Adversarial Reasoning: A Logical Approach for Computer Go

by

Tamir Klinger

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 2001

Ernest Davis

© Tamir Klinger

All Rights Reserved, 2001

To Mom and Dad

Acknowledgment

The research and writing of this thesis has been my preeminent preoccupation for more than six years. My colleague in this project, David Mechner, originally suggested the ideas on which this work was based and was an equal partner – sometimes more than equal – in every aspect of the program design, implementation, and analysis. Without David's optimism, dedication, and hard work this thesis would not have been written.

My family and friends have all endured this difficult time with the greatest support and encouragement. To them I offer my sincerest thanks and gratitude.

Finally, my advisor, Ernie Davis, helped me immensely both as a mentor and reader. He fought through countless drafts of countless papers, listened carefully to half-worked ideas, and never wavered in his commitment.

Thanks to all of you.

Tim Klinger, May 2001

Abstract

Go is a game with simple rules but complex strategy requiring ability in almost all aspects of human reasoning. A good *Go* player must be able to hypothesize moves and analyze their consequences; to judge which areas are relevant to the analysis at hand; to learn from successes and failures; to generalize that knowledge to other "similar" situations; and to make inferences from knowledge about a position. Unlike computer chess, which has seen steady progress since Shannon's [23] and Turing's [24] original papers on the subject, progress on computer *Go* remains in relative infancy. In computer chess, minimax search with α-β pruning based on a simple evaluation function can beat a beginner handily. No such simple evaluation function is known for *Go*. To accurately evaluate a *Go* position requires knowledge of the life and death status of the points on the board. Since the player with the most live points at the end of the game wins, a small mistake in this analysis can be disastrous.

In this dissertation we describe the design, performance, and underlying logic of a knowledge-based program that solves life and death problems in the game of *Go*. Our algorithm applies life and death theory coupled with knowledge about which moves are reasonable for each relevant goal in that theory to restrict the search space to a tractable size. Our results show that simple depth-first search armed with a goal theory and heuristic move knowledge yields very positive results on standard life and death test problems – even without sophisticated move ordering heuristics.

In addition to a description of the program and its internals we present a modal

logic useful for describing *strategic theories* in games and use it to give a life and death theory and to formally state the rules of *Go*. We also give an axiomatization for this logic using the modal μ-calculus [15] and prove some basic theorems of the system.

Contents

Dedication **iii**

Acknowledgment **iv**

Abstract **v**

List of Figures **x**

List of Tables **xii**

1 Introduction **1**

1.1 Introduction 2

1.2 Difficulties in Computer Go 3

1.3 Contributions of this thesis 5

1.4 Previous work on computer *Go* 6

1.5 Previous work on Game Logic 8

1.6 Organization of this thesis 8

2 Adversarial Reasoning **9**

2.1 Adversarial Operators 10
2.2 Game Knowledge 13
2.3 Solving Life and Death Problems 17
2.4 The language $\mathcal{GO}$ 19
2.5 A strategic theory for life and death in *Go* 23
2.6 The goal graph: Using the strategic theory to solve life and death problems 28
2.7 Correctness of the algorithm 39

3 Program **41**

3.1 Introduction 42
3.2 The Board 43
3.3 Blocks 43
3.4 Connections 45
3.5 Groups 46
3.6 Cores 47
3.7 Eyes 48
3.8 Aura 50
3.9 Knowledge base 51
3.10 Rule filtering 52
3.11 Constraint Manager 54
3.12 Assertion Database 56
3.13 State change – The Reversion Manager 58
3.14 History of the project 59

4 Results 61

4.1 Testing and Results 62

4.2 Two problems: A success and a failure 63

5 Game Logic 72

5.1 Introduction 73

5.2 Introduction to Modal Logic 73

5.3 Models 74

5.4 The first-order modal μ-calculus 77

5.5 First-Order Modal Game Logic 79

5.6 Game Logic 81

5.7 Axioms and Definitions 81

5.8 Semantics 84

5.9 Some useful theorems 85

6 Rules for *Go* 90

6.1 Introduction 91

6.2 Meta-language conventions 91

6.3 Rules of Go 94

7 Conclusion 101

7.1 Conclusions and Directions for Future Work 102

Bibliography 105

List of Figures

1.1 Five stone handicap game between David Mechner (white) and Greg Ecker (black) 5

2.1 A reasonable way to connect p to q. 15

2.2 Some examples of *Go* concepts. 24

2.3 Black can connect her stones at $C2$ and $F3$ by playing at $E2$. 26

2.4 The AND/OR graph for Save(P17) 32

2.5 A life and death problem Save(P17) on the left; its goal graph on the right. 33

2.6 The problem after Black plays at 1 (White's turn). 34

2.7 The problem after Black plays at 1, and White responds at 2. 35

2.8 Black captures all the White stones, saving P17. 36

2.9 Black can win the race to capture by playing on points $G1$ or $F1$. 37

2.10 Black can make life by playing on point $F1$. 37

2.11 Black is alive. 37

3.1 The program structure. 44

3.2 The black and white stones form a *crosscut*. 46

3.3 Two rules for recognizing eyes. Stones marked with a triangle are optional. 48
3.4 A variety of rules. 53
3.5 Applying rule 3.6 incorrectly gives Black two disjoint eyes (borders marked with “B”, centers with “C”.). 54
3.6 An eye rule. 54
3.7 A more specific eye rule. 54

4.1 Correctly solved problem 1-124. 64
4.2 Problem 1-124. Black chooses the wrong path. 64
4.3 Problem 1-124. Black chooses the right path. 65
4.4 Incorrectly solved problem 2-42. 66
4.5 Incorrectly solved problem 2-108 (gets lost). 67

List of Tables

2.1 Definitions of some basic *Go* concepts 21

3.1 Number of rules for each purpose. 52

4.1 Results on Kano's Graded Go Problems for Beginners: vols. 1-2 68

4.2 Results on Kano's Graded Go Problems for Beginners: vols. 1-2 69

4.3 Results on Kano's Graded Go Problems for Beginners: vols. 1-2 cont. . 70

4.4 Results on Kano's Graded Go Problems for Beginners: vols. 1-2 cont. . 71

4.5 Summary of results on Kano's Graded Go Problems for Beginners: vols. 1-2 71

Chapter 1

Introduction

1.1 Introduction

One of the reasons games have been of interest to AI researchers, aside from the popular belief that they are worthy intellectual pursuits, is the fact that they are usually self-contained, circumscribed activities not subject to the difficulties associated with the formalization of many "real-world" problems. Games are micro-worlds with none of the messiness and unpredictability of the real world but with some very challenging problems to be solved and strong human intuitions about how to solve them. Also, unlike in other human domains, it is relatively easy to measure the performance of a game-playing program on standard problem sets and in competition with other players.

Chess has traditionally been the game of choice for researchers interested in games. Within the computer chess world some, like Wilkins [26] with his PARADISE chess program, have pursued the knowledge-based approach, but the incredible success of big-search programs, like Deep Blue, have all but completely eclipsed these efforts. *Go*, on the other hand, has proved much more intractable.

There are really two problems in applying big-search methods to *Go*. The most often-cited problem is just the huge size of the search space. As Mechner [17] puts it:

> On average, at any given moment in a chess game, a player has twenty-five legal moves to choose from. Various refinements have enabled the minimax algorithm to effectively consider only the five or six best choices, pruning away the rest. In *Go*, on average, a player can choose from 250 legal moves per turn. Even if a *Go* algorithm could prune that number down to fifteen or sixteen, and consider 200 million positions per second, it would need seventy years to explore as deeply as Deep Blue does in three minutes.

But a more subtle and less-often mentioned problem is the quality of static evaluation functions. Mechner [17] continues:

> Very well, one might say; computers will get faster soon enough. But the problem goes deeper than that. Go programs not only fail to evaluate 200 million *Go* positions a second; they cannot accurately evaluate a single one.

Big-search techniques with sophisticated move-ordering heuristics have been successfully applied to solving life and death problems in *Go* by Wolf [28] and others, but these programs require very circumscribed positions that occur relatively rarely in real games. Many of even the simplest life and death problems we consider in this thesis are not solvable by these programs. All the successful full-playing programs and open-position life and death problem solvers are knowledge-based to some degree.

1.2 Difficulties in Computer Go

In Chess, a relatively simple positional evaluation function based on a tally of features like material advantage, while definitely not up to expert standards, provides a sufficiently good measure of the position to allow big-search techniques to work. In *Go*, humans evaluate a position by determining the life and death status of the stones on the board. Since dead stones are removed from the board at the end of the game and leave the space they occupied as enemy territory, a mistake in life and death analysis can lead to a big error in positional evaluation. Furthermore, this analysis is not one that can easily be made statically.

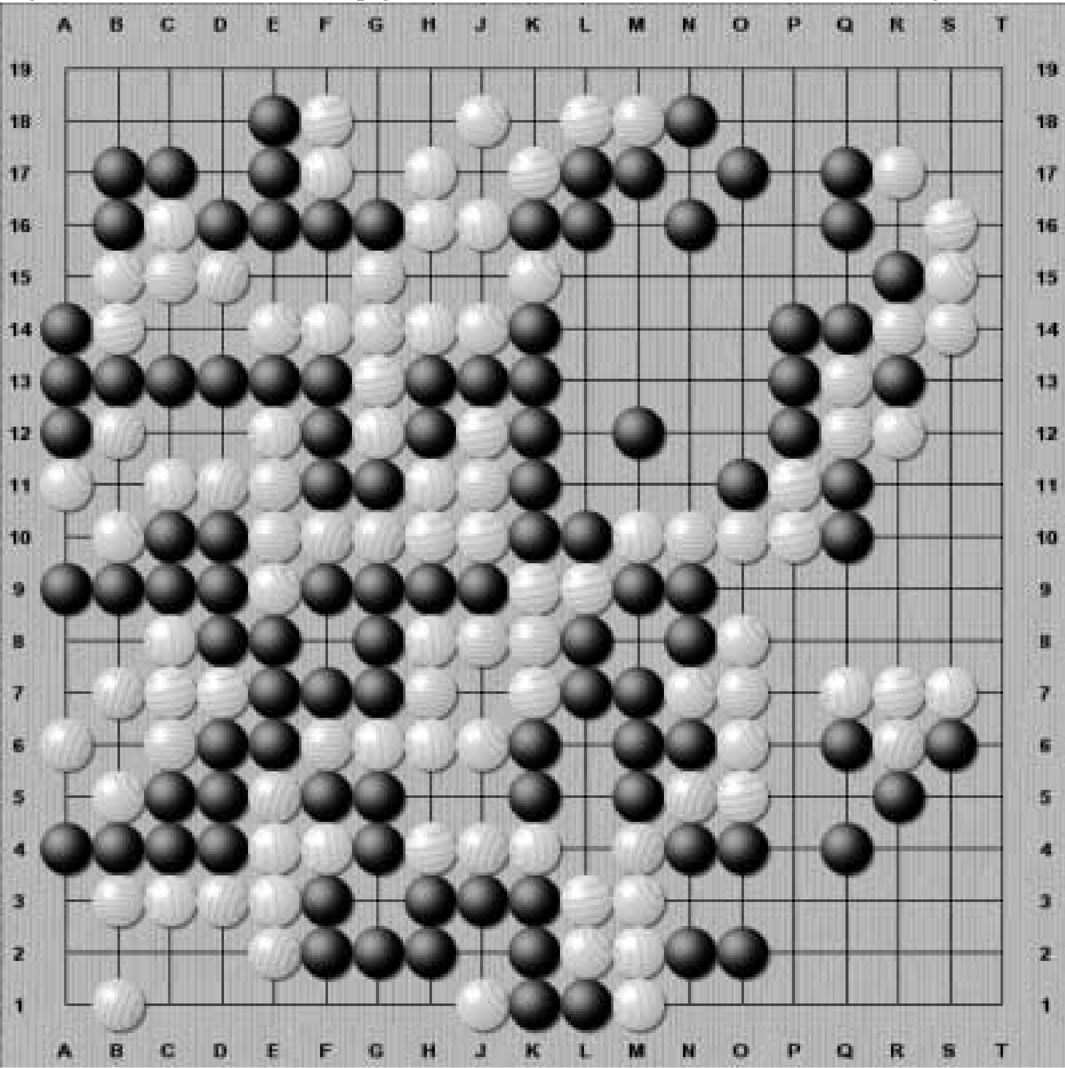
For human players, determining whether stones will live or die requires an understanding of which other stones on the board are important to the problem. Once the

context of the problem is understood, if the position is very simple, it may be possible to evaluate it statically. More often it requires further search to positions which are statically determinable. This is not only a limitation of *Go* programs. No human player can statically determine the life and death status of stones in all positions; there are too many subtle variations and interactions for a player to be able to look at an arbitrary configuration of stones and pronounce them alive or dead. People do, however, have a lot of knowledge they bring to bear about what is *reasonable* in a situation that allows them to limit the amount of reading they perform. By formalizing the techniques and knowledge *Go* players apply, we have a useful model for automation. Once it is determined that a position is unclear and further reading is required, one very strong heuristic for limiting the moves considered in the search is to try only moves that have worked for similar goals in similar contexts in the past. This is the basis for our approach.

