domain theory, a playing program using even the new theory so far would still be inefficient, due to the overhead of interpreting the theory. However, it is in fact possible to eliminate this interpretation overhead altogether, by folding the meta-interpreter directly into the clauses in the domain theory.

This transformation rewrites the now game-specific domain theory of Figure 12.3 into the Prolog program in Figure 12.4. It works by replacing the primitive constructs of Section 12.2.1 (like true(P) and add(GIn)) with their interpreted counterparts from Figure 12.2 (like true_in(P,SIn) and add_in(P,SIn,SOut)), and threading the SIn and SOut variables through all defined (i.e. non-primitive) goals which *possibly require* them.

```
chess_pseudo_op(move(Piece,Player,SqF,SqT),SIn,SOut) :-
        in_control(Player,SIn),
        true_in(on(Piece,Player,SqF),SIn),
        chess_general_moving(Piece,Player,SqF,SqT,SIn,SOut).

chess_general_moving(Piece,Player,SqF,SqT,SIn,SOut) :-
        chess_capturing(Piece,Player,SqF,SqT,SIn,SOut).
chess_general_moving(Piece,Player,SqF,SqT,SIn,SOut) :-
        chess_moving(Piece,Player,SqF,SqT,SIn,SOut).
```

Figure 12.4: Optimised chess-specific program after abstract interpretation.

Thus, in Figure 12.4, the goal `chess_pseudo_op` is converted into a new goal which contains both an input and output state, as some of its subgoals can change state. Those defined goals which, if called, could possibly lead to calling a subgoal which *tests* state, but not to one which could possibly *change* it, get extended by a SIn variable only.

12.3.2.1 Abstract interpretation for state-dependency analysis of domain predicates

In performing this transformation, it is necessary to know, for each defined goal, whether it has a recursive subgoal that could possibly change, or at least test, state. This is a question which partial evaluation alone does not answer, as it is concerned with specialising a theory, not gathering information about it. However, this question can be answered using *abstract interpretation* [Cousot and Cousot, 1977; Cousot and Cousot, 1992].

Abstract interpretation is a technique by which we "generalise" a program to make a new approximate program. It consists of mapping the *objects* in a program into *abstract objects*, mapping the *operators* into *abstract operators*, and then executing the program, over all abstract inputs, in the *abstract space* defined by these mappings. As a result of this abstract execution, we will have an approximate characterisation of its behaviour.

An example of abstract interpretation, from [Cousot and Cousot, 1977], is the use of the *rule of signs* to determine that the result of $12638 * -156$ is negative, without actually doing the multiplication. In this rule, we replace integers with (-),0,(+), and ANY, and replace the multiplication operator with a table of how it maps pairs of these abstract objects into new ones. Two entries from this table would be:

$$
\begin{array}{lll}
(+) & *(-) & \mapsto (-) \\
(-) & *(0) & \mapsto (0)
\end{array}
$$

The abstract execution $12638{*}{-}156 \Longrightarrow (+){*}{-}(+) \Longrightarrow (+){*}(-) \Longrightarrow (-)$ proves that $12638{*}{-}156$ is a negative number. Although this example is simple, the method is very powerful, and has been used in applications ranging from *data-flow analysis* to *mode inference* in logic programs [Warren, 1992]. As this technique, like partial evaluation, is thoroughly described elsewhere, we shall not discuss the formal foundations here, but shall instead detail its application to the game analysis problem concerning us in this section.

Abstract Space Thus, our program needs to know the state-dependency of each defined goal in the domain theory. This depends on the definition of that goal, which will either be one of the primitive constructs of Section 12.2.1, an operational goal, another defined goal, or a compound structure relating two or more goals (this is just a summary of the language defined by the meta-interpreter for *gdl* in Figure 12.1). In the framework of abstract interpretation, we thus take the *concrete objects* in our theory to be the primitive constructs and operational goals, and the *concrete operators* in our theory to be the defined and compound goals. Our *abstract space* groups objects into classes based on their state-dependency requirements, and thus consists of the following three *abstract objects*, with the associated abstract interpretations:

2 : we know that calling this goal could possibly lead to calling a goal which changes state.

1 : we know that calling this goal could possibly lead to calling a goal which tests state, but not to one which possibly changes state.

0 : we do not know that calling this goal could possibly lead to calling a goal which tests or changes state.

Abstract Objects In terms of this abstract space, it is easy to define the mapping α from concrete objects to abstract objects:

$$\begin{array}{lcl}
\alpha : & \texttt{true(P)} & \mapsto [1] \\
\alpha : & \texttt{add(P)} & \mapsto [2] \\
\alpha : & \texttt{del(P)} & \mapsto [2] \\
\alpha : & \texttt{control(P)} & \mapsto [1] \\
\alpha : & \texttt{transfer_control(P)} & \mapsto [2] \\
\alpha : & \texttt{G} & \mapsto [0], \text{for operational(G)}
\end{array}$$

This mapping formalises the notion that primitive goals which are quantified solely upon `SIn` have abstract value [1], those quantified on `SOut` as well, have abstract value [2], and operational predicates, which are state-independent, have abstract value [0]. Note that we do not need to define a mapping for `game:Pred`, as by this point all such predicates have been partially evaluated away.

Abstract Operators We then map our concrete operators, the defined goals and logical constructs, into abstract operators. We shall denote an abstract operator as $\alpha(Op)$. These abstract operators are then defined as:

$$\begin{aligned}
\alpha((\mathtt{A},\mathtt{B})) &= \max(A, B) \\
\alpha((\mathtt{A};\mathtt{B})) &= \max(A, B) \\
\alpha(\mathtt{if}(\mathtt{A},\mathtt{B},\mathtt{C})) &= \max(A, B, C) \\
\alpha(\mathtt{not}\,\mathtt{A}) &= \min(A, 1) \\
\alpha(\mathtt{setof}(\mathtt{X},\mathtt{A},\mathtt{Xs})) &= \min(A, 1) \\
\alpha(\mathtt{X} \wedge \mathtt{A}) &= A \\
\alpha(\mathtt{H} : \exists(B, \mathtt{H} ==> \mathtt{B})) &= \max(B : \mathtt{H} ==> \mathtt{B})
\end{aligned}$$

These abstract operators represent the fact that a compound or defined goal can possibly cause any of its components or subgoals to be called, in which case their abstract value is the maximum of any of their components. Note that in the case of the logical operators `not` and `setof`, these would never lead to the state being changed, even if their arguments could cause a modified state to be envisioned. Thus they never map to an abstract value greater than 1.

Abstract Execution Now that we have defined the relation between our abstract program and the concrete program, we can execute the abstract program in order to determine the abstract values of each of our defined goals. Our analysis proceeds in a sequence of iterations. We start in the most abstract space, in which we know nothing about the abstract values of any of our defined goals, and execute the program in this abstract space, to determine if we necessarily gain any information which forces us to move to a less abstract space. If the execution in an abstract space leaves us in the same space, then clearly all further executions will leave us in this space also, implying that we have reached a *least fixed point* in our approximation. At this point we are finished with the abstract interpretation, and by the construction of our abstract mapping we are guaranteed that we have correctly classified all of our defined goals in terms of their state-dependency number (a proof of this can be found in [Cousot and Cousot, 1977]).

An Example In case the above description was too abstract, this analysis can be interpreted algorithmically, as follows: we begin by assuming that all defined goals have state-dependency (henceforth *stativity*) 0, which means we know nothing about their stativity. At each iteration, we update our assumptions on stativity for each defined predicate based on our assumptions from the previous iteration.

For example, our theory might consist of the rules:

$$A \implies (B, C). \tag{12.1}$$

$$B \implies true(P). \quad (12.2)$$
$$C \implies D. \quad (12.3)$$
$$D \implies add(Q). \quad (12.4)$$

We would begin our analysis assuming that all predicates A, B, C and D have stativity 0. Based on this knowledge, our first iteration reveals that B has stativity 1 (as calling B immediately calls a `true(P)`, which performs a test on state), and D has stativity 2. As we assumed at the start that all predicates had stativity 0, at the end of this first iteration we have not determined anything new about A or C. Then on the next iteration, from ($B = 1$ and $C = 0$) we conclude that A is at least 1. And from $D = 2$, we conclude that $C = 2$. On our third iteration, from ($B = 1$ and $C = 2$), we conclude $A = 2$. On our fourth iteration, we gain no new knowledge (our assumptions about all the goals stays the same), which means that every further iteration will have the same result. Thus, our final knowledge of the stativity of this theory is: $(A = 2, B = 1, C = 2, D = 2)$, which is correct.

The result: A chess-specific program Having completed the state dependency analysis, the program then transforms the theory to eliminate the interpreter, thus yielding the final efficient and specialised Prolog chess program in Figure 12.4. As all of the transformations performed here were logic-preserving, this program is still logical and bidirectional, but is now optimised to generating moves and evaluating goals only in chess positions: most of the inefficiency due to the general representation has been eliminated automatically.

Thus, the net result of all this optimisation can be viewed as a game-specific program generator which takes as input the rules of any game within the class of symmetric chess-like games and, by analysing the rules of this game as an instance of the general class, produces a special-purpose, efficient program to play just that game.

12.3.2.2 Implementation Details

Under Sicstus Prolog 2.1 patch 6 on a SUN4 SPARC-2, finding all the legal moves in the initial position of Chess under the original (general) representation takes 314 seconds. On the specialised representation resulting from the processing discussed above, the same computation takes 1 second. This speedup enables the current program to search 5 half-ply deep in chess in 45 minutes. It should be noted that this is still very slow by the standards of specialised game-playing programs. Further efficiency improvement is a major area for future work.

Although these optimisations eliminate overhead and conditionality, the underlying state representation used by the current program is still inefficient. For example, a board is represented as a tree instead of a constant-time array. Because of this, checking the occupancy status of each square on a chess board (8x8) requires 16 steps instead of 1.

I anticipate further speedup from other optimisations using the automatic partial evaluator in MIXTUS-PROLOG [Sahlin, 1991]. As this is one of the largest applications of MIXTUS to date, more work on it appears necessary before it can be applied usefully to this domain theory [Sahlin, 1992].

12.4 Summary

This chapter dealt with the initial issues involved in realizing a SCL-Metagame-playing program, given the rigorous definition of the problem developed in Part II. As SCL-Metagame is a more general problem, it is natural that one of the most pressing issues in the construction of a program to address this problem should be the tradeoff between the representational goals of generality and flexibility, on the one hand, and the operational goals of specialisation and efficiency on the other.

While these goals seemed incompatible at first, the chapter showed that it was possible to achieve them both to some extent, by shifting some of the work of building special-purpose programs, normally the task of the human researcher, onto the program itself. It is interesting to note that this was achieved by first developing a naive game player which was *extremely* general, flexible, and inefficient, and only then automatically transforming this program into a more efficient specialised player of a particular game.

To summarise, the approach taken in this chapter to realize a practical SCL-Metagame-player consisted of the following steps:

1. **gdl:** design a general-purpose language for describing games, and represent the entire class of games as one big meta-game, where a legal move is one which could be legal in any possible situation in any game in the class.

2. **gseval:** implement a naive player as a declarative and flexible meta-interpreter for this language.

3. **peval:** partially evaluate this interpreter to specialise (a copy of) the domain theory for a specific game.

4. **stativity:** use abstract interpretation to determine the state-dependency requirements of the predicates in the specialised domain theory.

5. **transform:** use the results of this interpretation to fold the interpreter into the specialised domain theory, to eliminate the overhead of meta-interpretation. The result is an efficient Prolog program to play a specific game, without the inefficiencies due to the generality of the class definition or the flexibility of meta-interpretation.

In concluding this chapter, it is interesting to observe that the issue of efficient representations in game-playing systems is often seen as an engineering concern which is somehow separate from the scientific work of playing the game in a given representation. However, the issue of *automatic* efficient change of representation is a very important scientific problem [Benjamin, 1990]. This chapter has shown how techniques from logic programming can greatly assist in this endeavour.

Chapter 13

Basic Metagame Players

Elephants don't play chess – Brooks

Chess is not skittles – Kasparov

13.1 Introduction

This chapter discusses how the basic game-playing components produced by the game-specialiser in Chapter 12 can be put together to construct a variety of basic Metagame-playing programs using only game-tree-search and minimal evaluation functions. To begin with, Section 13 develops some simple players which play mostly randomly. These serve as baselines against which to compare later players, and also demonstrate that it is at least possible to play *legally* any game in this wide class of games. Section 13.3 then develops a general search engine based on standard game-tree-search techniques used in CGP and discusses some difficulties of using search on this class of games. Section 13.4 summarises the chapter and discusses the performance of the search engine using a minimal evaluation function.

13.2 Baseline Players

The baseline players developed here are extremely simplistic. They only make use of the legal move generator and goal detector for a given game, and beyond this perform no analysis of the game rules. The first few players are based on making random moves. The final basic player has a simple evaluation function which counts the number of possible moves and the number of pieces.

13.2.1 Random Players

The simplest possible player, called *random*, ignores goal detection altogether, and uses the move generator to choose a random legal move. Although the moves it plays are all legal, games played by the random player are obviously of extremely low quality.

A slightly more sophisticated random player, called *random-cautious*, incorporates goal detection into level move selection. This player plays a random move so long as the resulting state is not a lost position. As expected, *random-cautious* performs much better than *random* on games where it is possible to play a move which loses immediately (like losing chess), but their performance is equal (i.e. both random) on games where this is not possible (like chess).

Continuing the progression, *random-aggressive* plays a move which wins immediately if one exists, and cautiously otherwise, and *random-aggressive-defensive* (RAD) defends against immediate threats to win while playing *random-aggressive* when there is no such threat.

Each refinement has consisted in adding more search to the random program, which basically results in producing a program which plays randomly in general, but somewhat more intelligently in near-terminal positions. The best player out of this set, RAD, actually makes a decent opponent for a human when playing new games; while the human has much better ability to search, the program has the advantage of understanding and remembering the rules perfectly! In fact, when time limits are taken into account, it appears that RAD is one of the most efficient possible defensive programs; it performs no search but that which is is necessary to ensure the best possible outcome when evaluating near-leaf nodes. Thus, any program which spends time evaluating non-terminal positions must gain a tangible advantage for this effort, otherwise it will be at a time-disadvantage against RAD. Furthermore, for programs which are not able to evaluate more than the first ply, RAD may actually be a stronger player, as it will never play a move which enables a winning response unless it is forced to do so. These considerations become important during the Metagame tournament in Chapter 16.

13.2.2 A Minimal Evaluation Function

After developing this sequence of random players, the next step was to attempt to provide a program with some minimal knowledge, in the form of an evaluation function, and implement a search engine. As suggested in Chapter 4, implementing any knowledge at all turned out to be much harder to do in the context of Metagame than in the context of a particular game. For example, even the most basic Chess programs are provided with a *material* feature, which weights each piece differently. But since the program will be playing brand new games, we do not even know in advance the names of the pieces that will be used, much less their powers and relative values. Thus, as a start, I implemented an evaluation function with two terms which can be determined to apply without any specific consideration of the rules of a given game. These general features are defined as follows:

- `general mobility difference`: the total number of moves available to **player**, minus the total number available to **opponent**, in the current position. Here the value for the player not on move is computed in a hypothetical state where the player on move has passed.
- `general material difference`: the total number of **player**'s pieces on the board, minus those of **opponent**.

An associated question was how to weight these terms (remember that we are not using a learning system yet). For purposes of experimentation, these weights were left as parameters.

13.3 Search Engine

The search engine incorporates several standard search techniques from the game-playing literature (see [Rich, 1983; Frey, 1983; Levy and Newborn, 1991; Kierulf, 1990]). It is based on a Prolog implementation of the *minimax* algorithm with *alpha-beta pruning*, as presented in [Bratko, 1986, page 366]. In order to cope with the dual issues of time limits and varying search spaces, this basic algorithm is extended with *iterative deepening* [Korf, 1985]. That is, the engine performs a search down to 1 ply, then 2 ply, and so on, until it has run out of time. As is common with iterative-deepening searches, the search engine uses the *principal continuation heuristic*, which orders the moves in the principal continuation (the best set of moves and responses found at the previous iteration) above the others. This serves two purposes in our case. First, it increases the benefits of pruning by ordering the most promising move first. Second, it ensures that a program which runs out of time *within an iteration* will fall back on the move it would have made after the last full iteration, unless it has fully explored another move during the final incomplete iteration which received a higher evaluation.

As there is nothing particularly new about the search engine, we shall not discuss the details of the implementation in the main text. The interested reader is referred to the literature mentioned above for a more thorough explanation of the general concepts involved in game-tree search. The following sections mention some fine points regarding the search engine as applied to Metagame in symmetric chess-like games.

13.3.1 Using Partial Iterations

The issue of failing within an iteration is particularly relevant to Metagame programs. Many game-specific programs check before starting the next iteration whether the program has enough time to search it fully. However, given the wide variety of games in this class, it is common that a program does not have enough time even to evaluate all moves at the first ply. In this case, the programs at least select the highest-valued choice among those they have evaluated. This technique also allows the programs to benefit from partial iterations after the first, with observable benefits (see Section 17.4).

13.3.2 Game-Assumptive Search Methods

In addition to the heuristics currently used by the search engine, a number of other standard search heuristics could all in principle improve the performance of the search engine. Additional heuristics include *windows*, *singular extensions*, *conspiracy search*, *killer heuristics*, *quiescence search*, and *hashing* Some of these have not been implemented due to time constraints, and others have not because their use is *game-assumptive* (in the sense of Section 3.2.1.2).

For example, *quiescence search* is a technique in which unstable positions (i.e. those whose estimated value is likely to change after further search) are searched more deeply. A common application is that a position is always expanded when piece captures are possible. The problem with applying this to symmetric chess-like games is that it assumes that captures are infrequent. On the contrary, several games have been generated in which every move is a capture. Applying such a heuristic to these games would thus result all positions appearing non-quiescent, rendering the quiescence search ineffective. However, one way around this problem might be to have the playing programs decide for themselves whether the assumptions behind such game-assumptive search methods actually held of a given game, in which case improved performance could result. This is an idea for future work.

13.3.3 Move Ordering

When expanding a position during search, the search engine uses the legal-move generator for the given game (described in Chapter 12) to produce a list of moves possible from that position. As the efficiency of alpha-beta search is improved by better move orderings, a standard technique in CGP is to use heuristic ordering functions at this point [Levy and Newborn, 1991, page 172]. The current search engine chooses one of two possible orderings, depending on the value of an internal parameter (called `ordering`):

fixed Uses the moves in the order in which they are generated. This is in general arbitrary, and depends on the game definition, the *gdl* encoding of symmetric chess-like games, the order of Prolog clause selection, and the process of game compilation (Chapter 12).

random The move order is randomised.

In either case, if one of the possible moves is on the principal continuation based on the previous iteration of search, that move is then moved to the front of the list.

In the experiments reported in this thesis, the *random* ordering is used. The point of this is as follows: if the program does not have enough time to evaluate even the first ply, or if it found several nodes equal in value to it but not better, choosing its current best move will result in a *random* choice being made, instead of the first generated move. This can be important in Metagame matches, as otherwise two programs with indiscriminant evaluation functions or strict time controls wind up playing the same game every match. As an example in Section 17.4 shows (see the discussion following Diagram 9, page 174), using a random ordering may be suboptimal if the move on the principal continuation is bad. Experimenting with different general move-ordering schemes is another area for future work on the search engine, and could considerably improve the performance of future Metagame-playing programs.

13.3.4 Time Management

Time limits are an important part of the competitive context. Different limits may be specified which constrain any or all of the following:

move-time-limit: The maximum amount of time within which a player must make each move.

game-time-limit: The amount of time allotted to a player to play the entire game.

tournament-time-limit: The amount of time allotted to a player to play an entire tournament.

Effective time management [Hyatt, 1984; Markovitch and Sella, 1993] is an important and difficult topic even within the context of a specific game known in advance. The importance lies in the fact that a player who spends less time thinking about *easy* positions in order to spend more time on *hard* positions[1] has a significant advantage over a player who allocates time uniformly across positions within a contest. While this type of decision is hard enough, an additional complication is that the length of the game is often variable, which means that in addition to guessing how important a move is relative to others, a player must also estimate the number of moves remaining to be played in the game. To make matters worse, the number of moves remaining may in general depend on the quality of move chosen, which depends in turn on the amount of time allocated to the present move.

These considerations, which are difficult enough for playing known games, are even harder to manage effectively in the context of a Metagame tournament on games not known in advance. Whereas for a known game players can begin with a reasonable estimate of game-length, even this information is unavailable when playing a new game. Observing this, it appears that most present time-management strategies are game-assumptive, which implies that so far humans have performed this important

[1] Of course, what constitutes an easy or hard position is a difficult question in itself.

aspect of game-analysis instead of programs. Metagame thus presents a challenge, and a necessity, to understand and automate this style of reasoning.[2]

The current search engine assumes that time-management decisions have already been made by the players using it. The players thus invoke the search with a specified maximum amount of time to spend on the search. The search engine uses all time allotted, with the exception that it stops early when one of the following two conditions hold:

1. A move is forced in the root position (there is only one legal move).

2. The search has determined the true value of the root position (by reaching the end of the game).

In the future, more responsibility for time management may be transferred to the search engine. To bypass this issue until later developments, the games and tournaments played in this thesis take place within a competitive context which sets a limit only on the maximum time-per-move.

13.4 Summary

This brief chapter has discussed the development of some baseline SCL-Metagame players and a general search engine, using the basic game-playing components produced by the game-specialiser in Chapter 12.

Initial experiments using the search engine with the minimal evaluation functions (and similar functions) over a variety of games yielded some interesting results. Many of the moves chosen by the programs made a lot of sense. For example, a program using search with this evaluation function automatically prefers putting some pieces in the centre of the board, moving riding pieces onto long open lines, and capturing opponent pieces with greater mobility. However, none of the programs were capable of long-range planning.

As was also expected, a set of weights to the minimal features which was good on one game often tended to be poor on another game. For example, some weights should have a positive sign when the goal is to stalemate the opponent, and a negative sign otherwise. Unfortunately, as will be discussed in Section 14.2, heuristics as simple as this are insufficient for weight adjustment in the general case.

Before moving on to more sophisticated Metagame-players, we can summarise a few tentative conclusions from the early experience with search and minimal evaluation functions applied to SCL-Metagame.

[2][Markovitch and Sella, 1993] presents an algorithm which learns resource allocation strategies by observing the change in performance when different types of moves are allocated additional resources. One limitation of the work is that it assumes that the number of moves to be played in the game is known in advance. There is certainly much to be done in this area.

Value of Mobility First, the minimal evaluation function combined with deep search performed significantly better than the random players on some games, and thus appears to contain some strategic value with respect to this class of games. In particular, mobility appears to be a very useful heuristic for this whole class of games, even for those games where a player's goal is to have no more moves available. While we found the general notion of material (number of pieces) to be an important feature in almost every game, its sign and value naturally varied from game to game. No fixed function combining these simple terms performed best on all games.

Lacking Goal Direction Second, a real problem with these players was that they were not goal-directed, and for many games these simple features were not powerful enough to guide them to forcing a win, even from positions in which a human would have been able to win easily. For example, the immediate mobility information was not enough to enable programs to checkmate, as they offer no guidance when a player's mobility is maximised.

Human Analysis Uses Abstraction Third, and related to this last point, a recurrent observation when watching humans play new games against these programs is that humans do not get immersed in the details of the rules, nor do they even attempt to consider all possible moves even at the first ply. Instead, they focus on just those rules which are relevant to achieving their goals. For example, if the goal of the game is `stalemate player`, a typical method of analysis is to determine which pieces are relevant to achieving this goal, and effectively ignore the rest.

Deep Search May Still Win Finally, it is important to stress that these initial observations have not ruled out the success of a brute-force approach to SCL-Metagame. It might seem obvious in any case from the analysis of the class performed in Chapter 9 that deep search alone will not be useful because of the combinatorial nature of some of the games in the class (see Section 9.3, page 83). However, those results could be interpreted to mean that *no* programs are likely to perform well on many games in the class, in which case it is possible that a program based solely on deep search would at least be better than other competitors.

Thus, the issue is largely empirical, and to address it properly might well require a significant effort to construct special-purpose machines to provide a search-engine with maximum efficiency, as has been done for chess [Ebeling, 1986]. However, the observations in this chapter indicate at least that it is not *easy* to apply deep search to this problem without having access to a reasonable evaluation function, and that no good evaluation function seems available without a more sophisticated analysis of the class of games, either by a human or a program. The construction of a good evaluation function for this class of games is the subject of the next chapter.

Chapter 14

Metagame-Analysis

14.1 Introduction

With the search engine in place, using the optimised primitive operations, we have a program which can search deeply (subject to resource constraints) in any position in any game in this class. However, in order to play successfully, the program requires a good evaluation function for each game the program plays. Moreover, the human programmer cannot possibly create an evaluation function for a specific game, as the rules of specific games are only presented to the program at the time of competition. The question facing us, then, is: what can we do, using the information we have in advance of competition, to enable the program to construct an evaluation function using the information *it* will have at the time of competition? In all cases, our goal is to have the program win as many games as possible in the competition.

The reader may at this point notice that this is almost identical to the challenge faced in the *gamer's dilemma* considered in Section 3.3, and that we are now in the position of the hypothetical researcher considered in that section! The difference is that we know that the game the program will play is an instance of a well-defined class of games, about which class we have full information.

As suggested in that discussion, the field of CGP offers a number of different approaches to this problem, including knowledge-engineering, machine learning, and self-play. Unfortunately, the section noted that most current approaches, even those using learning, rely on a great deal of human analysis of specific games, which is in our case impossible. But as discussed above, we do have a major source of information that we can analyse if we choose: the definition of the class of games as constrained by the generator.

As this is the first attempt at addressing this new problem, it is interesting to see just how far traditional approaches can be applied here. Thus, I shall attempt to apply traditional CGP techniques, in which the human, instead of the program, analyses the problem, to SCL-Metagame. But unlike in traditional work in CGP applied to known games, in this new context I do not know the rules of the game in advance, but know instead only that the game is in the given *class* of games. So, instead of performing game-assumptive knowledge-engineering, I am forced to perform *class-assumptive* engineering, or what I will call *Metagame-analysis*. As I will not know the details of specific games in advance, I will perform my analysis of the class with an aim to determine how the program can perform analysis of each specific game.

This chapter discusses the general techniques and detailed analysis which has been useful in this effort. Several of the resulting considerations have influenced the construction of advisors and analysis-tables for METAGAMER, the program I have developed to play SCL-Metagame (see Chapter 15). In those cases, the relevant components are indicated in boldface. The proposed game changes have also revealed some game-assumptive properties of current work on specific games, as discussed in Section 3.2.1.2. It should be noted that the analysis presented here is not intended to be complete or definitive. Rather, it provides the motivation for the current implementation, and also furthers this case-study in Metagame-analysis performed by a human. Most importantly, the questions asked and the broad nature of the responses highlights the difficulties encountered when attempting to analyse in advance a class of unknown games.

Before proceeding, it should be noted that this problem turned out to be very hard to approach well by human analysis alone, which indicates that the construction of the problem is successful in this respect.

14.2 Generalising existing features

The first and most obvious approach to finding class-wide knowledge for chess-like games was to examine the knowledge currently used by programs which play specific games in the class. By isolating the assumptions relevant to a game-assumptive concept, it was in some cases possible to generalise the concept to apply to the class as a whole. Three significant and general features emerged in this way: *mobility*, *centrality*, and *promotion*.

14.2.1 Mobility

The concept of mobility is used in some respect in almost all game-playing programs. The common factor in most mobility features is that they compute a set of properties which are necessary, but not always sufficient, for a player to have a legal move in a position.[1] Both chess and checkers programs contain terms in their evaluation

[1] [Rosenbloom, 1982] developed a set of mobility-related features for the game of Othello. [Fawcett and Utgoff, 1992] showed how many of these features could be derived automatically through a process of transformations on the game rules. The sequence of transformations starts with the necessary and sufficient conditions for making a legal move, and then drops conditions. The result is an abstract indicator of mobility, which is necessary but not sufficient. [de Grey, 1985] independently developed a similar approach, and used it to derive automatically mobility-related features in several other games,

functions which count the moves available to each piece owned by each player in the current position. Note that this is not sufficient to guarantee that the player really has those moves available. One example comes from checkers: although many pieces have possible movements, a player might be forced to capture an enemy piece, in which case he may in reality have only one legal move. Another example comes from chess: a player may be credited with a large number of queen moves, although none of them might actually be legal if the player's king were in check.

When trying to develop a general concept of mobility which could be used in this class, it was necessary to determine whether mobility was always desirable, all else being equal, or if its validity assumed some properties of the game. For example, in both chess and checkers, a player wins by capturing all of the enemy pieces. Thus it seemed possible that for *positive* goals of this form, in which a player wins by weakening the enemy, mobility might be useful. In *negative* goals, in which a player wins by weakening himself, it seemed that mobility might be detrimental. One example of such a negative game is *lose-chess*. In this game, each player must make a capture whenever possible, and the goal is to have no more moves (`stalemate-player`, in the language of symmetric chess-like games). Another negative game is the ordinary version of *Othello*, in which a player wins by having the most pieces on the board at the end of the game, but mobility often decreases with each piece a player captures.

After examining successful strategies in both the win and lose versions of these games, the opposite conclusion was reached. That is, mobility appears to be valuable for either type of goal, all else equal. Some evidence for this is that in both win and lose versions of chess and Othello, the openings are almost identical regardless of the final goal. That is, both players strive for increased mobility in the opening, as this gives them greater control. With greater control, they then go on to achieve later goals. Conversely, it turns out that attempts to reach the final goal in spite of reduced mobility often result instead in early defeat. For example, a player who immediately tries to give away all pieces in lose-chess quickly winds up with only a few moves available. Although this means the player has almost achieved the final goal, the opponent is then in such control of the game that he can force the first player to capture all of the opponent's remaining pieces. These points have been illustrated in the strategy for Turncoat-Chess in Section 8.3.3.

14.2.2 Centrality and Eventual Mobility

The concept of mobility as used by current chess and checkers programs can be termed *immediate dynamic mobility*. That is, it is assessed by counting for each piece the number of moves immediately available to that piece *in the present position*. The effect of this is to encourage programs to place pieces on squares where they have many moves, and to restrict the moves available to enemy pieces. However, this concept alone is insufficient for strong performance in two respects. First, it provides

including checkers. None of this work was applied to games unknown to the researchers.

no guidance when a player has reached a square which maximises immediate mobility. For example, a chess king has the same immediate mobility on any non-edge square (namely, 8 moves). Longer-term strategy requires placing pieces on squares from which they can reach a large number of squares quickly, whether or not they reach them on the very next move. Thus a king in the centre of a chess-board can reach every square within four moves, while a king on the edge may take 7 moves to reach squares on the opposite edge.

To help programs distinguish squares and pieces having the same value in terms of immediate mobility, they are often provided with *piece-square tables* or *centrality bonuses* [Hartmann, 1987] which provide bonuses for having pieces on more central squares. As our programs will play unknown games, it was necessary to enable them to construct and use similar tables directly from the rules of the game. This resulted in the **eventual-mobility** advisor and table, which are discussed in more detail later. These concepts are also illustrated in the Knight-Zone chess example in Section 3.2.2.2. One important consequence of including the eventual-mobility advisor is that it enables a program to force checkmate in chess endgames, whereas the immediate mobility advisor alone does not. The reason is that the program first centralises its own pieces, and then forces the enemy pieces to move to squares from which it would take more moves to reach other squares (in chess this results in forcing them first to the edge of the board, and then to the corner).

Static Mobility A second problem with the traditional use of mobility is that it is solely dynamic, based on the present position. If this is used as the sole basis for determining the value of having a piece on a square in a position, it leads a program to dramatically underestimate the value of pieces which are blocked in a particular position. Thus if an enemy knight attacks a program's queen which has only one available move, a program using only dynamic mobility would not move the queen to its safe square. The reason is that in the position under consideration, the queen would be worth only one point of mobility, while the knight would be worth several more. This problem was solved by the addition of a *static-mobility* advisor, which credits a piece on a square with the number of moves it would have from that square on an *otherwise empty board*. Using this advisor, a program would defend its queen against the attack as it realises that the pieces blocking the queen may eventually move, after which the mobility of the queen would be worth more than that of the knight.

Constrained Mobility A further complication in mobility considerations is that while it may be *legal* for a piece to move from one square to another, this may in some cases always lead to immediate loss. This came up when representing Knight-Zone Chess and Chinese-Chess as instances of this class. In both of these games, pieces are excluded from a portion of the board by rules declaring that the opponent wins whenever such a piece reaches one of the excluded squares. A program which did not take this into account when making mobility calculations would attribute much more mobility to restricted pieces than they could ever have when playing the game. This problem was solved by having the procedures which compute piece transitions eliminate those transitions which would never be played by a player without losing immediately (**constrained-matrix**).

14.2.3 Promotion

Similar to centrality-bonuses discussed above, many programs for existing games have promotion-bonuses which encourage the program to move pieces on the path to promotion. For example, chess programs calculate the distance a pawn is from the promotion rank, and give points as that distance is reduced. In those games, however, there are no pieces which can continue promoting into new pieces after they have once promoted, no pieces for which the opponent gets to promote them instead of the owner, and no pieces which promote into enemy pieces. These issues force us to generalise the treatment of promotion, and are discussed in more detail in Section 14.4.

14.3 Step Functions

The second approach attempts to encode knowledge which would allow a program to achieve its goals using only one-ply of search (i.e. considering only the possible moves in the current position). The idea is that under very tight time-constraints, a program should still be able to make progress against a random player. An example of an advisor motivated by this thinking is **arrival-distance**, which measures in abstract terms the progress a player has made toward achieving an arrival goal. Using only this advisor, a program is guided to clear the destination square of any friendly or hostile pieces, and to move the goal piece to the destination square from any other location. The result of this is that a player could achieve goals which require thousands of moves by making only immediate decisions.

The static analysis table constructed for this purpose (**distance-matrix**) was motivated by a dynamic version of the same computation developed by [Botvinnik, 1970] and later extended by [Church and Church, 1979] and [Snyder, 1993]. Two systems exist which derive step functions directly from problem specifications: Zenith [Fawcett and Utgoff, 1992] and CINDI [Callan and Utgoff, 1991]. An important area for future work would be to apply these automatic methods to the formal class definition for symmetric chess-like games provided in Appendix B.