To illustrate how subtle differences in the position can have dramatic positional significance consider the position in figure 1.1 taken from a game between two amateur players. It is White's turn to move.

In order to evaluate this position, an analysis must be made that is deep enough to reveal that the Black group on $A14$ is alive. An intermediate level human player will see this quite easily by reasoning that the White group on $C11$ can be captured more quickly than either of the surrounding black groups (on A14 and A9). Also, the one-eyed white group at $M1$ is alive because it can capture the black M5 before it is itself captured. And therefore the black group at K1 is dead. Such an analysis is beyond current go-playing programs and is the focus of our research.

Figure 1.1: Five stone handicap game between David Mechner (white) and Greg Ecker (black)



1.3 Contributions of this thesis

Our work focuses on a number of difficult and important issues in *Go* and Artificial Intelligence. First, we have devised a novel algorithm that applies human game knowledge to solve uncircumscribed life and death problems. To implement this algorithm required finding appropriate knowledge representation and truth maintenance schemes

and developing a class library of data structures to support incremental update and reversion in search. Our solutions to these problems are discussed in more detail in chapter 3.

Our second major contribution is in the development of a modal logic for knowledge representation in games, and the presentation of our program in terms of this logic. Logic is the *lingua franca* for researchers in different areas of AI. By casting our problem and solution in a logical language, we increase the possibility that it will prove useful to others both in the specifics of the formalization, as well as, more generally, a template for research in other domains.

We give an axiomatization for our logic and provide semantics in terms of game tree models. We use the logic to prove some general game theorems, to state a rule set for *Go*, and to describe an algorithm for *adversarial reasoning*. The algorithm we give applies a *strategic theory*, to find the relevant context for a given adversarial decision problem. When the context is known, knowledge about which moves are *reasonable* to achieve the required goals is applied to focus the search. This algorithm applied to a strategic theory of life and death is the basis of our life and death problem solver, but can more generally be applied to the solution of adversarial reasoning problems in any domain in which it is possible to state a strategic theory.

1.4 Previous work on computer *Go*

There have been a number of papers and theses published on computer *Go* since Zobrist's thesis [29] in 1970. Müller [18] lists among others: Erbach [9], Kierulf [3], and Ryder [21]. The interested reader will find most of the relevant references and a his-

tory of computer *Go* in these documents. Further references are also available from the American Go Association web page [1].

The best existing full-playing programs perform shallow life and death analysis through recognition of features associated with life, like eyes[1] or surrounding open space (lots of "friendly" empty space means room to make eyes). This is adequate in simple positions where the block in question is clearly safe or hopelessly dead, but falls down in complex situations like the one above, which often occur in real games. Most current programs also perform some kind of limited, "tactical" reading to determine whether stones with few liberties can be captured, but this too falls down when the status of a larger group is in question, or when the stones in question have many liberties but are still dead.

Relatively little has been written on the problem of life and death in *Go*. Dave Dyer and Thomas Wolf [28] have developed life and death problem solvers; however both use brute-force search and are therefore limited in their application to highly constrained positions with no access to the center of the board and little empty space allowed within the confines of the problem. Problems such as these are rare in actual games where the life and death status of stones must be determined before positions become so well-defined.

An interesting and relevant recent piece of work that is similar in spirit to our own is Willmott's master's thesis [27]. Willmott's program is a prototype of a life and death problem solver based on the notion of hierarchical planning in an adversarial setting. In this context he represents a theory of life and death somewhat similar to our own and

[1] An eye is a compartment each point of which is empty or occupied by dead enemy stones. A block of stones with the ability to form two adjacent disjoint eyes will live, so eyes play a fundamental role in life and death strategy.

shows how it can be used to solve a small set of problems.

1.5 Previous work on Game Logic

Game *semantics* has been widely used by researchers in a variety of fields, but the notion of a *game logic*, a proof system and semantics for games, is relatively recent and unexplored. Game logic was first proposed in a paper by Parikh [19] and more recently considered by Pauly [2]. Both of these papers consider games as atomic entities in the language. Our interest lies in the analysis of the *internal* structure and strategies of games. To accommodate this added expressivity requirement, we have chosen the first-order modal μ-calculus as the basis for our logic.

1.6 Organization of this thesis

In chapter 2 we introduce a modal language useful for adversarial reasoning and use it to present a theory of life and death, a high-level algorithm for solving life and death problems, and a detailed example problem. In chapter 3 we present our program's structure and algorithms. Chapter 4 gives results on a set of graded *Go* problems. In chapter 5 we consider the problem of adversarial reasoning in games like *Go* from a logical perspective. We present an axiomatization for the first-order modal logic of games and prove some basic results.In chapter 6 we use our game logic to provide an axiomatization of the rules of *Go*.

Chapter 2

Adversarial Reasoning

In this chapter we will describe, at a high-level, our algorithm for analyzing and solving life and death problems. The basic idea is to define a logical theory of life and death in *Go* and use this theory to find the set of goals that are relevant to the problem at hand. We represent the relevant goals and their logical relationships in an AND/OR graph and use a knowledge base to find the reasonable moves for each goal in this graph. With luck, the set of reasonable moves is considerably smaller than the set of legal moves. The reasonable moves for the root goal are each explored in turn, and the resulting position analyzed recursively until the problem is solved or we run out of time. The algorithm may be applied more generally in any domain in which it is possible to state a strategic theory, though we have implemented it only for life and death.

2.1 Adversarial Operators

In this section we introduce adversarial operators which augment the first-order language presented above to allow statements about strategies. Such statements take the form:

Player 1 has a move such that

    For all Player 2's moves

        ...

            Player 1 has a move such that

                For all Player 2's moves

                    some goal is true

In a first-order notation we could express this as

$$\exists m_1 \forall m_2 \exists m_1 \cdots \exists m_{n-1} \forall m_n \phi$$

for some goal ϕ and some number n. However the number n would have to be fixed; there is no way to quantify over the number of quantifiers in the statement and no facility for adding a "there exists n" in front of the above statement. To allow this kind of statement we propose the addition of three new *adversarial* operators, two of which are primitive and one which is the composition of the primitive operators. We will not here explore the logical properties of these operators but refer the interested reader to Chapter 5 where we provide a proof system and semantics.

- $\mathbf{est}_i(\phi)$ ("establishable ϕ") which means that player i has a legal strategy to make the goal ϕ true.
- $\mathbf{irr}_i(\phi)$ ("irrefutably ϕ") which means ϕ is true now and player i has a legal strategy to maintain its truth until there are no legal moves available (the end of the game).
- $\mathbf{ach}_i(\phi)$ ("achievable ϕ") which is $\mathbf{est}_i(\mathbf{irr}_i(\phi))$ and means player i has a legal strategy to make ϕ irrefutably true.

If we want to say that there is a strategy to establish ϕ from some particular position or state s, we write "$s \models \mathbf{est}_i(\phi)$" which is read "s models $\mathbf{est}_i(\phi)$" or "$\mathbf{est}_i(\phi)$ is true in state s." Similarly with $\mathbf{irr}_i$ and $\mathbf{ach}_i$.

To illustrate the definitions consider how we could express the fact that black (on her turn) can capture either a bishop or a knight, in the game of chess:

$$ToPlay(BLACK) \wedge$$
$$\mathbf{est}_{Black}(Captured(bishopQ) \vee Captured(bishopK) \vee$$
$$Captured(knightQ) \vee Captured(knightK)))$$

where it is assumed the predicates $ToPlay$ (whose turn it is to play), and $Captured$ (indicating that a piece has been removed from the board), are already defined. $bishopQ$ refers to the queen-side bishop; $bishopK$ refers to the king-side bishop; $knightQ$ refers to the queen-side knight; $knightK$ to the king-side knight.

Note that saying $\mathbf{est}_i(P \vee Q)$ is not the same as $\mathbf{est}_i(P) \vee \mathbf{est}_i(Q)$. It is true that $[\mathbf{est}_i(P) \vee \mathbf{est}_i(Q)] \longrightarrow \mathbf{est}_i(P \vee Q)$, but not conversely in general. The reason for this is that there may be a situation in which player i a strategy to make either P or Q true, but which one is up to the opponent. Imagine the case where the player has simultaneously attacked (forked) a knight and a bishop. It is the opponent's decision which one to save. If the player wants to capture a bishop, the opponent denies her by sacrificing the knight; if the player wants to capture a knight, the opponent denies her by sacrificing the bishop. Both strategies fail individually but the strategy to capture either a bishop *or* a knight is a success.

Another example from chess is the notion of *checkmate*. A checkmate occurs in chess when the opponent's king is in check and has no legal move to escape. The existence of a strategy for White to checkmate Black from a state s can be expressed in our language as:

$$\mathbf{ach}_{White}(Check(Black))$$

where it is assumed that $Check$ has already been defined. This says that it is White's turn to play and she has a legal strategy from state s to reach a state in which Black is in check and there are no legal moves available to escape from it. It must be Black's turn in such a state since it is not (legally) possible for Black to be in check on White's turn.

As a another example, in the game Othello, we can state the fact that square P is White at the end of the game:

$$\mathbf{ach}_i(White(P))$$

If we assume that the predicate $White$ has already been defined, this says that player i can make square P ultimately white (forever after a certain point), even though it may change color from white to black and back again many times during the course of the game.

2.2 Game Knowledge

The adversarial operators $\mathbf{est}_i$, $\mathbf{irr}_i$, and $\mathbf{ach}_i$ allow us to talk about legal game strategies in a logical language. However, they do not take into consideration the practical difficulties in the computation of such strategies. To decide whether White has a forced win in the game of NxN *Go*, for instance, is known to be PSPACE complete [16].

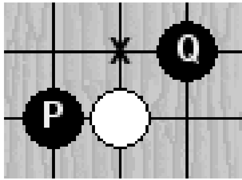
In our model we assume non-modal sentences of interest can be statically decided. Examples (in English) of such sentences for *Go* include, "How many black stones are adjacent to the block on point p?" and "How many white blocks are there on the board?" With a reasonable representation for the board, these kinds of questions are trivial to answer quickly and efficiently.

When we allow modal sentences (including sentences with the adversarial operators) that refer implicitly to other states in the game the decision process is no longer so simple. Search is generally required and such a search may be intractable without additional knowledge to limit it in some way. For example, "Does Black have a strategy to capture the white block on point p?". Answering this question without any knowledge of which moves are "reasonable" to *Go* about capturing a block will likely be intractable.

To combat this difficulty we try to model how humans simplify the problem. Human players when confronted with a decision problem like $s \models \mathbf{ach}_i(\phi)$ seem to engage in several simplifying processes. One is to find the set of goals that have some bearing on the goal ϕ of interest. If successful, this heuristic may narrow down the set of possibilities dramatically. We model this human ability in life and death problems by constructing an AND/OR graph representing the relevant goals or *context* for the problem at hand.

In addition to finding the context of a problem, human players seem to have at their disposal a repertoire of moves that have worked in the past to accomplish similar goals in similar situations. Sometimes, it is clear that the current situation is similar enough to one encountered previously that the same move that worked there can be statically determined to work in the current situation. More often though, finding the set of moves that have worked in similar situations in the past merely serves as a good heuristic for focusing on the likely candidates. Combined with knowledge about the set of relevant *goals* to the problem goal, a human player can use this heuristic to determine the set of all relevant *moves* to try to achieve the problem goal by finding the relevant moves for each relevant goal. Obviously for such a strategy to be correct with respect to perfect

Figure 2.1: A reasonable way to connect p to q.



play, the set of moves considered must include at least one move that does in fact work, when such a move exists.

An example of a rule suggesting a reasonable move for connecting p and q shown in figure 2.1. This rule states that playing on the point marked "X" is a reasonable way to connect the p and q stones. It is reasonable because, while it is not guaranteed to work in every context, there are many contexts in which it will work to connect the stones. Our program makes use of hundreds of such rules for a number of different strategic goals. A more detailed discussion of the rule language and goals is provided in chapter 3 section 3.9

The question arises, why break the problem down into these two components: finding the relevant goals, and using rules such as the connect rule in figure 2.1 to generate moves for each such relevant goal? Why not just "build in" the context into the preconditions of the rules? The answer to this question is that there are so many slightly different positions where the same goal has very different reasonable moves that trying to capture them in "if-then" rules is very difficult. The structure of a given life and death problem can be very complex requiring a recursive analysis of the position. Using a goal theory to model it is a succinct and efficient approach: the goal theory handles the problem structure while the knowledge base concerns itself only with which moves are reasonable to achieve a goal in isolation – ignoring its relationships to other goals.

The strategic goal theory and knowledge base of rules suggesting moves for goals in that theory form the basis of our approach. However, there are some complicating subtleties.

One wrinkle is the fact that goals which may be achievable in isolation may not be in conjunction with other necessary goals; there can be interactions between the two which allow one or the other to be achieved but not both simultaneously. The most well-known example of such a situation occurs in Chess when the opponent has two pieces forked. The player may have one move which is guaranteed to save one of the pieces, and a different move which is guaranteed to save the other, but no move exists in common which will save both. Similar examples in *Go* occur frequently. The question of whether there is a fork between two goals is really a question of their *independence* or, more accurately, their *dependence*.

Intuitively a goal ϕ is *disjunctively independent* from a goal ψ just if whenever it is possible to achieve $\phi \vee \psi$, it is possible either to achieve ϕ or to achieve ψ. In a state s we write:

$$s \models \mathbf{ach}_i(\phi \vee \psi) \longrightarrow [\mathbf{ach}_i(\phi) \vee \mathbf{ach}_i(\psi)]$$

A player has a *fork attack* between two goals ϕ and ψ if and only if they are not disjunctively independent. If, for example, the opponent has a fork attack between a bishop and a knight, then she may capture one or other, but which one is the player's choice. In logical terms, there is a fork attack between ϕ and ψ iff

$$s \models \mathbf{ach}_i(\phi \vee \psi) \wedge (\neg\mathbf{ach}_i(\phi) \wedge (\neg\mathbf{ach}_i(\psi))$$

There is a dual notion of *conjunctive independence*. Two goals are conjunctively

independent just if whenever they are both achievable individually they are achievable collectively. In logical terms:

$$s \models [\mathbf{ach}_i(\phi) \land \mathbf{ach}_i(\psi)] \longrightarrow \mathbf{ach}_i(\phi \land \psi)$$

A player has a *fork defense* between two goals ϕ and ψ if and only if they are not conjunctively independent. For example, after the opponent has played her move to fork a bishop and a knight, the player has a fork defense. She may save one or the other piece, but not both. In logical terms, there is a fork defense between ϕ and ψ iff

$$s \models \mathbf{ach}_i(\phi) \land \mathbf{ach}_i(\psi) \land (\neg \mathbf{ach}_i(\phi \land \psi))$$

It is a difficult problem to determine independence between goals in a given position, but there are times when independence assumptions are plausible and seem to correspond with human intuition as well as leading to the correct result. In the move generation algorithm presented in the next section, we make independence assumptions about goals whose value is statically known. These assumptions are heuristic but seem to work well in practice.

2.3 Solving Life and Death Problems

A life and death problem is a problem that asks whether a point on the board can be captured (by the opponent) or saved (by the player). A captured point is one that cannot be occupied by the opponent at the end of the game. Such points may ultimately be empty or occupied with friendly stones. Dead enemy stones are assumed removed.

The problem of capturing a point p from state s for player i can be formalized as the problem of deciding:

$$s \models \mathbf{ach}_i(TwoEyes(p))$$

where the predicate $TwoEyes(p)$ will be defined more precisely in section 2.4.

It is possible to show that a block is saved (uncapturable by the opponent) iff it has a strategy to maintain at least two eyes. This is because capturing a block requires filling in all adjacent empty points. In theory, it need only have a strategy to maintain a single eye, but there are no such strategies since it is always legal for the opponent to just fill the single empty point on his turn, capturing the block.

One way of making two eyes which is sometimes possible is just to play near to the point to be saved and to build defenses around two empty disjoint points. Such a strategy is required when the point to be saved is hemmed in by enemy stones. When there is more space to "run out", another kind of strategy for making life is to establish an unbreakable connection between the point to be saved and another friendly point which can be saved. Ultimately these two strategies amount to the same thing: the point in question is saved because it can make and maintain two eyes, but the techniques applied by *Go* players to pursue one or the other of these strategies are qualitatively different and knowledge about how to accomplish each of them is quite reasonably kept separate in the knowledge base.

In the next section we present a simple language for life and death in *Go* that will allow us to formulate life and death problems and theories in a logical language. In section 2.1 we extend this language to include adversarial operators which allow us to make statements about the existence of strategies to achieve goals. Chapter 5 has a

more formal presentation of the adversarial game logic with proofs of some of the basic properties of the system. Section 2.5 works through a life and death problem in some detail.

2.4 The language $\mathcal{GO}$

There are really only a handful of predicates necessary to express basic life and death ideas. We need the primitive concepts of

- *occupancy* – which points are occupied with black or white stones, and which are empty.
- *adjacency* – which points are adjacent on the board.
- *nearness* – whether two occupied points are close enough together for the life of one to affect the other.
- *turn* – whose turn it is to play.

From these, we derive the concepts of

- *friendly* – whether two stones are of the same color.
- *enemy* – whether two stones are of different colors.
- *block adjacent* – there is a path of adjacent stones of the same color from a stone to a point.

The reader should note that our definition of an "eye" captures the concept of a *potential* eye, or a block-adjacent empty point that may or may not remain empty under

attack by the opponent. Having two eyes, as we have defined them, does not guarantee anything about the block's ability to live. In the next section, we will define operators which can be applied to our definition of eyes, that say, in effect, "there is a strategy to make the block have two potential eyes forever." This statement will be our definition of "alive".

Table 2.1 gives the formal definitions in our language for these concepts. As is usual, free variables are assumed universally quantified.

Ideally, we would define $SameBlock$ as a least-fixpoint of the definition given – in fact, in chapter 6 that is exactly how we define it. At this point, because we have not yet discussed fixpoints or the modal μ-calculus, we give this less precise characterization.

Figure 2.2 shows examples of $BlockAdj$, $Enemies$ and $\neg Enemies$.

Each of these predicates can be easily and efficiently statically decided using a simple board representation. Chapter 3 discusses our program's particular representation scheme in more detail.

Currently, we define $Near(p, q)$ to be true if p and q are both occupied and within a Manhattan distance of 3 on the board, with no interposing enemy stones on the path between them.

This language is useful for stating facts about a particular position, but lacks the "temporal" operators necessary to describe future or past positions or to talk about strategies to reach a goal position. These kinds of modal statements require additional logical machinery. The next section introduces some adversarial operators which allow us to extend the basic language to handle these kinds of statements.

In the next section we will present our strategic theory for *Go*. The goals represented in the theory correspond roughly to human notions of life and death theory though we

Table 2.1: Definitions of some basic *Go* concepts

- $Occ(p, c)$ – The point p is occupied with color c, with $c \in \{BLACK, WHITE, EMPTY\}\}$.
- $Adj(p, q)$ – The point p is immediately north, south, east, or west of q.
- $Near(p, q)$ – The point q is close enough to p to be of strategic interest.
- $ToPlay(i)$ – It is player i's turn to play.
- $<, \leq, =, \geq, >$ – The relational operators for each sort of interest.

$$Black(p) \longleftrightarrow Occ(p, BLACK)$$

The point p is occupied by a black stone.

$$White(p) \longleftrightarrow Occ(p, WHITE)$$

The point p is occupied by a white stone.

$$Empty(p) \longleftrightarrow Occ(p, EMPTY)$$

The point p is empty.

$$Enemies(p, q) \longleftrightarrow [Black(p) \wedge White(q)] \vee [White(p) \wedge Black(q)]$$

The points p and q are occupied with stones of different colors.

$$EnemyNbrs(p, q) \longleftrightarrow Enemies(p, q) \wedge Adj(p, q)$$

Points p and q are adjacent enemies.

$$Friends(p, q) \longleftrightarrow [Black(p) \wedge Black(q)] \vee [White(p) \wedge White(q)]$$

The points p and q are occupied with stones of the same color.

$$FriendlyNbrs(p, q) \longleftrightarrow Friends(p, q) \wedge Adj(p, q)$$

The points p and q are adjacent friends.

$$SameBlock(p,q,i) \longleftrightarrow Occ(p,i) \wedge \exists r\ Occ(r,i) \wedge Adj(p,r) \wedge [r = q \vee SameBlock(r,q,i)]$$

Point p is in the same block as point q iff
p and q are the same point, occupied with stones of color i or
p is adjacent to a point r which is itself
in the same block as q.

$$BlockAdj(p,q,i) \longleftrightarrow \exists r\ SameBlock(p,r,i) \wedge Adj(r,q)$$

Point p is block-adjacent to point q iff
it is in the same block as a point adjacent to q.

$$Liberty(p,q,i) \longleftrightarrow Empty(q) \wedge BlockAdj(p,q,i)$$

Point q is a liberty for p iff
p is block-adjacent to q and q is empty.

$$Atari(p,i) \longleftrightarrow [\forall q,r\ [Liberty(p,q,i) \wedge Liberty(p,r,i)] \longrightarrow q = r] \wedge \exists t\ Liberty(p,t,i)$$

A point p is in atari if it is has exactly 1 liberty.

$$Suicide(p,i) \longleftrightarrow Empty(p) \wedge \forall q\ Adj(p,q) \longrightarrow [\neg Empty(q) \wedge [Occ(q,i) \longrightarrow Atari(q)] \wedge [Occ(q,\bar{i}) \longrightarrow \neg Atari(q)]]$$

An empty point p is suicide for player i iff all its neighbors
are occupied – the friendly neighbors (of color i)
are in atari, and the enemy neighbors (of color $\bar{i}$) are not.

$$
\begin{aligned}
Eye(p,q,i) &\longleftrightarrow Liberty(p,q,i) \wedge Suicide(q,\bar{i})
\end{aligned}
$$

A point q is an eye for point p iff q is a liberty of the block on p and q is suicide for the opponent.