14.4 Game-variant analysis

Given the class of games and chess as an instance, I thought about how various parts of the definition could change to produce new games. If the current evaluation function was insensitive to these differences (i.e. gave the same suggestions as before), this suggested a gap in knowledge and/or built-in assumptions. I then tried to think what additional information could allow discriminations. Either (a) changing some existing knowledge, or (b) realizing that there was a conflicting goal. If (b), I added a new advisor with intention that program could someday work out the tradeoff for itself.

For example, arrival and mobility are important (sub)goals, but in some ways conflict. The arrival advisor encourages a player to get a piece to a destination as quickly as possible. The mobility advisors encourages a player to put the piece where it has many options. When the two give conflicting advice, a gap in knowledge exists. This could be corrected in one of two ways. One way would be to assume that a weight-learning program could eventually modify weights for existing advisors to make the tradeoff more appropriately based on experience with a given game. For example, a program might find that mobility considerations tend to dominate in a particular game, and thus weigh mobility more heavily. The second correction would be to locate special conditions under which the advisors conflict, and make separate advisors which respond to those situations only, and which serve to patch the differences. This type of goal conflict is particularly apparent between self-eradicate goals and mobility goals, as illustrated by the discussion on win and lose versions of chess.

14.4.1 Hypothetical Chess Variants

Some examples of games in which some properties or rules are changed from an existing game (in this case, chess) follow. The question in each case is: other things staying equal, how should strategies change if the following rule change were made to the game of chess?

- If queens could not capture knights?
- If players have two kings instead of one in the initial position and win by capturing both enemy kings?
- If a player has an alternative goal to eradicate his own knights?
- If rooks must capture whenever possible (or not move at all)?
- If a player wins by eradicating everything but the opponent's king?
- If knights additionally travel in a special zone outside the original board?
- If pieces captured by pawns are player-possessed instead of removed?
- If knights capture by moving away from instead of onto a piece?
- If pawns capture like queens?

- If queens capture like pawns?
- If the game is played on a wrap-around board?
- If a player has an alternative goal to arrive her own knight at a8?
- If knights can promote to pawns when reaching the penultimate rank?
- If the opponent decides how a player's pawn is promoted?
- If some combination of the above changes are made?

In the following, I shall discuss some intuitive answers to these questions.

If queens could not capture knights? In chess and checkers, all pieces can capture all enemy pieces. An evaluation function which did not differentiate pieces on the basis of what they could capture would be insensitive to this change in rules. This property is obviously significant, however. Relative to the other pieces, it would seem to diminish the value of the queen, as it can no longer achieve the same effects as the other pieces (**victims**). Also, it would seem to increase the value of the knight, as it is safer from capture than other pieces (**immunity**). Finally, it would lessen the value of the pawn, whose value is influenced by its ability to promote into a queen (**promote**).

If players have two kings instead of one in the initial position and win by capturing both enemy kings? Chess programs have two values for kings [Hooper and Whyld, 1984]: one when used as an ordinary piece and another when considered during piece exchanges. In the first case the king is similar in value to a knight (based on mobility). In the second case the king is given almost infinite value, to ensure both that a program never tries to exchange it, and that a program never leaves its king attacked. When we consider games with goals to eliminate more than one such target, this special treatment in the second case must be changed. We can no longer attach absolute importance to each such target. For example, a player should probably exchange one of his kings for an enemy queen if the chance arises. Instead, it seems that the importance of each target piece in terms of exchanges should be inversely proportional to the number of such pieces remaining (**vital**). When only one piece remained, this treatment would naturally reduce to that in current chess programs.

If a player has an alternative goal to eradicate his own knights? Recall that in a game where players cannot capture their own pieces, a player may eradicate his own pieces by forcing his opponent to capture them, or by promoting them into pieces owned by the opponent (see the discussion on strategy of Turncoat Chess, Section 8.3.2, page 70). In chess and checkers there is no intrinsic disadvantage to capturing any enemy piece, so the naive strategy "capture all enemy pieces" is viable. The above change renders this strategy inapplicable. Also, it implies that both players will have at least one knight on the board throughout the game. One consequence of this is that, at least for purposes of exchange, a knight should be worth *less* to the player who owns it (**eradicate**). Another consequence is that a player should ignore threats against his last remaining knight. Also, a player need not consider whether pieces attacked by his last knight are defended, as is the case in chess (**potent**).

If rooks must capture whenever possible (or not move at all)? With this constraint placed on rooks, their relative value decreases dramatically. For example, an enemy rook may be effectively immobilised by placing a defended pawn under its attack. The rook would not have to capture it, because the player may still move other pieces. Also, the rook would still control the other squares it can move to, since if the enemy moved an undefended piece to one of these squares the rook could capture that piece instead. But the rook would be restricted from serving any active function until it had no captures available.[2] Exactly how these considerations should be used by a playing program is at present unclear. One implication is that it may be useful to own pieces which can be captured by our own pieces, as these may provide extra mobility for otherwise restricted pieces (**giveaway**).

If a player wins by eradicating everything but the opponent's king? In this example game the player need not eradicate the king, i.e. there is no penalty if she also eradicates it. In this game it would seem that all other pieces should increase in value relative to the king. That is, beyond the value derived from the other functions performed by each piece, the presence of a piece has value for a player merely because the opponent derives value from eliminating it (**eradicate** and **vital**).

If knights additionally travel in a special zone outside the original board?

This is the first version of Knight-Zone chess, discussed already in Section 3.2.2.2). The knight should clearly increase in value as it now has additional squares to move to. The intuitiveness of this conclusion shows that piece value derives not only from the squares a piece could reach in one move (**immediate-mobility**), but also from the squares it could reach in several moves (**eventual-mobility**). It also shows that board topology must be taken into account when determining piece values, rather than the structural definition of the piece in isolation. Finally, when this extra freedom for the knight is represented as restrictions on all the other pieces, it becomes clear that such restricted squares should not be taken into account in mobility calculations (**constrained-matrix**).

If pieces captured by pawns are player-possessed instead of removed?

As discussed in Section 7.2.4.2, the *possess* capture-effects allow a designated player to possess the piece upon capture, after which they can place the piece on any square on the board at any later turn (**possess**). This change makes pawn-captures more lethal, and thus increases both the offensive and defensive value of pawns. This means they should increase in overall value relative to the other pieces (**victims**). For example, pieces protected by pawns can no longer be safely captured by enemy pieces of the same type. When the enemy makes the capture, the player's piece is removed, but when the player recaptures, he would in effect win an additional piece. Just the opposite considerations apply if the effect were instead that the opponent

[2] One way the attack could be cancelled is to use friendly pieces to block the attack, which might give rise to some interesting strategies.

would possess the captured piece, in which case the piece would almost never be used for capturing (unless it were forced to capture, of course). This also shows that calculations of the value of threats should take into consideration the *effects* of the threat (**potent**). The fact that work on chess and checkers assumes a uniform capture effect has also made it difficult to transfer some of that work toward the development of Shogi programs.

If knights capture by moving away from instead of onto a piece?

In chess, all pieces capture by landing on (clobbering) a target piece. This means it is possible in chess to focus on a particular square when determining the extent to which a piece is attacked or defended, or to which a square is controlled. For example, it can be determined statically that a pawn defends a piece because it could capture any enemy piece on the same square as the defended piece, and this replacement necessarily occurs whenever the enemy captures the defended piece. This property is exploited by several strategic chess programs [Church and Church, 1979; Botvinnik, 1970; Snyder, 1993]. However, the assumptions behind this approach become clear when we consider pieces with capturing methods other than clobbering. In this changed game, it is no longer correct to assume a pawn defends a piece just because it could capture an enemy which replaced the piece. In this case, a knight could move *away* from the defended piece to capture it. This greatly increases the difficulty of determining the consequences of threats without exploring full moves and responses (**potent**).

If pawns capture like queens?

This change must increase the relative value of pawns, which now control as many squares as a queen. The change in value becomes even more clear when we imagine a piece which moves like a pawn but can capture a piece on any other square. It is tempting to conclude that the latter piece would be worth more than a queen, and that this shows that capturing mobility determines piece value independently of moving mobility. However, while this increases the pawn's ability to attack other pieces, it does not alter its ability to get out of danger. Unless the pawn can capture another piece safely, it is an easy target to capture due to its limited moving mobility. Moreover, while the new pawn may at any time attack as many squares as a queen, it cannot suddenly attack as many new squares as a queen. This corresponds to the distinction between immediate and eventual mobility. In this case, however, the important property is not having a piece on a square, but having an attack against a piece from a square.

If queens capture like pawns? This is the converse of the preceding question. This should reduce the relative value of a queen. It is hard to assess the relative value of (a) a queen which captures like a pawn, and (b) a pawn which captures like a queen.

In general terms, the question is whether it is intuitively better to have a piece with limited moving mobility than to have one with limited capturing mobility. This issue does not arise in chess (or Shogi), in which each piece moves and captures with the same movements. It does arise in checkers. But as kings are more mobile than men both in terms of moving and capturing, this issue has not been important in that context either.

In pre-computing the static-value of each type of piece, our current advisors do not capture this distinction, as they base all static mobility considerations on the moving powers of each piece, not on the capturing powers. A dynamic advisor (**capturing-mobility**) does enable a program to make this distinction to some extent, as it awards pieces points for each other piece they could capture from the current square.

If the game is played on a wrap-around board?

As discussed in Section 14.2.2, most game-playing programs have a concept of centre control which favours attacking and occupying the squares in the physical centre of the board. If the topology is changed to a wrap-around board, the importance of the physical centre is replaced by the logical centre, which contains all squares near the midline of the board. As our programs determine centrality in terms of eventual-mobility, changing board topology automatically results in changes of piece values and piece-square values.

If a player has an alternative goal to arrive her own knight at a8?

This should increase the relative value of knights, which can now fulfill an extra function (**arrival**). Exactly *how much* it should increase in value is difficult to determine in advance. It should also increase the dynamic value of a knight depending on which square it is on, as squares fewer moves from the goal should be worth more (**arrival-distance**). Interestingly, it might also increase the value of a pawn, by virtue of its ability to promote into a knight which could then achieve this new goal (**promote**). It is unclear whether pawn value should in fact increase, as the value may be predicated already on the best piece into which it could promote (i.e. the queen), rather than on a sum of values corresponding to each promotion option. In any case, pawns on the a-file should increase in value as they get closer to promotion, as promoting to a knight would immediately achieve the goal (**promote-distance**).

If knights can promote to pawns when reaching the penultimate rank? While a pawn in chess derives some value from the ability to promote directly to a queen, the knight in this context may derive similar value indirectly (**promote**). It is not obvious how to quantify such indirect value. An added complication is that the knight in effect must *demote* into the pawn before it can *promote* to a queen. In the meantime, the player's mobility would be greatly reduced.

If the opponent decides how a player's pawn is promoted? A pawn in ordinary chess has four possible promoting options (knight, bishop, rook, and queen), but in practice the queen is almost always the best option. This suggests that value derived from the possibility of promotion is related to the *maximum* of the value of the options. If the opponent, instead of the player, gets to decide which of those pieces are chosen, he will almost always choose the knight, which suggests the value in this case is the *minimum* of the values of the options. This should be taken into account when determining piece material values (**promote**) and dynamic positional values (**promote-distance**). Also, when statically evaluating a position where the opponent is about to initially-promote an enemy piece, a player could anticipate the results of this promotion (**init-promote**). If some combination of the above changes are made?

Each of the above changes in isolation suggested factors which should influence an evaluation function for symmetric chess-like games. In each case the change revealed some difficulty for extending chess-specific work to a more general class of games. When several changes are made simultaneously, it becomes increasingly difficult to determine an appropriate structure for an evaluation function. One reason for this is as follows. In the case of single changes, the new evaluation function could be seen as a change to a chess-specific evaluation function. In the case of multiple changes, the changes begin to influence each other non-locally.

A particularly difficult instance of this interaction concerns the value of pieces as they relate indirectly to the value of other pieces. For example, a type of piece gains value by its potential ability to capture other piece types, and it would seem that it should get more value based on the value of the pieces it captures. If one type of pawn could only capture rooks while the other could only capture knights, it would seem that the first type should be slightly more valuable. But the value of the captured pieces in turn may be indirectly influenced by the value of other pieces. For example, the knights may be those which can promote into pawns in one of the examples above. An appropriate method of handling this type of circular influence remains an open issue with this approach.

14.4.2 Generated Games

The method of game-variant analysis presented above can be viewed as *active* analysis. I also used a passive form of this analysis, where instead of designing hypothetical games to focus my analysis, I considered randomly generated games, observed how my programs did on them, and applied the methods discussed in this chapter to analyse the results.

14.5 Summary

This chapter has addressed the issue of knowledge-acquisition for Metagame-playing in general, and for SCL-Metagame in particular. The approach taken in this chapter has been to determine how strong a player could be designed using a *knowledge engineering* approach, without having the program use learning or analyse the semantics of the class of games itself. The intention of this was (a) to determine how far traditional methods could be applied to this new problem, and (b) to construct a competent challenger against which to test the more general methods in the future. The approach to developing a metagamer by having the human analyse the class of games was called Metagame-analysis.

This chapter discussed several techniques which were used for this analysis. The techniques can be summarised as follows:

Generalising Existing Features: Several features (heuristics) used in existing game-playing programs were generalised to this new context. This process revealed some of the assumptions implicit in these heuristics, which had to be made explicit in order to apply them to this more general problem.

Step Functions: In order to enable programs to achieve long-range planning behaviour, it was necessary to have them pre-compile tables which measure progress on goals which may not be achieved for many moves.

Game-Variant Analysis: This is an original knowledge-acquisition technique for constructing general programs to play a class of games, using a known game as a basis. We begin with a known game and an evaluation function which is effective on that game. We represent the known game as an instance of the class of games, and generalise the evaluation function slightly so that its language applies to the class instead of the specific game alone. We then consider ways in which the grammatical definition of the game could change. For a given change, we consider the behaviour of the evaluation function on the resulting game. If the evaluation function has not changed but our analysis of the new game suggests it should change, this reveals a new source of knowledge (or an assumption within current knowledge) with which to modify the evaluation function.

Generated Games: Instead of imagining game variants, we generate instances and apply the preceding methods on the instance games.

This analysis serves a case study in Metagame-analysis, which may be useful for future work on this class or different classes of games.

As the details of the specific games to be played are not known in advance, it is necessary to transfer some of the detailed game-analysis onto the program which will play unknown the games. The following chapter (Chapter 15) discusses how the results of this analysis have been incorporated into a program, METAGAMER, which produces its own game-specific evaluation function when given the rules of a new game.

Chapter 15

Metagamer

15.1 Introduction

This chapter discusses the architecture and strategic knowledge implemented METAGAMER, a sophisticated SCL-Metagame-player which embodies the results of the analysis in Chapter 14. This program takes as input the *rules* of any game in the class and analyses those rules to produce a set of game-specific analysis tables. Those tables are then used by a set of general knowledge sources which do not mention the rules of any specific game. Together these components form a game-specific evaluation function, without requiring any human contact with the rules of the specific game. This evaluation function is used by the search engine in the standard manner.

Section 15.2 discusses the architecture of METAGAMER, and is followed by sections which discuss the particular components of this architecture along with their implementation. Section 15.3 covers the general knowledge sources used by the program. Section 15.4 discusses the game-specific analysis tables produced by the program which are used by those knowledge sources. Section 15.5 addresses the important issue of setting weights for the advisors. Section 15.6 provides examples of the game-analysis performed by METAGAMER on several games discussed in this thesis. Section 15.7 summarises the chapter.

15.2 Overview of Metagamer

In order to represent knowledge in a general and flexible fashion, I have used an approach similar to that used already in Chapter 12. That is, I represent knowledge for use with the evaluation function in a game-independent form which does not refer to the details of any game.[1] When given the rules of a specific game, the program

[1] It should be remembered that, by construction of the SCL-Metagame problem, it impossible to encode knowledge which is game-specific, because the program is given the rules of specific games only before competition, at which point the human is no longer allowed to modify the program.

using this evaluation function first performs a pre-processing stage during which it specialises some of the general knowledge to take account of the details of each game it is given. In this way, some of the responsibility for game-specific analysis is transferred onto the program.

The architecture of METAGAMER is rather sophisticated, and is shown schematically in Figure 15.1 and Figure 15.2.

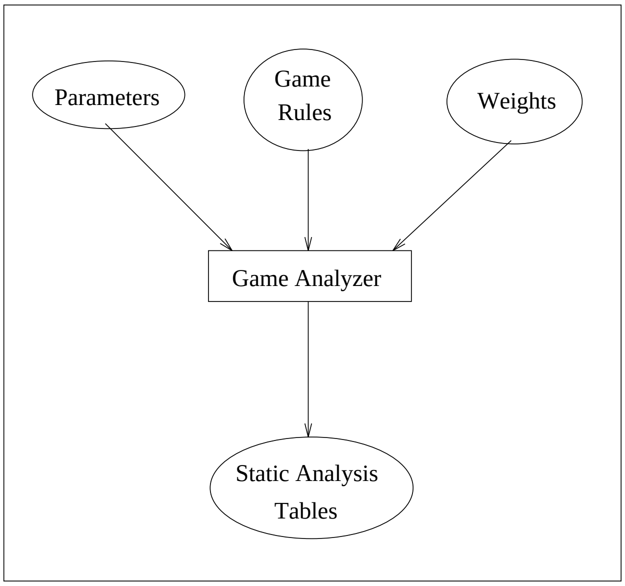


Figure 15.1: Game-Analyzer of METAGAMER.

When presented with a set of game rules, the game-analysing component of METAGAMER (Figure 15.1) constructs a set of *static analysis tables* based on the rules of that game. When later evaluating a position, while playing that game, the evaluation-function component (Figure 15.2) takes as input these static tables and the rules of the game. The program then constructs a set of *dynamic analysis tables*, based on the current position. The dynamic tables, along with the static tables, the position, and the game rules, are then used by a set of *advisors*, each of which may provide specialised *advice* to be used in evaluating the position. The whole set of advice is then passed to a *mediator*, which combines it into a global evaluation of the position.[2] This evaluation is a number, which is the estimate of the position from the perspective of the *white* player. This number is used by the search engine in the standard manner.

15.2.1 Overview of Advisors

Following the approach used in HOYLE [Epstein, 1989b], we view each component of the evaluation function as an *advisor*, which encapsulates a piece of advice about why

[2] In the current system, the mediator returns just a weighted sum of the advice, where each piece of advice is weighted according to the weight attached to the advisor who offered it.

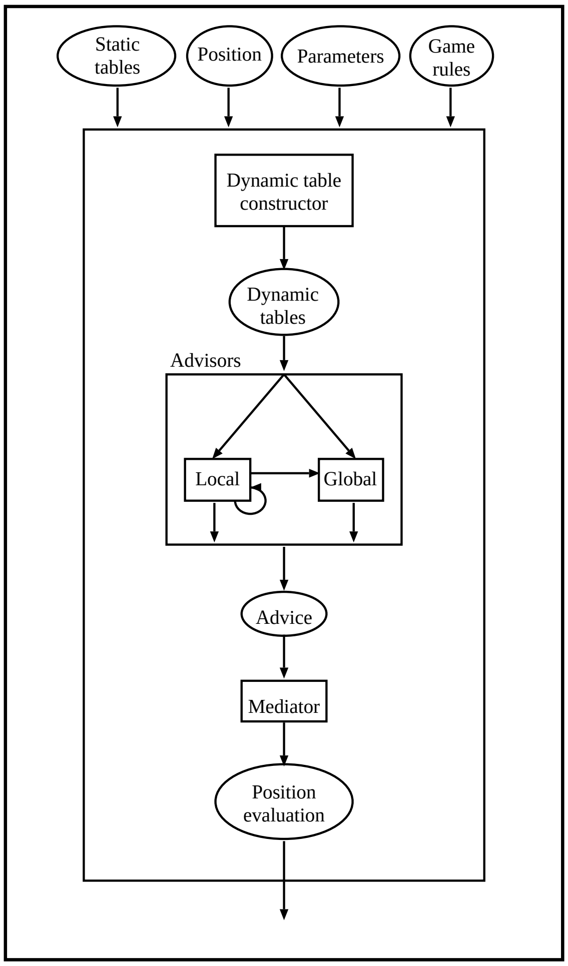


Figure 15.2: Evaluation Function of METAGAMER.

some aspect of a position may be favourable or unfavourable to one of the players. The advisors recognise aspects of a position which are favourable for one of the players, other things being equal, and express their observations in the form of numerical values (*advice*).

Local and Global Advisors There are two types of advisors: *local* and *global*. Local advisors comment on the value derived from having a particular piece on a particular square of the board, or a particular relation holding between a set of pieces. An example of a local advisor is *local-threat*, which awards points to a position in which a piece threatens to capture an enemy piece. Global advisors comment on global properties of a position, often returning the maximum of the values of a set of local advisors. The corresponding example of a global advisor is *global-threat*, which awards points to a position based on the most valuable threat by a player. As suggested by this example, the arrows between the advisors in the figure indicate that some advisors refer to other advisors in making their assessments.

15.2.2 Representation of Advisors

In terms of the representation of the advisors, I follow an approach similar to that used in Zenith [Fawcett and Utgoff, 1992], in which each advisor is defined by a non-deterministic rule for assigning additional value to a position. The total contribution (value) of the advisor is the sum of the values for each solution of the rule. This method of representation is extremely general and flexible, and facilitates the entry and modification of knowledge.

15.3 Advisors

This section briefly explains the advisors currently implemented for METAGAMER. It should be noted that this set is not final, and there are several important general heuristics which are not yet incorporated (such as *distance* and *control* [Snyder, 1993]).

The advisors can be categorised into four groups, based on the general concept from which they derive.

15.3.1 Mobility Advisors

The first group is concerned with different indicators of *mobility*. These advisors were inspired in part by [Church and Church, 1979] and [Botvinnik, 1970], and are motivated in Section 14.2 and Section 14.4.

• `dynamic-mobility`:counts the number of squares to which a piece can move directly from its current square on the current board, using a *moving* power.[3]

• `static-mobility`:a static version of immediate-mobility, this counts the number of squares to which a piece could move directly from its current square on an otherwise empty board, using a *moving* power.

• `capturing-mobility`:counts the number of captures each piece could make in the current position, regardless of the usefulness of the capture. It does not even distinguish whether the victim is a friendly or enemy piece. There is no static version of this advisor. For future work, one way to approximate the potential capturing ability of a piece might be to play random games and count the pieces attacked by each piece in each position.

• `eventual-mobility`:measures the total value of all squares to which a piece could move eventually from its current square on an otherwise empty board, using a *moving* power. The value of each square decreases (by a parameter-controlled function) with the number of moves required for the piece to get there. Thus while a bishop has 32

[3] I may elsewhere in this thesis refer to this as *immediate-mobility* or *moving-mobility*.

eventual moves and a knight has 64 from any square, the bishop can reach most of its squares more quickly, a fact captured by this advisor.

15.3.2 Threats and Capturing

The second group of advisors deals with capturing interactions (in general, threats and conditions enabling threats):

• `local-threat`:for each target piece which a piece could capture in the current position, this measures the value of executing this threat (based on the other advisors), but reduces this value based on which player is to move. Thus a threat in a position where a player is to move is almost as valuable for that player as the value derived from executing it (i.e. he can capture if he likes), but if it is the opponent's move it is less valuable, as it is less likely to happen. Thus attacking an enemy piece while leaving one's own piece attacked (if the pieces are of equal value) is a losing proposition, but if the threatened enemy piece is worth much more this may be a good idea. The value of these threats are also augmented based on the *effect* of the capture.

• `potent-threat`:this extracts from the local-threat analysis just those threats which are obviously potent. A threat is *potent* for the player on move if the target is either undefended or more valuable (based on the other advisors) than the threatening piece. A threat is potent for the non-moving player only if the attacker is less valuable than the target and the moving-player does not already have a potent threat against the attacker.

• `global-threat`:The two threat advisors above exist in both *local* and *global* versions. The local version credits a player for each threat she has in a position, while the global version credits the player with only the *maximum* of those local threat values.

• `possess`:In a position where a player has a piece *in-hand*, the player is awarded the dynamic value (using the local advisors) that the piece would receive, *averaged* over all empty board squares. Note that if the maximum value were used instead of the average, a program searching one-ply would never choose to place a piece on the board once possessed.[4]

15.3.3 Goals and Step Functions

The third group of advisors is concerned with goals and regressed goals for this class of games.

• `vital`:this measures dynamic progress by both players on their goals to eradicate some set of piece types. As a given goal becomes closer to achievement, exponentially more points are awarded to that player. In addition, if the number of such remaining

[4]This analysis would have to be improved substantially to capture the complexities of possessed pieces in games like Shogi.

pieces is below some threshold, the remaining pieces are considered *vital*, in which case any potential threats against them automatically become potent.

• `arrival-distance`:this is a decreasing function of the abstract number of moves it would take a piece to *move* (i.e. without capturing) from its current square to a goal *destination* on an otherwise empty board, where this abstract number is based on the minimum distance to the destination plus the cost/difficulty of clearing the path. This applies only to destinations for which the game has a defined arrival-goal for a piece-type consistent with the piece, and succeeds separately for each way in which this is true.

• `promote-distance`:for each *target-piece* that a piece could promote into (by the player's choice), this measures the value of achieving this promotion (using the other advisors), but reduces this value based on the difficulty of achieving this promotion, as discussed for arrival-distance above. The result is the maximum net value (to player) of the discounted promotion options, or the minimum value if the opponent gets to promote the piece instead.[5]

• `init-promote`:This applies in a position where a player is about to promote the opponent's piece (on some square) before making a normal move. For each promoting-option defined for that piece, this calculates the value that option would have on the given square in the current position. The value returned is the maximum of the values of the options, from the perspective of the promoting player (and thus the minimum from the perspective of the player who owned the piece originally).

15.3.4 Material Value

The final group of advisors are used for assigning a fixed material value to each type of piece, which is later awarded to a player for each piece of that type he owns in a given position. This value does not depend on the position of the piece or of the other pieces on the board. As a player's score from these material values does not change whenever the piece makes an ordinary move, the effect is that the program is especially sensitive to moves which change the material balance. In chess and checkers, these changes are captures (as pieces are removed from the board) and promotions (as the type of a piece changes).

One global advisor, `material`, computes a set of material values for each type of piece in a given game, after the static analysis tables used by all the other advisors have been constructed. The material value for a piece is a weighted sum of the values returned by the advisors listed in this section. Note that *white rook* and *black rook* are viewed as different types of pieces during the computations below.[6]

[5] If we take the sum instead of the maximum here, a piece with many promotion options could be more valuable than its combined options, and thus it would never be desirable to promote it. For example, a pawn on the seventh rank would sit there forever enjoying all its options, but never cashing them in.

[6] Thus if a piece gets one point for each piece it can possibly capture, and there are 5 distinct piece

- `max-static-mob`:The maximum static-mobility for this piece over all board squares.
- `avg-static-mob`:The average static-mobility for this piece over all board squares.
- `max-eventual-mob`:The maximum eventual-mobility for this piece over all board squares.
- `avg-eventual-mob`:The average eventual-mobility for this piece over all board squares.
- `eradicate`:Awards 1 point for each opponent goal to eradicate pieces which match this type, and minus one point for each player goal to eradicate our own piece matching this type.
- `victims`:Awards 1 point for each type of piece this piece has a power to capture (i.e. the number of pieces matching one of its *capture-types*). A bonus is provided for the effects of each capture, as discussed for the local-threat advisor above. It would be interesting to have a dynamic version of this which gave preference to pieces which could capture other pieces actually present in a given position.[7]
- `immunity`:Awards 1 point for each type of enemy piece that *cannot* capture this piece.
- `giveaway`:Awards 1 point for each type of friendly piece that *can* capture this piece.
- `stalemate`:This views the goal to stalemate a player as if it were a goal to eradicate all of the player's pieces, and performs the same computation as *eradicate* above.
- `arrive`:Awards a piece 1 point if the player has an arrival goal predicated on that type of piece. It awards $1/n$ points if the piece can promote to an arrival-goal piece in n promotions. Values are negated for opponent arrival goals.[8]
- `promote`:This is computed in a separate pass after all the other material values. It awards a piece a fraction of the material value (computed so far) of each piece it can promote into.

Section 15.6 provides concrete examples of the application of these advisors to the rules different games discussed in this thesis.

15.4 Static Analysis Tables

Many of the advisors draw on information which would be extremely slow to compute dynamically at each position to be evaluated. For example, it would be wasteful to determine in each position the number of squares a piece could eventually reach on an otherwise empty board. Chapter 12 showed how some wasteful computation in the primitive search components could be eliminated through a pre-compilation stage. A similar approach is used here to overcome inefficiency in the evaluation

names, it is possible to score 10 points if the piece can capture all pieces.

[7]A more sophisticated version of this feature, not fully implemented yet, takes into account the value of each victim, as determined by other static advisors.

[8]The net effect of this is that pieces which help a player to achieve arrival goals receive positive points, and those which help the opponent receive negative points.

function. To this end, the program compiles a set of tables after it receives the rules of each game, and then uses those tables thereafter. As the tables encode the results of analysis applied to a specific game, the advisors can be written in a game-independent manner. Another advantage of this game-specialisation beyond efficiency is that after receiving the game rules, the program can take advantage of the rules to fold some of the goal tests directly into the tables.

As an example, one of the most frequently used tables is the **Constrained-Transition** table. This table is used by all the static mobility advisors, and records for each piece and square, the other squares the piece could move to directly on an otherwise empty board. However, the table *excludes* all transitions from this set which can easily be shown either to be impossible or immediately losing for the player moving that piece. A transition is *impossible* if the starting square is one in which an arrival goal would already have been achieved when the piece first arrived at the square. A transition is *immediately losing* if the player would necessarily lose the game whenever his piece arrives on the destination square. While these considerations do not obviously apply to any known games, they prove highly relevant in generated games.

15.4.1 List of Tables

This section describes in more detail the actual tables constructed by the current system.

`Transition-matrix:` This contains all tuples of the form $\langle P, SqF, SqT\rangle$ such that P is a piece-type in the game, SqF and SqT are squares on the board, and P could use a *moving* power in the current position to go in one move from SqF to SqT, based on the definition of legal moves applied to the rules of this game.

`Constrained-matrix:` This removes from the transition matrix all tuples of the form $\langle P, SqF, SqT\rangle$ such that the player who owns P would lose the game if he ever had a piece on SqF. This thus gives all transitions which would ever happen without losing the game (continued-capture sequences are not relevant here, since we are looking only at *moving* moves). We also remove all triples where SqT is a losing square but not in promotion region (as those squares in promotion region may allow us to promote into a new kind of piece).

`Distance-matrix:` Based on the constrained-matrix, this matrix contains the solution of the all-pairs-shortest-paths (APSP) problem [Aho *et al.*, 1983] to determine for each piece P and pair of squares $\langle SqF, SqT\rangle$, the number of moves P would take to move from SqF to SqT. This is constructed in $O(N^3)$ time, where N is the number of squares on the board.

`Mobility-matrix`**:** This is based on the distance-matrix, and contains all quadruples $\langle P, Sq, N, C\rangle$ such that piece P could move from square Sq to C different squares, each in minimum N moves. For example, a rook in chess can move from any square to 14 different squares in one move, and to 49 other squares in two moves.

`Promote-Transition-Matrix:` Maps each piece P to the set of pieces $PieceT$ such that P could promote directly into $PieceT$, by player's choice. This table for now ignores the cases where the opponent has the choice, as this is complicated.

`Promote-Distance-Matrix:` The APSP solution to the promote-transition-matrix, analogous to the case for the distance matrix above.

`Promote-Square-Matrix:` For each $\langle Player, Piece, SqF\rangle$ triple, contains the minimum distance in moves that $Piece$ on SqF would have to travel to reach the closest square in promotion region for $Player$. This is based on the promote-distance matrix above. An extension to this table also contains the next square on this path.

`Material-Table:` This table contains the pre-computed material values used by the advisor called *material* (see Section 15.3.4).

15.4.2 Static vs. Dynamic Advisors and Tables

There are two types of advisors and tables used, depending on whether the information they use is based on the present position, or on a more abstract one.

Static advisors are based on tables built when the program is given the rules of a new game. They are constructed on the assumption of an empty board, and thus give a kind of abstraction or relaxation from a particular position.

Dynamic advisors are based on the current board, and are more expensive to compute. In principle, all of the static tables could also be based on the current position, but in practice with the current implementation this would take prohibitively long to construct (table construction for checkers takes 7 seconds on a SUN4 SPARC-10, for chess it takes 50 seconds).

Empirical studies show that both static and dynamic versions of these tables could lead to much improved performance. For example, a chess queen which has no moves in the current position (0 dynamic mobility) is still very valuable (high static mobility), and the squares a piece could reach in N moves in a blocked position (dynamic-n-mobility) may be dramatically different than those it could reach on an empty board. Certainly the shortest paths vary based on the position.

It is possible to construct these tables in parallel in short enough time to be useful for real-time search. Future work will explore this possibility further.

15.4.3 Use Of Tables

The way the various advisors make reference to these tables is straightforward given their description. The static mobility advisors simply refer directly to the corresponding tables, while the dynamic mobility advisor interprets the rules on the current position. Arrival-distance uses the distance matrix and some reference to the current board for those squares on the path. Promote-distance uses the promote-distance-matrix to determine the possible pieces a certain piece could promote into (and the number of promotions necessary to achieve this), and the promote-square matrix to determine the number of moves necessary to reach promotion region in the first place. Then the critical squares along this path are examined dynamically (i.e. with reference to the current board).[9]

The threat analysis at present determines captures dynamically, based on the current position. It is thus rather slow, almost as slow as performing another ply of search. However, as this feature measures all the threats available, it is more abstract than counting the number of enemy pieces in resulting positions. Empirical testing has shown that using the threat feature more than pays for the time necessary in terms of performance. That is, a program using threats performs better than a program which does not use threats, when given the same amount of time per move. This is thus a clear case where expensive knowledge is more valuable than cheap search, observed on a wide variety of games.[10]

15.5 Weights for Advisors

The last major issue concerning the construction of the strategic evaluation function involves assigning weights to each advisor, or more generally, developing a function for *mediation among advisors* [Epstein, 1989a]. While this issue is already difficult in the case of existing games, it is correspondingly more difficult when we move to unknown games, where we are not even assured of the presence of a strong opponent from which to learn.

15.5.1 Constraints on Weights

By the construction of some of the advisors, we can determine at least a few significant constraints on their possible weights.