$$
TwoEyes(p,q,r,i) \longleftrightarrow Eye(p,q,i) \wedge Eye(p,r,i) \wedge q \neq r
$$

Points q and r together give p two eyes iff they are both distinct eyes.

$$
Point(p,i) \longleftrightarrow Occ(p,i) \vee \exists q \; Eye(q,p,i)
$$

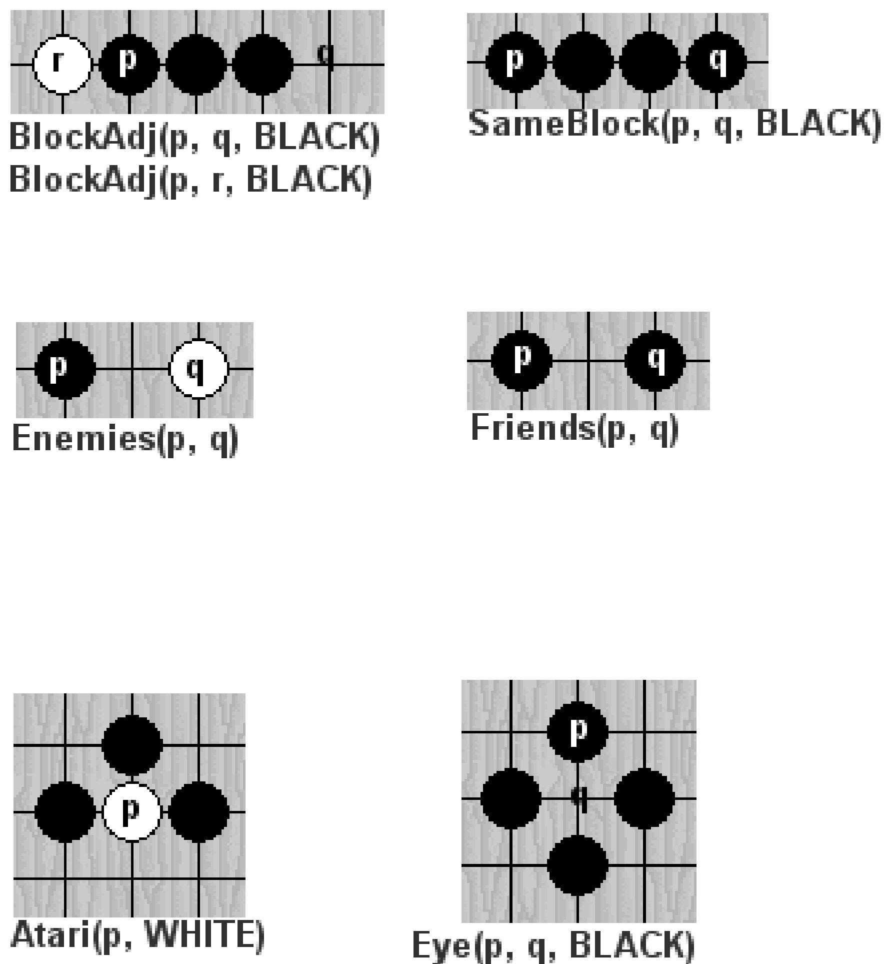
The board point p is a point of territory for player i iff p is occupied with a stone of color i or p is an eye for a block of color i.[1]

do not elaborate the goals of saving and capturing into a more detailed theory involving eyes. Mainly the theory addresses capturing, saving, connecting to points, and severing from points. The purpose is to explicitly state the logical relationships between these goals so that we can identify the relevant goals in a particular life and death situation.

2.5 A strategic theory for life and death in *Go*

There are four basic strategic concepts in our theory: *capture*, *save*, *connect*, and *sever*. $Capture$ and $Save$ are the primary goals in the theory. All life and death problems that we will consider will be framed in terms of deciding whether a given stone can be captured or saved. When the object block is completely surrounded with (statically)

Figure 2.2: Some examples of *Go* concepts.



safe enemy stones (such as in figure 2.10) then it is not necessary to consider other capture or save goals to analyze the problem. The problem is simply a matter of deciding whether the object block can make two eyes or not. In figure 2.10, block $D1$ can make two eyes by playing on point $F1$. The resulting shape is shown in figure 2.11. The goals Capture and Save are defined formally below. The point p is the object block to be saved or captured. The points q and r refer to eye points.

1. $Save(p, i) \equiv \mathbf{ach}_i(\exists q, r\ TwoEyes(p, q, r, i))$

2. $Capture(p, i) \equiv \mathbf{ach}_i(\forall q, r\ \neg TwoEyes(p, q, r, \bar{i}))$

Note that these goals are about the existence of *strategies* to make something true. $Capture(p)$, for instance, is satisfied when there is a strategy to make the point p irrefutably a point of territory at some time in the future.

Not all life and death problems are so simple as 2.10. Often the question of whether a given block can be captured hinges on whether other blocks can be captured or saved. The purpose of our goal theory is to relate the capturing or saving of an object block to the other capture and save goals that have some relevance. This is where the secondary goals, Connect and Sever come into play. Connect is the goal of establishing block-adjacency and, Sever, its dual, is the goal of preventing block-adjacency.

If a block is to be saved and it is not completely surrounded as in figure 2.10, it may live if there is a strategy to expand the block until it has enough additional surrounding empty space, or an increased ability to articulate eyes. As an example, consider figure 2.3. Here the Black stone on $F3$ is in danger of being cut off by a White move at $D2$. To save it, she can play at $D2$ herself, as part of a strategy to connect it to the other Black stone on point $C2$.

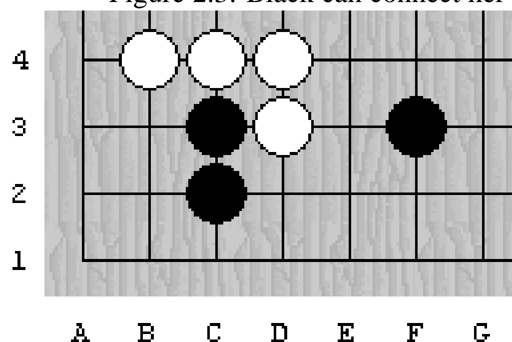
The goals Connect, and Sever are formally defined as follows:

1. $Connect(p, q, i) \equiv \mathbf{est}_i(BlockAdj(p, q, i))$

2. $Sever(p, q, i) \equiv \mathbf{irr}_i(\neg BlockAdj(p, q, \bar{i}))$

Note that for these goals the point q need not be occupied. This may be confusing for *Go* players who typically use the term "sever" when talking about cutting two occupied

Figure 2.3: Black can connect her stones at $C2$ and $F3$ by playing at $E2$.



stones apart. We use it in that conventional sense, but also to mean keeping an occupied point p from extending itself to a block which is adjacent to the point q, regardless of whether q is empty or occupied with an enemy stone.

We now give a strategic theory that relates the goals of capturing and saving, using Connect and Sever [2].

When reading these axioms be aware of the following subtleties:

1. It is perfectly possible to connect a block to, or sever a block from, an empty point.

2. In the $Capture$ axiom, the point p is of a different color than the player i so an enemy of p is actually a friend of the player who is trying to capture p.

3. For axiom 2.5.1 the right-to-left implication is trivial since it includes the case $q = p$. Similarly, for the left-to-right implication of 2.5.2. When there are no points q different from p satisfying the right-hand side of axiom 2.5.2, no other capture or save goals have relevance to $Save(p, i)$ and we are in a situation like

[2]The theory is given as a pair of axioms, both of which should, in fact, be theorems derivable from the rules given in chapter 6 for some fixed definition of $Near$. We won't attempt that derivation in this thesis.

that of figure 2.10 where to live p must articulate eyes internally. Similarly for $Capture(p, i)$ in axiom 2.5.1.

The axioms and their English meanings are now given below:

Axiom 2.5.1.

$$Capture(p, i) \longleftrightarrow \forall q \; Sever(p, q, i) \vee$$
$$[[(Near(p, q) \wedge Enemies(p, q)) \longrightarrow Save(q, i)] \wedge$$
$$[(Near(p, q) \wedge \neg Enemies(p, q)) \longrightarrow Capture(q, i)]]$$

Axiom 2.5.1 gives the complete relationship between capturing and saving. It says that a point p is captured iff for each point q, either p can be severed from q, or if q is an enemy stone, near to p then it can be captured, or if q is a friendly stone, near to p, then it can be saved. As mentioned, the predicate $Near$ is currently defined to mean "within a Manhattan distance of 3."

There is a dual axiom relating Save to Capture:

Axiom 2.5.2.

$$Save(p, i) \longleftrightarrow \exists q \; Connect(p, q, i) \wedge$$
$$[[(Near(p, q) \wedge \neg Enemies(p, q)) \wedge Save(q, i)] \vee$$
$$[(Near(p, q) \wedge Enemies(p, q)) \wedge Capture(q, i)]]$$

Axiom 2.5.2 says that a point p is saved iff it can be connected to a point q, near to p that is either friendly and saved, or enemy and captured.

2.6 The goal graph: Using the strategic theory to solve life and death problems

In this section we show how the strategic theory presented in the previous section can be applied to find the relevant goals or context of a life and death problem. We begin by building a model of the theory using an AND/OR graph. The reader should be careful not to confuse this graph with the graph of the game state space – they are entirely different. This graph is a model of the strategic theory. The nodes correspond to goals in the theory; they do not represent positions in the game.

A strategic theory for a particular problem can be modeled as an AND/OR graph which we call the *goal graph*. Figure 2.4 shows the AND/OR graph for the goal of saving point $P17$. We've abbreviated the goals for Capture, Save, Connect, and Sever as "C" and "S", "Con", and "Sev" respectively. The predicate Enemies is abbreviated "En".

Each branch from the root goal corresponds to one valuation of the point variable q in the existential quantifier $\exists q$ in the definition of $Save(p, i)$ given in axiom 2.5.2 above. All points ranging from $(1, 1)$ to $(19, 19)$ are considered in this hypothetical model. The definitions in the goal theory are recursive so goals can occur as descendants of themselves in the graph creating cycles.

We originally formulated the question of whether or not a point p could be saved from a state s as the decision problem $s \models Save(p, i)$. In the goal graph model, the question of the life of the point p is now whether the root goal of the AND/OR graph is true or false. In principle, evaluation of a node in the graph just requires searching for a strategy. For instance, if the node is $Save(p)$ for some point p, we are looking for a

strategy to achieve $TwoEyes(p, q, r, i)$ for some points q and r. If such a strategy can be found then the value of the node is true, and if no such strategy exists then it's false. However, since, this sort of search is intractable in general (which is why we were led to this kind of analysis in the first place), we cannot know what the value of the goal is, unless it can be statically determined. Using the goal graph, we may be able to use the static determination of other nodes to decide a node of interest. For each node that is statically unknown, we examine its children in the graph. If they can be statically determined, then the node itself will be known.

To evaluate a goal graph with root node R, we determine the static value of all the nodes in the graph and then, starting from the root, recursively evaluate the nodes in depth-first order. The algorithm is sketched below:

- Initialize the value of each node in the graph to "true", if it is statically true, "false", if it is statically false, and "unknown" otherwise.
- if node N is known, return its value, otherwise mark N "visited" and recursively evaluate each of N's unvisited children.
- If node N is an AND-node, then it is "true" if all children are true, "false" if any child is false, and "unknown" otherwise.
- If node N is an OR node then it is "true" if at least one child is "true", "false" if all children are "false", and "unknown" otherwise.

In our model we assume that the truth value of non-modal formulas can be decided quickly. For example, it is easy to check, for any points p and q, if $s \models Near(p, q)$. Modal, and specifically adversarial, formulas are assumed to be neither true nor false

but “unknown”, unless we have specific static knowledge otherwise. This reflects the fact that it is generally much more computationally involved to determine their truth, often requiring a search of positions other than the current one for information.

We can use our static knowledge of the truth of predicates such as $Near$ to avoid having to actually build the full AND/OR graph shown in figure 2.4. We use the following observation to eliminate nodes from the graph which are logically redundant. Suppose the theory gives us

$$A \longleftrightarrow B \wedge C$$

for some goals A, B, and C. If B is known statically to be false then A is known statically to be false as well. Similarly, if

$$A \longleftrightarrow B \vee C$$

and B is known statically to be true, then A is known statically to be true as well. In figure 2.4, for instance, we can safely (without affecting the value of the root of the tree) remove any subgraph of $Save(P17)$ for which the leaf node $Near(P17, q) \wedge Enemies(P17, q)$ is statically known to be false, and similarly for any subgraph with a leaf node $Near(P17, q) \wedge \neg Enemies(P17, q)$ which is statically known to be false.

The right side of figure 2.5 shows the AND/OR graph (without the redundant nodes) for the ground goal $Save(P17, BLACK)$ in the life and death problem given on the left of the figure. The nodes of the graph are labeled with the goal they represent. The player argument of the goal is omitted in the graph, so, for instance, $Save(P17, BLACK)$ is written $Save(P17)$. No confusion should result since the player can be determined by the color of the object – $Save$ and $Connect$ goals have the same player color as their objects; $Capture$ and $Sever$ have different player colors from their objects. The object

points are given in the notation <column letter><row number>. The goal whose truth value is being decided is called the *root goal* and is identified in the figure by a double border. In this example, the root goal is "Save(P17)".

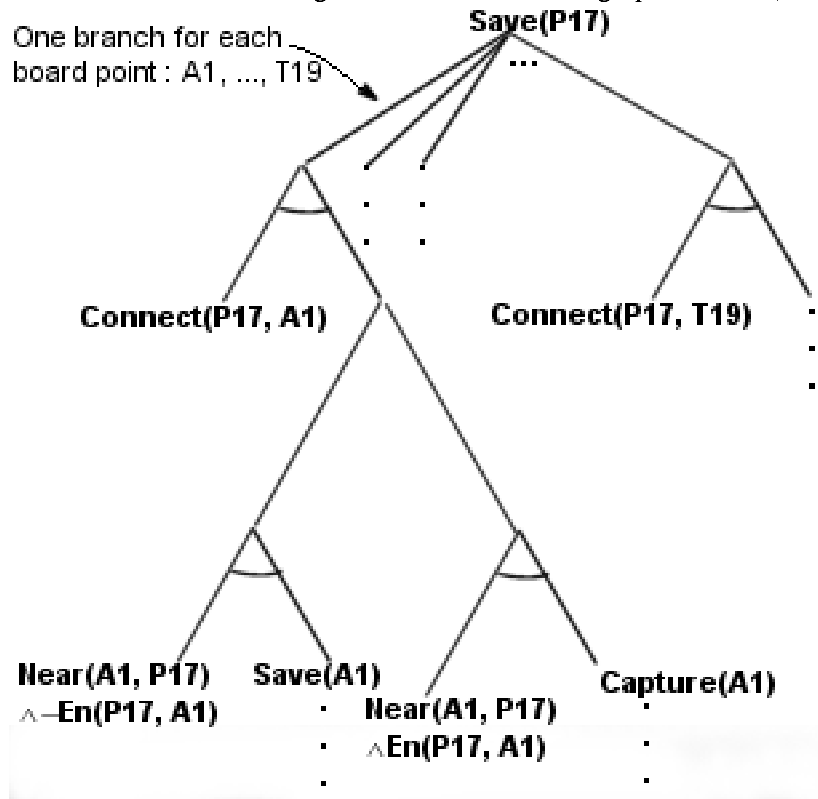
To save space in the diagram we do not always interpret quantified formulas as AND/OR graphs. Normally, an existential quantifier is interpreted as an OR-node with children for each possible interpretation of the quantified variable and a universal quantifier is interpreted as an AND-node similarly. However, if the quantified variable ranges over points there may be a considerable number of nodes that need to be represented in the graph. In practice our rules limit the number of points to a small number but to avoid enumerating all the rules used in the problem we just make one node which stands for the whole sub-graph in the tree.

Some nodes in the graph are decorated on the top with a "+" or "-". These are nodes whose value is known statically to be true or false, respectively. For example, the node corresponding to $Save(O15) \wedge Save(P12) \wedge Save(Q13)$ is marked with a "+". Each of these three goals is statically known to be true. Recall that static knowledge of a goal such as $Save(O15)$ means that we are confident that a strategy exists to establish two eyes for $O15$. When a goal of the form $Save(q)$ is statically known to be true, or a goal of the form $Capture(q)$ is statically known to be false, we say q is *safe*. The safe points are marked in the *Go* position on the left with triangles.

We do not show the expansion of nodes whose value is statically known (marked with a "+" or "-"). Goals whose value is unknown and that have children that are not shown in the graph are marked on the underside with an ellipsis. The goal,

$$\exists q \; Connect(R14, q) \wedge Save(q)$$

Figure 2.4: The AND/OR graph for Save(P17)

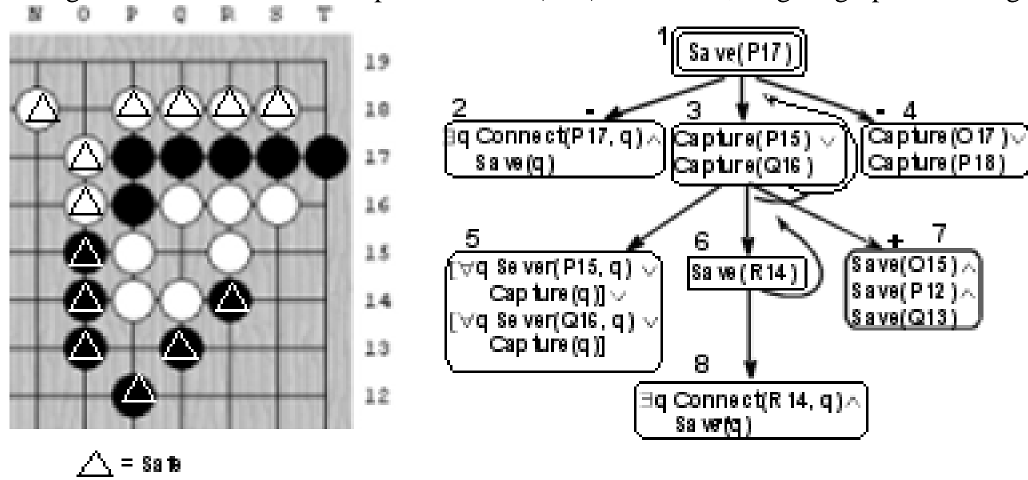


for instance, has children which are elided.

Cycles in the graph indicate mutual dependency. Mutual dependency among goals of interest is very common in human analysis of problems. In figure 2.9, for example, there are only two goals whose value is unknown: $Save(D1)$ and $Capture(G2)$ and they are mutually dependent. This situation is given the name *semeiai* in *Go* terminology.

Figure 2.10 shows a position where there is just the goal to save. Black must play a move to directly save his block by making two eyes. Figure 2.11 shows the position after black has played. Now the root goal graph is statically known and both the black and white stones are safe. In this case, the goal theory is not helpful to us since there

Figure 2.5: A life and death problem Save(P17) on the left; its goal graph on the right.

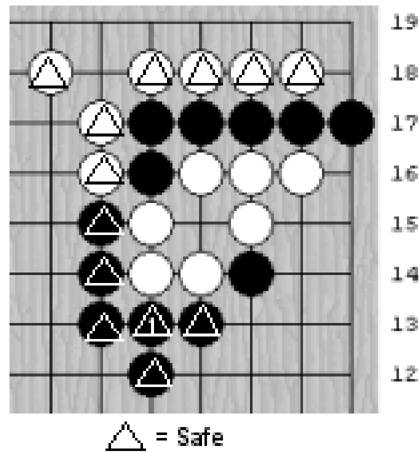


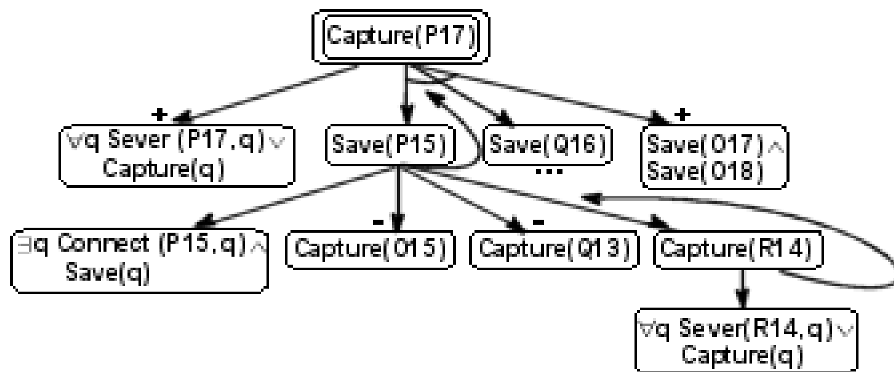
are no friendly blocks to connect to and no enemy blocks to attack. Axiom 2.5.2 tells us only that $Save(D1) \longleftrightarrow Save(D1)$. In this case the move $F1$ is generated directly from the goal $Save(D1)$.

The goal graph represents the universe of relevant goals for solving problem. As long as the root of the goal graph is not statically known (marked with a "+" or "-"), the problem's solution is still unknown and we must continue to search for a strategy to make it statically true. If no such strategy can be found then it is false. We use the goal graph to narrow down the set of moves considered in the strategy search by applying rules from a knowledge base to generate moves for each goal in the graph.

In the starting position, figure 2.5, Black must save her block on $P17$. To do this, she may play moves (like liberty extending moves) to directly save the block, but the goal graph suggests other alternatives too. She might try to connect $P17$ to something that is itself safe. This goal is labeled "2" in the diagram. In this case there are no blocks to connect to and this goal is statically false. Another possibility is to capture the surrounding White blocks: $O17$ or $O18$. This goal is labeled "4" in the diagram

Figure 2.6: The problem after Black plays at 1 (White's turn).

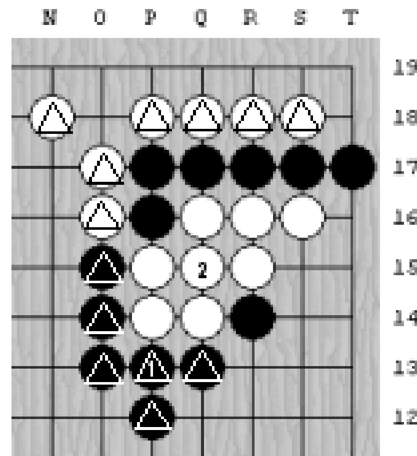


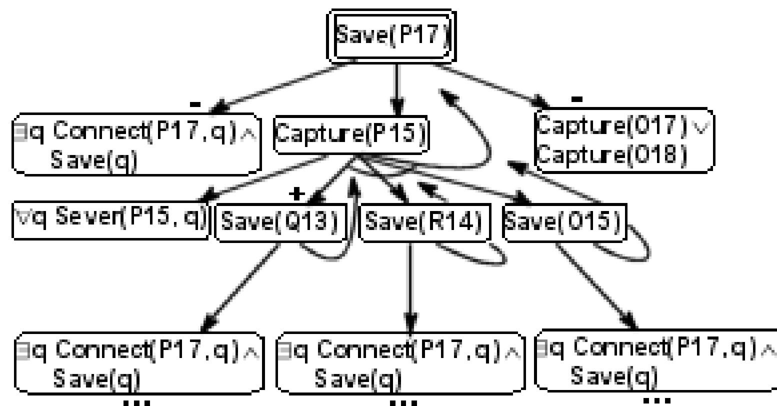


and again is recognized statically as being hopeless. The remaining goal, labeled "3", to capture $P15$ or $P16$ is possible.