[9] In the current implementation, only the final promotion square is inspected dynamically, due to efficiency considerations.

[10] Berliner [Berliner, 1984] discussed at length this issue of the tradeoff between knowledge and search in game-playing programs.

15.5.1.1 Regressed Goals are Always Fractional

First, for advisors which anticipate goal-achievement (**promote-distance** and **threats**), it seems that their weights should always be at most 1. The reason is that the value they return is some fraction of the value which would be derived if the goal they anticipate were to be achieved. If such an advisor were weighted double, for example, the value of the threat would be greater than the anticipated value of its execution, and the program would not in general choose to execute its threats. This type of constraint is commonly used when tuning weights for chess programs, as otherwise a pawn on the seventh rank may be seen as more valuable than the queen into which it could promote.[11]

15.5.1.2 Advice is Always Constructive

Second, the philosophy behind the advisors suggests that weights should never be negative. The knowledge represented in each advisor recognises properties of a position which should be valuable to a player, other things being equal (sharing the view of goals advocated in [Wellman and Doyle, 1991], see also Section 3.2.2.2). Aspects of a position which are bad for a player are designed to be recognised as aspects which are good for the opponent, and so need not be recognised separately. In this way the advisors can be viewed as being always constructive: they never tell a player that some aspect is bad for the player, but they sometimes point out that an aspect is good for the opponent.

15.5.2 Internal Consistency is Weight-Independent

It should be emphasised that the problem of finding good weights for the advisors for use in a specific game is different from the problem of finding weights for game-specific features such as material or positional piece values.[12] Since one set of weights is used to determine the values for all of the pieces in the game, this imposes an internal consistency on the resulting piece values. A concrete example will illustrate the point.

The **victims** advisor awards a piece 1 point for each type of piece it can capture, while **avg-static-mob** awards points based on the average mobility the piece has on an empty board. The raw numbers returned by these advisors are obviously on different scales, and the issue of finding weights for them on a specific games involves determining the relative impact of differences in capturing ability as compared to differences in mobility. In chess, where all pieces capture the same number of pieces,

[11]The chess-master Tartakover (quoted in [Hunvald, 1972]) said, "A threat is more powerful than its execution." But he did not mean by this that a player should prefer to have threats against pieces than to have captured them for free.

[12]These concepts have already been discussed, but I will state them here to avoid ambiguity. A material value is a value of a piece independent of any board position. A positional value is the value from having a piece on a square, independent of the rest of the board position.

the differences in weights will have no impact on the relative values of the pieces determined by the **material** advisor relying on these advisors. Moreover, for any game in which two pieces capture the same number of pieces but one has higher mobility than the other, there is *no possible assignment* of positive weights to only these two advisors that will result in the program valuing the lower-mobility piece over the higher-mobility one. As a consequence of this, it can be seen that even a random setting of weights to advisors (subject to the constraints discussed above) should still result in a competitive advantage over a player that assigned random weights directly to the game-specific features (assuming that the general encoded knowledge is useful at all).

Abramson [Abramson, 1990] found a similar result when learning weights for base-level features in chess: a program using a random assignment of weights for chess pieces (i.e. material values) performed significantly better than a program which assigned random values to positions as a whole. The meta-level knowledge in our case constrains evaluations even more than in the case of base-level features, so we should expect to gain even more advantage over using random weights.

It is of course possible that a parameter-learning system would find some advisors to be negatively correlated with success, and that negating those weights could produce improved performance. This would suggest that other things were not, in fact, equal, and should motivate a program (or its designer) to search for an advisor which recognised the aspects which become favourable for the opponent as a player increases the value of the negatively-correlated advisor. These issues were discussed more fully in Section 14.2.1 and Section 14.4.

15.5.3 Summary of Weights

One way to summarise the discussion above is with the observation that the advisors are in effect *feature-generators*, which serve to *modularise* more general knowledge. When applied to a specific game, the advisors yield values which function in the same way as the base-level features of a game-specific program. But since a small number of general advisors generate a large number of specific values, it is clear that the game-specific values produced by the advisors are much more constrained than the corresponding values in a game-specific program.[13]

But even given the constraints and modularity imposed by these advisors, it is still the case that for different games, some sets of weights may lead to performance which is dramatically stronger than other sets of weights. This will be demonstrated

[13]There is an important connection here to the subfield of linguistics called *universal grammar* [Chomsky, 1965] or more recently *principles-based parsing* [Fong, 1991]. Rather than writing detailed parsers for specific languages, this field develops general linguistic *principles*. Given a set of *parameters* which affect the principles, and a high-level grammar for the specific language, the principles are compiled to produce a parser for the specific language. A more detailed discussion of relation between the use of linguistic principles and the use of advisors is beyond the scope of this thesis.

empirically in the experiments in Chapter 16. The issue of finding weights for advisors used by METAGAMER is thus an important area for future research (see Section 19.4.1).

In the meantime, I have experimented with METAGAMER with a variety of weight settings. As will be shown in the next section and in Chapter 17, even setting all weights equally leads to reasonable performance.

15.6 Examples of Material Analysis

This chapter has discussed the advisors and analysis tables used by METAGAMER, which were motivated by the analysis discussed in Chapter 14. One important aspect of METAGAMER's game analysis, which was discussed in Section 15.3.4, is concerned with determining relative values for each type of piece in a given game. This type of analysis is called *material analysis*, and the resulting values are called *material values* or *static piece values*. This section illustrates the results of METAGAMER's material analysis when applied to some of the games discussed in this thesis. In all cases, METAGAMER took as input only the rules of the games as encoded in the text (see Appendix D).

In conducting this material analysis, METAGAMER used the material advisors shown in Table 15.1, all with equal weight of one point each.

Advisor	Weight
dynamic-mobility	1
capture-mobility	1
global-threat	1
eventual-mobility	1
promote-distance	1
eradicate	1
vital	1
material	1
victims	1
max-static-mob	1
max-eventual-mob	1
eradicate	1
stalemate	1
arrive	1
giveaway	1
immunity	1

Table 15.1: Advisor weights for material analysis examples.

15.6.1 Checkers

Table 15.2 lists material values determined by METAGAMER for the game of checkers, given only the encoding of the rules as presented in Figure 7.3, page 59. In the table, K stands for $king$, and M stands for man.

Material Analysis: checkers		
	Piece	
Advisor	K	M
max-static-mob	4	2
max-eventual-mob	6.94	3.72
avg-static-mob	3.06	1.53
avg-eventual-mob	5.19	2.64
eradicate	1	1
victims	2	2
immunity	0	0
giveaway	0	0
stalemate	1	1
arrive	0	0
Total	23.2	13.9

Table 15.2: Material value analysis for checkers.

METAGAMER concludes that a king is worth almost two men. According to expert knowledge,[14] this is a gross underestimate of the value of a man. The reason that men are undervalued here is that METAGAMER does not yet consider the static value of a piece based on its possibility to promote into other pieces (see Section 15.3.4). When actually playing a game, METAGAMER does take this into consideration, based on the dynamic **promote-distance** advisor.

15.6.2 Chess

Table 15.3 lists material values determined by METAGAMER for the game of chess, given only the encoding of the rules as presented in Appendix D.2.2, page 247. In the table, the names of the pieces are just the first letters of the piece names in the game definition.

As discussed for checkers above, pawns are here undervalued because METAGAMER does not consider their potential to promote into queens, rooks, bishops, or knights. According to its present analysis, a pawn has increasingly less eventual-mobility as

[14] I am thankful to Nick Flann for serving as a checkers expert.

Material Analysis: chess						
	Piece					
Advisor	B	K	N	P	Q	R
max-static-mob	13	8	8	1	27	14
max-eventual-mob	12	12.9	14.8	1.99	23.5	20.2
avg-static-mob	8.75	6.56	5.25	0.875	22.8	14
avg-eventual-mob	10.9	9.65	11.8	1.75	22.4	20.2
eradicate	0	1	0	0	0	0
victims	6	6	6	6	6	6
immunity	0	0	0	0	0	0
giveaway	0	0	0	0	0	0
stalemate	1	1	1	1	1	1
arrive	0	0	0	0	0	0
Total	51.7	45.1	46.9	12.6	103	75.5

Table 15.3: Material value analysis for chess.

it gets closer to the promotion rank. Beyond this, the relative value of the pieces is surprisingly close to the values used in conventional chess programs, given that the analysis was so simplistic.

15.6.3 Fairy Chess

Table 15.3 lists material values determined by METAGAMER for some fairy-chess pieces [Dickins, 1971], which are useful for comparison with the standard chess pieces. The definitions for these pieces are presented in Appendix D.2.3, page 250. In the table, D is short for $diagleap$, a piece which can move only one square diagonally. O is short for $ortholeap$, a piece that moves one square orthogonally. A chess king is thus a piece which moves as either of these two pieces. NB, NQ, and NR are short for *knight-bishop*, *knight-queen*, and *knight-rook*. These pieces have movement powers of a knight disjoined with a bishop, queen, or rook, respectively. A normal chess queen can be viewed as a bishop-rook, for comparison.

One interesting point to note from this is that METAGAMER's values are fully consistent, in the sense that giving a piece an additional moving movement (like turning a queen into a combined knight and queen) results in a corresponding increase in piece value.

Material Analysis: fairy-chess					
	Piece				
Advisor	D	NB	NQ	NR	O
max-static-mob	4	21	35	22	4
max-eventual-mob	6.94	22	25.5	22.2	7.91
avg-static-mob	3.06	14	28	19.2	3.5
avg-eventual-mob	5.19	19.7	23.8	21.6	6.26
eradicate	0	0	0	0	0
victims	5	5	5	5	5
immunity	0	0	0	0	0
giveaway	0	0	0	0	0
stalemate	1	1	1	1	1
arrive	0	0	0	0	0
Total	25.2	82.7	118	91.1	27.7

Table 15.4: Material value analysis for fairy-chess.

15.6.4 Knight-Zone Chess

Table 15.5 lists material values determined by METAGAMER for the first version of knight-zone chess with Rule 1. This game was discussed in Section 3.2.2.2 on page 18, and the definition input to METAGAMER is listed in Appendix D.2.4. The only difference between this game and chess is that the board is extended by two on all sides, and other pieces are restricted from landing on the outside squares except for the knight. This restriction is represented as a goal in which the opponent wins if a player ever puts a piece other than a knight in that zone.

It is interesting to observe that given this change to the rules of normal chess, METAGAMER concludes that a knight is now worth a small amount more than a rook. This also provides a good example of the importance of using the **constrained-matrix** when determining mobility instead of the ordinary **transition-matrix**. The transition-matrix alone would have indicated that rooks, for example, could move to every square of the board and so would have higher mobility than the knight. The constrained-matrix folds the goal conditions directly into the move transition analysis, and enables the program to determine statically that the knight is the only piece which can travel in this outer zone.

15.6.5 Turncoat Chess

Table 15.6 lists material values determined by METAGAMER for the game of turncoat chess. This game was produced by the game generator and is discussed in Section 8.3

Material Analysis: knight-zone chess						
	Piece					
Advisor	B	K	N	P	Q	R
max-static-mob	13	8	8	1	27	14
max-eventual-mob	12	12.9	23.1	1.99	23.5	20.2
avg-static-mob	3.89	2.92	6.11	0.389	10.1	6.22
avg-eventual-mob	5.42	4.84	17.3	1.33	10.5	9.56
eradicate	0	1	0	0	0	0
victims	6	6	6	6	6	6
immunity	0	0	0	0	0	0
giveaway	0	0	0	0	0	0
stalemate	1	1	1	1	1	1
arrive	-1	-1	0	-1	-1	-1
Total	40.3	35.7	61.4	10.7	77.1	56

Table 15.5: Material value analysis for knight-zone chess.

on page 66.

METAGAMER here values a firefly (F) as the best piece by far, and considers the termite (T) and slug (S) to be approximately equal in value, with the termite only slightly better. This example reveals how simplistic the current analysis is, when compared to the sophisticated strategy for this game discussed in Section 8.3.3.

15.6.6 Discussion

This section has provided some examples of the game analysis produced by METAGAMER after receiving just the rules of a set of games discussed in this thesis. In all examples, METAGAMER used the same set of weights for its advisors, and all advisors were weighted equally. Despite its simplicity, the analysis produced useful piece values for a wide variety of games, which agree qualitatively with the assessment of experts on some of these games. This illustrates that the general knowledge encoded in METAGAMER's advisors and analysis methods is an appropriate generalisation of game-specific knowledge, which was the point of the analysis presented in Chapter 14.

This appears to be the first instance of a game-playing program automatically deriving material values based on active analysis when given only the rules of different games. It also appears to be the first instance of a program deriving useful piece values for games unknown to the developer of the program. The following sections compare METAGAMER to previous work with respect to determination of feature values.

Material Analysis: turncoat-chess			
	Piece		
Advisor	F	S	T
max-static-mob	12	4	4
max-eventual-mob	10	4	4
avg-static-mob	8.96	2.4	3.2
avg-eventual-mob	8.8	2.56	3.24
eradicate	0	0	0
victims	6	6	6
immunity	0	0	0
giveaway	3	3	3
stalemate	-1	-1	-1
arrive	0	0	0
Total	47.8	21	22.4

Table 15.6: Material value analysis for turncoat-chess.

15.6.6.1 Expected Outcome

Abramson [Abramson, 1990] developed a technique for determining feature values based on predicting the *expected-outcome* of a position in which particular features (not only piece values) were present. The expected-outcome of a position is the fraction of games a player expects to win from a position if the rest of the game after that position were played randomly. He suggested that this method was an *indirect* means of measuring the mobility afforded by certain pieces. The method is statistical, computationally intensive, and requires playing out many thousands of games. On the other hand, the analysis performed by METAGAMER is a *direct* means of determining piece values, which follows from the application of general principles to the rules of a game. It took METAGAMER under one minute to derive piece values for each of the games discussed in this section, and it conducted the analysis without playing out even a single contest. This same consideration also applies to other work on *self-play*, which was discussed in Section 3.3.4.

15.6.6.2 Automatic Feature Generation

There has recently been much progress in developing programs which generate features automatically from the rules of games [de Grey, 1985; Callan and Utgoff, 1991; Fawcett and Utgoff, 1992]. When applied to chess such programs produce features which count the number of chess pieces of each type, and when applied to Othello they produce features which measure different aspects of positions which are correlated with mobility. The methods operate on any problems encoded in an extended logical representation, and are more general than the methods currently used by METAGAMER. However, these methods do not generate the *values* of these features, and instead serve as input to systems which may learn their weights from experience or through observation of expert problem-solving. While METAGAMER's analysis is specialised to the class of symmetric chess-like games, and thus less general than these other methods, it produces piece values which are immediately useful, even for a program which does not perform any learning.

15.6.6.3 Evaluation Function Learning

There has been much work on learning feature values by experience or observation (e.g. [Samuels, 1967; Lee and Mahajan, 1988; Levinson and Snyder, 1991; Callan *et al.*, 1991; Tunstall-Pedoe, 1991; van Tiggelen, 1991]). These are all examples of passive analysis, and are not of use to a program until it has had significant experience with strong players. This issue is discussed in detail in Section 3.3.

15.6.6.4 Hoyle

HOYLE [Epstein, 1989b] is a program, similar in spirit to METAGAMER, in which general knowledge is encapsulated using the metaphor of *advisors*. HOYLE has an advisor responsible for guiding the program into positions in which it has high mobility. However, HOYLE does not analyse the rules of the games it plays, and instead uses the naive notion of immediate-mobility as the number of moves available to a player in a particular position. The power of material values is that they abstract away from the mobility a piece has in a particular position, and characterise the potential options and goals which are furthered by the existence of each type of piece, whether or not these options are realised in a particular position. As HOYLE does not perform any analysis of the rules or construct analysis tables as does METAGAMER, it is unable to benefit from this important source of power.

15.7 Summary

This chapter discussed METAGAMER, a first-principles game-playing program for the class of symmetric chess-like games. The program takes in the rules of the specific game and builds a set of analysis tables specialised for that game which are used as the basis for a game-specific evaluation function. These tables determine such factors as the short-term and long-term mobility each piece has from each board square, the number of moves it takes a piece to move between any pair of squares on an empty board, and which pieces can promote into which other pieces.

When the program later plays a game, these tables are used by the features in the general evaluation function, which are called *advisors*. Thus the overall architecture of the evaluation function is divided between (a) a set of game-independent knowledge sources, and (b) a set of analysis methods which construct game-specific tables based on the rules of the game. When actually playing a specific game, the program functions as if its knowledge were specialised to just that specific game. The overall process is a form of *knowledge-compilation* [Flann, 1992], as general knowledge is specialised to a particular problem.

Section 15.5 discussed the important issue of weights for advisors. The section showed that the advisors can be viewed as *feature generators* which produce specific features with weight settings, and that any setting of weights for the advisors still imposes an internal consistency on the base-level features which result from their use with a specific game (via table construction). This means that the successful performance of a program using the advisors is not as dependent on the weights of those advisors as a game-specific program would be dependent on the weights of its game-specific features. In other words, a random setting of weights to advisors (subject to constraints discussed in the section) is still expected to perform significantly better than a random setting of weights to base-level features. A concrete example was provided which showed that all settings of weights to advisors that were consistent with the constraints discussed in that section would still result in METAGAMER preferring a piece with more mobility over one with less, other factors being equal.

However, the section also observed that a better setting of weights to advisors for a given game would still result in a competitive advantage, and that no one set of advisor weights appeared likely to be good for all games in the class. The general problem of learning advisor weights was suggested as an important area for future research.

Section 15.6 grounded the discussion of advisors, weights, and analysis by providing some examples of the game analysis produced by METAGAMER after receiving just the rules of a set of games discussed in this thesis. In all examples, METAGAMER used the same set of weights for its advisors, and all advisors were weighted equally. Despite its simplicity, the analysis produced useful piece values for a wide variety of games, which agree qualitatively with the assessment of experts on some of these games. This illustrates that the general knowledge encoded in METAGAMER's advisors and analysis methods is an appropriate generalisation of game-specific knowledge, which was the point of the analysis undertaken in Chapter 14. The section pointed out that METAGAMER appears to be the first instance of a game-playing program automatically deriving material values based on active analysis when given only the rules of different games.

The game specialiser, search engine, and class-wide strategic evaluation function together comprise METAGAMER, a first-principles SCL-Metagame-player. Chapter 16 will now examine the success of the knowledge encoded in the program within a context of competition on generated games.

Chapter 16

A Metagame Tournament

16.1 Introduction

This chapter discusses the results of an experiment in which a set of different versions of METAGAMER competed against each other and against some baseline programs on a set of games which were generated only at the time of competition. The experiment assessed the performance of METAGAMER against a set of baseline players, the competitive advantage derived from different weights for advisors, and the potential for future work on learning to provide programs with a competitive advantage relative to those which do not learn. In all cases, the results were positive and significant.

16.2 Motivation

The preceding chapter developed METAGAMER, a SCL-Metagame-playing program which plays arbitrary generated games from the class of symmetric chess-like games without human intervention. The general strategies implemented in METAGAMER were derived by generalising some of the strategies implemented in traditional specialised game-players, and also by extensive human analysis of the semantics of the class of games as a whole. As will be shown in Chapter 17, METAGAMER performs reasonably well on a variety of *known* games, which indicates that the knowledge it embodies is in some sense an effective and efficient generalisation of some strategies used in specific games. However, the real task for which METAGAMER was designed, and the task on which it must be evaluated, is SCL-Metagame: the only criterion is its performance on *unknown* games within the class of symmetric chess-like games, as generated only at the time of competition.

The ultimate test will come when other researchers construct programs to play SCL-Metagame also, at which point it will be possible to compare the programs, and the theories they embody, in a context of competition. But even in the absence of strong opponents, it is important to assess whether METAGAMER is at all successful on this extremely difficult problem:

- Does *any* setting of weights for METAGAMER actually give it a competitive advantage against the baseline players on generated games?

Although the analysis which motivated the strategies, architecture, and implementation of METAGAMER seemed sound, there are several reasons why the performance of METAGAMER might be no better (perhaps even worse) than random on new games. Among other reasons, the knowledge as implemented in METAGAMER may be *incomplete*, *incorrect*, or *inefficient*:

incomplete: The knowledge may be useful for games similar to known instance games from which it was derived, but still too incomplete to give useful guidance on games actually generated.

incorrect: The knowledge may actually be misleading on generated games.

inefficient: The knowledge may be broadly applicable and strategicly sound, but nevertheless too inefficient to be useful under strict time constraints. In other words, the time taken to apply the knowledge during problem-solving may be better spent searching more deeply instead. This problem has been called the *utility problem* [Minton, 1990], and the tradeoff between knowledge and search has been considered in depth within both the game-playing and problem-solving communities [Berliner, 1984; Callan, 1993; Fawcett and Utgoff, 1992].

Moreover, it could also be possible that the generator produced only drawn or hopelessly complicated games, in which case METAGAMER could not demonstrate an advantage even if its knowledge was correct and efficient.

On the other hand, if it turned out that some version of METAGAMER is a worthy competitor, then it is of interest to answer the following additional questions:

- Can success be attributed to just a few simple advisors (such as static mobility), or do more advisors increase performance strength?

- Are different combinations of weights for advisors more effective on different generated games?

An affirmative answer to the second question would indicate that learning could result in a competitive advantage in SCL-Metagame.

To address these questions, I conducted an experiment in the form of a SCL-Metagame-tournament.

16.3 Experimental Procedure

The procedure used in the experiment was as follows. I began by choosing settings of the *generator* which would further the goals of the experiment. I then decided on the layout of the tournament, in terms of the number of players, the number of generated games, the number of contests per game against each opponent, the maximum game-length before a draw is declared, and the time-constraints under which a given contest was played. Then I chose a set of players to play whatever games were later generated.

Once the entire format of the tournament was specified, all of these details were fixed thereafter. That is, I did not change the details of any programs or their parameters, alter any generated games in any way, or experiment with different time-limits: I ceased to play a role in the process. The games were then generated and the rules fed directly to the programs, which then played the tournament without human intervention. It should be noted here that none of the players modified themselves, either, in the course of the tournament (except for building analysis tables when presented with the games). That is, no passive learning took place.

It is important to emphasise the order of the decision made, and the extent of human involvement, as it is fundamental to the concept of SCL-Metagame in the autonomous context (see Section 4.4) that the human has no contact with the rules of the games before the programs actually play them, and that they then do so without human interaction.

The details of competition are discussed in the following sections.

16.3.1 Generator Constraints

I chose settings of the generator which would further the goals of the experiment. I set the parameters and constraints on the generator to favour medium-sized boards (5 or 6 squares per axis) and moderately complex rules, and added a constraint which rejected games having only one goal, of the form *stalemate-player*. The reason for this is that my programs are not in general capable of achieving these goals, nor are the random players, so games which had this as the only goal would always result in draws and so provide no useful information.[1] The resulting games produced by the generator (i.e. those games actually used in the experiments in this chapter) are listed in Appendix D.1.

16.3.2 Players

I selected a set of six SCL-Metagame-players, which consisted of two baseline players and four versions of METAGAMER with different weight settings. The baseline players

[1]Recall that the full definition of the generator is always available to the humans in advance of the competition. No claims of generality are made beyond the class as constrained by the particular generator known in advance of the competition.

were *random* and *random-aggressive-defensive* (RAD), as defined in Chapter 13. The different versions of METAGAMER are shown in Table 16.1.

Tournament Advisor Weights				
	Player			
Advisor	3	4	5	6
dynamic-mobility	0	0	0	1
capturing-mobility	0	0	0	1
global-threat	0	0	0	1
vital	0	0	0	1
eventual-mobility	0	1	1	1
possess	0	0	1	1
arrive-distance	1	0	1	1
promote-distance	1	0	1	1
init-promote	0	0	1	1
material	0	1	1	1
max-static-mob	0	1	1	1
max-eventual-mob	0	1	1	1
eradicate	0	1	1	1
victims	0	1	1	1
immunity	0	1	1	1
giveaway	0	1	1	1
stalemate	0	1	1	1
arrive	0	1	1	1

Table 16.1: Advisor weights for versions of METAGAMER in tournament.

The players can be summarised as follows:

1. Random
2. Random-aggressive-defensive (RAD)
3. Arrival and promotion only
4. Eventual-mobility and material value[2]
5. Player 3 + Player 4 + possession and init-promotion

[2]Recall from Section 15.3.4 (page 138) that the *material* advisor relies on a set of advisors. As seen from Table 16.1, all players using material had the same values for all the advisors upon which it depends.

6. Player 5 + dynamic mobility, capturing-mobility, global-threats, and vital.

In the above description, the notation *Player A + Player B* is shorthand to indicate that the player uses all the advisors with non-zero weights used by Player A or Player B.

For a full discussion about the function of the basic players, see Section 13.2.1. For a full discussion about these advisors and their motivation, see Section 15.3 and Chapter 14, respectively. Note that not all advisors were used in the tournament, and also that advisors with weight 0 are not evaluated and thus incur no cost in the evaluation function. Thus Player 6 takes longer to evaluate each position than does Player 5, and as a consequence gets to examine fewer positions.

16.3.3 Tournament Format

Each of the programs played each other program in a match of 20 contests (10 as white, 10 as black) on each of the 5 generated games. With 15 pairings, 5 generated games, and 20 contests per game, there were thus a total of 1500 contests. Each contest was declared a draw if 200 moves were played without a goal achieved, and programs were given 30 seconds (0.5 minutes) to make each move. The programs were not constrained as to game-time-limit or tournament-time-limit (see Section 13.3.4). Thus the maximum time for running the entire tournament (if every game was a draw) was $1500 \times 200 \times 0.5 = 150,000$ minutes, or 2500 hours of cpu-time.

The experiments were divided among 50 machines (DEC-Station 2500's) running in parallel, giving a maximum total time of 50 hours per machine. In practice the time was on average 25 hours per machine since at least half the games ended in a player winning.

A note on time-limits Although the versions of METAGAMER were constrained to a *move-time-limit* of 30 seconds per move, the random players were not constrained. The *random* player responds instantly, but RAD sometimes exceeded the time-limit dramatically, taking in some cases 5 minutes to move. This is because some positions in the generated games had branching factors of several hundred, and RAD may do a full two-ply search in order find an otherwise random move which does not allow a winning response. As all versions of METAGAMER were restricted in their time, it can be seen that defeating RAD may be a formidable challenge, especially in positions in which the branching factors are high enough that the players do not have time even to evaluate all first-ply options.

16.3.4 Significance Testing

All results discussed in this experiment have been tested for significance using a one-sided t-test for comparing two samples from binomial populations [Huntsberger

Game 1							
	Score vs. Opponent						Total
Player	1	2	3	4	5	6	Score
1	—	7.0	8.5	5.5	6.5	5.0	32.5
2	13.0	—	11.5	7.5	7.0	3.5	42.5
3	11.5	8.5	—	7.0	5.5	7.5	40.0
4	14.5	12.5	13.0	—	8.5	7.5	56.0
5	13.5	13.0	14.5	11.5	—	7.5	60.0
6	15.0	16.5	12.5	12.5	12.5	—	69.0

Table 16.2: Results of tournament on Game 1.

and Billingsley, 1981, pages 321-322]. Unless indicated otherwise, the results are significant at the .01 (1%) level. In using this test, draws have been absorbed into the total outcomes such that two draws are considered equivalent to one win and one loss (this absorption was also used by [Abramson, 1990]).

16.4 Results

The results of the competition on each game are presented in Table 16.2–Table 16.6. Table 16.7 summarises the results of the competition between each pair of players summed across all 5 games. Table 16.8 summarises the results of each player on each game. For each entry in the tables, players were awarded 1 point for a win, 0.5 points for a draw, and 0 points for a loss on each contest. As a match between two players on a game consisted of 20 games, the maximum score per match was 20 points. As each player played 500 contests, the maximum possible score over the whole tournament was 500 points, and the minimum possible score was 0 points. For presentation purposes, the tables list only the total score, but full tables indicating the breakdown between wins and draws can be found in Appendix E.

16.5 Discussion

The results of the experiment provide direct answers to the questions in which we were interested.

Does *any* setting of weights for METAGAMER actually give it a competitive advantage against the baseline players on generated games? As the overall results of the tournament across all games show (Table 16.7), all players performed

Game 2							
Player	Score vs. Opponent						Total Score
	1	2	3	4	5	6	
1	—	0.5	0.0	0.0	0.0	1.0	1.5
2	19.5	—	11.0	15.0	7.0	8.5	61.0
3	20.0	9.0	—	15.0	11.0	16.0	71.0
4	20.0	5.0	5.0	—	7.0	7.0	44.0
5	20.0	13.0	9.0	13.0	—	10.0	65.0
6	19.0	11.5	4.0	13.0	10.0	—	57.5

Table 16.3: Results of tournament on Game 2.

Game 3							
Player	Score vs. Opponent						Total Score
	1	2	3	4	5	6	
1	—	0.0	0.0	0.0	0.0	0.0	0.0
2	20.0	—	12.0	9.5	4.0	4.5	50.0
3	20.0	8.0	—	17.0	2.5	3.5	51.0
4	20.0	10.5	3.0	—	8.0	0.5	42.0
5	20.0	16.0	17.5	12.0	—	8.5	74.0
6	20.0	15.5	16.5	19.5	11.5	—	83.0

Table 16.4: Results of tournament on Game 3.

Game 4							
Player	Score vs. Opponent						Total Score
	1	2	3	4	5	6	
1	—	0.0	1.5	1.0	0.0	0.0	2.5
2	20.0	—	16.0	11.0	10.5	6.5	64.0
3	18.5	4.0	—	5.0	3.0	3.0	33.5
4	19.0	9.0	15.0	—	18.0	6.5	67.5
5	20.0	9.5	17.0	2.0	—	6.5	55.0
6	20.0	13.5	17.0	13.5	13.5	—	77.5

Table 16.5: Results of tournament on Game 4.

Game 5							
	Score vs. Opponent						Total
Player	1	2	3	4	5	6	Score
1	—	0.0	0.0	0.0	0.5	0.5	1.0
2	20.0	—	7.5	4.0	4.0	5.0	40.5
3	20.0	12.5	—	4.0	5.5	4.0	46.0
4	20.0	16.0	16.0	—	11.5	10.0	73.5
5	19.5	16.0	14.5	8.5	—	9.0	67.5
6	19.5	15.0	16.0	10.0	11.0	—	71.5

Table 16.6: Results of tournament on Game 5.

Overall Results: by Opponent							
	Score vs. Opponent						Total
Player	1	2	3	4	5	6	Score
1	—	7.5	10.0	6.5	7.0	6.5	37.5
2	92.5	—	58.0	47.0	32.5	28.0	258.0
3	90.0	42.0	—	48.0	27.5	34.0	241.5
4	93.5	53.0	52.0	—	53.0	31.5	283.0
5	93.0	67.5	72.5	47.0	—	41.5	321.5
6	93.5	72.0	66.0	68.5	58.5	—	358.5

Table 16.7: Overall results of tournament against each opponent.

Overall Results: by Game						
	Score on Game					Total
Player	1	2	3	4	5	Score
1	32.5	1.5	0.0	2.5	1.0	37.5
2	42.5	61.0	50.0	64.0	40.5	258.0
3	40.0	71.0	51.0	33.5	46.0	241.5
4	56.0	44.0	42.0	67.5	73.5	283.0
5	60.0	65.0	74.0	55.0	67.5	321.5
6	69.0	57.5	83.0	77.5	71.5	358.5

Table 16.8: Overall results of tournament on each game.

significantly better than the random player (Player 1). More importantly, all versions of METAGAMER except for Player 3 also scored significantly better than RAD. As discussed in Section 13.2.1 and restated in Section 16.3.3 above, this is actually a non-trivial accomplishment, as the only way to win against RAD is to achieve a position in which all moves it makes are losing.

One possible reservation to the above interpretation of the results might be that it was the search engine used by the versions of METAGAMER that gave them an advantage, rather than their knowledge. However, the results also indicate that this was not the case. For two of the five games (Game 1 and Game 4), Player 3 played using essentially a random evaluation function. This is because the only advisors it used (*promote-distance* and *arrival-distance*) were inapplicable since these games have no arrival goals. The fact that the other players defeated Player 3 so decisively on these games is thus evidence that the knowledge in their advisors is at least superior to a random evaluation function.[3]

Can success be attributed to just a few simple advisors (such as static mobility), or do more advisors increase performance strength? The results of the tournament demonstrate convincingly that additional advisors provide competitive advantage in terms of total score over the whole tournament. Player 4, Player 5, and Player 6 differ only in the successive addition of more advisors, and each addition accounts for a significant improvement in overall tournament score (see Table 16.8). The player with all advisors active, Player 6, won the overall tournament convincingly and had the highest score on 3 of the 5 games (Game 1, Game 3 and Game 4, all significant at the 0.1 (10%) level). These results demonstrate that, in terms of *general performance* summed over all games in the tournament, additional advisors pay for their cost in computation and provide useful guidance to players using them.

Are different combinations of weights for advisors more effective on different generated games? While additional advisors resulted in significant improvement overall, no one combination of weights (in this case no one set of active advisors) performed best on each game individually. The most striking example of this is in Game 2 (Table 16.3), in which the overall champion (Player 6) performed significantly worse than Player 3, both in terms of score over all opponents (.025 level) and in head-to-head competition (the score was 16-4 in favour of Player 3, which is significant at the .0001 level).

The explanation for this particular result is interesting. Game 2 (see Appendix D.1) has an arrival goal which can be achieved relatively quickly unless the opponent takes precautions. The game also has a high branching factor, which meant that the versions of METAGAMER did not manage to search much of the second ply (if they even got that

[3]Note that the search engine in these experiments used a *random* move-ordering (see Section 13.3), so that this behaviour cannot be attributed to a bad default move-ordering either.

far). As Player 3 focussed only on arrival goals (these were the only advisors it had), every move made progress toward this goal, while Player 6 deliberated about each position, examining and valuing mobility and threats as well as arrival goals. While Player 6's broad-generality paid off overall in the tournament, in this particular case Player 3's single-mindedness gave it a marked advantage.

Could learning produce a competitive advantage in SCL-Metagame? This above result provides concrete evidence that learning could indeed result in competitive advantage. If Player 6 had been able to vary its combination of strategies across different games (for example, to play like Player 3 on Game 2 and like Player 4 on Game 5, its overall score might have been much higher. This statement must remain only suggestive, however, because the entire structure of the tournament might have been different if players learned while playing. At the very least, though, if Player 6 had played like RAD against Player 3 on Game 2, it might have increased its score by 7 points. This also suggests a competitive improvement could be obtained by learning which strategies to use against specific opponents in addition to learning which strategies to use on specific games.