When we have analyzed the position fully and determined the statically known

Figure 2.7: The problem after Black plays at 1, and White responds at 2.

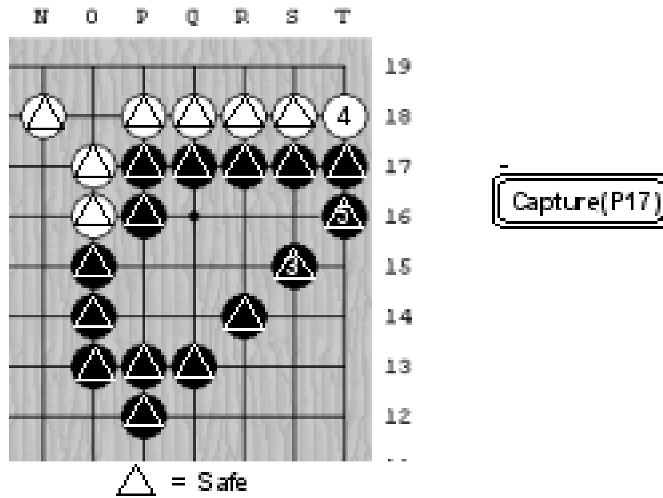




goals, we are ready to generate candidate moves to try to determine if the root goal is true or false (i.e. if there is a strategy to achieve safety for $P17$).

The move generation algorithm generates moves for each node in the graph in

Figure 2.8: Black captures all the White stones, saving P17.



depth-first order starting with the root. No moves are generated for goals which are statically known.

For example, in figure 2.5, to generate moves for the graph rooted at goal 3, we would generate moves for goals 3, 5, 6 and 8 but not from goal 7, since that goal is statically known to be true. No moves would be collected from goal 1 – even though it is a child node of 3 – since it has already been visited on the way to goal 3. The move generation algorithm is given below. R is the root of a goal (sub)graph for which we want to generate moves.

Figure 2.9: Black can win the race to capture by playing on points $G1$ or $F1$.

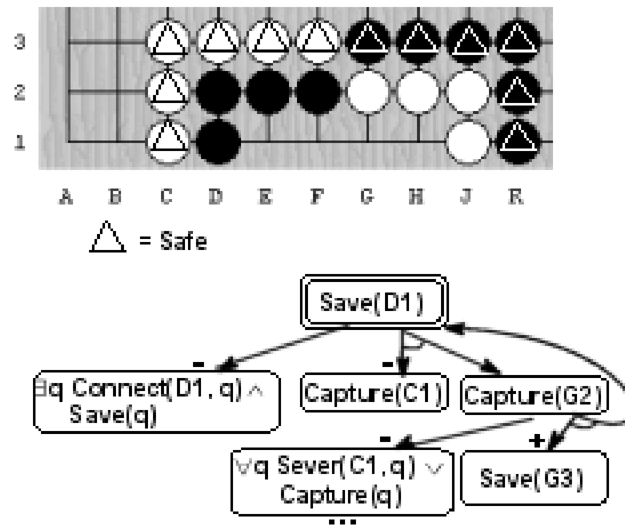


Figure 2.10: Black can make life by playing on point $F1$.

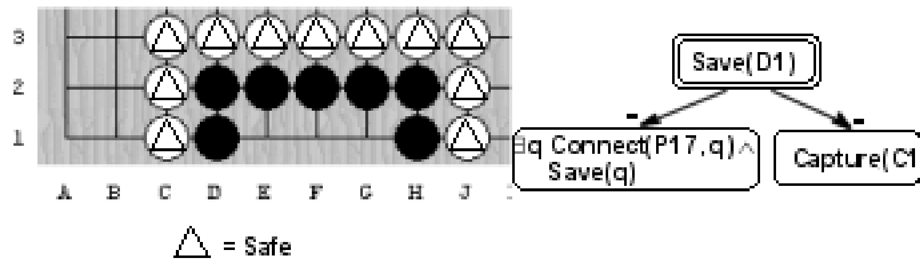
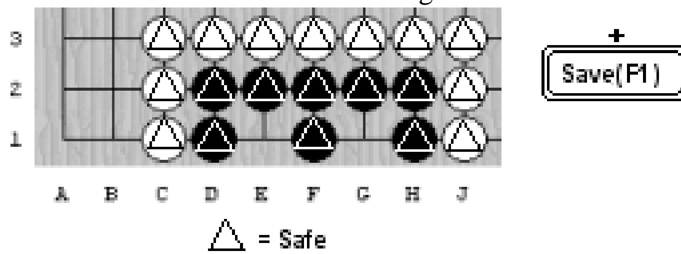


Figure 2.11: Black is alive.



GENERATE_MOVES(R)

(1) **if** R value is statically known

(2) **return** $\emptyset$

(3) Mark R as visited

(4) $Moves \leftarrow$knowledge-base generated moves for$R$

(5) **foreach** unvisited child C of R

(6) $Moves \leftarrow Moves \cup Generate_Moves(C)$

(7) **return** $Moves$

After a move is played, the opponent's goal graph is generated and moves are collected for her to play. When one is selected and played, we generate the player's goal graph in the resulting state, and so on, until the problem is determined or we run out of time. Because positions often do not change radically from move to move, the goal graph after such an exchange is likely quite similar to its original structure. However, sometimes – after a big capture for instance – the structure can change radically. We regenerate the goal graph after each move rather than attempting to update it incrementally.

Continuing with our example, figure 2.6 shows the position after Black tries the candidate move $P13$ [3]. Now it is White's turn and her root goal is the negation of Black's, namely to Capture $P17$. Her goal graph (shown to the right of the diagram) dictates that she must try to save $P15$ and $Q16$. She chose to play at $Q15$ and the resulting position is shown in figure 2.7.

Now it is Black's turn to play and she must again choose a move that works to

[3]For illustration we have selected a move that actually works to solve the problem. The move generation algorithm, of course, does not always choose the correct move to try first.

capture $P15$ or $Q16$. Figure 2.8 shows the position after Black plays at $S15$, White responds at $T18$, Black captures the White stones at $T16$. In this position it is White's turn to play and her root goal, to capture $P17$ is statically determined to be false. At this point the search algorithm would backtrack to the last choice point and continue searching for a solution. When it is determined, finally, that White has no means of preventing Black from saving $P17$, the problem is over.

2.7 Correctness of the algorithm

There are two important assumptions that underlie the correctness of the move generation algorithm. First, the knowledge base must be *complete* in the sense that if there is a move that is part of a strategy to achieve goal R then it is suggested by the knowledge base for R or a sub-goal of R in the goal graph. Second, when a goal is statically known, it is *independent* of its siblings in the graph. More specifically, suppose $G_1 = \mathbf{ach}_i(P_1)$, $G_2 = \mathbf{ach}_i(P_2)$, $G_k = \mathbf{ach}_i(P_k)$. If $G = AND(G_1, G_2, \ldots, G_k)$ is statically known, then P_1 is conjunctively independent of $P_2 \wedge P_3 \wedge \cdots \wedge P_k$. If $G = OR(G_1, G_2, \ldots, G_n)$ then P_1 is disjunctively independent of $P_2 \vee P_3 \vee \cdots \vee P_k$.

This assumption comes from the following analysis. Recall, earlier in section 2.2 we discussed conjunctive and disjunctive independence. There we said that a goal P is conjunctively independent of a goal Q just if whenever it is possible to achieve P and Q separately, there is a strategy to achieve them jointly. In logical terms,

$$[\mathbf{ach}_i(P) \wedge \mathbf{ach}_i(Q)] \longrightarrow \mathbf{ach}_i(P \wedge Q)$$

If we have $\mathbf{ach}(A) \longleftrightarrow \mathbf{ach}(B) \wedge \mathbf{ach}(C)$ for some goals $\mathbf{ach}(A), \mathbf{ach}(B)$, and $\mathbf{ach}(C)$, and $\mathbf{ach}(B)$ is statically known to be true, then, conjunctive independence

of B and C amounts to the statement

$$\mathbf{ach}_i(C) \longleftrightarrow \mathbf{ach}_i(A)$$

This means that in generating moves to try to achieve and AND node $\mathbf{ach}_i(A)$ we are justified in considering only moves directly suggested by the knowledge base for $\mathbf{ach}_i(A)$ and for $\mathbf{ach}_i(C)$ and ignoring moves for $\mathbf{ach}_i(B)$.

Similarly, if we have $\mathbf{ach}_i(A) \longleftrightarrow \mathbf{ach}_i(B) \vee \mathbf{ach}_i(C)$ and $\mathbf{ach}_i(B)$ is statically known to be false then disjunctive independence of B and C amounts to the statement that

$$\mathbf{ach}_i(A) \longleftrightarrow \mathbf{ach}_i(C)$$

This justifies ignoring moves generated by statically false subgoals of an OR-node.

Chapter 3

Program

3.1 Introduction

In chapter 2 we showed how to use a strategic theory of life and death to find moves relevant to a given life and death problem. The move generation algorithm we gave there uses a goal graph of the goals relevant to the problem and a knowledge base of moves to find reasonable moves for each goal in the graph whose value is not statically known. The collection of all such moves is the set of moves that are explored to determine whether the root goal is true or false.

In this chapter we present the data structures and algorithms that are used to statically decide and generate moves for each of the strategic goals. Many of the structures we create to do this will be familiar to go players and programmers; they include: blocks, groups, cores (what some programmers call strings), eyes, and an implementation of the goal graph discussed in chapter 2.

Figure 3.1 shows a schematic of the program. The major data structures occupy the left-hand-side of the"State" box. The right-hand-side shows modules used to interface with the knowledge base and support incremental state update. The Rule Matcher reads rules from the knowledge base and matches them to the board. Rules that match have their assertions stored in an (in-memory) Assertion Database. The Constraint Manager is a truth maintenance system that manages constraints on the board and blocks required for the validity of assertions in the Assertion Database. The Reversion Manager interacts with all the incremental data structures in the state to support backtracking. Since almost every data structure has at least some incremental components, we do not show the connection between the Reversion manager and the rest of the state (it's just marked with a double-line border).

Outside the State box and to the right are the knowledge base, containing the heuristic rules, and PET, the editor used to enter and maintain them.

At the bottom, there is a box for the search algorithm that interacts with the data structures in the state to determine which moves to try at each ply of a problem. The search algorithm is just a standard boolean minimax search with no frills or move-ordering heuristics.

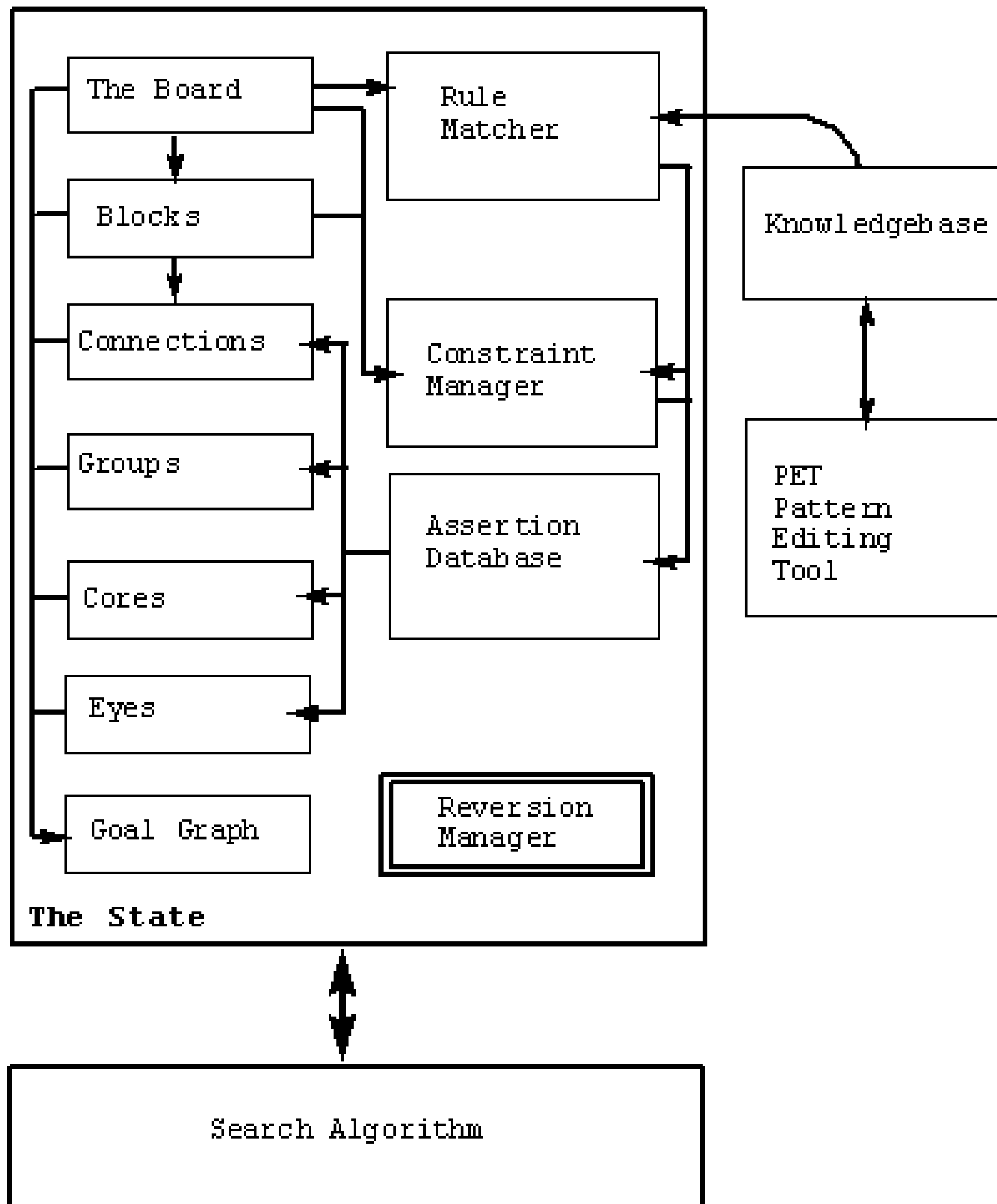
3.2 The Board

Obviously the starting point for any go program is a representation of the board. The board records the color of each of the 361 board points. When a legal move is played on a point, the board is updated to reflect the change. We refer to the color of a point as its *occupancy*. When the occupancy of a point is black or white, we refer to it as a *stone*.

3.3 Blocks

Two points are *adjacent* if one is directly north, south, east, or west of the other on the board; diagonal points are not adjacent. Two points are in the same *block* if they are of the same color (black or white) and connected by a path of adjacent stones of that color. An occupied point p is *block-adjacent* to an empty point q if q is adjacent to some stone in p's block. The set of all such empty block-adjacent points is called the block's *liberties*. The rules of go dictate that when all the liberties of a block are filled, the block is *captured* and all its stones removed from the board. All stones in the block share the same fate – if one is captured they are all captured – so blocks are the basic

Figure 3.1: The program structure.



tactical unit in the game.

It is also useful to consider blocks more loosely related than the adjacency required of the stones in each block.

3.4 Connections

The strategic theory presented in chapter 2 for capturing and saving stones, includes the two goals:

$$Connect(p, q, i) \longleftrightarrow \mathbf{est}_i(BlockAdj(p, q, i))$$

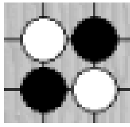
and

$$Sever(p, q, i) \longleftrightarrow \mathbf{irr}_i(\neg BlockAdj(p, q, \bar{i}))$$

To statically decide if $Connect(p, q, i)$ is true, we first search for "connect" rules in the knowledge base that match between the points p and q. If a connect rule is found, we create a *connection* data structure to record this fact. However, the existence of a move to connect p and q is not itself enough to pronounce $Connect(p, q, i)$ statically true. The opponent may still have moves to sever them. If a connect rule is found (and connection structure created) we say that p and q are *possibly connected*. If, in addition, the opponent has no moves to sever p and q, then we say that p is *unassailably connected* to q and consider $Connect(p, q, i)$ statically true [1].

[1] By considering opponent moves in a position in which it is the player's turn to play, we are implicitly assuming that if the opponent has no moves in this state, she will not have any moves after some move by the player. In go this is a reasonable assumption since the player may just pass her turn leaving the state unchanged except for whose turn it is to play.

Figure 3.2: The black and white stones form a *crosscut*.



If no connect rule is found in the knowledge base that matches for points p and q then $Connect(p, q, i)$ is considered statically false and no connection structure is created. The existence of a connection structure means that $Connect(p, q, i)$ is possible (i.e. not statically known to be false).

Connection rules are constructed according the heuristic that, in some (usually many) contexts, the points to be connected must be able to *at least* form a *crosscut*. Figure 3.2 shows a crosscut pattern. Crosscuts are significant because they indicate that there is some hope that the black or white stones will be able to connect by capturing one of the adjacent enemy stones. Without the ability to form a crosscut pattern there is no hope that they will be able to connect if the opponent actively opposes it. Each connection rule in the knowledge base satisfies the property that, when the stones are considered in isolation, with no enemy stones nearby, they will at the very least be able to join in a crosscut pattern.

3.5 Groups

A maximal set of possibly connected blocks is called a *group*. Groups on a point p are computed using a standard transitive closure algorithm on the set of connections from p.

Groups are the "loosest" tactical unit considered in our life and death problem solver. They represent the most optimistic assessment of which blocks will be able

to merge (from the owner of the block's perspective). Very often, however, this assessment is overly optimistic. The opponent may have ways of cutting apart the group by attacking and severing its connections.

Groups are not very useful for solving life and death problems but they are useful for getting a reasonable upper bound on the territory to be gained or lost by attacking a block – something which is necessary for a full-playing go program.

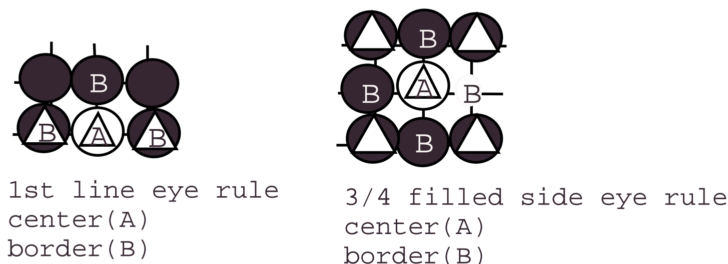
3.6 Cores

The maximal set of blocks that are unassailably connected is called a *core*. Since it is possible that the opponent may not be able to sever either of two unassailable connections but may have a fork to sever one of the two, unassailable connectivity is not transitive. Put another way, even if $Connect(p, q, i)$ and $Connect(q, r, i)$, it is not necessarily the case that $Connect(p, r, i)$.

After all connection structures have been created, we create core structures to record information about the collection of blocks in the core. Except for the possibility of forks, all the blocks in the core can be assumed to live or die together. Cores are thus a sort of natural generalization of the block concept to include stones that are not adjacent but nevertheless known to be able to connect. The largest possible core for a block is the whole group, which reflects the most optimistic assessment of connection between the blocks. Some go programmers call cores, "strings".

When a core is tightly surrounded by the opponent it must be able to articulate eyes internally to save itself. When there is a lot of "friendly" space around the core, it has more options to either articulate eyes if attacked, or connect to another core which has

Figure 3.3: Two rules for recognizing eyes. Stones marked with a triangle are optional.



eyes (or friendly space). We call the measure of the amount of friendly empty space around a core its *aura*.

The life or death of a core depends on its ability to create eyes, either by making them, reinforcing existing ones, gaining access to enough friendly space to be guaranteed to be able to articulate eyes, or to connect to another friendly core that is able to do any of these things.

The next two sections describe data structures that, in conjunction with cores, allow us to statically decide save and capture goals.

3.7 Eyes

Chapter 2 defines an eye for a point p as a liberty point q that is suicide for the opponent. Such points are necessarily surrounded by stones of the same color as p (since these are the only points that are suicide for the opponent).

To statically decide if $Save(p, i)$ is true based on its ability to form eyes, we match eye rules from the knowledge base for the point p. If a rule matches, we create a data structure called an eye structure to record this fact. Figure 3.3 shows two eye rules.

The eye rules tell us the *border* and *center* of the eye. The border is the set of points surrounding the space, the center the set of points that are surrounded. We say two eyes e_1 and e_2 are *disjoint* if the border of e_1 is disjoint from the center of e_2 and the border of e_2 is disjoint from the center of e_1. The borders of two disjoint eyes may overlap.

If the core on a point p has two eyes then we say that core is *possibly alive*. An eye for which the opponent has no attacking moves is called *unassailable*.

If the core on a point p has two disjoint unassailable eyes, then $Save(p, i)$ is considered statically true.

If two disjoint eyes cannot be found for a core, and the core cannot obtain resources in other ways (see section 3.8 below), then $Save(p, i)$ is statically false.

The following rules determine a minimum (min) and maximum (max) number of disjoint eyes for a block on point p.

1. if there are two non-overlapping eyes both of which are unassailable, min = 2 and max = 2.

2. if there are two non-overlapping eyes, exactly one of which is unassailable, min = 1 and max = 2.

3. if there are two non-overlapping eyes, neither of which is unassailable, min = 0 and max = 2.

4. if there is one eye which is unassailable, min = 1 and max = 1.

5. if there is one eye which is not unassailable, min = 0 and max = 1.

6. if there are no eyes, min = 0 and max = 0.

3.8 Aura

Eyes provide one method of statically determining $Save(p, i)$. However, our eye knowledge is not complete. Sometimes, a core may have no eyes (or just one eye) according to the knowledge base, yet still be able to make two eyes by virtue of the amount of empty space surrounding it. Such cases are better characterized algorithmically than by the kinds of rules we have in our knowledge base.

To determine whether a core has enough surrounding friendly space to articulate two eyes under alternating play we use a measure called *aura*. This is a weighted count of the space surrounding the core. The weights are determined by the presence of friendly vs. enemy surrounding cores.

The amount of aura a core has is recorded in the core structure directly. A core with sufficient aura to be able to articulate two eyes (a value determined by trial and error) is *possibly alive*.

If the opponent has no attacks to reduce the aura (drastically) then we say the aura is *unassailable*.

If a core on point p is possibly alive and the core's aura is unassailable then we consider $Save(p, i)$ statically true.

If the core is not possibly alive then we consider $Save(p, i)$ statically false.

Eyes and aura are the only ways a core can be considered possibly alive and $Save(p, i)$ statically true or false.

If it is player i's turn and she has a core on point p for which $Save(p, i)$ is statically true or $Capture(p, i)$ is statically false, then we say the core on point p is *safe*. In the examples given in chapter 2, the safe stones were marked with triangles.

3.9 Knowledge base

Most of the declarative knowledge in our system is knowledge about which moves are reasonable to establish their goal. This knowledge is broken down into a series of *if-then* rules of the form:

> If precondition X holds for object p then move M is reasonable to establish goal ϕ.

The kinds of conditions that X can represent divide naturally into two categories: strict occupancy constraints, such as "There must be a Black stone on point p" and attribute constraints such as "The Black block must have 2 or more liberties." In addition we can have boolean combinations of constraints.

Because board occupancy plays such a central role among the constraints in most of our rules, we developed a graphical pattern editor called PET that allows the knowledge maintainer to enter these constraints graphically on rectangular board sections called "maps". Figure 3.4 shows a variety of typical pattern maps. Rules may have multiple maps linked by some attribute constraint that identifies common objects in the two.