Game Generator The fact that different versions of METAGAMER were the best on different games also attests to the variety of games produced by the generator. If all games had been largely similar in some sense, we would have expected the same results on each game. On the contrary, the rank ordering for each game is qualitatively different.

16.6 Summary

This chapter has assessed the performance of METAGAMER against a set of baseline players, the competitive advantage derived from different weights for METAGAMER's general knowledge, and the potential for future work on learning to provide programs with a competitive advantage relative to those which do not learn. These issues were investigated by means of an experiment which took the form of a *Metagame-tournament*. In the experiment, several versions of METAGAMER with different settings of weights for their advisors played against each other and against baseline players on a set of generated games which were *unknown* to the human designer in advance of the competition. The rules were provided directly to the programs, and they played the games without further human intervention. The main results of the experiment can be summarised as follows:

- Several versions of METAGAMER performed significantly better than the baseline players across a set of generated games.

- The version of METAGAMER which made use of the most knowledge clearly outperformed all opponents in terms of total score on the tournament. This is true in spite of the added evaluation cost incurred when using more knowledge.

- No single version of METAGAMER (as defined by weight settings) performed the best on each individual game.

These results support the following main conclusions:

- The knowledge implemented in METAGAMER provides it with efficient and effective guidance across a variety of generated games unknown to its designer in advance of the competition.

- METAGAMER represents the state-of-the-art in SCL-Metagame and is a useful starting point against which to test future developments in this field.

- Future programs that incorporate learning and more sophisticated active-analysis techniques will have a demonstrable competitive advantage on this new problem.

At a meta-level, these results also attest to the success of the problem itself:

- SCL-Metagame is a new and challenging research problem which can be practically addressed. It fulfills its desiderata as a concrete instantiation of the idea of Metagame.

Chapter 17

Examples on Known Games

> There were other human players in the Culture who could beat him – though they were all specialists at the game, not general game-players as he was – but not one of them could guarantee a win, and they were few and far between, probably only ten in the whole population.
>
> – Ian Banks, *The Player of Games* ([Banks, 1988])

17.1 Introduction

One of the advantages of playing Metagame with a class containing known games is that we can assess the extent to which the general theories implemented in our programs are sufficient to explain the development of strategies for games with which we are familiar. If the program does not play known games well, we gain an understanding of what may be general limitations and directions for improvement. But unlike in traditional game-playing on known games, in Metagame we do not interpret strong performance on known games as evidence of more general ability, although we can take such performance as encouragement.

This chapter provides examples of METAGAMER playing the known games of chess and checkers against human and computer opponents. Section 17.2 discusses the manner in which the examples should be interpreted. With that motivation, Section 17.3 then comments on games played against a specialised checkers program, and Section 17.4 provides examples of METAGAMER playing chess against a specialised chess program and against a human novice. Section 17.5 summarises the chapter.

17.2 Using Known Games

As pointed out in Section 6.2.1 (page 43), playing Metagame in a class which contains known games has several advantages. It is helpful to quote directly from that section:

> Existing games have a known standard of performance and existing bodies of theory. These can be used to assess the extent to which a metagamer's game-analysis yields competitive performance on specific games. If a metagamer is unable to derive important strategies or is otherwise weak on a known game, this reveals areas for future research. Note that the opposite of this does not necessarily hold. That is, a metagamer that plays a set of known games well is not by virtue of that to be evaluated as a strong Metagame-player. As the games are known in advance, it cannot be proven that this strength does not derive from human analysis of those specific games.

This is the spirit in which the examples to follow should be interpreted. Examples in which METAGAMER plays well using its own analysis are encouraging, but should not be taken as any indication whatsoever of generality beyond these specific games. Chapter 14 has already discussed how the knowledge implemented in METAGAMER was derived by attempting to generalise strategies and organisation used in known games, and it is likely that METAGAMER is already in some sense optimised toward exactly these known games as a consequence of this (whether or not this optimisation was performed consciously). On the other hand, examples which reveal weaknesses of METAGAMER in competition against strong opponents can be useful. If these weaknesses can be corrected in a general way, it might help to improve the performance on *unknown* games which share some structure with the known games under consideration.

With that said, the following sections provide anecdotal evidence regarding the performance of METAGAMER on some known games. I have provided my own analysis of the games, and have attempted to point out some of the more indicative strengths and weaknesses of METAGAMER. Ultimately, the examples should enable the readers to draw their own conclusions about METAGAMER's performance on known games, and these examples are not intended to establish any claim to the success of METAGAMER on the tasks for which it was designed. Extensive experiments which *do* establish such a claim have been discussed already in Chapter 16.

17.3 Checkers

When playing checkers, the version of METAGAMER used was Player 6, the program that won the Metagame tournament in Chapter 16 (see Section 16.3.2, page 155). METAGAMER played after being given the rules of checkers as listed in Figure 7.3, and after it analysed the rules for 30 seconds (on a SUN4 SPARC-10). The program then played the game with a move-time-limit of one minute.

17.3.1 One Man Handicap against Chinook

The following game is a typical instance of a small set of handicap games played between METAGAMER and Chinook [Schaeffer *et al.*, 1991]. Chinook is the world's strongest computer checkers player, and the second strongest checkers player in general.

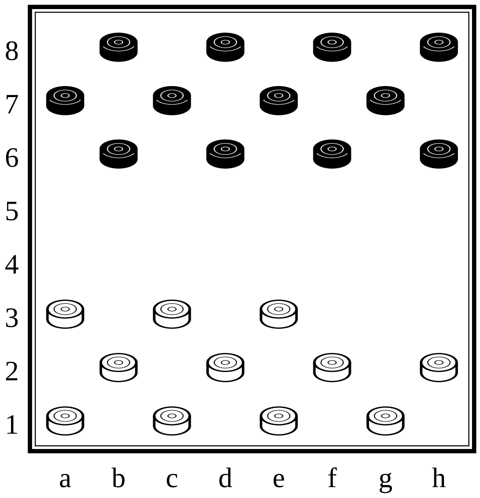
As it is a highly optimised and specialised program, it is not surprising that METAGAMER always loses to it (on checkers, of course!) when playing an even game, even when Chinook plays on its easiest level and responds virtually instantly, and without its opening or endgame database. However, to get a baseline for METAGAMER's performance relative to other possible programs when playing against Chinook, I have evaluated the programs when given various piece handicaps (number of men taken from Chinook at the start of the game), in order to determine the size of the handicap necessary to draw with Chinook on its easiest level. The results were that METAGAMER competes evenly with the weakest possible Chinook (without opening or endgame databases, searching 5-ply, and responding almost instantly) when given a handicap of one man. This is compared to a deep-searching greedy material program (i.e. METAGAMER's search engine using the minimal evaluation function with only the feature for general material difference, as discussed in Section 13.2.2) which requires a handicap of 4 men, and to the random player, which requires a handicap of 8 men.

These experiments were by no means thorough, and are only provided to indicate that drawing with Chinook with a one-man handicap even on its easiest level is by no means easy.[1]

In the following game, Chinook plays white, without one man. METAGAMER plays well to maintain the material advantage throughout the game, eventually reaching a won endgame with two kings against king. Unfortunately, METAGAMER is unable to win the endgame because it lacks a strategy to force Chinook's king out of its double-corner hiding place, so the game ends in a draw. I am not a checkers expert, but I did not find a serious mistake in any move played by METAGAMER throughout the game.

[1]Nick Flann [personal communication,1993] has pointed out that giving up one man in checkers is a significant handicap which, between players of roughly equal standards, would always result in a loss for the player offering the handicap.

Diagram 1 *Initial Position.*



1.	a3-b4	d6-c5

METAGAMER calculates that this move helps it to centralise its pieces quickly and increase mobility.

2.	b4-d6	c7-e5
3.	h2-g3	e7-d6
4.	c3-d4	e5-c3
5.	b2-d4	

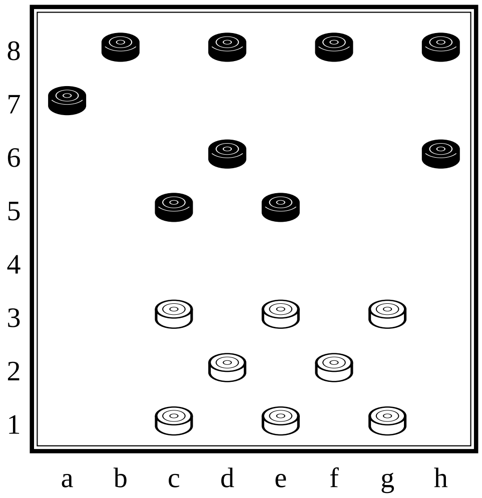
Chinook follows a similar strategy.

5.	...	f6-e5
6.	d4-f6	g7-e5
7.	a1-b2	b6-c5

METAGAMER has so far chosen not to move any of its back-men. This is a well-known strategy in checkers, as moving the back men makes it easier for the opponent to promote his pieces. This behaviour is typical when it plays checkers. It turned out that this strategy emerged from the promote-distance advisor, operating defensively instead of in its "intended" offensive function to encourage a player to promote pieces when the destination squares are vacant. In effect, METAGAMER realised that by moving its back men, it made the promotion square more accessible to the opponent, thus increasing the opponent's value, and decreasing its own.

8. b2-c3

Diagram 2 *After 8. b2-c3.*



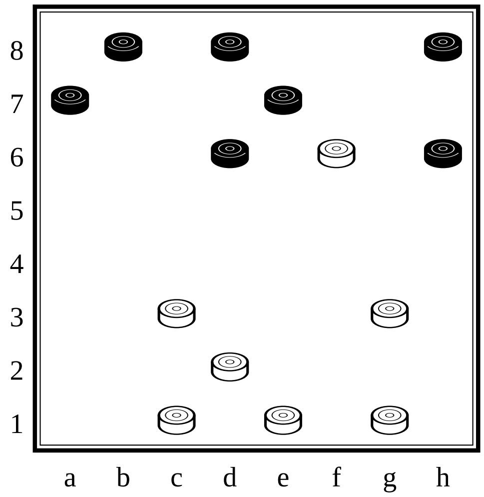
White sets a trap, as 9.e3–d4, c5–e3; 10.f2–d4–f6, threatens to win a man unless Black can recapture (if 11. . . ., f8–g7; 12.d2–e3, g7–e5; 13.e3–d4 wins the piece).

8.	...	f8-e7

One of the few defences. METAGAMER searched to a depth of 4-ply to find this move, before which it preferred 8. . . ., a7–b6. A trace of METAGAMER as it made this move is presented in Appendix C.1 It is also interesting that METAGAMER was forced to move one of its back men, which it would have resisted doing otherwise.

9.	e3-d4	c5-e3
10.	f2-d4-f6	

Diagram 3 *After 10. f2-d4-f6.*



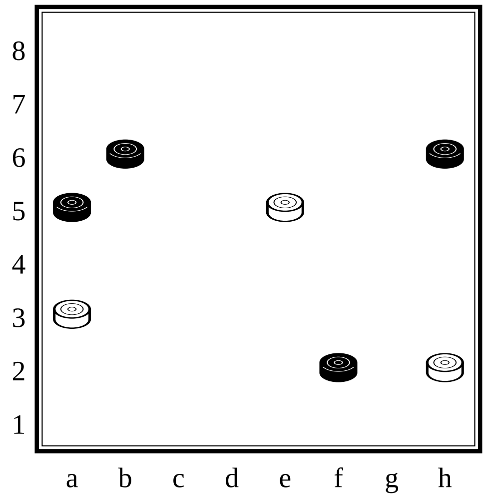
This was the position evaluated at 4-ply above. METAGAMER knew it would not be a piece down here, as its threat analysis revealed that it would win a piece on the next move.

10.	...	e7-g5
11.	g3-f4	g5-e3
12.	d2-f4	a7-b6
13.	e1-d2	h8-g7

Now METAGAMER moves the corner man from the back row, as (a) it estimates that the opponent could promote on f8 anyway (the promotion advisor at present focusses only on the final square, not on the accessibility to it), and (b) it determines that the increase in mobility is worth the cost. The idea of moving the corner piece but keeping the other back men in place is a refinement on the naive strategy of not moving back men, and is used by checkers experts.[2]

14.	g1-h2	b6-c5
15.	d2-e3	g7-f6
16.	c1-b2	f6-g5
17.	b2-a3	b8-c7
18.	c3-d4	g5-h4
19.	d4-b6	c7-a5
20.	e3-d4	h4-g3
21.	d4-e5	g3-f2
22.	e5-c7	d8-b6
23.	f4-e5	

Diagram 4 *After 23. f4-e5.*



Guided by its promote-distance advisor, METAGAMER has brought a man to the point of promotion. Chinook will be able to promote soon also.

23.	...	h6-g5

Instead of promoting the man on f2, METAGAMER prefers to bring the man at h6 closer to promotion. It is searching into the sixth ply here.

24.	h2-g3	g5-f4
25.	g3-h4	b6-c5
26.	e5-d6	f4-e3
27.	d6-c7	e3-d2

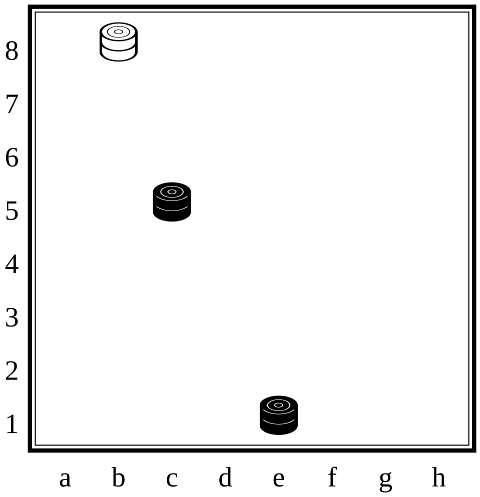
Having brought the other man close to promotion also, METAGAMER finally promotes into a king.

28.	c7-b8(k)	f2-e1(k)
29.	h4-g5	e1-f2
30.	b8-c7	f2-e3

[2]I am grateful to Nick Flann for this observation.

31.	c7-b6	c5-d4
32.	g5-f6	d4-c3
33.	b6-c5	c3-b2
34.	c5-d6	e3-d4
35.	d6-e5	d4-c5
36.	f6-g7	a5-b4
37.	g7-f8(k)	b4-c3
38.	e5-d6	c5-e7
39.	f8-d6	d2-e1(k)

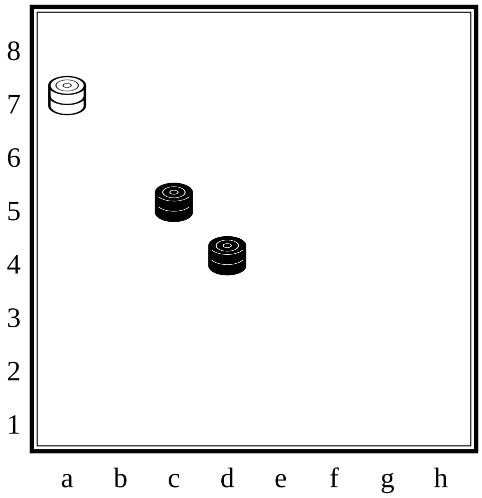
Diagram 5 *After 51. . . . , d2-e1(k).*



After promoting this king, METAGAMER will now centralise it, while Chinook hides in the double-corner.

40.	a3-b4	e1-f2
41.	b4-a5	c3-d2
42.	a5-b6	f2-e3
43.	d6-e5	b2-c1(k)
44.	b6-c7	c1-b2
45.	c7-b8(k)	b2-c3
46.	b8-a7	e3-d4
47.	e5-d6	d4-c5
48.	d6-b4	c3-a5
49.	a7-b8	a5-b6
50.	b8-a7	b6-c5
51.	a7-b8	d2-e1(k)
52.	b8-a7	e1-d2
53.	a7-b8	c5-d6
54.	b8-a7	d2-e3
55.	a7-b8	e3-d4
56.	b8-a7	d4-e3
57.	a7-b8	e3-d4
58.	b8-a7	d4-e5
59.	a7-b8	d6-c5
60.	b8-a7	c5-d6
61.	a7-b8	e5-d4
62.	b8-a7	d6-c5

Diagram 6 *After 62. . . . , d6-c5.*



METAGAMER has maximised its own mobility while confining Chinook's king to the corner, but does not know how to win this won endgame. Thus the game ended in a draw.

17.3.2 Discussion of Checkers Games

In this game METAGAMER appeared to play well, and used several well-known checkers strategies which all followed from its application of the general advisors to the specific rules of checkers. The significant advisors in this game were material, dynamic-mobility, global threat analysis, and promote-distance. One obvious limitation of METAGAMER revealed in this game is that it cannot win in the final position. This is actually a difficult position for checkers programs in general, and many have evaluation functions specifically tuned to guide it to win positions like this. In fact, the version of Chinook used in the experiment (without opening or endgame databases) required 15-ply of search to find the winning sequence.

It would be an interesting area for future work to develop a general reasoning method which could be applied to checkers and similar games to solve problems like this. One possibility would be a generalisation of work on using knowledge and plans to control search [Wilkins, 1982; Campbell, 1988; Tadepalli, 1989a]. Another possibility is to add an advisor to METAGAMER which is able to construct simple endgame databases, perhaps generating first those which are simplest and most common. A general method for achieving precisely this task was developed by Flann [Flann, 1992], and has been applied to construct databases for chess and checkers end-games using only a logical representation of the rules for those games.

17.4 Chess

This section discusses the results of two games of chess played by METAGAMER when given as input only the rules of the game. The encoding given to the program is listed in Appendix D.2.2 (page 247). The first game is against a strong specialised program, and the second is against a human novice. In both games, METAGAMER played with 60 seconds per move on a SUN4 SPARC-10. It played with the same settings of advisors as Player 6, the best overall player in the experiments in Chapter 16 (see Section 16.3.2, page 155), with two exceptions: promote-distance was set to 0, and capture-mobility was set to 5. The full set of advisors with their weights is shown in Table 17.1.

These changes in setting resulted from my own experience observing METAGAMER play chess, and are thus the result of a small amount of hand-tuning of the program to improve its performance on this game. As indicated in Section 15.5, a major area of future work is to have the program optimise its own weights for each game, and it seems straightforward to implement the same kind of tuning as I performed in order to develop this behaviour automatically.

It may be of interest to describe briefly the reasons I changed these two weights. First, capture-mobility was set to 5 points per attack because when it was set to 1 the program preferred not to attack the enemy, in order to maximise the dynamic-mobility of its pieces. Setting it to 5 instead was my first try at correcting this, and yielded much stronger performance. Second, using promote-distance on chess increased the time to evaluate each position by a significant amount. This is because my current implementation of this advisor considers, for each piece which can promote, the dynamic value (using the rest of the advisors, as discussed in Section 15.3) of each possible promoting option on the final square in the promotion-region. In chess, this meant that the program evaluated the potential of each pawn on the board to promote

Advisor	Weight
dynamic-mobility	1
capturing-mobility	5
global-threat	1
vital	1
eventual-mobility	1
possess	1
arrive-distance	1
promote-distance	0
material	1
max-static-mob	1
max-eventual-mob	1
eradicate	1
victims	1
immunity	1
giveaway	1
stalemate	1
arrive	1

Table 17.1: Advisor weights in METAGAMER chess games

into four different pieces, so even in the initial position it was in effect considering the dynamic value of 64 additional pieces. Although the resulting knowledge was useful, the cost of evaluating it dramatically reduced the total number of positions METAGAMER could consider.

After making these two small changes, it will be seen in the following chess games that the resulting set of advisors provide METAGAMER with a reasonable degree of chess skill, when given as input just the rules of the game.

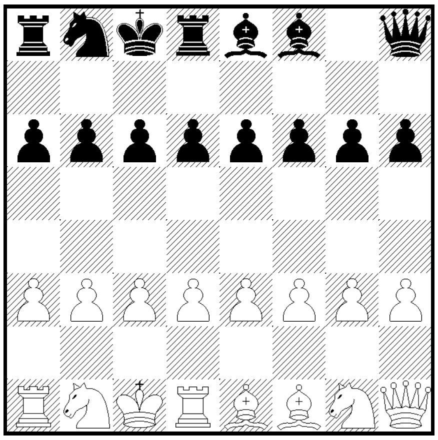
17.4.1 Knight's Handicap against GnuChess

The following game is a typical instance of a small set of handicap games played between METAGAMER and GnuChess. GnuChess is a strong publicly available chess program, winner of the C Language division of the 1992 uniform Platform Chess Competition. GnuChess is vastly superior to METAGAMER (at chess, of course!), unless it is handicapped severely in time and moderately in material.

In this game, GnuChess (black) played on level 1 with depth 1. This means it searches 1-ply in general but can still search deeply in non-quiescent positions. METAGAMER (white) played with the settings and time-control already described.

The initial position for this game, shown in Figure 7, was generated by randomly permuting the pieces on White's first rank, generating Black's setup by symmetry, and removing a random knight from Black's position. The pawns are placed on the third rank instead of the second to ensure that the game is played without double pawn-moves or the *en-passant* rule, as these rules are absent from the encoding of chess as a symmetric chess-like game (see Appendix D.2.2). The position was permuted randomly because this was one of a small set of matches between the two players at various handicaps.

Diagram 7 *Initial Position.*



1. d3–d4

Opening a diagonal for the ♗f1, attacking a6, and increasing the mobility of ♖d1.

1. ... d6–d5

Black seems to have the same idea.

2. f3–f4

Opening a diagonal for the ♕h1, attacking d5, and giving a good square on which to develop the ♘g1.

2. ... ♗f8–d6

3. ♖a1–a2

Increasing the mobility of the rook.

3. ... ♘b8–d7

4. ♗f1–d3

Developing the bishop, and attacking g6.

4. ... ♗e8–f7

5. a3–a4 ♔c8–b7

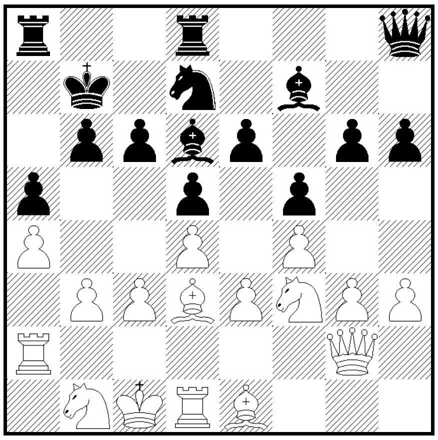
6. ♕h1–g2 a6–a5

7. ♘g1–f3

Developing the knight to a good square.

7. ... f6–f5

Diagram 8 *After 10.f5.*



Black's last move creates a weakness on e5.

8. ♘f3–e5

METAGAMER plants its knight on this weak square. Although this move was played without considering all responses at the second ply, METAGAMER's potent threat advisor determines that the knight is safe on that square, in that any one piece that can capture it could be recaptured by a pawn. In addition to being a central location for the knight with high eventual mobility, from this square the knight attacks four pieces, giving a high capture-mobility.

8. … ♘d7×e5
9. f4×e5 ♗d6–e7
10. ♕g2–f3

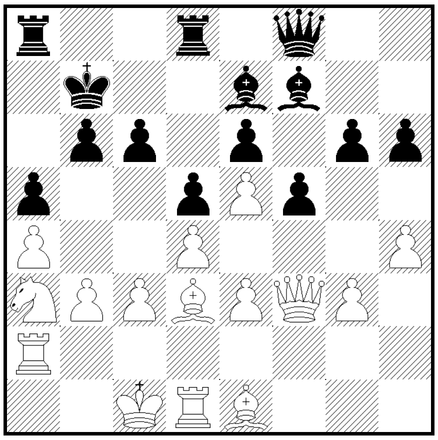
This move clears White's second rank for the ♖a2. It also increases White's capture-mobility by adding another attacker to f5, although that square is securely under Black's control.

10. … ♗e7–g5
11. ♘b1–a3 ♕h8–f8
12. h3–h4

Forcing Black's bishop to retreat from its aggressive location.

12. … ♗g5–e7

Diagram 9 *After 12. …, ♗e7.*



Black now threatens to capture the ♘a3.

13. ♘a3–c2

After an iteration of search at ply 1, METAGAMER's best choice move (13.h5), did not defend the knight, as its potent threat analysis indicated incorrectly that the knight was already defended. It *was* defended against any *single* capture (as the rook can recapture), but the threat analysis does not continue beyond that point. METAGAMER then reconsidered the search at ply 2, starting with the preferred variation from the previous iteration, 13.h5, at which point it saw the response 13. …, ♗×a3. At that point the threat analysis revealed (now correctly) that Black would have won a piece (14.♖×a3, ♕×a3). Eight other 1-ply moves were then considered in random order (out of a total 44 choices), the best of which was 13.♘c2, before METAGAMER ran out of time. It is important to note that as only 3 out of 44 moves for White in this position avoid the loss of a piece, and METAGAMER was in effect sampling a random 9 moves in the time it had left (as the principal continuation was not good), METAGAMER got somewhat lucky to have found a non-losing move. This also explains why, when faced with the same choice two moves later, METAGAMER makes a different decision.[3]

13. … ♕f8–g7
14. ♘c2–a3 ♕g7–f8
15. ♘a3–b1

This time METAGAMER makes a different choice, as discussed above.

15. … ♕f8–g7
16. c3–c4

A good move. METAGAMER increases the mobility of its ♗e1 and clears the square c3 for its ♘b1.

16. … d5×c4
17. ♗d3×c4

[3]In general, if there are M options, of which g are good, and a random N-subset are considered, the probability of considering a good choice is $(M-g)!(M-N)!/(M-g-N)!M!$. In the present context $(M=44, g=3, N=9)$, this means METAGAMER had a 49.4% chance of even *considering* a saving move at ply-2.

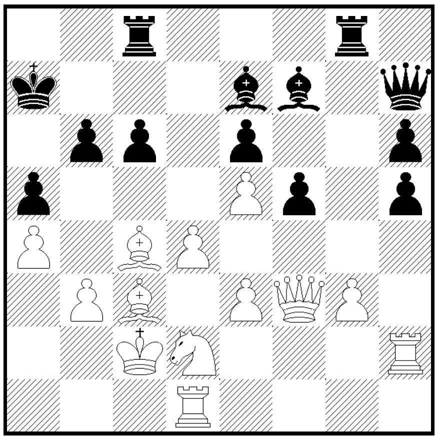
METAGAMER recaptures with the ♗d3 instead of the pawn as this puts pressure on Black's pawn at e6.

17.	...	♖a8–c8
18.	♗e1–c3	♔b7–a7
19.	♖a2–h2	♕g7–h7
20.	h4–h5 ?!	

If 18. ..., g×h5, METAGAMER views the position as favourable because it will have two pieces attacking the pawn at h5, thus giving 10 points of capture-mobility, and giving up the −5 points Black received for attacking the pawn when it was on h4. The net difference of 15 points is worth more than the pawn. This does not seem like a good move, though, and probably reflects an over-estimate of the value of attacking defended pawns.

20.	...	g6×h5
21.	♔c1–c2	♖d8–g8
22.	♘b1–d2	

Diagram 10 *After 21.♘d2.*



22.	...	♖g8–g4

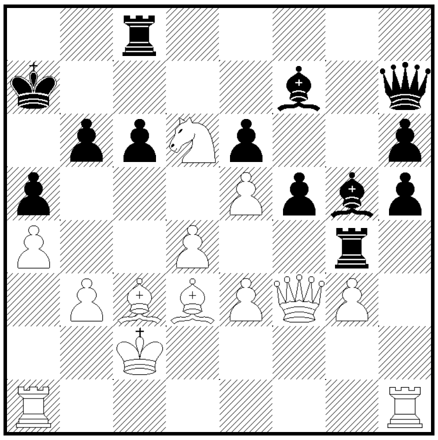
On 21. ..., ♖×g3; 22.♕×g3, f4+; 23.♗d3, f×g3; 24.♗×h7, g×h2 does not lead to anything. It easily might have, though, and METAGAMER could benefit from having discovered attacks added to its current threat analysis to anticipate such possibilities.

23.	♖d1–e1	♗e7–g5
24.	♖e1–a1	♕h7–h8
25.	♖h2–h1	♕h8–f8
26.	♗c4–d3	

Clearing c4 for the ♘d2.

26.	...	♕f8–h8
27.	♘d2–c4	♕h8–h7
28.	♘c4–d6	

Diagram 11 *After 24.♘d6.*



METAGAMER, as White, has achieved a dominating position. All its pieces are well-placed on active squares, with Black rendered passive due to White's strong centre.

28.	...	♖c8–g8 ?

A strange choice for the rook, as it leaves c6 undefended.

29.	♔c2–d2	

Sadly, METAGAMER does not seize the opportunity to ply 28.♕×c6, which threatens 29.♕b7 mate immediately and forces a win within a few moves (e.g. 28. ..., ♗g6; 29.♘b5+, ♔a6; 30.♗×a5 (threatening 31.♕×a6 mate), ♖b8 (if 30. ...,

♔×a5; 31.b4+, ♔a6; 32.a5, followed by 33.a×b6 mate); 31.♗×b6, ♖×b6; 32.♘c7 dbl+, ♔a7; 33.♕a8 mate). Instead, METAGAMER prefers to keep g3 defended and further centralise its king. (Remember that METAGAMER is searching only a little more than one ply here).

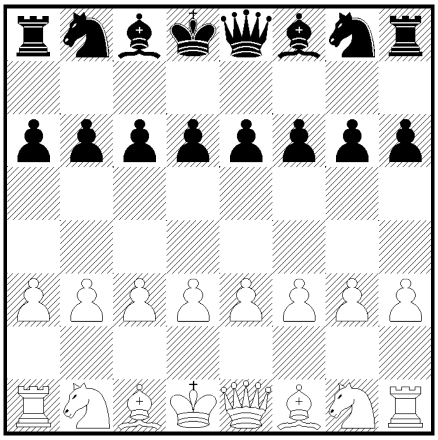
29.	...	♗f7–g6
30.	♔d2–c2	♗g6–f7
31.	♔c2–d2	♗f7–g6
32.	♔d2–c2	♗g6–f7
33.	♔c2–d2	♗f7–g6
34.	♔d2–c2	♗g6–f7

GnuChess sees nothing better to do than wait. But since METAGAMER sees neither the winning line above beginning ♕×c6, nor other any way to improve its position on the next move, both players just move back and forth. Thus the game is drawn by repetition.

17.4.2 Even Match against Human Novice

In the following game, METAGAMER played Black with the same settings as in Section 17.4. White was played by a human novice (unrated) chess-player. The players played two contests, one as each colour, and METAGAMER won both games. As discussed in Section 17.4, the pawns are placed on the third rank instead of the second to ensure that the game is played without double pawn-moves or en-passant, as these rules are absent from the encoding of Chess as a symmetric chess-like game (see Appendix B).[4]

Diagram 12 *Initial Position.*



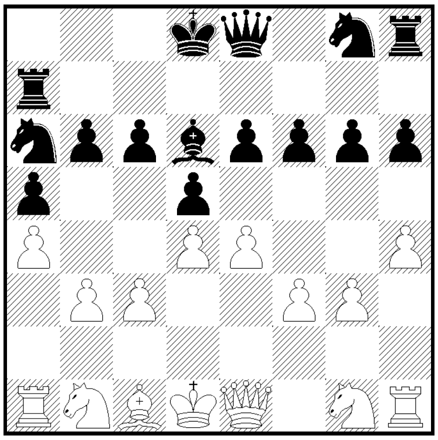
1.	h3–h4	d6–d5
2.	d3–d4	♗f8–d6
3.	a3–a4	♖a8–a7

METAGAMER likes to increase the mobility of the rook, just as in the game against Gnuchess.

4.	♗f1–d3	a6–a5
5.	e3–e4	♗c8–a6
6.	♗d3×a6	♘b8×a6

[4] The initial position also has king and queen starting on the wrong squares, to remind the novice that castling was not an option in the current encoding of chess as a symmetric chess-like game (see Appendix D.2.2).

Diagram 13 *After 6. . . ., ♘×a6.*



So far the game is pretty ordinary. White now advances in the centre.

7. e4–e5 f6×e5

8. d4×e5 ♗d6–c5

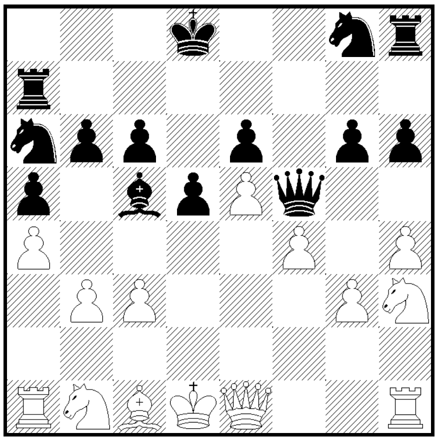
METAGAMER places the bishop on its best square.

9. ♘g1–h3 ♕e8–f7

Developing the queen and attacking f3.

10. f3–f4 ♕f7–f5

Diagram 14 *After 10. . . ., ♕f5.*



METAGAMER has established a strong position and has a marked advantage in development.

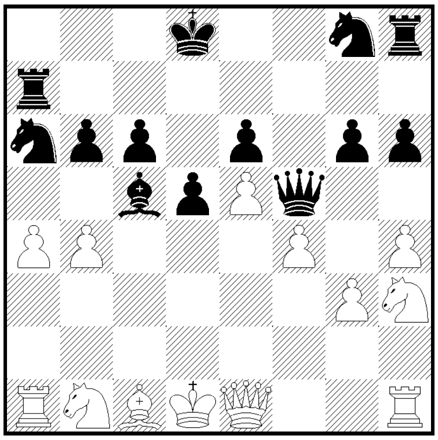
11. b3–b4 ?

This is a bad move which loses a pawn. It is funny because it is the kind of move METAGAMER sometimes plays using limited threat analysis: before the sequence of captures the pawn is twice defended.

11. . . . a5×b4

12. c3×b4

Diagram 15 *After 12.cd.*



Black can now win the b pawn.

12. . . . ♗c5–d4

Instead of winning the pawn immediately, METAGAMER attacks ♖a1. It calculates that the pawn is still hanging after the forced response 13.♘c3, as moving the ♖a1 loses to 13. . . ., ♕×b1.

13. ♘b1–c3

Forced.

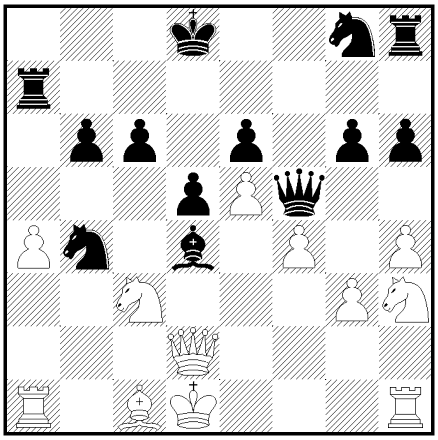
13. . . . ♘a6×b4 ?

Now Black captures the pawn. 13. . . ., ♕d3+ instead would have won a piece (if 14.♕d2, ♕×c3, and if 14.♗d2, ♗×c3 wins as the ♗d2 is pinned). METAGAMER was unable to search this deeply.

14. ♕e1–d2

White does not give Black a second chance to play ♕d3+.

Diagram 16 *After 15. Qd2*



14. ... ♘b4–d3 ?

Black penetrates with the knight. This misses 13. ..., ♕g4+; 14.♔e1, ♕xg3+ followed by 15. ..., ♕×c3, winning a piece. Deeper search would really be helpful here!

15. ♖h1–f1 ?

This allows Black to play 16. ..., ♕×h3; 17.♕×d3, ♗×c3; which wins a piece as 18.♕×c3, ♕xf1+.

15. ... ♗d4×c3

METAGAMER thinks that after the forced response 16.♕×c3 the ♘h3 is still hanging (it is, but only because Black can play 16. ..., ♘×c1 first; 16. ..., ♕×h3 allows 17.♕×d3, and it is not looking that far ahead).

16. ♕d2×c3

Forced.

16. ... ♘g8–e7 ?

Missing the line discussed above. METAGAMER only examined 7 other choices at the second iteration (out of 42 total possibilities); if it had searched the entire ply it would have seen that 16. ..., ♘×c1 wins the piece. Still, this move is not bad, as it develops another piece.

17. ♖f1–g1 ?

White is not searching very deeply either, so it seems like a fairly even match! According to White, this move prepares 17.g4, attacking ♕f5 and blocking the attack on ♘h3. 16.♗a3 here would have avoided the chance to lose a piece.

17. ... ♔d8–d7 ?

Still missing the win of a piece. This time METAGAMER examined 10 other choices, out of 40 possible. The move it played brings the king toward the centre and increases the mobility of the ♖h8.

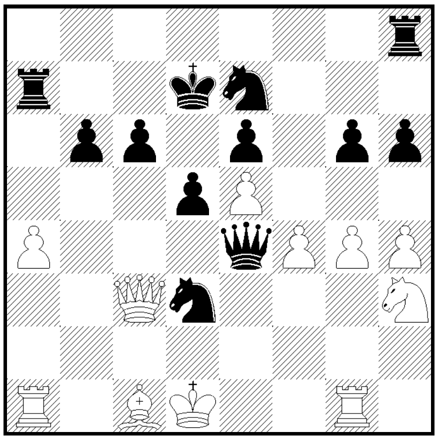
18. g3–g4

White carries out his plan and at last defends the ♘h3.

18. ... ♕f5–e4

Forced, as any other move loses the ♘d3.

Diagram 17 *After 18. ..., ♕e4.*



19. ♗c1–a3 ?

This loses the pawn at a4, but creates complications. White is starting to get active now.

19. ... ♖a7×a4
20. ♗a3×e7 ♔d7×e7
21. ♔d1–d2

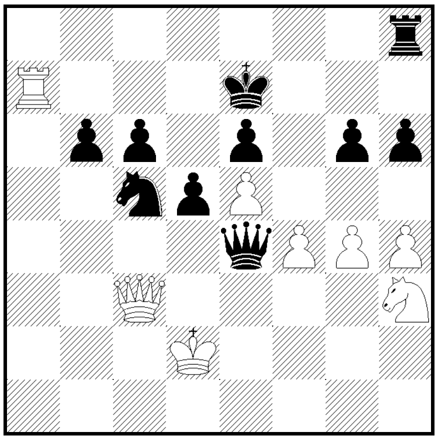
Connecting rooks, and threatening to win the ♘d3. White's position is improving.

21. ... ♖a4×a1
22. ♖g1×a1 ♘d3–c5

Retreating the threatened knight and blocking White's attack on c6.

23. ♖a1–a7 +

Diagram 18 *After 23.♖a7+.*



A strong move, putting Black on the defensive.

23. ... ♔e7–d8 ?

As METAGAMER is still only searching part-way into the second iteration, it is unable to see that this move loses a rook. To see this would require searching well into the third ply, even with METAGAMER's potent-threat analysis. Although METAGAMER prefers to keep its king centralised, it does not like 23. ..., ♘d7 at 1-ply because it leaves the c6 pawn hanging.

24. ♖a7–a8

White doesn't miss the opportunity.

24. ... ♔d8–e7
25. ♖a8×h8

White has now won a rook. His king is somewhat exposed, but he should be able to win from this position.

25. ... h6–h5

METAGAMER thinks this defends the h-pawn.

26. ♘h3–f2 ?

White tries to bring the knight into the action and attack Black's queen, but there were better ways to do this. This loses at least a pawn.

26. ... ♕e4×f4 +

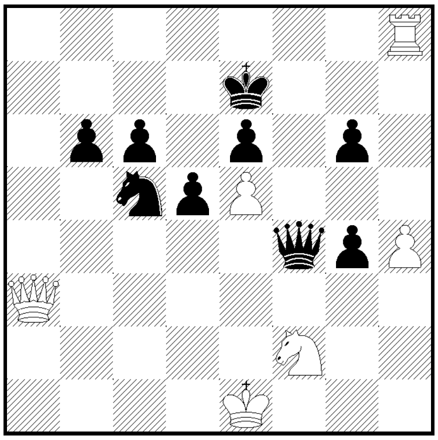
Black takes the free pawn with check. Although White is still up a rook, with these players the game can still go either way.

27. ♔d2–e1 h5×g4

Black misses 27. ..., ♕×f2; 28.♔×f2, ♘e4+, which wins back a piece. The move played is still reasonable.

28. ♕c3–a3 ?

Diagram 19 *After 28.♕a3.*



White prepares 29.♕a7+, which would win the game, but he misses something ...

28. ... ♕f4×e5 +

Black forks White's king and rook, winning back the rook!

29. ♘f2–e4 ?!

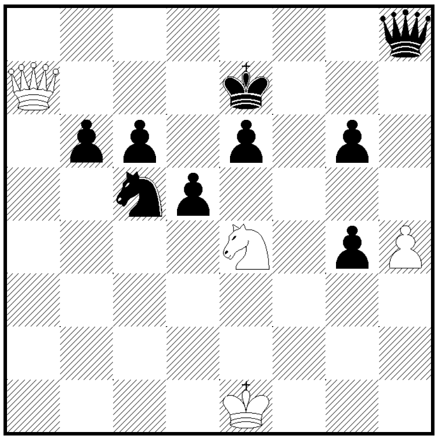
An interesting move. White explained that he was trying to distract METAGAMER from taking his rook.

29. ... ♕e5×h8

Black is not distracted, and now captures the rook.

30. ♕a3–a7 +

Diagram 20 *After 30.♕a7+.*



White hopes Black will play 30 ...,.♔d8 or ♔e8, after which 31.♕a8+ would win Black's queen in the same way as White won Black's rook after the position in Diagram 18.

30. ... ♘c5–d7

This time METAGAMER defends with the knight. Whereas in Diagram 18 defending with the knight left a pawn attacked, this time it even defends the b6 pawn.

31. ♕a7–a3

White just wants to keep checking.

31. ... ♔e7–f7

31. ..., c5 is much better, but METAGAMER sees that move as increasing White's capture-mobility by 2, as the c-pawn would be attacked both by ♕a3 and ♘e4. This shows that the current use of capture-mobility is naive.

32. ♘e4–d6 + ♔f7–e7

This allows a discovered double check, but white has nothing to threaten on the next move.

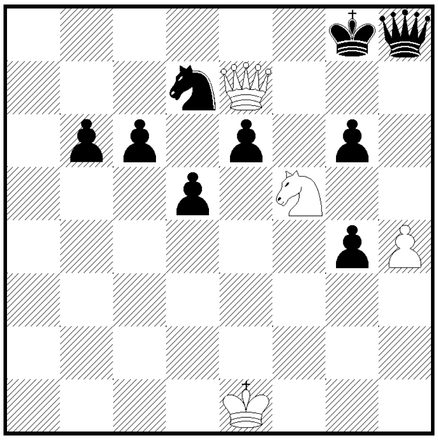
33. ♘d6–f5 + ♔e7–f7 ?

33. ..., ♔f6 seems safer. Now White has a very dangerous attack.

34. ♕a3–e7 + ♔f7–g8

Black's response is forced.

Diagram 21 *After 34. ..., ♔g8.*



35.♕xe6, winning the pawn with check, now wins a piece (35. ..., ♔f8; 36.♕d6+, ♔g8; 37.♕×g6+, ♔f8; 38.♕d6+ ♔g8; 39.♕×d7).

35. ♕e7–e8 +?

White misses the sequence above, and Black gets a chance to defend.

35. ... ♘d7–f8

This is the right move. METAGAMER rejects 35. ..., ♔h7 at the first ply because it sees the ♘d7 hanging. It should also be noted that METAGAMER could move its king into check given the encoding of chess as a symmetric chess-like game (see Appendix D.2.2). The reason it never does, even when searching only 1-ply, is because the **vital** advisor detects the possible loss of a piece which would achieve an *eradicate* goal for the opponent.

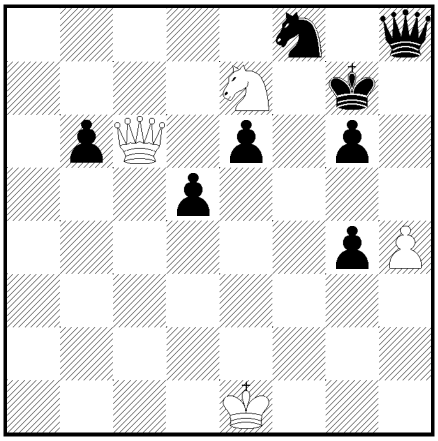
36. ♘f5–e7 + ♔g8–g7

METAGAMER preferred this move over 36. . . ., ♔h7 because it keeps its king more centralised and does not cut off the mobility (and attack on h4) of the ♕h8. White's attack has now been repulsed.

37. ♕e8×c6 ?

White grabs a pawn, but this loses a piece. In any case, White cannot defend the pawn at h4, he will be down 6 pawns, and his king is totally exposed.

Diagram 22 *After 37.♕×c6.*



37. . . . ♕h8×h4 +

Black captures a pawn with check, forking and winning the ♘e7.

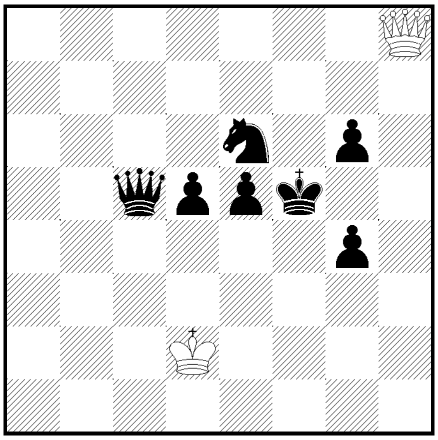
38. ♔e1–d2 ♕h4×e7

Black captures the knight. White's position is now hopeless. He tries to attack black but quickly gets overwhelmed. METAGAMER is now searching into the third-ply, as the position is now simplified.

39. ♕c6×b6 e6–e5
40. ♕b6–b3 ♕e7–c5
41. ♕b3–b7 ♔g7–f6
42. ♕b7–b8 ♘f8–e6
43. ♕b8–h8 ♔f6–f5

White resigned. METAGAMER as Black has now centralised its king and will checkmate shortly.

Diagram 23 *Final Position.*



17.4.3 Games against Chess Material Function

For purposes of comparison, a version of METAGAMER with only a standard hand-encoded material evaluation function (queen=9, rook=5, bishop=3.25, knight=3, and pawn=1) [Botvinnik, 1970; Abramson, 1990] played against the versions of METAGAMER and GnuChess used in the example games here. The result was that the material program lost every game at knight's handicap against GnuChess, and lost every game at even material against METAGAMER. This showed again that METAGAMER's performance in the example games illustrated here was not due to its search abilities, but rather to the knowledge in its evaluation function.

17.4.4 Discussion of Chess Games

These games have demonstrated the applicability of METAGAMER's general advisors to the known game of chess. We have seen the program develop its pieces quickly, place them on active central squares, put pressure on enemy pieces, make favourable exchanges while avoiding bad ones, and restrict the freedom of its opponent. In all, it is clear that METAGAMER's knowledge gives it a reasonable *positional* sense and enables it to achieve longer-term strategic goals while searching only one or two-ply deep. This is actually quite impressive, given that none of the knowledge encoded in METAGAMER's advisors or static analyser makes reference to any properties specific to the game of chess—METAGAMER worked out its own set of material values for each of the pieces (see Section 15.6, page 145), and its own concept of the value of each piece on each square.

On the other hand, the most obvious immediate limitation of METAGAMER revealed in these games is a weakness in *tactics* caused in part by an inability to search more deeply within the time constraints, by a lack of quiescence search, and also by the reliance on full-width tree-search. One way to address this problem might be through a tactics analyzer as developed by Berliner [Berliner, 1974] or a knowledge-based planner as developed by Wilkins [Wilkins, 1982]. These would need to be extended to games beyond just chess.

To summarise METAGAMER's performance in chess, it's play is reasonable and it looks like it has a basic understanding of some chess strategy. With its current inability to search more deeply, it performs similarly to a human novice. It seems likely that a small improvement in search ability (either through improved efficiency, quiescence search, or more sophisticated search techniques) would enable it to compete evenly with GnuChess on its easiest level.

17.5 Summary

One of the advantages of playing Metagame with a class containing known games is that we can assess the extent to which the general theories implemented in our programs are sufficient to explain the development of strategies for these games with which we are familiar. This chapter has discussed the performance of different versions of METAGAMER on the known games of chess and checkers. Three example matches were analysed in detail. Against specialised programs in chess and checkers, METAGAMER played reasonably well to draw when given a significant handicap in time and material. Against a human novice in chess, METAGAMER won both of two games (one of which was discussed), but in general the players appeared evenly matched.

The discussion of the example games illustrated some of the strengths and weaknesses of METAGAMER on these games, which can be summarised as follows:

- METAGAMER has a reasonable positional sense for both games. It uses some strategies which are familiar to players of those games and which are hard-wired in many game-specific programs.
- METAGAMER is weak in tactics compared to the specialised programs, and also weak in deductive problem-solving and planning.

In drawing conclusions from the discussion on known games in this chapter, two important points should be noted. First, there is nothing unusual about a program playing chess or checkers if it is given an evaluation function for each of those games. What is unusual here is that METAGAMER has no knowledge specific to any one of these known games (no mention of kings, pawns, or men), and plays these games with some degree of success given just the rules (and an alteration of two weights in the case of chess). In reading through the games, it is easy to assume that METAGAMER is a specialised checkers-player or a specialised chess-player, whereas in fact it is essentially the same program (exactly the same in the case of checkers) which was the winner of the Metagame-tournament in Chapter 16. This appears to be the first case of a program taking in the rules of two games as complicated as chess and checkers and playing both of them with some degree of skill.

The only other program which can play multiple games like this at all is HOYLE [Epstein, 1989b], but there is no evidence to date that HOYLE's advisors give it any useful guidance on these games. Moreover, HOYLE is designed to learn from a strong opponent, whereas METAGAMER plays these games after analysing the rules, without relying on any help from a human or a good opponent.

All of this notwithstanding, the second important point to note is more fundamental, and has been made already in Section 17.2 and throughout Part I of the thesis: any apparent success of METAGAMER on these known games is actually *no* evidence of general ability beyond those specific games. As I knew the games in advance, it is possible (and even likely) that I modified the program consciously or unconsciously until it played exactly these known games reasonably well. This is the methodological problem identified in part I, and this is the problem which is solved by the new paradigm of Metagame. The performance of METAGAMER on these known games has been encouraging and has indicated several areas for improvement, but the real evidence of general ability over the class of games was presented in Chapter 16, in which METAGAMER with all its advisors significantly outperformed a set of opponents across a set of games unknown to its programmer in advance of the competition.

Chapter 18

Summary of Part III

This part of the thesis has discussed both the general issues involved in the construction of Metagame-playing programs and the specific directions which have been taken in the development and implementation of the first programs to address this new problem.

Chapter 12 dealt with one of the most immediate and pressing issues in designing more general problem-solvers: the tradeoff between the representational goals of generality and flexibility, on the one hand, and the operational goals of specialisation and efficiency on the other. While these goals seemed incompatible at first, the chapter showed that it was possible to achieve them both to some extent, by shifting some of the work of building special-purpose programs, normally the task of the human researcher, onto the program itself. This was achieved by first developing a naive game player which was *extremely* general, flexible, and inefficient, and then applying techniques from logic-programming to automatically transform this program into a more efficient specialised player of a particular game.

Chapter 13 discussed how the basic game-playing components produced by the game-specialiser developed in Chapter 12 can be put together to construct a variety of basic Metagame-playing programs using only game-tree-search and minimal evaluation functions. The chapter pointed out that several techniques used in conventional game-tree-search relied on detailed knowledge of specific games, and suggested that applying these ideas to SCL-Metagame will require transferring some of the responsibility for finding appropriate search strategies onto the programs themselves. Experiments with the search engine on a variety of generated games indicated that deep search using a naive evaluation function was not enough for strong performance across this class of games.

Motivated by the need for a more sophisticated evaluation function, Chapter 14 addressed the issue of knowledge-acquisition for Metagame-playing in general, and for SCL-Metagame in particular. The approach taken in that chapter was to analyse by hand the semantics of the class of games in order to discover general strategies which could be used by a playing program to construct its own specialised evaluation function when given the rules of a game. This approach was called *Metagame-analysis*. The chapter found several techniques to be useful in this analysis. These included (a) generalising existing features used in game-specific programs, (b) searching for knowledge in the form of step-functions which would guide a program to achieve long-term goals while using only one-ply of search, and (c) game-variant analysis, in which we represent a known game as an instance of the class of games and then systematically make rule changes to discover the class-specific structure upon which the strategies for known games may operate.

Chapter 15 then discussed how the results of the analysis in Chapter 14 were embodied in a program, called METAGAMER. This program takes as input the *rules* of a specific game and a set of parameter settings (or *weights*), and analyses the rules to construct its own evaluation function for that game. The analysis performed by the program relates a set of general knowledge sources (called *advisors*) to the details of the particular game in a manner which can be viewed as a form of knowledge-compilation. Among other properties, METAGAMER's analysis determines the relative value of the different pieces in a given game, and for each piece the relative value of placing it on different board squares. Although METAGAMER does not learn from experience, the values resulting from its analysis are qualitatively similar to values used by experts on known games, and are sufficient to produce competitive performance the first time the program actually plays each game it is given. This appears to be the first time a program has derived useful piece values directly from analysis of the rules of different games.

This was possible because the process of combining the general knowledge with the specific game-analysis imposes internal consistency on the values of the base-level features, independent of the weights assigned to the general knowledge itself. While traditional specialised chess programs must be told, for example, the relative value of bishops and knights, METAGAMER instead must be told, for example, the relative value of immediate and long-range mobility. From this high-level tradeoff, the values for specific pieces follow from first-principles. Of course, programs which make better tradeoffs in terms of the general knowledge may have a competitive advantage over those with worse priorities on different games, so determining or learning these high-level tradeoffs was suggested as an important area for future research.

Chapter 16 then assessed the performance of METAGAMER against a set of baseline players, the competitive advantage derived from different weights for METAGAMER's general knowledge, and the potential for future work on learning to provide programs with a competitive advantage relative to those which do not learn. These issues were investigated by means of an experiment which took the form of a *Metagame-tournament*. In the experiment, different versions of METAGAMER played against each other and against baseline players on a set of generated games which were *unknown* to the human designer in advance of the competition. The rules were provided directly to the programs, and they played the games without further human intervention.

The significant results of the experiment were that (a) several versions of META-

GAMER performed significantly better than the baseline players across a set of generated games, (b) the version of METAGAMER which made use of the most knowledge clearly outperformed all opponents in terms of total score on the tournament, and (c) no single version of METAGAMER (as defined by weight settings) performed the best on each individual game. The main conclusions derived from these results were that (a) the knowledge implemented in METAGAMER is useful on games unknown to its programmer in advance of the competition, (b) METAGAMER represents the state-of-the-art in SCL-Metagame and is a useful starting point against which to test future developments in this field, and (c) future programs that incorporate learning and more sophisticated active-analysis techniques will have a demonstrable competitive advantage on this new problem.

While the experiments in Chapter 16 established METAGAMER's success on the task for which it was designed (playing new games), Chapter 17 examined the performance of METAGAMER on the known games of chess and checkers, when playing against humans and specialised programs. One of the advantages of playing Metagame with a class containing known games is that we can assess the extent to which the general theories implemented in our programs are sufficient to explain the development of strategies for games with which we are familiar. If the program does not play known games well, we gain an understanding of what may be general limitations and directions for improvement. But unlike in traditional game-playing on known games, in Metagame we do not interpret strong performance on known games as evidence of more general ability, although we can take such performance as encouragement.

With this difference noted, the discussion of the example games in the chapter illustrated some of the strengths and weaknesses of METAGAMER on these games. A main strength was that METAGAMER has a reasonable positional sense for both games. It uses some strategies which are familiar to players of those games and which are hard-wired in many game-specific programs. However, METAGAMER is weak in tactics compared to the specialised programs, and also weak in deductive problem-solving and planning. These are therefore promising areas for future work. The discussion of known games concluded with an important caveat about drawing conclusions from known games. The performance of METAGAMER on these games looks general and is encouraging, but any apparent success of METAGAMER on these known games is actually *no* evidence of general ability beyond those specific games. As the games were known in advance, it is possible (and even likely) that METAGAMER was modified consciously or unconsciously until it played exactly these known games reasonably well. This is the methodological problem identified in part I, and this is the problem which is solved by the new paradigm of Metagame. The performance of METAGAMER on these known games has been encouraging and has indicated several areas for improvement, but the real evidence of general ability over the class of games was presented in Chapter 16, in which METAGAMER with all its advisors significantly outperformed a set of opponents across a set of games unknown to its programmer in advance of the competition.

Chapter 19

Conclusion

> Difficult problems hypothesis: There are too many ways to solve simple problems. Raising the level and breadth of competence we demand of a system makes it *easier* to test—and raise—its intelligence.
>
> – Lenat and Feigenbaum, "On the Thresholds of Knowledge" ([Lenat and Feigenbaum, 1991], p. 188)

19.1 Introduction

This chapter concludes the thesis. Section 19.2 provides a high-level summary of the thesis, Section 19.3 outlines the major contributions, and Section 19.4 points out some limitations and areas for future work.

19.2 Summary of the Thesis

Game-analysis is the general process that exploits the abstract representation of a specific game to produce a competitive advantage on that game. It is the process which relates structure to power in game-playing, and is of central importance to Artificial Intelligence (AI) as a science of intelligence. Some of the problems currently faced by the field of Computer Game-Playing (CGP) stem from the fact that, in all current work done in this field, humans have full information about the rules of the games their programs play. This makes it impossible to infer, even from strong performance on those games, that the theories which a program embodies are applicable to anything but the specific games the program has played. In particular, success on a known game is no evidence that the *program*, and not its *programmer*, has performed game-analysis. Therefore, success on a known game is not evidence that the program is interesting from an AI perspective.

These problems are alleviated by working within the new paradigm of *Meta-Game Playing* (*Metagame*). Rather than designing programs to play an existing game known in advance, we design programs (*metagamers*) to play *new* games, from a well-defined class, taking as input only the rules of the games produced by a *game generator*. The performance criterion is still competition: all programs eventually compete in a *tournament*, at which point they are provided with a set of games produced by the generator. The programs then compete against each other through many contests in each of these new games, and the winner is the one which has won the most contests by the end of the tournament. While humans still have full information about the generator and thus can still perform analysis instead of the program, the details of the specific games must be treated by the program alone. The class can also be made more general in a controlled manner as scientific knowledge advances.

Metagame in symmetric chess-like games, or simply SCL-Metagame, is a concrete Metagame research problem based around the class of symmetric chess-like games. The class generalises many features of the chess-like games, instances of which have received much of the attention in CGP, and represents games in a manner which preserves the compact structure which makes them appear similar. The class has been defined in sufficient detail necessary to enable SCL-Metagame to be used as a testbed by other researchers.

New games for this class are produced by a generator based on a new method of problem generation called *Constrained Stochastic Context-Free Generation*. It operates by making statistical choices at each decision point in the grammar defining the class of problems, as controlled by a set of user-defined constraints. The generator has produced games which are interesting to humans as objects of interest in their own right, despite the apparent complexity and unfamiliarity of the rules. Analysis of the class of games as constrained by the generator showed that it measures reasonably well in terms of coverage, diversity, structure, varying complexity, and extensibility. The conclusion from this is that the problem of SCL-Metagame is a good instance of a Metagame research problem, and that competitive performance on this problem will be evidence of increased general ability in game-playing.

The results of an extensive human analysis of the class of games have been embodied in a program, called METAGAMER, which plays SCL-Metagame. The program takes as input the *rules* of a specific game and a set of parameter settings (or *weights*), and analyses the rules to construct (a) a more efficient representation of the general semantics of the class of games, specialised to just the input game, and (b) its own evaluation function for that game, for use with a generic search engine. The strategic analysis performed by the program relates a set of general knowledge sources (called *advisors*) to the details of the particular game in a manner which can be viewed as a form of knowledge-compilation. Among other properties, METAGAMER's analysis determines the relative value of the different pieces in a given game, and for each piece the relative value of placing it on different board squares. Although METAGAMER does not learn from experience, the values resulting from its analysis are qualitatively similar to values used by experts on known games, and are sufficient to produce competitive performance the first time the program actually plays each game it is given. This appears to be the first time a program has derived useful piece values directly from analysis of the rules of different games.

An extensive experiment was carried out to assess the strength of different versions of METAGAMER across a set of unknown games produced by the generator. The conclusions from the experiment were that (a) the knowledge implemented in METAGAMER is useful on games unknown to its programmer in advance of the competition, (b) METAGAMER represents the state-of-the-art in SCL-Metagame and is a useful starting point against which to test future developments in this field, and (c) future programs that incorporate learning and more sophisticated active-analysis techniques will have a demonstrable competitive advantage on this new problem.

Examination of the performance of METAGAMER on the known games of chess and checkers, when playing against humans and specialised programs, suggested that METAGAMER has a reasonable positional sense for both games. It derives from more general principles some strategies which are familiar to players of those games and which are hard-wired in many game-specific programs. However, METAGAMER is weak in tactics compared to the specialised programs, and also weak in deductive problem-solving and planning. These are therefore promising areas for future work.

19.3 Contributions

The contributions from the three parts of the thesis are here listed separately.

The major contributions of Part I were the following:

- A characterisation of game-analysis.
- An assessment of past and present research in CGP as it relates to game-analysis.
- An analysis of the methodological underpinnings of CGP. This analysis revealed a set of difficulties with the present methodology.
- The creation of a new research paradigm for CGP, called *Meta-Game Playing* (or *Metagame*), which overcomes the difficulties revealed by the preceding analysis.

The major contributions of Part II were the following:

- The definition of a class of games to serve as a basis for Metagame-playing. The class is called *symmetric chess-like games*. The definition consisted of both a formal syntax and semantics for the class, and provided sufficient detail that the problem could be addressed by other researchers.
- A general method for automatically generating problem definitions for instances of a class of problems. The method is called *Constrained Stochastic Context-Free Generation*.

- The application of this general method to the class of symmetric chess-like games, resulting in the implementation of a game generator for this class. The resulting generator has produced interesting games which have been studied by humans as objects of research in their own right.
- The analysis of the class of games, as constrained by the implemented generator, with respect to the explicit goals of Metagame-playing.

Together, the work in this part makes two additional high-level contributions:

- The construction of a specific research problem, SCL-Metagame, designed specifically to encourage and enable researchers to generalise work on chess-like games. This problem overcomes some of the methodological limitations of existing work.
- A case-study of the construction of concrete Metagame problems. This involves the definition of a class, implementation of a game-generator, and analysis of those two components with respect to a set of desiderata. The approaches taken in this part should be useful for constructing different Metagame research problems in the future.

The major contributions of Part III were the the following:

- The application of logic programming techniques (*partial evaluation* and *abstract interpretation*) to increase the efficiency of game-playing systems.
- The construction of the first programs to play SCL-Metagame legally. These serve as baselines against which to compare future work.
- The introduction and application of *game-variant analysis*, a knowledge-acquisition technique for generalising game-specific strategies to apply to a class of games.
- A case study in strategic analysis of the class of symmetric chess-like games. This may be useful for future work on this class or different classes of games.
- An architecture for general game-playing programs based on knowledge-compilation, in which general knowledge is related to the rules of a specific game by a process of game-analysis.
- The implementation within this architecture of what appears to be the first program to derive useful material values for games by active analysis of the game rules.
- An experimental demonstration of the effectiveness of a game-playing program across a set of games unknown to its designer in advance of the competition.

- An analysis of the performance of a program playing a set of known games (chess and checkers) against human and computer opponents, using its own active analysis of the rules.

Together, the work in this part makes three additional high-level contributions:

- The construction and establishment of the state-of-the-art in SCL-Metagame programs.
- A generalisation of previous work in computer game-playing.
- An empirical confirmation of the diversity of games produced by the generator developed in Part II.

One additional contribution which resulted from the work in this thesis is the development of a general platform which supports research in Metagame. This platform contains the formal class specification expressed in *gdl* (see Appendix B), the symmetric chess-like game generator, interpreters, baseline players, efficiency optimisation routines, a generic search engine, a library of game definitions, and most of the code implementing METAGAMER. This platform has been made publicly available, under the name *Metagame Workbench*.[1] The reception to this workbench, and to the idea of Metagame, has been favourable. It was used as part of a course on computer game-playing, and is currently being used as a testbed by a small number of researchers. At present, METAGAMER has no competition—will any readers of this thesis rise to the challenge and put their generality to the test?

19.4 Limitations and Future Work

Part III of the thesis has demonstrated that SCL-Metagame can be practically addressed and that doing so necessitates increased understanding and implementation of many aspects of the process of game-analysis. However this work is just the start, and virtually everything that was done here opens an interesting area of research, with the advantage that increased understanding should translate directly into competitive advantage. Three areas for future work are apparent from the performance of METAGAMER as observed in Chapter 16 and Chapter 17. These areas are learning weights for advisors, deriving new advisors automatically from the class definition, and more advanced search techniques.

[1] The Metagame Workbench has been placed on the world chess ftp server and can be retrieved from:

`ftp@chess.uoknor.edu:pub/chess/projects/metagame/`

It has also been stored on the Prime Time AI CD-Rom, and can be retrieved from:

`ftp.cs.cmu.edu:user/ai/software/games/metagame/`

19.4.1 Learning Weights

Chapter 15 emphasised that the current version of METAGAMER does not perform any active or passive adjustment of the weights to its advisors. While Chapter 16 and Chapter 17 demonstrated that even weighing all advisors equally results in competitive performance across a variety of games, those chapters both presented results which indicated that programs that can modify their general weights based on analysis or experience with a particular game will have a marked competitive advantage over those which do not. The issue of finding weights for advisors used by METAGAMER or similar programs is thus an important area for research.

One idea for future research would be to apply temporal-difference learning [Sutton, 1984] and self-play [Tesauro, 1993] to this problem. It would be interesting to investigate whether the "knowledge-free" approach which was so successful in learning backgammon [Tesauro, 1993] also transfers to these different games, or whether it depends for its success on properties specific to backgammon.

Another idea would be to use a genetic approach with competition among a population of players with different weights (subject to the constraints on weights listed in Section 15.5). One interesting recent approach to weight learning [Angeline and Pollack, 1993] has a population of programs compete repeatedly in a knockout tournament, in which winners advance to the next round and losers play against each other. This appears to be a more efficient method to extract information and compare programs than the format used in the Metagame-tournament in Chapter 16, as the losing programs get weeded out of the competition early so that more effort can be focussed on the stronger programs.

19.4.2 Deriving New Advisors

While METAGAMER performs its own analysis of the rules of each game it plays, the analysis methods and the knowledge which draws on them were both the products of a human analysis of the whole class of games. The result is that the analysis methods implemented in METAGAMER are still very simplistic and are only linked indirectly to the semantics of the class of games. As a consequence, for a wide variety of generated games METAGAMER's analysis offers no useful guidance. For example, all static analysis at present is based on the assumption of an otherwise empty board. Pieces which move only by hopping over other pieces are determined to have no mobility by this measure. One consequence of this is that the program would have no success on games which correspond to constraint satisfaction problems (see Section 9.2.2). In short, while the advisors are well-motivated by the analysis in Chapter 14, it would be desirable and competitively advantageous to have a more systematic method of deriving new general advisors directly from the semantics of the class definition.

One approach to this problem would be to extend the techniques used for efficiency specialisation in Chapter 12 to derive useful subgoals and invariant properties for a given game by abstract interpretation. It might also be possible to apply *knowledge-*

based feature construction [Callan, 1993; Fawcett and Utgoff, 1992] directly to this problem, as it is designed to extract functional features from a logical encoding of problem definitions.

19.4.3 Advanced Search Techniques

Chapter 17 noted that while METAGAMER's analysis of each game seemed to give it a basic understanding of strategy, it is noticeably weak in tactics. This weakness is caused in part by an inability to search more deeply within the time constraints, by a lack of quiescence search, and also by the reliance on full-width tree-search. Almost any improvement should produce a markedly better player. One way to address this problem might be through a tactics analyser as developed by Berliner [Berliner, 1974] or a knowledge-based planner as developed by Wilkins [Wilkins, 1982]. These would need to be extended to games beyond just chess.

Another approach would be to apply some of the recent developments in *rational game-tree search* [Russell and Wefald, 1992; Baum and Smith, 1993; Good, 1968] to SCL-Metagame. These methods appear to be powerful and general tools, but within the traditional approach to games it has been impossible to demonstrate their advantage against programs which have already been carefully engineered by humans to play a single game well. It would be exciting to apply these techniques to this new context, where programs are required to perform game-specific optimisations without human assistance.

Appendix A

Grammars for Symmetric Chess-Like Games

This appendix presents the formal grammar in which symmetric chess-like games are encoded. A formal move grammar has also been defined to enable programs to communicate on games in this class, and this is listed in Section A.2.

A.1 Class Definition

This grammar for *symmetric chess-like games* is presented in an extended-*BNF* notation. Capitalised and quoted words are terminal symbols, except for `IDENTIFIER` and `NUMBER`, which stand for any identifier and any number, respectively. Text in unquoted braces is optional. The grammar is not case-sensitive, so that game definitions may use uncapitalised words for clarity.

Comments can appear within game definitions. Comments begin with percent symbols ('%') and end with the start of a new line.

```
game --> GAME IDENTIFIER
         goal_defs
         board
         SETUP assignment_list
         {CONSTRAINTS MUST_CAPTURE}
         {piece_defs}
         END GAME '.'

goal_defs --> GOALS goals

goals -->  goal | goal goals

goal --> ARRIVE description AT square_list
```

```
       | ERADICATE description
       | STALEMATE player

description --> '[' player_gen piece_names ']'

player_gen --> '{' player '}' | ANY_PLAYER

player -->  PLAYER | OPPONENT

piece_names --> '{' identifiers '}'
              | ANY_PIECE

identifiers --> IDENTIFIER | IDENTIFIER identifiers

square_list --> '{' squares '}'

squares --> square | square squares

square --> '(' NUMBER ',' NUMBER ')'

board --> BOARD_SIZE NUMBER BY NUMBER
          BOARD_TYPE board_type
          {INVERSION inversion_type}
          PROMOTE_RANK NUMBER

board_type --> PLANAR | VERTICAL_CYLINDER

inversion_type --> FORWARD | DIAGONAL

assignment_list --> assignment_decision
                  | assignments

assignment_decision --> DECISION assigner ASSIGNS
                        piece_names TO square_list
                        END DECISION

assigner --> player | RANDOM

assignments --> assignment | assignment assignments

assignment --> IDENTIFIER  AT square_list
```

```
piece_defs --> piece_def | piece_def piece_defs

piece_def --> DEFINE IDENTIFIER
              {MOVING movement_def END MOVING}
              {CAPTURING capture_def END CAPTURING}
              {PROMOTING promote_def END PROMOTING}
              {CONSTRAINTS constraint_def}
              END DEFINE

movement_def --> movement | movement movement_def

movement --> MOVEMENT
             movement_type
             direction
             symmetries
             END MOVEMENT

movement_type --> leaper | rider | hopper

leaper --> LEAP

rider --> RIDE  {MIN NUMBER} {MAX NUMBER} {LONGEST}

hopper --> HOP BEFORE compare_eq
               OVER compare_eq
               AFTER compare_eq
           HOP_OVER description

compare_eq --> '[' X comparative NUMBER ']'

comparative --> '>=' | '=' | '<='

direction --> '<'  NUMBER ',' NUMBER '>'

symmetries --> SYMMETRY symmetry_set

symmetry_set --> ALL_SYMMETRY
               | '{' {FORWARD} {SIDE} {ROTATION} '}'

capture_def --> capture | capture capture_def

capture --> CAPTURE
```

```
            BY capture_methods
            TYPE description
            EFFECT effect
            movement_def
            END CAPTURE

capture_methods --> '{' {RETRIEVE} {CLOBBER} {HOP} '}'

effect --> REMOVE | player POSSESSES | player DISPLACES

promote_def --> PROMOTE_TO IDENTIFIER
              | promotion_decision

promotion_decision -->  DECISION player
                        OPTIONS description

constraint_def --> {MUST_CAPTURE} {CONTINUE_CAPTURES}
```

A.2 Move Grammar

In order to have programs communicate directly on games unknown to humans in advance of a competition, there must be a *defined language*, in which to express a player's choice of moves, which will be adequate and unambiguous for any game within the entire class. This section discusses the move grammar for *symmetric chess-like games*. The grammar is based on the standard notations for moves used in Chess and Shogi, but extended to describe unambiguously all the changes which can happen as part of a move in this class. As the move grammar should be clear, we will only provide a few example descriptions of different types of moves.

Basic Movements and Captures The basic movement of a piece P from square $(x1, y1)$ to square $(x2, y2)$ is written: `P` $(x1, y1) \rightarrow (x2, y2)$. If this move had the *capture effect* of *removing* a piece Q at square $(x3, y3)$, the full move would be (replacing symbols for squares): `P sq1` $\rightarrow$ `sq2 X Q sq3`.

Possession If the effect of a given capture were *player possesses* instead of removal, the captured piece Q would then be in the possession of the player who moved. If white had just moved, this would be denoted: `P sq1` $\rightarrow$ `sq2 X Q sq3 / (white)`. If the effect were instead *opponent possesses*, the captured piece would go to the opponent. The above move thus would be: `P sq1` $\rightarrow$ `sq2 X Q sq3 / (black)`.

Multiple Captures A single capture movement can result in the capture of several pieces. For example, a piece may hop over one piece to land on another, thus capturing both. Such multiple captures are denoted by listing each piece and square captured: `P sq1 → sq2 X Q sq3 R sq4`. If the effect were *player possesses* instead of *remove*, such a move would be denoted: `P sq1 → sq2 X Q sq3 R sq4 (white)`.

Placing a Possessed Piece A player in possession of a piece can at any later move *place* this piece on any empty square, instead of making a normal piece movement. So if the piece captured and possessed on $sq3$ in the last move above was Q, a later move for *white*, placing this piece on square sq, would be: `Q(white) → sq`.

Promotion by Player The notation for a move which promotes a piece includes the square the piece is on, the player who will now own the piece,[1] and the piece-type being promoted to. If white moves piece P to $sq2$ in his promotion territory, P has a `fixed` or `player_promotes` promotion power, and white decides to promote it to a king, this would be denoted: `P sq1 → sq2; promote sq2 white king`.

Promotion by Opponent When a player must begin his turn by promoting a piece which has just been moved by the opponent, the notation for this promotion precedes the notation for the rest of the move. For example, suppose a move by *white* moves a piece P from $sq1$ to $sq2$, capturing some piece Q on $sq3$, with the `opponent_possesses` capture effect, and also that $sq2$ is in promotion territory for white, and that piece P has the `opponent_promotes` promotion power. As this promotion is not denoted in the player's move (he makes no choice here, so it can be inferred), the notation for white's move would be: `P sq1 → sq2 X Q sq3 (black)`.

White's turn would then end, and black would then have to promote white's moved piece into some piece $P2$ consistent with its promotion power, and then make a normal move (suppose he places the captured piece Q from his hand on square $sq4$). Black's move would then be denoted:
`promote sq2 P2; Q(black) → sq4`.

Continued Captures This sequential notation (with the semicolon) is also used when a player *continues capturing* (see section 7.2.4.3). For example, a continued captures move in Checkers might be written: `White Man (a 1) → (c 3) X (b 2) ; White Man (c 3) → (e 5) X (d 4)`.

Summary of Move Grammar This notation is complete and unambiguous, and thus allows humans and programs to communicate their moves in any game in the class of symmetric chess-like games.

[1] Recall that promotion may involve a change of ownership.

```
move --> {promote ';'}
         main_move '.'

promote --> PROMOTE square piece

main_move --> placement
            | transfers {';' promote}

placement --> piece '(' color ')' '->' square

transfers --> transfer ';' {transfers}

transfer --> moving {capture}

moving --> piece square '->' square

capture --> X piece square effect {capture}

effect --> remove
         | possess

remove --> {}

possess --> '/' '(' color ')'

piece --> color piece_name

color --> WHITE
        | BLACK

piece_name --> IDENTIFIER

square --> '(' NUMBER ',' NUMBER ')'
```

Appendix B

Formalisation of Symmetric Chess-Like Games

A general overview of the class of symmetric chess-like games is provided in Chapter 7. The game grammar given in Appendix A.1 defines the syntax of a game definition. This Appendix provides a formalisation of symmetric chess-like games, in which the semantics is represented using the Game Description Language presented in Chapter 12.

It should be noted that there are many possible representations of this theory, each of which may have different implications for efficiency and flexibility when used by a playing program. The particular representation presented here is intended to convey the semantics clearly, and in places differs in implementation details from the encoding used by the playing programs discussed in the thesis.

When documenting procedures, Prolog convention is used to indicate the intended mode of arguments, where `+`, `-`, and `?` indicate that the argument functions as input, output, or either, respectively.

The details of game-specific predicates (such as `game:piece_has_movement`), which primarily access the data structure corresponding to the parsed game definition, are straightforward and have thus been omitted from this presentation.

B.1 Overview

The following theory provides the representation of initial state, goal achievement and complete legal moves for symmetric chess-like games. When interpreted using the Game Description Language meta-interpreter on a given game, these routines form the skeleton of the game definition, defining three procedures:

1. legal-move(Move,Player,StateIn,StateOut,Game): True when Move is legal for Player in StateIn, and Produces StateOut, in Game.

2. game-outcome(FinalState,Outcome,Game): Outcome is the final outcome of Game, which ends in FinalState. Outcome is either **player**, **opponent**, or **draw**.

3. start-game(InitState,Game): True when InitState is set to a valid start state for a run of Game.

Any 2-player game must provide at least these procedures in order to be fully defined, and an interface which relies on these is thus capable of playing legally all 2-player games.

The rest of this chapter provides the domain theory for each of these items in turn.

B.2 Legal Moves

We first provide an overview of the basic move sequence, and then present the encoding of this in the Game Description Language.

B.2.1 Overview of Basic Move Sequence

Each individual legal move is itself composed of a sequence of *pseudo-operators*, each of which are applicable in different *stages* of the move sequence. The pseudo-moves applicable at each stage are as follows:

- **Assignment:** If the game begins with a decision-assignment stage (where players place pieces on the board before moving anything), the only legal move for each player is to *assign* a piece and end his move. When there are no more pieces to assign, the stage is set to *move*.

- **Init-promote:**: If the last player moved a piece into promotion region, and the piece has an opponent-promotes promotion power, then this move begins with the player now in control *init-promoting* the piece on its square, after which the stage is set to *move*.

- **Move:** In an ordinary move, a player decides whether to *place* or *move* a piece. The two operators, either of which can be selected, are as follows:

  – **Place a piece:** If a player has a piece *in-hand* (i.e. one given to him as a result of an earlier possession capture-effect), he can now place it on an empty square of his choice. This ends the move.

  – **Move a piece:** A piece is moved from one square to another, possibly enabling some capture effects. For games which have a global *must-capture* constraint, if some capturing moves are available, any one of them can be made; otherwise the player is free to make any non-capturing move. For games which have only local *must-capture* constraints, any piece can be moved, subject to the constraint that if it moves and has some capture choices, it cannot make a non-capturing move. In this case, unconstrained pieces can make either capturing or non-capturing moves.

- **Capture:** If the movement chosen ended in some pieces being subjected to capture effects, these are now executed. There are two types of effect, which may apply to many pieces as a consequence of a given move:
  - *Removed* pieces just disappear.
  - *Possessed* pieces go to the hand and colour of the possessing player.
- **Continue captures:** If a piece that just captured is *permitted* to *continue-capturing*, a next capture movement for it may be chosen (but only if this really captures), after which the stage goes back to *capture*. If the piece is *required* to continue capturing (when it is both permitted to continue, and forced to capture whenever it can), then if it can do so, it must. When a player cannot continue capturing, or decides to stop (unless forced to continue, of course), the stage transfers to *promote*.
- **Promote:** If a piece has just finished being *moved* (as opposed to being *placed*), and is now on a square in promotion territory for the moving player, the promotion effect is then executed. There are two types of promotion effects:
  - *player-promotes:* the moving player *selects* a valid piece to promote into (which depends on the piece definition), which replaces the original piece on its final square. The player's turn then ends, and control transfers to the opponent, whose move will begin in the *move* stage.
  - *opponent-promotes:* the piece is removed from the board, and the player's turn ends. The opponent will begin the next move in the *init-promote* stage, and will thus start the next turn putting a valid promoted piece on the indicated square.

The set of legal moves available to a player in a position is thus the set of all possible sequences of legal pseudo-moves available to that player which terminate in a transfer of control to the other player.

B.2.2 Legal Move Domain Theory

legal_move(?Move,?Player): *A move begins with one of the players being in control, after which a sequence of sub-moves is made, which ends in transfer of control to the other player. A move-count is tracked throughout the game.*

```
legal_move(Move,Player) ==>
     control(Player),
     legal_move_seq(Move,Player),
     increment_move_count.

legal_move_seq([M|Rest],PlayerIn) ==>
     pseudo_op(M),
     control(PlayerOut),
     if( opposite_role(PlayerIn,PlayerOut),
         true,
         legal_move(Rest,PlayerIn) ).

increment_move_count ==>
     move_count(M),
     M1 is M + 1,
     del(move_count(M)),
     add(move_count(M1)).
```

B.2.2.1 Pseudo-Operators

Initial Assignments *For games in which players begin placing pieces on the board, the interpreter program should have already put the correct pieces in the hand of each player. At this point there are two operators.*

- **assign:** *Place a piece on an assignable square, ending the move. Opponent will start next turn still in assign stage.*
- **end_assign:** *Nothing to place, so proceed to move stage.*

```
pseudo_op(assign(Piece,Square)) ==>
     true(stage(assign)),
     true(in_hand(_Piece,_Player)),
     game:assignable_squares(Squares),
     member(Square,Squares),
     placeable(Piece,Player,Square),
     place_piece_from_hand(Piece,Player,Square),
     transfer_control.

pseudo_op(end_assign) ==>
     true(stage(assign)),
     not true(in_hand(_Piece,_Player)),
     transfer_stage(assign,move).
```

```
place_piece_from_hand(Piece,Player,Square) ==>
     del(in_hand(Piece,Player)),
     add(on(Piece,Square)).
```

Opponent Promotion *When a piece, owned by a player, has finished a move by moving into that player's promotion region, the player designated in the piece's definition gets to promote it to some piece matching the defined description, and then continue to the movement stage.*

```
pseudo_op(opponent_promote(Sq,NewPiece)) ==>
     true(stage(init_promote)),
     true(opponent_promotes(OldPiece,Sq,Descr)),
     del(opponent_promotes(OldPiece,Sq,Descr)),
     matches(Descr,NewPiece),
     add(on(NewPiece,Sq)),
     transfer_stage(init_promote,move).

pseudo_op(no_opponent_promote) ==>
     true(stage(init_promote)),
     not true(opponent_promotes(_OldPiece,_Sq,_Descr)),
     transfer_stage(init_promote,move).
```

Placement Operator *Unlike in the assignment stage, a player can* **place** *a piece in his possession on any empty square. This ends his move.*

```
pseudo_op(place(Piece,Square)) ==>
     true(stage(move)),
     control(Player),
     true(in_hand(Piece,Player)),
     placeable(Piece,Player,Square),
     place_piece_from_hand(Piece,Player,Square),
     transfer_control.

placeable(Piece,Player,Square) ==>
     in_hand(Piece,Player),
     empty(Square).
```

Move Operator *Make a capturing or non-capturing movement, subject to global or local must-capture constraints. Movements begin by* selecting *a piece on the board to move, and taking it off the board before considering where to move it. For capturing moves, we will also note the capture effects.*

```
pseudo_op(move(Piece,Player,SqF,SqT)) ==>
     control(Player),
     if( game:global_must_capture,
         global_prefer_capture(Piece,Player,SqF,SqT),
         local_move(Piece,Player,SqF,SqT) ).

global_prefer_capture(Piece,Player,SqF,SqT) ==>
     if( ( select_piece(Piece,Player,SqF),
           capturing(Piece,Player,SqF,SqT) ),
         transfer_stage(move,capture),
         ( select_piece(Piece,Player,SqF),
           moving(Piece,Player,SqF,SqT),
           transfer_stage(move,promote) ) ).

local_move(Piece,Player,SqF,SqT) ==>
     select_piece(Piece,Player,SqF),
     if( game:piece_must_capture(Piece),
         local_prefer_capture(Piece,Player,SqF,SqT),
         general_moving(Piece,Player,SqF,SqT)).

local_prefer_capture(Piece,Player,SqF,SqT,Stage) ==>
     if( capturing(Piece,Player,SqF,SqT),
         transfer_stage(move,capture),
         ( moving(Piece,Player,SqF,SqT),
           transfer_stage(move,promote) ) ).

general_moving(Piece,Player,SqF,SqT) ==>
     moving(Piece,Player,SqF,SqT).
general_moving(Piece,Player,SqF,SqT) ==>
     capturing(Piece,Player,SqF,SqT).

select_piece(Piece,Player,SqF) ==>
     true(on(Piece,SqF)),
     piece_struct_owner(Piece,Player),
     del(on(Piece,SqF)).
```

Capture Operator *Executes the capture effects for whichever pieces were noted to be captured during the movement stage. The next stage will give the moving piece a chance to continue capturing.*

```
pseudo_op(capture(Effect,Captured)) ==>
     true(stage(capture)),
```

```
     true(effect(Effect,Captured)),
     del(effect(Effect,Captured)),
     effect_captures(Captured,Effect),
     transfer_stage(capture,continue).

effect_captures([],_).
effect_captures([Cap|Caps],Effect) ==>
     effect_capture(Effect,Cap),
     effect_captures(Caps,Effect).

effect_capture(remove,Cap) ==>
     captured(Cap,Piece,Sq),
     del(on(Piece,Sq)).
effect_capture(possess(Owner),Cap) ==>
     captured(Cap,Piece,Sq),
     del(on(Piece,Sq)).
     piece_struct(Piece,Type,_Player),
     piece_struct(NewPiece,Type,Owner),
     add(in_hand(Piece,Player)).

captured(Piece@Sq,Piece,Sq).
```

Continue Captures *After making a capture, a piece may be permitted or forced to continue capturing as part of its definition. The results determine whether the resulting pseuo-operator is another capture movement, or a decision to end the captures now.*

```
pseudo_op(Move) ==>
     true(stage(continue)),
     if( may_continue(Piece,Sq),
         if( must_continue(Piece),
             continue_captures(Piece,Player,Sq,_SqT,Move),
          continue_or_end(Piece,Player,Sq,_SqT,Move) ),
         discontinue(Move) ).

may_continue(Piece,Sq) ==>
     true(moved_onto(Piece,Sq)),
     game:piece_may_continue(Piece).

must_continue(Piece) ==> game:piece_must_continue(Piece).
```

Optional continued captures: *The piece can either make another capture, and move back to the* **capture** *stage, or end the capture sequence and move to the* **promote**

stage.

```
continue_or_end(Piece,Player,Sq,SqT,move(Piece,Player,Sq,SqT)) ==>
     del(moved_onto(Piece,Sq)),
     capturing(Piece,Player,Sq,SqT),
     transfer_stage(continue,capture).
continue_or_end(_,_,_,_,end_continues) ==>
     transfer_stage(continue,promote).
```

Forced continued captures: *If the piece is forced to continue capturing, and has some capture movements available, it must take one of them, in which case we first note that the this square is no longer the destination square of the piece. Otherwise, the sequence ends.*

```
continue_captures(Piece,Player,Sq,SqT,Move) ==>
     del(moved_onto(Piece,Sq)),
     if( capturing(Piece,Player,Sq,SqT),
         ( Move = move(Piece,Player,Sq,SqT),
           transfer_stage(continue,capture) ),
         ( add(moved_onto(Piece,Sq)),
           Move = end_continues,
           transfer_stage(continue,promote) ) ).
```

Promoting Pieces *When a piece, owned by a player, has moved into that player's promotion region, the player designated in the piece's definition gets to promote it to some piece matching the defined description. In any case, the promoted piece (the original one if it does not now promote) is now put down on the final square, as thus far it was only noted that it had moved there.*

```
pseudo_op(try_promote(Square,OldPiece,NewPiece)) ==>
     true(stage(promote)),
     control(Player),
     moved_onto(OldPiece,Sq),
     promote_if(OldPiece,Player,Sq,NewPiece).

promote_if(OldPiece,Player,Sq,NewPiece) ==>
     game:in_promote_region(Sq,Player),
     game:piece_promoting(OldPiece,Promoting),
     promote_if1(OldPiece,Promoting,Player,Sq,NewPiece).
promote_if(OldPiece,Player1,Sq,[]) ==>
     not game:in_promote_region(Sq,Player1),
     del(moved_onto(OldPiece,Sq)),
```

```
     add(on(OldPiece,Sq)),
     transfer_stage(promote,move),
     transfer_control.
```

promote_if1(+OldPiece,+Promoting,+Player,+Sq,-NewPiece): *If the promotion is simple (i.e. only one choice), it gets done now. Otherwise there will be a selection.*

```
promote_if1(OldPiece,Promoting,Player,Sq,NewPiece) ==>
     simple_promote(Promoting,Player1,NewPiece),
     del(moved_onto(OldPiece,Sq)),
     add(on(NewPiece,Sq)),
     transfer_stage(promote,move),
     transfer_control.
promote_if1(OldPiece,Promoting,Player,Sq,selection) ==>
     promoter(Promoting,Player,Promoter),
     promote_role(Promoter,Player,OldPiece,Sq).

simple_promote(Promoting,Player,NewPiece) ==>
     promoting_info(Promoting,promote(Type)),
     piece_struct(NewPiece,Type,Player).

promoter(Promoting,_Player,Promoter) ==>
     Promoting = promote(Promoter,_Descr).
```

promote_role(+Promoter,+Player,+OldPiece,+Sq): *If the piece is to be promoted by the player who owns the it, transfer to the promote-select stage. If the opponent is to promote it, then record this fact and end the move.*

```
promote_role(Player,Player,_,_) ==>
     transfer_stage(promote,promote_select).
promote_role(Opponent,Player,OldPiece,Sq) ==>
     opposite_role(Player,Opponent),
     game:piece_promoting(OldPiece,Promoting),
     promoting_options(Promoting,Descr),
     add(opponent_promotes(OldPiece,Sq,Descr)),
     del(moved_onto(OldPiece,Sq,Descr)),
     transfer_stage(promote,init_promote),
     transfer_control.
```

Promote-Select Operator *The player promotes the piece to a valid piece matching the description in the definition of the promoting power for that piece. The record of this piece movement is deleted, and the piece is placed on its final square, as the move* *sequence is now finished. This ends the turn, and the next player's move will start in the* **move** *stage.*

```
pseudo_op(promote_select(Sq,OldPiece,NewPiece)) ==>
     true(stage(promote_select)),
     control(Player),
     moved_onto(OldPiece,Sq),
     game:piece_promoting(OldPiece,Promoting),
     promoting_options(Promoting,Descr),
     matches(Descr,NewPiece),
     del(moved_onto(OldPiece,Sq)),
     add(on(NewPiece,Sq)),
     transfer_stage(promote_select,move),
     transfer_control.

promoting_options(Promoting,Descr) ==>
     Promoting = promote(_Actor,Descr).
```

B.2.3 Moving and Capturing Powers

Pieces have separate powers of moving and capturing. A moving power defines a disjunctive set of movements, each with a disjunctive set of directions, based on the symmetries attached to the base direction of the movement. A capturing power defines a disjunctive set of capture definitions, each containing a set of movements, capture-methods, and capture effects.

The main difference between using moving and capturing powers is as follows:

Moving Powers: Find a final square SqT such that there is a path from the current square SqF to SqT using one of the defined movements within the moving power definition, and SqT is presently empty.

Capturing Powers: Find a similar path, but using a movement within the capturing power definition, and after determining the captured pieces based on the defined *capture-methods* for this capturing movement, something must get captured and SqT must become empty if it was not empty already.

B.2.3.1 Moving Powers

moving(?Piece,+Player,?SqF,?SqT): *True when player owns a piece and there is some valid* moving-power *defined for that piece, a movement within that power, and a direction within the symmetry-set for the movement, which can be used to move the piece from SqF to SqT, without capturing anything. This routine does not require that the piece is on the board, as presumably it has already been lifted. The moving piece* *will not be placed after this movement, but we note that this piece has moved to a square, with the state predicate* `moved_onto(Piece,SqT)`.

```
moving(Piece,Player,SqF,SqT) ==>
     game:piece_has_movement(Piece,Movement),
     movement_has_symmetric_direction(Movement,Dir),
     moving_movement_for_piece(Piece,SqF,SqT,Player,Movement,Dir,_Hop),
     add(moved_onto(Piece,SqT)).
```

If the movement selected is a riding movement with the **longest-ride** *property, and the board for this game is planar, the piece is constrained to ride as far as possible in the selected direction. Otherwise, any movement is ok. As this is not a capturing movement, the final square must be empty.*

```
moving_movement_for_piece(Piece,SqF,SqT,Player,Movement,Dir,_Hop) ==>
     if(  ( ride(Movement,Dir,Min,Max,Longest),
            longest(Longest),
            game:board_type(planar) ),
          longest_moving_ride(SqF,Dir,Min,Max,SqT),
          movement_for_piece(Piece,SqF,SqT,Player,Movement,Dir,_Hop)
       ),
     empty(SqT).
```

B.2.3.2 Capturing Powers

capturing(?Piece,+Player,?SqF,?SqT): *True when player owns a piece and there is some valid* capturing-power *defined for that piece, a movement within that power, and a direction within the symmetry-set for the movement, which can be used to move it from SqF to SqT, capturing at least one piece. As in the case for using moving-powers, this routine does not require that the piece is on the board, or place it back at the end. Capture effects are not executed here, but the captured pieces and the capture effect are noted with the state predicate* `effects(Effect,Captured)`.

```
capturing(Piece,Player,SqF,SqT) ==>
     captures(Piece,Player,SqF,SqT,Effect,Captured),
     add(moved_onto(Piece,SqT)),
     add(effects(Effect,Captured)).
```

captures(+Piece,+Player,+SqF,-SqT,-Effect,-Captured): *True when Piece, from SqF, could use a capture power to move to SqT, making the list of captures in Captured, with capture effect Effect.*

```
captures(Piece,Player,SqF,SqT,Effect,Captured) ==>
     game:piece_has_capture(Piece,Capture),
     capture_has_movement(Capture,Movement),
     movement_has_symmetric_direction(Movement,Dir),
     capture_effect(Capture,Effect),
     capturing_movement_for_piece(Piece,SqF,SqT,Player,
                                  Movement,Dir,Capture,Captured).
```

If the movement selected is a riding movement with the **longest-ride** *property, and the board for this game is planar, the piece is constrained to make the longest riding movement in the given direction which results in a capture. Otherwise, any capturing movement is ok. In determining the captured pieces resulting from this movement, it is ensured that the final square would be empty.*

```
capturing_movement_for_piece(Piece,SqF,SqT,Player,
                             Movement,Dir,Capture,Captured) ==>
     if( ( ride(Movement,Dir,Min,Max,Longest),
           longest(Longest),
           game:board_type(planar) ),
         longest_capturing_ride(SqF,Dir,Min,Max,SqT,Capture,Captured),
         ( movement_for_piece(Piece,SqF,SqT,Player,Movement,Dir,Hopped),
           captured_pieces(SqF,SqT,Capture,Dir,Hopped,Captured) ) ).
```

B.2.4 Piece Movements

There are three types of movements, characterised as follows:

- **leap:** The piece moves to the next square along a given direction vector.
- **ride:** The piece moves along an open line of squares along a given direction vector.
- **hop:** The piece leaps through a number (possibly 0) of empty squares, then through a number (possibly 0) of squares occupied by certain types of pieces, and then through a number (at least 1) of empty squares.

Leaping movement: *The piece moves to the next connected square along the selected direction. This is equivalent to a ride of distance 1.*

```
movement_for_piece(Piece,SqF,SqT,Player,Movement,Dir,[]) ==>
     leap(Movement,Dir),
     connected(SqF,SqT,Dir).
```

Riding movements: *Traverse in the given direction at least the minimum number, and at most the maximum number of empty squares starting from our current square. Then leap one further to the final square (which need not be empty). A separate definition applies to* **longest-ride** *movements.*

```
movement_for_piece(Piece,SqF,SqT,Player,Movement,Dir,[]) ==>
     ride(Movement,Dir,Min,Max,_Longest),
     open_line(SqF,Dir,Min,Max,empty,_Squares,SqT).
```

A hopping movement has constraints on the number of squares traversed before and after any hopped-over pieces, and a constraint on the number of hopped-over pieces as well. The constraints are of the form: `compare(Op,Val)`, *where Op is* geq, eq, *or* leq, *and Val is an integer. The meaning is that the number of leaps must be* at least N, equal to N, *or* at most N, *respectively. The set of hopped squares is returned.*

```
movement_for_piece(Piece,SqF,SqT,Player,Movement,Dir,Hopped) ==>
     hop(Movement,Dir,Before,Over,After,Description),
     hoplines(SqF,SqT,Dir,Before,Over,After,Description,Hopped).

comparative_interval(compare(geq,N),Dir,N,MaxLeaps) ==>
     valid_max(_,Dir,MaxLeaps).
comparative_interval(compare(eq,N),_,N,N).
comparative_interval(compare(leq,N),_,0,N).

hoplines(SqF,SqT,Dir,Before,Over,After,Description,Hopped) ==>
     hopline(SqF,Before,Dir,empty,_,SqB),
     hopline(SqB,Over,Dir,Description,Hopped,SqO),
     hopline(SqO,After,Dir,empty,_,SqL),
     connected(SqL,SqT,Dir).

hopline(SqF,Range,Dir,Descr,Squares,SqT) ==>
     comparative_interval(Range,Dir,MinB,MaxB),
     constrained_line(SqF,Dir,MinB,MaxB,Descr,Squares,SqT).
```

Longest Rides *The piece tries riding the maximum distance in the given direction. If this is not legal, it tries one less, and so on. The routine fails when the distance would be closer than the minimum defined for this movement. When making the longest moving ride, the farthest empty square is selected. When making the longest capturing ride, the farthest empty square is selected which would result in the capture of a piece.*

```
longest_moving_ride(SqF,Dir,Min,Max,SqT) ==> Min > Max, !, fail.
longest_moving_ride(SqF,Dir,Min,Max,SqT) ==>
```

```
     open_line(SqF,Dir,Max,Max,empty,Squares,SqT),
     empty(SqT), !.
longest_moving_ride(SqF,Dir,Min,Max,SqT) ==>
     Max1 is Max - 1,
     longest_moving_ride(SqF,Dir,Min,Max1,SqT).

longest_capturing_ride(_SqF,_Dir,Min,Max,_SqT,_Capture,_) ==>
     Min > Max, !, fail.
longest_capturing_ride(SqF,Dir,_Min,Max,SqT,Capture,Captured) ==>
     open_line(SqF,Dir,Max,Max,empty,_Squares,SqT),
     captured_pieces(SqF,SqT,Capture,Dir,[],Captured), !.
longest_capturing_ride(SqF,Dir,Min,Max,SqT,Capture,Captured) ==>
     Max1 is Max - 1,
     longest_capturing_ride(SqF,Dir,Min,Max1,SqT,Capture,Captured).
```

B.2.4.1 Open Lines

open_line(SqF,Dir,Min,Max,Cond,Squares,SqT): Traverse between Min-1 and Max-1 leaps along Dir, starting after SqF, where each such square traversed satisfies COND (either 'empty' or a piece description). Then take one more leap, which brings us to SqT (not necessarily empty). The difference between an **open** and a **constrained** line (below) is as follows: an open-line finishes with a step, so we first decrement the min and max counters, as we know we will take one leap at the end.

```
open_line(SqF,Dir,Min,Max,Cond,Squares,SqT) ==>
     Min1 is Min-1,
     Max1 is Max-1,
     constrained_line(SqF,Dir,Min1,Max1,Cond,Squares,Sq1),
     connected(Sq1,SqT,Dir).
```

constrained_line(SqF,Dir,Min,Max,Cond,Squares,SqT): *True when N squares, starting with the square after SqF, satisfy Cond, where* $Min <= N <= Max$*. The set of squares are collected in the variable* $Squares$*.*

```
constrained_line(SqF,Dir,Min,Max,Cond,Squares,SqT) ==>
     all_sat(SqF,Dir,Cond,0,Min,Max,Squares,SqT).

all_sat(SqF,_Dir,_Cond,Count,Min,Max,[],SqF) ==>
     Count >= Min, Count =< Max.
all_sat(SqF,Dir,Cond,Count,Min,Max,[Sq1|Squares],SqT) ==>
     Count < Max,
     connected(SqF,Sq1,Dir),
     crossable(Cond,Sq1),
```

```
    Count1 is Count + 1,
    all_sat(Sq1,Dir,Cond,Count1,Min,Max,Squares,SqT).
```

Crossable(+Descr,?Sq): *If Descr is Empty, then true if square is empty. If a piece description, then true if the piece on the square matches.*

```
crossable(empty,Sq) ==> empty(Sq).
crossable(Descr,Sq) ==> true(on(Piece,Sq)), matches(Descr,Piece).
```

B.2.4.2 Determining Captured Pieces

A capture definition for contains some some set of **capture methods**, which determine the pieces might be captured following a given capture-movement. The candidate targets for each method and a given movement are as follows:

- **clobber:** Selects the destination of the movement as a target square to be captured.
- **retrieve:** Selects the square adjacent to the starting square but in the opposite direction of the movement.
- **hop:** Selects any of the pieces which were hopped-over if the movement was a hopping movement.

Having determined the set of selected candidate squares, the pieces actually captured are those on any selected square which match the **capture-type** defined for the capturing movement.

captured_pieces(SqF,SqT,Capturing,Dir,Hopped,-Captures): *Given the capture type and method, determines the set of* Piece@Square *entries which will be captured. For the capture to be valid, something must be captured, and the destination square SqT must wind up empty.*

```
captured_pieces(SqF,SqT,Capturing,Dir,Hopped,Captures) ==>
    capture_type(Capturing,Type),
    if( clobbers(SqT,Capturing,Type,Clobs),
        true,
        Clobs = [] ),
    if( retrieves(SqF,Dir,Capturing,Type,Retrs),
        true,
        Retrs = []),
    if( hops(Hopped,Capturing,Type,Hops),
        true,
```

```
            Hops = []),
     append_lists([Retrs,Clobs,Hops],Captures),
     valid_capture(SqT,Captures).

valid_capture(Final,Captured) ==>
     Captured ≡ [],
     will_be_empty(Final,Captured).
```

A square will be empty if it is already empty, or if the piece now on it will be captured.

```
will_be_empty(Sq,Captured)  ==> captured_piece(_Piece,Sq,Captured).
will_be_empty(Sq,_Captured) ==> empty(Sq).

captured_piece(Victim,SqV,Captured) ==>
     member(Cap,Captured),
     captured(Cap,Victim,SqV).
```

Capture Methods *The three capture methods are* **clobber**, **retrieve** *and* **hop**.

```
clobbers(Sq,Capturing,Type,[Piece@Sq]) ==>
     capture_has_method(Capturing,clobber),
     true(on(Piece,Sq)),
     matches(Type,Piece).

retrieves(SqF,Dir,Capturing,Type,[Piece@Sq1]) ==>
     capture_has_method(Capturing,retrieve),
     connected(Sq1,SqF,Dir),
     true(on(Piece,Sq1)),
     matches(Type,Piece).

hops(Hopped,Capturing,Type,Hops) ==>
     capture_has_method(Capturing,hop),
     matchers_on_squares(Hopped,Type,Hops).
```

matchers_on_squares(+Hopped,+Type,-Matchers): *Matchers contains those Hopped-over pieces which match Type, and will thus be captured. Each element of Matchers is of the form:* `Piece@Square`.

```
matchers_on_squares([],_,[]).
matchers_on_squares([H|Hs],Type,Caps) ==>
     true(on(P,H)),
     if( matches(Type,P),
```

```
        Caps = [P@H|Rest],
        Caps = Rest ),
    matchers_on_squares(Hs,Type,Rest).
```

B.2.5 Matching Piece Descriptions

MATCHES(?Descr,?Piece): *A Piece matches a Description if both the player and type in the description are at least as general as those of the piece.*

```
matches(Descr,Piece) ==>
    piece_description(Descr,Player_Gen,Piece_Gen),
    piece_struct(Piece,Name,Player),
    matches_player(Player_Gen,Player),
    matches_name(Piece_Gen,Name).

matches_player(any_player,Player) ==> player_role(Player).
matches_player(player,player).
matches_player(opponent,opponent).
```

Note that a generalisation of a piece-type must be either `any_piece` *or a list of piece-names in the current game, not a singleton piece-name.*

```
matches_name(any_piece,Name) ==>
    game:piece_name(Name).
matches_name([H|T],Name) ==>
    game:piece_name(Name),
    member(Name,[H|T]).
```

B.2.6 Low-Level Representation

B.2.6.1 Data Structures

Piece Descriptions, Pieces, Squares, and Directions are represented as structures. A **piece-structure** has fields for the Player who owns the piece and the name of the piece. A **piece-description** has fields for the player-generalisation and piece-generalisation which match the description. A **square-structure** has fields for the X and Y coordinates of the square. A **direction** has fields for the X and Y offsets defininining the slope of the vector.

```
piece_description(piece_desc(Player,Piece),Player,Piece).

piece_description_player(piece_desc(Player,_Piece),Player).
piece_description_piece(piece_desc(_Player,Piece),Piece).
```

```
piece_struct_name(piece(Name,_Owner),Name).
piece_struct_owner(piece(_Name,Owner),Owner).

piece_struct(piece(Name,Owner),Name,Owner).

owns(Piece,Owner) ==> piece_struct_owner(Piece,Owner).

square(square(X,Y),X,Y).

direction(dir(X,Y),X,Y).
```

B.2.6.2 Support Predicates

The following procedures define support routines which are used throughout the domain theory.

Player Roles *The two roles are* **player** *and* **opponent**. *These are opposite roles.*

```
player_role(player).
player_role(opponent).

opposite_role(player,opponent).
opposite_role(opponent,player).
```

A square is empty if it is a valid square in the current game, and if there is no piece currently on that square.

```
empty(Sq) ==>
     game:square(Sq),
     not on(P,Sq).
```

Transferring from one stage to the next means deleting the fact that the move is in one stage, and adding the fact it is in the next.

```
transfer_stage(Old,New) ==>
     del(stage(Old)),
     add(stage(New)).
```

B.3 Goals and End of Game

Symmetric chess-like games have three types of goals:

Arrival: Achieved when a piece matching a description has arrived on one of a set of square.

Eradicate: Achieved when no pieces matching a description are currently on the board (even though they may be in the hands of a player). This only applies after the **assignment** stage has passed, otherwise all such goals would be true at the start of games which have initial-assignment stages.

Stalemate: Achieved when a player is in control (on move) but has no legal moves.

Goals are evaluated immediately before each player makes a move. A game ends when either of the following is true:

1. Some player (possibly both) has achieved a goal, or
2. The game-specific maximum number of moves have been played.

When the game is over, the outcome is then based on which players have achieved one of their goals, or is a draw if the game-specific move limit has been exceeded.

```
game_over ==> goal_achieved(_Player).
game_over ==> exceeded_move_limit.

exceeded_move_limit ==>
     game:max_moves(L),
     true(move_count(L)).

game_outcome(draw) ==>
     exceeded_move_limit, !.
game_outcome(Outcome) ==>
     player_outcome(player,WinP),
     player_outcome(opponent,WinO),
     outcome(WinP,WinO,Outcome).
```

The goals for each player are checked separately, and the outcomes are combined to yield one of **player**, **opponent**, *or* **draw**. *Note that a player achieves a goal if* any *of the defined goals for that player are true.*

```
player_outcome(Player,Outcome) ==>
     if( goal_achieved(Player),
            Outcome = yes,
```

```
            Outcome = no ).

outcome(yes,yes,draw).
outcome(yes,no,player).
outcome(no,yes,opponent).

goal_achieved(Player) ==>
     game:player_has_goal(Player,Goal),
     goal_true(Goal).

goal_true(arrive(Descr,Squares))  ==>
     member(Sq,Squares),
     true(on(Piece,Sq)),
     matches(Descr,Piece).

goal_true(eradicate(Descr)) ==>
     not still_assigning,
     not exists(Descr).

goal_true(stalemate(Player)) ==>
     control(Player),
     not legal_move(_M,Player).

still_assigning ==>
     stage(assign),
     in_hand(_Piece,_Player).

exists(Descr) ==>
     true(on(Piece,Sq)),
     matches(Descr,Piece).
```

B.4 Board Topology

There are two types of boards in symmetric chess-like games, **planar** and **vertical-cylinder**. Both are two-dimensional grids, and in the cylindrical case the x-axis has wrap-around connectivity. These issues are discussed fully in the main text, but the implementation of these topologies in the Game Description Language is provided as a convenience to the reader.

connected(?S1,?S2,?Dir): *True when S1 and S2 are Squares, Dir is a direction-vector, Dir maps (directly) S1 to S2, given the board definition for the current game.*

```
connected(S1,S2,Dir) ==>
     square(S1,Xf,Yf),
     square(S2,Xt,Yt),
     direction(Dir,DX,DY),
     Xt1 is Xf + Dx,
     Yt1 is Yt + Dy,
     game:board_type(T),
     corresponds_for_type(T,(Xt1,Yt1),(Xt,Yt)),
     on_board((Xt,Yt)).

corresponds_for_type(planar,(Xt1,Yt1),(Xt,Yt)) ==>
     Xt1 = Xt,
     Yt1 = Yt.
corresponds_for_type(vertical_cylinder,(Xt1,Yt1),(Xt,Yt)) ==>
     game:board_size(XN,_YN),
     Xt is ( (Xt1 + XN - 1) mod XN ) + 1,
     Yt = Y1.

on_board(X,Y) ==>
    game:board_size(XMax,YMax),
    X >= 1,
    X =< XMax,
    Y >= 1,
    Y =< YMax.
```

B.5 Initial Setup

Before play of a symmetric chess-like game commences, it is necessary to determine the initial configuration, based on the definition of the current game. There are two[1] types of initial setup:

Arbitrary: the game specifies which of **player**'s pieces begin on which squares.

Decision: the game specifies the sets of pieces and squares, and both players begins either with their own or their opponent's pieces in hands, after which play begins in the **assign** stage as they place these pieces on valid squares.

In both cases, the initial state is first changed for **player**, and then for **opponent** (by calling the same setup routine but with control transferred).

[1]A third type of setup, *random-setup* is converted to an *abritrary* setup after the random decision is made each time the game is played. Thus it is declared in the game definition, but not in the domain theory.

```
start_game ==>
     initialize_state_properties,
     create_initial_setup,
     transfer_control,
     create_initial_setup.

initialize_state_properties ==>
     add(stage(assign)),
     add(control(player)),
     add(move_count(0)).

create_initial_setup ==>
     game:assignments(Assigns),
     do_assignments(Assigns).

do_assignments(As) ==>
     if( assignment_decision(As,Hand,PieceNames,_Squares),
         put_pieces_in_hand(PieceNames,Player),
         do_arbitrary_assignments(As) ).

put_pieces_in_hand(PieceNames,Hand) ==>
     control(Player),
     put_pieces_in_hand(PieceNames,Player,Hand).

put_pieces_in_hand([],_,_).
put_pieces_in_hand([Name|Names],Player,Hand) ==>
     put_piece_in_hand(Name,Player,Hand),
     put_pieces_in_hand(Names,Player,Hand).

put_piece_in_hand(PieceName,Player,Hand) ==>
     piece_struct(Piece,PieceName,Player),
     add(in_hand(Piece,Hand)).

do_arbitrary_assignments(As) ==>
     control(Player),
     place_pieces_on_squares(As,Player).

place_pieces_on_squares([],_).
place_pieces_on_squares([Name@Square|Assigns],Player) ==>
     piece_struct(Piece,Name,Player),
     add(on(Piece,Square)),
     place_pieces_on_squares(As,Player).
```

B.6 Global Symmetry and Inversion

As discussed in the main text, each game is defined from the perspective of **player** (white), and a specified global inversion is used to determine the definition of the game from the perspective of **opponent** (black). The two game definitions are then used indexically within the *gdl* interpreter, the white version provided whenever **player** calls a game-dependent routine, and the inverted version provided whenever black calls one.

The following Prolog code implements the global inversions.

invert(?Term1,?Term2,+Game): *True when Term2 is Term1 inverted to the perspective of* **opponent**, *given Game.*

```
invert(opponent,player,_) :- !.
invert(player,opponent,_) :- !.
invert(Term,Term2,Game) :-
     ( atom(Term) -> Term=Term2
     ; invert_struct(Term,Term2,Game) ).

invert_struct(square(X1,Y1),Sq,Game) :- !,
     invert_square(square(X1,Y1),Sq,Game).
invert_struct(dir(X1,Y1),Dir,Game) :- !,
     invert_dir(dir(X1,Y1),Dir,Game).
invert_struct(Pred,PredOut,Game) :-
     functor(Pred,F,N),
     functor(PredOut,F,N),
     invert_args(N,Pred,PredOut,Game).

invert_args(0,_,_,_) :- !.
invert_args(N,Pred,PredOut,Game) :-
     arg(N,Pred,A),
     invert(A,A1,Game),
     arg(N,PredOut,A1),
     N1 is N-1,
     invert_args(N1,Pred,PredOut,Game).

invert_square(Sq1,Sq,Game) :-
     board_size(XN,YN,Game),
     board_inversion(Inv,Game),
     invert_square_dim(Inv,XN,YN,Sq1,Sq).
```

Each game definition specifies whether to use **diagonal** *or* **forward** *inversion (described in the main text). These give different results for inverting squares and direction* *vectors, as a forward inversion reflects only about the X axis, while a diagonal inversion reflects about both axes.*

```
invert_square_dim(diagonal,XN,YN,square(X1,Y1),square(X2,Y2)) :-
     X2 is XN - X1 + 1,
     Y2 is YN - Y1 + 1.
invert_square_dim(forward,_XN,YN,square(X1,Y1),square(X2,Y2)) :-
     X1=X2,
     Y2 is YN - Y1 + 1.

invert_dir(dir(X1,Y1),dir(X2,Y2),Game) :-
     board_inversion(Inv,Game),
     inv_negate_dir(Inv,x,X1,X2),
     inv_negate_dir(Inv,y,Y1,Y2).

inv_negate_dir(diagonal,_Axis,X,XNeg) :- negates(X,XNeg).
inv_negate_dir(forward,Axis,X1,X2) :-
     ( Axis = y ->
       negates(X1,X2)
     ; X1 = X2 ).
```

Appendix C

Metagamer in Action

C.1 Metagamer Playing Checkers against Chinook

The following output is extracted from a trace of the checkers game between METAGAMER and Chinook as discussed in Section 17.3. The position is that after Chinook's eighth move, and METAGAMER finds a the non-losing move at its fourth-ply of search.

METAGAMER played with a move-time-limit of 60 seconds, while Chinook played with search depth 5 (its easiest level, responding almost instantly) and no opening or end-game databases. Chinook gave METAGAMER a piece handicap of one man. For more details of the game from which this sample was taken, see Section 17.3, page 166.

```
----------------------------------------
Clock times (in seconds):
<white>:        8.000 used,     99991.999 left
<black>:        393.000 used,   99606.999 left
----------------------------------------
To Chinook: I played g3f4
 g7f6
+---+---+---+---+---+---+---+---+
|   |*M*|   |*M*|   |*M*|   |*M*| 8
+---+---+---+---+---+---+---+---+
|*M*|   |...|   |...|   |...|   | 7
+---+---+---+---+---+---+---+---+
|   |...|   |*M*|   |...|   |*M*| 6
+---+---+---+---+---+---+---+---+
|...|   |*M*|   |*M*|   |...|   | 5
+---+---+---+---+---+---+---+---+
|   |...|   |...|   |...|   |...| 4
+---+---+---+---+---+---+---+---+
|...|   | M |   | M |   | M |   | 3
```

```
+---+---+---+---+---+---+---+---+
|   |...|   | M |   | M |   |...| 2
+---+---+---+---+---+---+---+---+
|...|   | M |   | M |   | M |   | 1
+---+---+---+---+---+---+---+---+
  a   b   c   d   e   f   g   h


Move Number: 15
Control: black
Notated Move played:
white man ( b , 2 ) -> ( c , 3 ) .


===============================================================

State checkpointed under index: <checkpoint426>
----------------------------------------
Clock times (in seconds):
<white>:        9.000 used,     99990.999 left
<black>:        393.000 used,   99606.999 left
----------------------------------------
Success: yes
Time is 2.209 sec.
Searching at depth <1>:
The best move found has a value of: -17.912220001220703
Number of nodes expanded: <1>
Number of nodes statically evaluated: <12>
Number of terminal nodes encountered: <0>
Number of terminal node tests:  <13>
The principal continuation here is:
black man ( a , 7 ) -> ( b , 6 ) .

Success: yes
Time is 4.869 sec.
Searching at depth <2>:
The best move found has a value of: -19.196252822875977
Number of nodes expanded: <13>
Number of nodes statically evaluated: <23>
Number of terminal nodes encountered: <0>
Number of terminal node tests:  <36>
The principal continuation here is:
```

```
black man ( a , 7 ) -> ( b , 6 ) .
white man ( e , 3 ) -> ( f , 4 ) .

Success: yes
Time is 22.340 sec.
Searching at depth <3>:
The best move found has a value of: -17.81200332641602
Number of nodes expanded: <34>
Number of nodes statically evaluated: <119>
Number of terminal nodes encountered: <0>
Number of terminal node tests:  <153>
The principal continuation here is:
black man ( a , 7 ) -> ( b , 6 ) .
white man ( e , 3 ) -> ( d , 4 ) .
black man ( c , 5 ) -> ( e , 3 ) x white man ( d , 4 ) .

Success: yes
Time is 30.561 sec.
Searching at depth <4>:
The best move found has a value of: -9.31275024414063
Number of nodes expanded: <79>
Number of nodes statically evaluated: <150>
Number of terminal nodes encountered: <0>
Number of terminal node tests:  <229>
The principal continuation here is:
black man ( f , 8 ) -> ( e , 7 ) .
white man ( e , 3 ) -> ( d , 4 ) .
black man ( c , 5 ) -> ( e , 3 ) x white man ( d , 4 ) .
white man ( f , 2 ) -> ( d , 4 ) x black man ( e , 3 ) ;
white man ( d , 4 ) -> ( f , 6 ) x black man ( e , 5 ) .

Success: yes
Time is 60.110 sec.
Success: yes
Time is 60.140 sec.

+---+---+---+---+---+---+---+---+
|   |*M*|   |*M*|   |...|   |*M*| 8
+---+---+---+---+---+---+---+---+
|*M*|   |...|   |*M*|   |...|   | 7
+---+---+---+---+---+---+---+---+
|   |...|   |*M*|   |...|   |*M*| 6
```

```
+---+---+---+---+---+---+---+---+
|...|   |*M*|   |*M*|   |...|   | 5
+---+---+---+---+---+---+---+---+
|   |...|   |...|   |...|   |...| 4
+---+---+---+---+---+---+---+---+
|...|   | M |   | M |   | M |   | 3
+---+---+---+---+---+---+---+---+
|   |...|   | M |   | M |   |...| 2
+---+---+---+---+---+---+---+---+
|...|   | M |   | M |   | M |   | 1
+---+---+---+---+---+---+---+---+
  a   b   c   d   e   f   g   h
```

```
Move Number: 16
Control: white
Notated Move played:
black man ( f , 8 ) -> ( e , 7 ) .
```

Appendix D

Game Definitions

This Appendix contains the game definitions for games discussed in the main thesis, represented as symmetric chess-like games. These listings are fully grammatical game definitions (according to the grammar presented in Appendix A.1), and are input and processed directly by the Metagame players.

Definitions of checkers and turncoat-chess are presented in full in the main text, and thus are not included here.

D.1 Generated Games in Tournament

The following games are those generated by the game generator, and played in the Metagame Tournament discussed in Chapter 16. Note that all symbols, including the piece names, are exactly as produced by the generator.[1]

D.1.1 Game: game1

```
game game1
goals stalemate opponent

board_size 6 by 6
board_type planar
promote_rank 6
setup
 decision player assigns
  {albino badger casket
   dumbo fairy handler}
  to {(a,1) (b,1) (c,1)
      (d,1) (e,1) (f,1)}
 end decision

DEFINE albino
 moving
    movement
     ride
     <3,3> symmetry all_symmetry
    end movement
 end moving
```

[1]The table of possible piece names used by the generator can be customised by the user. The current set was produced very late one night, and should not be taken as a statement about the psychological balance of the author.

```
 capturing
  capture by {clobber}
   type [{opponent}
         {badger casket jester}]
   effect player possesses
     movement
      hop before [x>=0]
          over [x=1] after [x<=6]
      hop_over [{opponent} any_piece]
      <1,1> symmetry all_symmetry
     end movement
  end capture
 end capturing
 promoting decision player
   options [{player} any_piece]
 end promoting
 constraints must_capture
end define

DEFINE badger
 moving
     movement
      leap
      <3,3> symmetry all_symmetry
     end movement
 end moving
 capturing
  capture by {retrieve clobber}
   type [{player} any_piece]
   effect player possesses
     movement
      hop before [x=0]
          over [x>=1] after [x>=0]
      hop_over [{player}
        {albino badger casket dumbo
         fairy handler jester lover}]
      <1,1> symmetry all_symmetry
     end movement
  end capture
 end capturing
 promoting decision opponent
   options [{opponent} any_piece]
 end promoting
 constraints must_capture
end define

DEFINE casket
 moving
     movement
      leap
      <2,2> symmetry all_symmetry
     end movement

     movement
      hop before [x>=0]
          over [x<=4] after [x>=0]
      hop_over [{opponent}
        {dumbo handler jester lover}]
      <2,0> symmetry all_symmetry
     end movement
 end moving
 capturing
  capture by {retrieve clobber}
   type [{player} any_piece]
   effect player possesses
     movement
      ride longest
      <2,0> symmetry all_symmetry
     end movement
  end capture
 end capturing
 promoting decision opponent
   options [any_player any_piece]
 end promoting
 constraints must_capture
             continue_captures
end define

DEFINE dumbo
 moving
     movement
      leap
      <1,0> symmetry all_symmetry
     end movement
```

```
 end moving
 capturing
  capture by {clobber}
   type [{opponent} {badger casket
          fairy jester morph}]
   effect remove
     movement
      ride
      <2,3> symmetry {forward side}
     end movement
  end capture
 end capturing
 promoting promote_to handler
 end promoting
 constraints must_capture
end define

DEFINE fairy
 moving
     movement
      leap
      <1,0> symmetry all_symmetry
     end movement
 end moving
 capturing
  capture by {clobber}
   type [{player} any_piece]
   effect remove
     movement
      hop before [x>=0]
         over [x>=1] after [x<=5]
      hop_over [{player} any_piece]
      <0,2> symmetry all_symmetry
     end movement

     movement
      leap
      <1,1> symmetry all_symmetry
     end movement
  end capture
 end capturing
 promoting promote_to albino
 end promoting
end define

DEFINE handler
 moving
     movement
      leap
      <0,1> symmetry all_symmetry
     end movement
 end moving
 capturing
  capture by {clobber}
   type [{opponent} {badger dumbo
          fairy jester morph}]
   effect remove
     movement
      ride max 4
      <1,1> symmetry all_symmetry
     end movement
  end capture
 end capturing
 promoting decision player
   options [any_player any_piece]
 end promoting
end define

DEFINE jester
 moving
     movement
      ride
      <2,3> symmetry all_symmetry
     end movement
 end moving
 capturing
  capture by {retrieve}
   type [{player} any_piece]
   effect player possesses
     movement
      leap
      <0,1> symmetry all_symmetry
     end movement
  end capture
```

```
 end capturing
 promoting promote_to lover
 end promoting
 constraints must_capture
end define

DEFINE lover
 moving
    movement
     hop before [x>=0]
        over [x=2] after [x>=0]
     hop_over
        [any_player any_piece]
     <1,1> symmetry all_symmetry
    end movement

    movement
     leap
     <2,0> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber hop}
   type [{opponent} any_piece]
   effect remove
    movement
     ride
     <1,0> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision player
   options [{player}
    {badger casket handler jester}]
 end promoting
 constraints continue_captures
end define

DEFINE morph
 moving
    movement
     ride max 3
     <1,1> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
   type [{player} any_piece]
   effect opponent possesses
    movement
     leap
     <1,0> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision player
   options [{player} any_piece]
 end promoting
 constraints must_capture
end define

end game .
```

D.1.2 Game: game2

```
game game2
goals stalemate player
 arrive [{player}
   {jerk llama monster plato}]
   at {(b,5)(c,5)}
 eradicate [{opponent} any_piece]
 arrive [{opponent}
   {drainer plato}] at {(e,5)}
 arrive [{opponent} any_piece]
     at {(a,2)}

board_size 6 by 6
board_type planar
inversion forward
promote_rank 6
setup
```

```
 decision opponent assigns
  {albino boy cleric
   drainer frenchman hooter} to
  {(a,1)(b,1)(c,1)(d,1)(e,1)(f,1)}
 end decision

DEFINE albino
 moving
    movement
     leap
     <0,1> symmetry all_symmetry
    end movement

    movement
     ride max 3
     <1,1> symmetry {forward}
    end movement

    movement
     ride
     <2,1> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
   type [{opponent} any_piece]
   effect remove
    movement
     ride
     <1,0> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting promote_to jerk
 end promoting
end define

DEFINE boy
 moving
    movement
     leap
     <0,1> symmetry all_symmetry
    end movement

    movement
     hop before [x=0]
        over [x<=3] after [x>=0]
     hop_over [any_player
         {albino drainer llama}]
     <2,3> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
   type [{player}
      {boy drainer frenchman
       jerk llama plato}]
   effect opponent possesses
    movement
     leap
     <1,1> symmetry all_symmetry
    end movement
  end capture

  capture by {clobber}
   type [{player} any_piece]
   effect remove
    movement
     hop before [x>=0]
        over [x<=2] after [x>=0]
     hop_over [{player} any_piece]
     <0,3> symmetry all_symmetry
    end movement

    movement
     ride
     <0,2> symmetry all_symmetry
    end movement

    movement
     leap
     <1,1> symmetry all_symmetry
    end movement
  end capture
```

```
 end capturing
 promoting promote_to albino
 end promoting
end define

DEFINE cleric
 moving
    movement
     ride max 2
     <0,1> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber hop}
   type [{opponent}
         {albino boy cleric
          drainer hooter jerk
          llama monster}]
   effect opponent possesses
    movement
     ride
     <1,1> symmetry {forward side}
    end movement

    movement
     hop before [x>=0]
         over [x<=1] after [x<=1]
     hop_over
         [any_player any_piece]
     <2,3> symmetry {forward side}
    end movement
  end capture
 end capturing
 promoting decision opponent
   options [any_player any_piece]
 end promoting
end define

DEFINE drainer
 moving
    movement
     leap
     <0,1> symmetry {forward side}
    end movement
end moving
capturing
 capture by {clobber hop}
  type [{player} {llama plato}]
  effect opponent possesses
    movement
     ride
     <1,1> symmetry all_symmetry
    end movement
 end capture

 capture by {hop}
  type [{player}
        {albino boy cleric
         hooter llama plato}]
  effect remove
    movement
     leap
     <1,0> symmetry all_symmetry
    end movement

    movement
     ride max 4 longest
     <1,1> symmetry all_symmetry
    end movement
 end capture

 capture by {clobber hop}
  type [{opponent} {albino
        drainer frenchman hooter}]
  effect opponent possesses
    movement
     ride
     <1,1> symmetry {forward side}
    end movement

    movement
     leap
     <1,1> symmetry all_symmetry
    end movement
```

```
  end capture
 end capturing
 promoting decision opponent
  options [any_player any_piece]
 end promoting
 constraints must_capture
end define

DEFINE frenchman
 moving
    movement
     ride
     <1,0> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber hop}
   type [{player} any_piece]
   effect remove
    movement
     leap
     <1,1> symmetry all_symmetry
    end movement
  end capture

  capture by {clobber hop}
   type [{opponent} any_piece]
   effect remove
    movement
     ride
     <2,2> symmetry all_symmetry
    end movement
  end capture

  capture by {clobber hop}
   type [{player} any_piece]
   effect remove
    movement
     ride
     <0,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision player
  options [{player} any_piece]
 end promoting
end define

DEFINE hooter
 moving
    movement
     leap
     <2,0> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {retrieve clobber}
   type [{opponent} any_piece]
   effect player possesses
    movement
     ride
     <3,0> symmetry all_symmetry
    end movement

    movement
     ride
     <0,1> symmetry all_symmetry
    end movement
  end capture

  capture by {clobber hop}
   type [{opponent}
       {albino boy cleric drainer
        frenchman hooter monster}]
   effect player possesses
    movement
     leap
     <1,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision opponent
  options [{player} any_piece]
 end promoting
```

```
 constraints must_capture
end define

DEFINE jerk
 moving
    movement
     leap
     <1,0> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
   type [any_player
        {cleric frenchman}]
   effect player possesses
    movement
     leap
     <2,0> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision player
   options [{player} any_piece]
 end promoting
end define

DEFINE llama
 moving
    movement
     hop before [x>=0]
        over [x<=3] after [x>=0]
     hop_over [{player}
        {albino boy frenchman
         jerk monster plato}]
     <2,3> symmetry {forward side}
    end movement
 end moving
 capturing
  capture by {clobber hop}
   type [{player} {albino cleric
        drainer hooter monster}]
   effect remove
    movement
     ride
     <1,2> symmetry all_symmetry
    end movement

    movement
     leap
     <1,0> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision player
   options [{player} any_piece]
 end promoting
end define

DEFINE monster
 moving
    movement
     ride max 2
     <1,0> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
   type [any_player {albino boy
        hooter jerk llama plato}]
   effect remove
    movement
     leap
     <0,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision opponent
   options [{player} {jerk}]
 end promoting
end define

DEFINE plato
 moving
    movement
```

```
    leap
    <1,0> symmetry all_symmetry
   end movement
 end moving
 capturing
  capture by {}
   type [{opponent} any_piece]
   effect opponent possesses
    movement
    leap
    <0,1> symmetry all_symmetry
   end movement
  end capture
 end capturing
 promoting promote_to frenchman
 end promoting
end define

end game .
```

D.1.3 Game: game3

```
game game3
goals stalemate opponent
  arrive [{player}
        {dumbo fairy heaven}]
      at {(e,5)}
  eradicate [{opponent}
     {albino bear
      cheeseman dumbo heaven}]
  arrive [{opponent} any_piece]
      at {(c,4)(c,3)(a,2)}
  arrive [{opponent} {dumbo}] at {(b,1)}

board_size 6 by 5
board_type planar
inversion forward
promote_rank 5
setup
  albino at {(a,1)}
  bear at {(e,1)}
  cheeseman at {(c,1) (d,1)}

DEFINE albino
 moving
    movement
     leap
     <1,0> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
   type [{opponent} any_piece]
   effect opponent possesses
    movement
     ride longest
     <1,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision opponent
  options [{opponent} any_piece]
 end promoting
 constraints must_capture
end define

DEFINE bear
 moving
    movement
     leap
     <1,0> symmetry all_symmetry
    end movement

    movement
     leap
     <0,2> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber hop}
```

```
  type [{player} {albino bear fairy}]
  effect opponent possesses
    movement
     hop before [x>=0]
         over [x=1] after [x>=0]
     hop_over [{player} any_piece]
     <2,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision opponent
  options [any_player
          {albino bear cheeseman
           dumbo fairy}]
 end promoting
 constraints must_capture
end define
```

DEFINE **cheeseman**

```
 moving
    movement
     leap
     <1,0> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
  type [{opponent}
      {albino bear dumbo heaven}]
  effect remove
    movement
     leap
     <2,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting promote_to heaven
 end promoting
end define
```

DEFINE **dumbo**

```
 moving
    movement
     ride max 4 longest
     <1,1> symmetry all_symmetry
    end movement

    movement
     ride max 3
     <1,1> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
   type [{player} any_piece]
   effect remove
    movement
     hop before [x=0]
         over [x=2] after [x>=2]
     hop_over [{opponent} {albino}]
     <1,0> symmetry all_symmetry
    end movement

    movement
     hop before [x>=0]
         over [x>=1] after [x>=0]
     hop_over
         [any_player any_piece]
     <2,1> symmetry all_symmetry
    end movement

    movement
     ride max 4
     <0,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision player
   options [{player} any_piece]
 end promoting
end define
```

DEFINE **fairy**

```
 moving
```

```
    movement
     ride longest
     <2,0> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {hop}
   type [any_player any_piece]
   effect remove
    movement
     ride max 3
     <0,1> symmetry {forward side}
    end movement
  end capture
 end capturing
 promoting promote_to albino
 end promoting
end define

DEFINE heaven
 moving
    movement
     leap
     <1,1> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber hop}
   type [{player} any_piece]
   effect remove
    movement
     hop before [x>=0]
         over [x>=1] after [x>=0]
     hop_over [{player}
        {albino bear dumbo heaven}]
     <1,1> symmetry all_symmetry
    end movement

    movement
     leap
     <1,0> symmetry {forward side}
    end movement

    movement
     ride max 2
     <1,1> symmetry all_symmetry
    end movement
  end capture

  capture by {clobber}
   type [{opponent} {albino bear
         cheeseman dumbo fairy}]
   effect remove
    movement
     hop before [x=1]
         over [x>=1] after [x>=0]
     hop_over
         [any_player any_piece]
     <0,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision player
   options [{player} {albino fairy}]
 end promoting
 constraints must_capture
end define

end game .
```

D.1.4 Game: game4

```
game game4
goals stalemate opponent
  eradicate [{opponent}
          {andover clinton}]
  arrive [{opponent} any_piece]
    at {(b,3)}

board_size 5 by 6
```

```
board_type planar
inversion forward
promote_rank 6
setup
  andover at {(d,1)}
  bishop at {(a,1)}
  clinton at {(b,1)}
  dumbo at {(e,1)}
  firefly at {(c,1)}

DEFINE andover
 moving
    movement
     ride
     <1,3> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {retrieve clobber hop}
   type [{player} {firefly}]
   effect remove
    movement
     hop before [x<=4]
        over [x>=1] after [x>=0]
     hop_over
        [any_player any_piece]
     <2,0> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision player
   options [any_player any_piece]
 end promoting
end define

DEFINE bishop
 moving
    movement
     leap
     <1,1> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {retrieve clobber hop}
   type [{opponent} any_piece]
   effect opponent possesses
    movement
     leap
     <0,2> symmetry all_symmetry
    end movement
  end capture

  capture by {clobber}
   type [any_player {jupiter lover}]
   effect remove
    movement
     leap
     <0,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision player
   options [{opponent} any_piece]
 end promoting
 constraints must_capture
end define

DEFINE clinton
 moving
    movement
     ride
     <1,0> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
   type [{player} {andover jupiter}]
   effect remove
    movement
     hop before [x>=0]
        over [x<=6] after [x>=0]
     hop_over [{opponent}
        {bishop firefly handler}]
     <1,2> symmetry all_symmetry
```

```
   end movement
  end capture
 end capturing
 promoting promote_to firefly
 end promoting
end define

DEFINE dumbo
 moving
    movement
     ride max 4
     <0,1> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
   type [{player} any_piece]
   effect opponent possesses
    movement
     leap
     <1,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision player
   options [{player} any_piece]
 end promoting
 constraints must_capture
end define

DEFINE firefly
 moving
    movement
     ride
     <1,2> symmetry {forward side}
    end movement
 end moving
 capturing
  capture by {clobber hop}
   type [{player} any_piece]
   effect opponent possesses
    movement
     ride
     <2,3> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting promote_to lover
 end promoting
 constraints must_capture
end define

DEFINE handler
 moving
    movement
     ride
     <3,2> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber hop}
   type [{player}
    {andover bishop clinton firefly
    handler jupiter lover}]
   effect opponent possesses
    movement
     leap
     <3,1> symmetry all_symmetry
    end movement

    movement
     ride max 4
     <0,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting promote_to lover
 end promoting
end define

DEFINE jupiter
 moving
    movement
     ride
```

```
    <2,1> symmetry all_symmetry
   end movement
 end moving
 capturing
  capture by {clobber}
   type [{player} any_piece]
   effect player possesses
     movement
      ride max 3
      <1,0> symmetry all_symmetry
     end movement
  end capture
 end capturing
 promoting promote_to andover
 end promoting
 constraints continue_captures
end define

DEFINE lover
 moving
     movement
      hop before [x=0]
          over [x<=2] after [x>=0]
      hop_over [{player} any_piece]
      <2,1> symmetry all_symmetry
     end movement
 end moving
 capturing
  capture by {retrieve clobber hop}
   type [{player}
     {bishop firefly handler lover}]
   effect player possesses
     movement
      leap
      <0,2> symmetry all_symmetry
     end movement

     movement
      leap
      <1,1> symmetry all_symmetry
     end movement

     movement
      ride
      <1,0> symmetry all_symmetry
     end movement

     movement
      leap
      <1,1> symmetry all_symmetry
     end movement
  end capture
 end capturing
 promoting decision opponent
  options [{opponent}
    {bishop clinton dumbo handler}]
 end promoting
 constraints continue_captures
end define

end game .
```

D.1.5 Game: game5

```
game game5
goals stalemate player
  arrive [{player} {frenchman heaven}]
     at {(f,2)}
  arrive [{player} {heaven}] at {(d,2)}
  eradicate [{opponent}
     {aardvark berkeley christ
      digger frenchman}]
  arrive [{opponent} any_piece]
     at {(e,4)}

board_size 6 by 6
board_type planar
promote_rank 6
setup
  aardvark at {(f,1)(e,1)(c,1)}
```

```
 berkeley at {(d,1)}
 christ at {(b,1)}
 digger at {(a,1)}

DEFINE aardvark
 moving
    movement
     ride
     <1,1> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {retrieve clobber hop}
   type [any_player {berkeley}]
   effect remove
    movement
     ride max 2
     <1,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision opponent
   options [{opponent} any_piece]
 end promoting
 constraints must_capture
end define

DEFINE berkeley
 moving
    movement
     ride max 2
     <1,1> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber hop}
   type [{player}
        {berkeley digger}]
   effect remove
    movement
     leap
     <2,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
 promoting decision player
   options [{opponent}
          {berkeley christ}]
 end promoting
 constraints must_capture
end define

DEFINE christ
 moving
    movement
     ride
     <1,1> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
   type [{opponent} any_piece]
   effect remove
    movement
     leap
     <0,1> symmetry {forward side}
    end movement
  end capture
 end capturing
 promoting decision opponent
   options [{opponent}
      {berkeley frenchman heaven}]
 end promoting
end define

DEFINE digger
 moving
    movement
     ride longest
     <1,0> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber hop}
```

```
   type [{player} {digger}]
   effect player possesses
     movement
      leap
      <0,3> symmetry all_symmetry
     end movement
  end capture
 end capturing
 promoting promote_to berkeley
 end promoting
end define

DEFINE frenchman
 moving
     movement
      ride max 2
      <1,0> symmetry {forward side}
     end movement
 end moving
 capturing
  capture by {hop}
   type [{player} any_piece]
   effect player possesses
     movement
      ride max 4
      <0,1> symmetry all_symmetry
     end movement

     movement
      hop before [x>=0]
          over [x<=6] after [x>=0]
      hop_over
          [{opponent} any_piece]
      <1,1> symmetry all_symmetry
     end movement
  end capture
 end capturing
 promoting decision opponent
   options [{player} any_piece]
 end promoting
end define

DEFINE heaven
 moving
     movement
      ride
      <0,2> symmetry all_symmetry
     end movement
 end moving
 capturing
  capture by {clobber}
   type [{player} any_piece]
   effect remove
     movement
      leap
      <1,1> symmetry all_symmetry
     end movement
  end capture

  capture by {clobber}
   type [{player} any_piece]
   effect remove
     movement
      ride
      <1,3> symmetry all_symmetry
     end movement
  end capture
 end capturing
 promoting promote_to christ
 end promoting
end define

end game .
```

D.2 Known Games

The following games are encodings of some known games as symmetric chess-like games which were not already listed in the main text.

D.2.1 Game: tic-tac-toe

This encoding of noughts and crosses (tic tac toe) was discussed in Section 9.2.1.2.

```
game tic_tac_toe
goals stalemate opponent
  stalemate player
  arrive [{player} {win}] at
          {(a,7) (b,7) (c,7)
           (a,6) (b,6) (c,6)
           (a,5) (b,5) (c,5)}

board_size 3 by 11
board_type planar
promote_rank 4
setup
  man at   {(a,1) (b,1) (c,1)
           (a,2) (b,2) (c,2)
           (a,3) (b,3) (c,3)}
  dummy at {(a,4) (b,4) (c,4)}
 constraints must_capture

DEFINE on_board
 capturing
  capture by {clobber}
   type [{player} {on_board}]
   effect remove
     movement
      ride min 2 max 2
      <1,0> symmetry {rotation}
     end movement

     movement
      ride min 2 max 2
      <1,1> symmetry {side}
     end movement
  end capture

  capture by {retrieve}
   type [{player} {on_board}]
   effect remove
     movement
      leap
      <1,0> symmetry all_symmetry
     end movement

     movement
      leap
      <1,1> symmetry all_symmetry
     end movement
  end capture
 end capturing
 promoting promote_to win
 end promoting
end define

DEFINE man
 moving
     movement
      leap
      <0,4> symmetry {}
     end movement
 end moving
 promoting promote_to on_board
 end promoting
end define

end game .
```

D.2.2 Game: chess

This is the encoding of chess used in the contests in Section 17.4. While the initial position in this encoding is the same as that in chess proper, the initial position used in the competitions with GnuChess was modified and randomised to eliminate situations in which double-pawn moves or castling could occur, as these rules are not covered in this encoding of the game. Note that in this version of chess, it is legal to leave a king in check, and a player wins by capturing the opponent's king. Also, the notion of *stalemate* here is that used in symmetric chess-like games: a player is stalemated when he starts a turn having no legal moves. As it is legal here to move a king into check, this representation does not have draws by stalemate as does chess proper.

```
game chess
goals stalemate opponent
  eradicate [{opponent} {king}]

board_size 8 by 8
board_type planar
inversion forward
promote_rank 8
setup
  pawn at {(a,2) (b,2) (c,2) (d,2)
          (e,2) (f,2) (g,2) (h,2)}
  night at {(b,1) (g,1)}
  bishop at {(c,1) (f,1)}
  rook at {(a,1) (h,1)}
  queen at {(d,1)}
  king at {(e,1)}

DEFINE pawn
 moving
    movement
     leap
     <0,1> symmetry {}
    end movement
 end moving
 capturing
  capture by {clobber}
   type [{opponent} any_piece]
   effect remove
    movement
     leap
     <1,1> symmetry {side}
    end movement
  end capture
 end capturing
 promoting decision player
   options [{player}
     {night bishop rook queen}]
 end promoting
end define

DEFINE night
 moving
    movement
     leap
     <2,1> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
   type [{opponent} any_piece]
   effect remove
    movement
     leap
     <2,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
end define

DEFINE bishop
 moving
    movement
     ride
     <1,1> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
   type [{opponent} any_piece]
   effect remove
    movement
     ride
```

```
   <1,1> symmetry all_symmetry
   end movement
  end capture
 end capturing
end define

DEFINE rook
 moving
   movement
    ride
    <1,0> symmetry all_symmetry
   end movement
 end moving
 capturing
  capture by {clobber}
   type [{opponent} any_piece]
   effect remove
   movement
    ride
    <1,0> symmetry all_symmetry
   end movement
  end capture
 end capturing
end define

DEFINE queen
 moving
   movement
    ride
    <1,0> symmetry all_symmetry
   end movement

   movement
    ride
    <1,1> symmetry all_symmetry
   end movement
 end moving
 capturing
  capture by {clobber}
   type [{opponent} any_piece]
   effect remove
   movement
    ride
    <1,0> symmetry all_symmetry
   end movement

   movement
    ride
    <1,1> symmetry all_symmetry
   end movement
  end capture
 end capturing
end define

DEFINE king
 moving
   movement
    leap
    <1,0> symmetry all_symmetry
   end movement

   movement
    leap
    <1,1> symmetry all_symmetry
   end movement
 end moving
 capturing
  capture by {clobber}
   type [{opponent} any_piece]
   effect remove
   movement
    leap
    <1,0> symmetry all_symmetry
   end movement

   movement
    leap
    <1,1> symmetry all_symmetry
   end movement
  end capture
 end capturing
end define

end game .
```

D.2.3 Game: fairy

This is an encoding of the definitions of some fairy-chess pieces, which are mentioned in Section 15.3.4. Section 15.6 shows the material values METAGAMER assigns to these pieces, for purposes of comparison with the ordinary chess pieces.

```
DEFINE nightbishop
 moving
    movement
     leap
     <2,1> symmetry all_symmetry
    end movement

    movement
     ride
     <1,1> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
   type [{opponent} any_piece]
   effect remove
    movement
     leap
     <2,1> symmetry all_symmetry
    end movement

    movement
     ride
     <1,1> symmetry all_symmetry
    end movement
  end capture
 end capturing
end define

DEFINE nightrook
 moving
    movement
     leap
     <2,1> symmetry all_symmetry
    end movement

    movement
     ride
     <1,0> symmetry all_symmetry
    end movement
 end moving
 capturing
  capture by {clobber}
   type [{opponent} any_piece]
   effect remove
    movement
     leap
     <2,1> symmetry all_symmetry
    end movement

    movement
     ride
     <1,0> symmetry all_symmetry
    end movement
  end capture
 end capturing
end define

DEFINE nightqueen
 moving
    movement
     leap
     <2,1> symmetry all_symmetry
    end movement

    movement
     ride
     <1,0> symmetry all_symmetry
    end movement

    movement
     ride
     <1,1> symmetry all_symmetry
    end movement
```

```
 end moving
 capturing
  capture by {clobber}
   type [{opponent} any_piece]
   effect remove
     movement
      leap
      <2,1> symmetry all_symmetry
     end movement

     movement
      ride
      <1,0> symmetry all_symmetry
     end movement

     movement
      ride
      <1,1> symmetry all_symmetry
     end movement
  end capture
 end capturing
end define


DEFINE ortholeap
 moving
     movement
      leap
      <1,0> symmetry all_symmetry
     end movement
 end moving
 capturing
  capture by {clobber}
   type [{opponent} any_piece]
   effect remove
     movement
      leap
      <1,0> symmetry all_symmetry
     end movement
  end capture
 end capturing
end define

DEFINE diagleap
 moving
     movement
      leap
      <1,1> symmetry all_symmetry
     end movement
 end moving
 capturing
  capture by {clobber}
   type [{opponent} any_piece]
   effect remove
     movement
      leap
      <1,1> symmetry all_symmetry
     end movement
  end capture
 end capturing
end define
```

D.2.4 Game: knight-zone

This is part of an encoding of the first version of knight-zone chess with Rule 1. This game was discussed in Section 3.2.2.2 on page 18. The only difference between this game and chess is that the board is extended by two on all sides, and other pieces are restricted from landing on the outside squares except for the knight. This restriction is represented as a goal in which the opponent wins if a player ever puts a piece other than a knight in that zone. The piece definitions are exactly the same as those in the chess encoding above, so they have been omitted here. Section 15.6 shows the material values METAGAMER assigns to each piece in this game based on its analysis of these rules.

```
game knight-zone-chess
goals stalemate opponent
  eradicate [{opponent} {king}]
  arrive [{opponent}
    {pawn bishop rook queen king}] at
   {(a,1) (b,1) (c,1) (d,1) (e,1) (f,1)
    (g,1) (h,1) (i,1) (j,1) (k,1) (l,1)
    (a,2) (b,2) (c,2) (d,2) (e,2) (f,2)
    (g,2) (h,2) (i,2) (j,2) (k,2) (l,2)
    (a,3) (b,3) (k,3) (l,3)
    (a,4) (b,4) (k,4) (l,4)
    (a,5) (b,5) (k,5) (l,5)
    (a,6) (b,6) (k,6) (l,6)
    (a,7) (b,7) (k,7) (l,7)
    (a,8) (b,8) (k,8) (l,8)
    (a,9) (b,9) (k,9) (l,9)
    (a,10) (b,10) (k,10) (l,10)
    (a,11) (b,11) (c,11) (d,11) (e,11) (f,11)
    (g,11) (h,11) (i,11) (j,11) (k,11) (l,11)
    (a,12) (b,12) (c,12) (d,12) (e,12) (f,12)
    (g,12) (h,12) (i,12) (j,12) (k,12) (l,12)}

board_size 12 by 12
board_type planar
promote_rank 12
setup
  pawn at {(i,4) (j,4) (c,4) (d,4)
         (e,4) (f,4) (g,4) (h,4)}
  night at {(d,3) (i,3)}
  bishop at {(e,3) (h,3)}
  rook at {(c,3) (j,3)}
  queen at {(f,3)}
  king at {(g,3)}
```

D.2.5 Games in Main Text

The rules of checkers are listed as Figure 7.3 on page 59. The rules of turncoat-chess are listed as Figure 8.2 and Figure 8.3 on page 68.

Appendix E

Experimental Results

This appendix contains the full results for the experiments in Chapter 16.

The results of the competition on each game are presented in Table E.1–Table E.5. Table E.6 summarises the results of the competition between each pair of players summed across all 5 games. Table E.7 summarises the results of each player on each game. To determine the total score for a player in a given table, players were awarded 1 point for a win, 0.5 points for a draw, and 0 points for a loss on each contest. Due to space considerations, the number of losses are not listed explicitly on the tables. Each pair of players played 20 contests on each game, so between a pair of players, $L = 20 - (W + D)$. Similarly, to find the number of contests lost by a player on a given game, $L = 100 - (W + D)$, and for the tournament, $L = 500 - (W + D)$.

For full details of the experiment, see Chapter 16.

Contest Results: Game 1															
Player	Opponent												Σ win	Σ draw	Total score
	1		2		3		4		5		6				
	W	D	W	D	W	D	W	D	W	D	W	D			
1	—	—	1	12	3	11	1	9	0	13	0	10	5	55	32.5
2	7	12	—	—	6	11	2	11	2	10	0	7	17	51	42.5
3	6	11	3	11	—	—	1	12	2	7	3	9	15	50	40.0
4	10	9	7	11	7	12	—	—	1	15	0	15	25	62	56.0
5	7	13	8	10	11	7	4	15	—	—	0	15	30	60	60.0
6	10	10	13	7	8	9	5	15	5	15	—	—	41	56	69.0

Table E.1: Results of tournament on Game 1.

Contest Results: Game 2															
Player	Opponent												Σ win	Σ draw	Total score
	1		2		3		4		5		6				
	W	D	W	D	W	D	W	D	W	D	W	D			
1	—	—	0	1	0	0	0	0	0	0	1	0	1	1	1.5
2	19	1	—	—	11	0	14	2	7	0	8	1	59	4	61.0
3	20	0	9	0	—	—	15	0	11	0	14	4	69	4	71.0
4	20	0	4	2	5	0	—	—	7	0	6	2	42	4	44.0
5	20	0	13	0	9	0	13	0	—	—	9	2	64	2	65.0
6	19	0	11	1	2	4	12	2	9	2	—	—	53	9	57.5

Table E.2: Results of tournament on Game 2.

Contest Results: Game 3															
Player	Opponent 1		2		3		4		5		6		Σ win	Σ draw	Total score
	W	D	W	D	W	D	W	D	W	D	W	D			
1	—	—	0	0	0	0	0	0	0	0	0	0	0	0	0.0
2	20	0	—	—	11	2	9	1	4	0	4	1	48	4	50.0
3	20	0	7	2	—	—	16	2	0	5	1	5	44	14	51.0
4	20	0	10	1	2	2	—	—	8	0	0	1	40	4	42.0
5	20	0	16	0	15	5	12	0	—	—	2	13	65	18	74.0
6	20	0	15	1	14	5	19	1	5	13	—	—	73	20	83.0

Table E.3: Results of tournament on Game 3.

Contest Results: Game 4															
Player	Opponent 1		2		3		4		5		6		Σ win	Σ draw	Total score
	W	D	W	D	W	D	W	D	W	D	W	D			
1	—	—	0	0	1	1	1	0	0	0	0	0	2	1	2.5
2	20	0	—	—	12	8	8	6	3	15	1	11	44	40	64.0
3	18	1	0	8	—	—	5	0	3	0	0	6	26	15	33.5
4	19	0	6	6	15	0	—	—	16	4	0	13	56	23	67.5
5	20	0	2	15	17	0	0	4	—	—	0	13	39	32	55.5
6	20	0	8	11	14	6	7	13	7	13	—	—	56	43	77.5

Table E.4: Results of tournament on Game 4.

Contest Results: Game 5															
Player	Opponent												Σ win	Σ draw	Total score
	1		2		3		4		5		6				
	W	D	W	D	W	D	W	D	W	D	W	D			
1	—	—	0	0	0	0	0	0	0	1	0	1	0	2	1.0
2	20	0	—	—	6	3	0	8	0	8	0	10	26	29	40.5
3	20	0	11	3	—	—	0	8	0	11	0	8	31	30	46.0
4	20	0	12	8	12	8	—	—	3	17	0	20	47	53	73.5
5	19	1	12	8	9	11	0	17	—	—	0	18	40	55	67.5
6	19	1	10	10	12	8	0	20	2	18	—	—	43	57	71.5

Table E.5: Results of tournament on Game 5.

Overall Results: by Opponent															
Player	Opponent												Σ win	Σ draw	Total score
	1		2		3		4		5		6				
	W	D	W	D	W	D	W	D	W	D	W	D			
1	—	—	1	13	4	12	2	9	0	14	1	11	8	59	37.5
2	86	13	—	—	46	24	33	28	16	33	13	30	194	128	258.0
3	84	12	30	24	—	—	37	22	16	23	18	32	185	113	241.5
4	89	9	39	28	41	22	—	—	35	36	6	51	210	146	283.0
5	86	14	51	33	61	23	29	36	—	—	11	61	238	167	321.5
6	88	11	57	30	50	32	43	51	28	61	—	—	266	185	358.5

Table E.6: Overall results of tournament against each opponent.

Overall Results: by Game													
Player	Game										Σ win	Σ draw	Total score
	1		2		3		4		5				
	W	D	W	D	W	D	W	D	W	D			
1	5	55	1	1	0	0	2	1	0	2	8	59	37.5
2	17	51	59	4	48	4	44	40	26	29	194	128	258.0
3	15	50	69	4	44	14	26	15	31	30	185	113	241.5
4	25	62	42	4	40	4	56	23	47	53	210	146	283.0
5	30	60	64	2	65	18	39	32	40	55	238	167	321.5
6	41	56	53	9	73	20	56	43	43	57	266	185	358.5

Table E.7: Overall results of tournament on each game.

Bibliography

[Abramson, 1990] Bruce Abramson. Expected-outcome: A general model of static evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2), February 1990.

[Aho *et al.*, 1983] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.

[Allis *et al.*, 1991a] L.V. Allis, H.J. van den Herik, and I.S. Herschberg. Which games will survive. In D.N.L. Levy and D.F. Beal, editors, *Heuristic Programming in Artificial Intelligence 2 – The Second Computer Olympiad*. Ellis Horwood, 1991.

[Allis *et al.*, 1991b] L.V. Allis, M. van der Meulen, and H.J. van den Herik. Databases in awari. In D.N.L. Levy and D.F. Beal, editors, *Heuristic Programming in Artificial Intelligence 2 – The Second Computer Olympiad*. Ellis Horwood, 1991.

[Allis, 1992] Victor Allis. Qubic solved again. In H.J. van den Herik and L.V. Allis, editors, *Heuristic Programming in Artificial Intelligence 3 – The Third Computer Olympiad*, pages 192–204. Ellis Horwood, 1992.

[Angeline and Pollack, 1993] P. Angeline and J. Pollack. Competitive environments evolve better solutions for complex tasks. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann, 1993.

[Axelrod, 1987] Robert Axelrod. The evolution of strategies in the iterated prisoner's dilemma. In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing*. Pitman, 1987.

[Banerji and Ernst, 1971] R. B. Banerji and G.W. Ernst. Changes of representation which preserve strategies in games. In *Proceedings of the Second International Joint Conference on AI*, 1971.

[Banks, 1988] Iain M. Banks. *The Player of Games*. Macmillan (London) Ltd, 1988.

[Baum and Smith, 1993] Eric B. Baum and Warren D. Smith. Best play for imperfect players and game tree search. In *Proceedings of the AAAI Fall Symposium on Games: Planning and Learning*. AAAI Press, 1993. To Appear.

[Bell, 1969] R.C. Bell. *Board and Table Games from Many Civilizations.* Oxford University Press, 1969.

[Benjamin, 1990] D. Paul Benjamin, editor. *Change of Representation and Inductive Bias.* Kluwer, 1990.

[Berlekamp *et al.*, 1982] E.R. Berlekamp, J.H. Conway, and R.K. Guy. *Winning Ways for your mathematical plays.* Academic Press, 1982.

[Berliner, 1974] Hans Berliner. *Chess as Problem Solving: The Developments of a Tactics Analyzer.* PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, 1974.

[Berliner, 1978] Hans Berliner. A chronology of computer chess and its literature. *Artificial Intelligence*, 10:201–214, 1978.

[Berliner, 1984] Hans Berliner. Search vs knowledge: An analysis from the domain of games. In Elithorn and Banerji, editors, *Artificial and Human Intelligence.* Elsevier Science Publishers, New York, 1984.

[Botvinnik, 1970] M. M. Botvinnik. *Computers, chess and long-range planning.* Springer-Verlag New York, Inc., 1970.

[Bratko, 1986] Ivan Bratko. *Prolog Programming for Artificial Intelligence.* Addison Wesley, 1986.

[Callan and Utgoff, 1991] J. P. Callan and P.E. Utgoff. Constructive induction on domain knowledge. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 614–619, 1991.

[Callan *et al.*, 1991] James P. Callan, Tom E. Fawcett, and Edwina L. Rissland. Adaptive case-based reasoning. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, 1991.

[Callan, 1993] James P. Callan. *Knowledge-Based Feature Generation for Inductive Learning.* PhD thesis, Department of Computer and Information Science, University of Massachusetts, January 1993.

[Campbell, 1988] Murray S. Campbell. *Chunking as an Abstraction Mechanism.* PhD thesis, Carnegie Mellon University, 1988.

[Chaitin, 1987] Gregory J. Chaitin. *Algorithmic information theory.* Cambridge University Press, Cambridge, 1987.

[Chomsky, 1965] Noam Chomsky. *Aspects of the Theory of Syntax.* MIT Press, Boston, 1965.

[Church and Church, 1979] Russell M. Church and Kenneth W. Church. Plans, goals, and search strategies for the selection of a move in chess. In Peter W. Frey, editor, *Chess Skill in Man and Machine*. Springer-Verlag, 1979.

[Collins and Birnbaum, 1988] Gregg Collins and Lawrence Birnbaum. Learning strategic concepts in competitive planning: An explanation-based approach to the transfer of knowledge across domains. Technical Report UIUCDCS-R-88-1443, Univesrity of Illinois, Dept. of Computer Science, Urbana, IL, 1988.

[Collins *et al.*, 1991] Gregg Collins, Lawrence Birnbaum, Bruce Krulwich, and Michael Freed. Plan debugging in an intentional system. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, 1991.

[Collins, 1987] Gregg Collins. *Plan Creation: Using Strategies as Blueprints*. PhD thesis, Yale University, Department of Computer Science, 1987.

[Cousot and Cousot, 1977] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, 1977.

[Cousot and Cousot, 1992] Patrick Cousot and Radhia Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[de Grey, 1985] Aubrey de Grey. Towards a versatile self-learning board game program. Final Project, Tripos in Computer Science, University of Cambridge, 1985.

[Dickins, 1971] Anthony Dickins. *A Guide to Fairy Chess*. Dover, 1971.

[Donninger, 1992] Ch. Donninger. The relation of mobility, strategy and the mean dead rabbit in chess. In H.J. van den Herik and L.V. Allis, editors, *Heuristic Programming in Artificial Intelligence 3 – The Third Computer Olympiad*. Ellis Horwood, 1992.

[Ebeling, 1986] C. Ebeling. *All the Right Moves: A VLSI Architecture for Chess*. PhD thesis, Carnegie-Mellon University, 1986.

[Epstein, 1989a] Susan Epstein. Mediation among Advisors. In *Proceedings on AI and Limited Rationality*, pages 35–39. AAAI Spring Symposium Series, 1989.

[Epstein, 1989b] Susan Epstein. The Intelligent Novice - Learning to Play Better. In D.N.L. Levy and D.F. Beal, editors, *Heuristic Programming in Artificial Intelligence – The First Computer Olympiad*. Ellis Horwood, 1989.

[Epstein, 1990] Susan Epstein. Learning Plans for Competitive Domains. In B. W. Porter and R. J. Mooney, editors, *Proceedings of the Seventh International Conference on Machine Learning*. Morgan Kaufman, 1990.

[Epstein, 1991] Susan Epstein. Deep Forks In Strategic Maps. In D.N.L. Levy and D.F. Beal, editors, *Heuristic Programming in Artificial Intelligence 2 – The Second Computer Olympiad*. Ellis Horwood, 1991.

[Epstein, 1992] Susan Epstein. Learning Expertise from the Opposition: The role of the trainer in a competitive environment. In *The Proceedings of AI 92*, pages 236–243, 1992.

[Fawcett and Utgoff, 1992] Tom E. Fawcett and Paul E. Utgoff. Automatic feature generation for problem solving systems. In Derek Sleeman and Peter Edwards, editors, *Proceedings of the Ninth Interntational Workshop on Machine Learning*, Aberdeen, Scotland, 1992.

[Flann and Dietterich, 1989] Nicholas S. Flann and Thomas G. Dietterich. A study of explanation-based methods for inductive learning. *Machine Learning*, 4:187–226, 1989.

[Flann, 1990] Nicholas S. Flann. Applying abstraction and simplification to learn in intractable domains. In *Proceedings of the International Machine Learning Conference*, pages 235–239, 1990.

[Flann, 1992] Nicholas S. Flann. *Correct Abstraction in Counter-planning: A Knowledge Compilation Approach*. PhD thesis, Oregon State University, 1992.

[Fong, 1991] Sandiway Fong. *Computational Properties of Principle-Based Grammatical Theories*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1991.

[Fraenkel *et al.*, 1978] A. S. Fraenkel, M.R. Garey, D.S. Johnson, T. Schaefer, and Y. Yesha. The complexity of checkers on an nxn board – preliminary report. In *Proceedings of the 19th Ann. Symp. on Foundations of Computer Science*. IEEE Computer Society, 1978.

[Frey, 1983] Peter W. Frey. The alpha-beta algorithm: Incremental updating, well behaved evaluation functions, and non-speculative forward pruning. In M. A. Bramer, editor, *Computer Game-Playing: theory and practice*, pages 285–289. Ellis–Horwood Publishers, Northwestern University, Evanston IL 60201, USA, 1983.

[Garey and Johnson, 1979] M.R. Garey and D.S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[Genesereth and Nilsson, 1987] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, 1987.

[Ginsberg and Geddis, 1991] Matthew L. Ginsberg and Donald F. Geddis. Is there any need for domain-dependent control information? In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 452–457, 1991.

[Good, 1968] I. J. Good. A five-year plan for automatic chess. In E. Dale and D. Michie, editors, *Machine Intelligence 2*, pages 89–118. Oliver and Boyd, 1968.

[Good, 1977] I. J. Good. Dynamic probability, computer chess, and the measurement of knowledge. In E. W. Elcock and D. Michie, editors, *Machine Intelligence 8*, pages 139–150. Ellis Horwood, Chichester, 1977.

[Hartmann, 1987] D. Hartmann. How to Extract Relevant Knowledge from Grand Master Games, part 1. *ICCA-Journal*, 10(1), March 1987.

[Hölldobler, 1992] Steffen Hölldobler. On deductive planning and the frame problem. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*. Springer-Verlag, 1992.

[Hooper and Whyld, 1984] David Hooper and Kenneth Whyld. *The Oxford Companion to Chess.* Oxford University Press, 1984.

[Huntsberger and Billingsley, 1981] David V. Huntsberger and Patrick Billingsley. *Elements of Statistical Inference.* Allyn and Bacon, Inc., fifth edition, 1981.

[Hunvald, 1972] Henry Hunvald. *Chess: Quotations from the Masters.* Peter Pauper Press, Mount Vernon, New York, 1972.

[Hyatt, 1984] R.M. Hyatt. Using time wisely. *ICCA Journal*, 7(1):4–9, 1984.

[Kierulf *et al.*, 1990] A. Kierulf, K. Chen, and J. Nievergelt. Smart Game Board and Go explorer: A case study in software and knowledge engineering. *Communications of the ACM*, 33(2), February 1990.

[Kierulf, 1990] Anders Kierulf. Smart Game Board: a Workbench for Game-Playing Programs, with Go and Othello as Case Studies. Technical Report NR 22, Informatik ETH, Zurich, 1990. (PhD Thesis).

[Koffman, 1968] Elliot B. Koffman. Learning games through pattern recognition. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(1):12–16, 1968.

[Korf, 1985] Richard E. Korf. Iteratively-Deepening-a*: An optimal admissible tree search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1034–1036, Los Angeles, California, 1985.

[Kuttner, 1983] Henry Kuttner. *Chessboard Planet*. Arrow Books Limited, 1983.

[Lee and Mahajan, 1988] Kai-Fu Lee and Sanjoy Mahajan. A pattern classification approach to evaluation function learning. *Artificial Intelligence*, 36:1–25, 1988.

[Lenat and Feigenbaum, 1991] D.B. Lenat and E.A. Feigenbaum. On the thresholds of knowledge. *Artificial Intelligence*, 47:185–250, 1991.

[Lenat, 1983] Douglas B. Lenat. The role of heuristics in learning by discovery: Three case studies. In R.S. Michalski, J.G. Carbonnel, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 1. Morgan-Kaufman, 1983.

[Levinson and Snyder, 1991] Robert A. Levinson and R. Snyder. Adaptive, pattern-oriented chess. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 1991.

[Levinson *et al.*, 1991] Robert Levinson, Feng-Hsiung Hsu, T. Anthony Marsland, Jonathan Schaeffer, and David E. Wilkins. Panel: The role of chess in artificial intelligence research. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, 1991.

[Levy and Newborn, 1991] David Levy and Monty Newborn. *How Computers Play Chess*. W.H. Freeman and Company, 1991.

[Markovitch and Sella, 1993] Shaul Markovitch and Yaron Sella. Learning of resource allocation strategies for game playing. In *Proceedings of IJCAI*, 1993.

[Minton, 1984] Steven Minton. Constraint-based generalization: Learning game-playing plans from single examples. In *Proceedings of the National Conference on Artificial Intelligence*, pages 251–254, 1984.

[Minton, 1990] Steven Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42:363–391, 1990.

[Newell *et al.*, 1963] Allen Newell, J.C. Shaw, and H.A. Simon. Chess-playing programs and the problem of complexity. In E.A. Feigenbaum and J. Feldman, editors, *Computers and Thought*. McGraw-Hill Book Company, 1963.

[Pell, 1992] Barney Pell. Metagame in Symmetric, Chess-Like Games. In H.J. van den Herik and L.V. Allis, editors, *Heuristic Programming in Artificial Intelligence 3 – The Third Computer Olympiad*. Ellis Horwood, 1992. Also appears as University of Cambridge Computer Laboratory Technical Report No. 277.

[Polya, 1945] G. Polya. *How To Solve It*. Penguin, 1945.

[Reznitsky and Chudakoff, 1990] A. Reznitsky and M. Chudakoff. A Chess Program Modeling a Chess Master's Mind. *ICCA-Journal*, 13(4):175–195, 1990.

[Rich, 1983] Elaine Rich. *Artificial Intelligence.* McGraw-Hill, 1983.

[Rosenbloom, 1982] Paul S. Rosenbloom. A world-championship-level othello program. *Artificial Intelligence*, 19:279–320, 1982.

[Roycroft, 1990] A. J. Roycroft. Expert Against Oracle. In D. Michie, editor, *Machine Intelligence 11*, pages 347–373. Ellis Horwood, Chichester, 1990.

[Russell and Wefald, 1992] Stuart Russell and Eric Wefald. *Do the Right Thing.* MIT Press, 1992.

[Sahlin, 1991] Dan Sahlin. An automatic partial evaluator for full prolog. Technical Report Research Report No. SICS/D–91/04–SE, Swedish Institute of Computer Science, 1991.

[Sahlin, 1992] Dan Sahlin. Personal communication, November 1992.

[Samuels, 1959] A. L. Samuels. Some studies in machine learning using the game of Checkers. *IBM Journal*, 3:210–229, 1959.

[Samuels, 1967] A. L. Samuels. Some studies in machine learning using the game of Checkers. ii. *IBM Journal*, 11:601–617, 1967.

[Schaeffer *et al.*, 1991] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. Reviving the game of checkers. In D.N.L. Levy and D.F. Beal, editors, *Heuristic Programming in Artificial Intelligence 2 – The Second Computer Olympiad*. Ellis Horwood, 1991.

[Snyder, 1993] Richard Snyder. *Distance*: Toward the unification of chess knowledge. Master's thesis, University of California, Santa Cruz, March 1993.

[Sutton, 1984] Richard S. Sutton. *Temporal Credit Assignment in Reinforcement Learning.* PhD thesis, Department of Computer and Information Science, University of Massachusetts, 1984.

[Tadepalli, 1989a] Prasad Tadepalli. Lazy explanation-based learning: A solution to the intractable theory problem. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 694–700, 1989.

[Tadepalli, 1989b] Prasad Tadepalli. Planning in games using approximately learned macros. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 221–223, 1989.

[Tesauro, 1993] G. Tesauro. TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural Computation*, 1993. To Appear.

[Tunstall-Pedoe, 1991] William Tunstall-Pedoe. Genetic Algorithms Optimizing Evaluation Functions. *ICCA-Journal*, 14(3):119–128, September 1991.

[van Harmelen and Bundy, 1988] Frank van Harmelen and Alan Bundy. Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, 36(3):401–412, 1988.

[van Tiggelen, 1991] A. van Tiggelen. Neural Networks as a Guide to Optimization. *ICCA-Journal*, 14(3):115–118, September 1991.

[von Neumann and Morgenstern, 1944] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[Warren, 1992] David Scott Warren. Memoing for Logic Programs. *Communications of the ACM*, 35(3), March 1992.

[Wellman and Doyle, 1991] Michael P. Wellman and John Doyle. Preferential semantics for goals. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, volume 2, pages 698–703, 1991.

[Wilkins, 1982] David E. Wilkins. Using knowledge to control tree search. *Artificial Intelligence*, 18:1–51, 1982.

[Williams, 1972] Thomas G. Williams. Some studies in game playing with a digital computer. In Siklossy and H. Simon, editors, *Representation and Meaning*. Prentice-Hall, 1972.

[Yee *et al.*, 1990] Richard C. Yee, Sharad Saxena, Paul E. Utgoff, and Andrew C. Barto. Explaining temporal-differences to create useful concepts for evaluating states. In *Proceedings of AAAI-90*, 1990.