The knowledge base currently has 725 rules. Table 3.1 shows the breakdown of rules by purpose. These purposes are a way of categorizing the knowledge more finely than the four strategic goals of chapter 2. Each purpose has associated rules that apply to one of the strategic goals in some particular circumstance. The strategic goal for each rule type is given in parentheses following the purpose name.

Figure 3.4 shows a variety of different rules.

Table 3.1: Number of rules for each purpose.

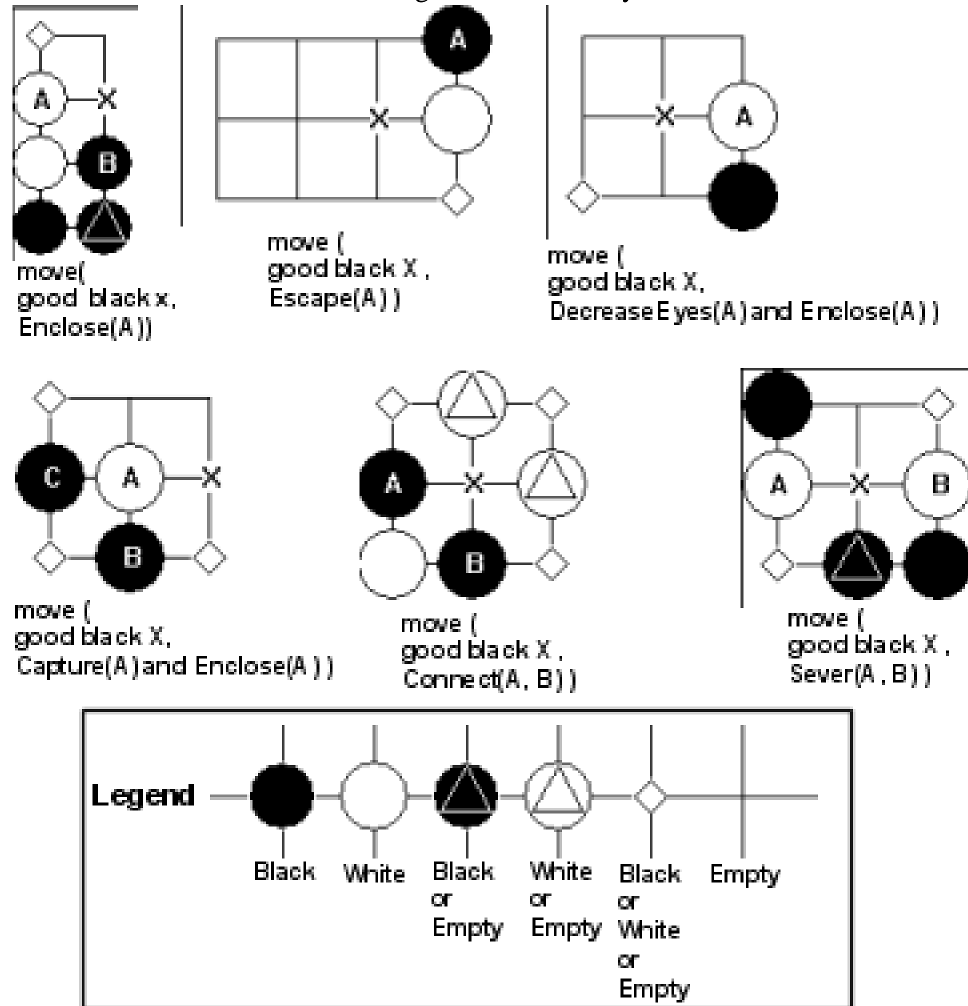
Purpose	*Number of rules*
Enclose (Sever)	8
Escape (Connect)	1
Eye fight (Save, Capture)	229
Eye recognition (Save, Capture)	126
Save and Capture (Save, Capture)	156
Connection recognition (Connect, Sever)	22
Sever/Split/Connect/Join (Connect, Sever)	183
Total	725

3.10 Rule filtering

As often happens in knowledge-based systems one would like to be able to override general rules when more specific ones apply. To determine a ranking among rules that match a given position, we assign a *specificity* to each rule computed as a measure of the information content of the pattern. This is determined at the time the pattern is created and stored with the pattern. Among all the patterns that hold in a given state s for a particular goal and objects, we prefer the most specific, which we call the *dominant* pattern. We say the dominant pattern *filters* the less specific ones.

For example, it is also useful to be able to override general eye knowledge with more specific knowledge. Figure 3.5 shows a position where Black might have two disjoint eyes by virtue of two matches in the corner of the rule in figure 3.6. The "B" indicates a border point, the "C" indicates a center point of the eye. Reading this position out we would determine that in fact, Black can make only one eye, but reading the situation out

Figure 3.4: A variety of rules.



takes time. To make a static determination that Black has just a single eye, we need an additional rule that matches the position and has higher specificity than the rule shown in figure 3.6. Figure 3.7 shows the occupancy constraints of such a rule. Typically, we would also require that the surrounding white blocks be safe, so that Black could not capture them as part of an escape.

Figure 3.5: Applying rule 3.6 incorrectly gives Black two disjoint eyes (borders marked with "B", centers with "C".).

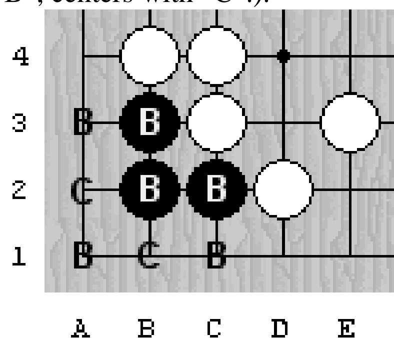
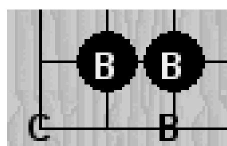


Figure 3.6: An eye rule.

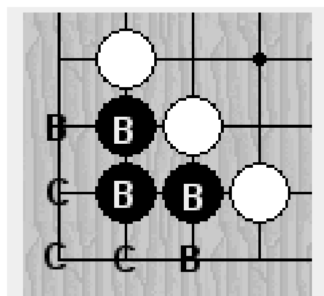


3.11 Constraint Manager

To maintain the integrity of the data structures representing board knowledge we employ a constraint manager to propagate board occupancy changes to any data structures that may be affected.

Each data structure may be either a *ward* or a *support* (or both). *Wards* are objects

Figure 3.7: A more specific eye rule.



that require notification if some object they depend on changes. *Supports* are objects that are depended upon by other objects. When a support object changes (including possibly when it is created and destroyed) it alerts any wards that have registered themselves as dependent. The notification received by the ward includes a message parameter that allows the support to pass information about the nature of the change. The ward itself may be a support of another ward. In this case when it receives an alert it may in turn alert another ward. Obviously, support cycles are prohibited.

There are two types of wards that behave differently when alerted: synchronous and asynchronous. Asynchronous wards respond by alerting their wards immediately upon receiving an alert themselves. Synchronous wards require a notification to propagate alerts to their wards.

Synchronous notification is useful in the case where a ward has multiple dependencies and requires knowledge of all of them before making a decision about alerting its ward. Consider for example an AND-node with two children X and Y whose values are 0 and 1, respectively. In the asynchronous case, if X changes value to 1, the AND-node itself changes value from 0 to 1 and anything that depends on it must be notified. However, if later, Y changes from 1 to 0, the AND-node's value changes back to 0, and again anything dependent on it must be notified. Using synchronous notification instead, we wait until both X and Y are finalized before alerting any wards, avoiding two potentially expensive updates.

The order in which the data structures of the program are provided synchronous notification is determined by an update cycle that reflects their natural dependencies: first the board occupancy, then blocks, relationships, groups etc.

The constraint manager takes board and block changes and notifies anything that

depends on them through a constraint system. An atomic *constraint* is an expression of the form:

$$\langle term \rangle \; \langle RelOp \rangle \; \langle term \rangle$$

A $\langle term \rangle$ is

$$Occ(p) \mid Libs(p) \mid number$$

A $\langle Relop \rangle$ is

$$< \; \mid \; <= \; \mid \; = \; \mid \; > \; \mid \; >=$$

$Occ(p)$ takes values $BLACK, WHITE$, or $EMPTY$ depending on the occupancy of the point p. $Libs(p)$ is the number of liberties of the block on point p.

Each atomic constraint has a value determined by evaluating its expression. In addition, we have AND/OR-constraints that can have an arbitrary number of constraint children. Using AND and OR constraints we can build arbitrary boolean constraints on the four terms listed above. The constraint manager notifies any constraints whose terms have changed since the last move and the constraint propagates the notification up to its parent recursively.

3.12 Assertion Database

The assertion database holds all of the rule-derived facts or assertions about the position. Assertions may be of different types; the data for the assertion varies according to type. Not all assertion purposes have rules associated with them yet in the knowledge base.

All assertions have the following fields:

- name – the name or purpose of the assertion. One of
  - Defensive purposes: {increase liberties, break out, escape, connect, join, increase eyes, reinforce eyes, save, live, increase aura, semeiai increase liberties }
  - Offensive purposes: {decrease liberties, trap, enclose, sever, split, decrease eyes, destroy eye, capture, kill, decrease aura, semeiai decrease liberties}
- move number – the move for which this assertion was created.
- specificity – an integer measuring the amount of information in the precondition of the rule that matched to make the assertion. Used for deciding between conflicting assertions of patterns that match the current position.

In addition, assertions have one or more of the following attributes:

- objects – Up to three object points. The objects for which the assertion holds.
- Rx point – The Rx or *prescribed* point. Where to play.
- color – The color of the player for whom the assertion obtains.
- recommendation – The recommendation value of the suggested move: {GOOD, BAD, THREAT }.
- strength – An integer value giving a rough measure of the strength of the connection (not currently used).
- border – the set of points surrounding the eye that would be occupied if the eye were fully articulated.
- center – the set of points inside the eye.

3.13 State change – The Reversion Manager

There are really three main options for handling state changes.

1. Redo state each time a move is made or backtracked.
2. Come up with some transparent system for automatically recording and undoing changes to the data structures.
3. A mixture of 1 and 2.

If the state is small, or there is not much code in the state update then option 1 is a viable alternative. However, for systems with heavy update requirements option 1 becomes inefficient if most changes to the state are localized; that is if, on average, more of the state remains unchanged from move to move than changes. This has the unfortunate side-effect of placing subtle psychological pressure on the programmer not to add more knowledge. More knowledge means more updating each ply and although it although it may be easier to program and faster to get up and running, it may become a bottleneck in the exploration of new ideas.

Options 2 and 3 are really the best choices in our estimation since they are computationally efficient, do not place pressure on the programmer to keep the state small, and can be implemented at the data structure level making it almost transparent to the programmer in the implementation of her algorithms.

Our system is based on the following observation: In an object-oriented setting all access to an object is through the class interface. If some preprocessing must be performed by the object before actually performing an operation, it can be hidden from the caller in a "wrapper" function. In our case the preprocessing consists of saving the

state of the object before any operation that changes it.

There are two key points to observe here. First, when we revert to an object's previous state after a backtrack we can copy the old version to the same memory location as the new version. This ensures that all references to the object remain valid. Second, we needn't copy an object every time it is changed, we can construct "frames" that group together changes to increase the granularity of change. After a backtrack, an object is reverted to its previous state at the beginning of the last frame. Changes made during a frame are not recorded and cannot be reverted. For the purposes of our go program we identify a frame with a state in the game. In this way the first change made to an object during a state update is recorded but subsequent changes made during the same update process are not.

3.14 History of the project

David Mechner and I started developing our program in 1992 with a grant from the Cambridge Center for Behavioral Studies (CCBS). Our initial goal was to develop a full playing knowledge-based go program. Just getting the infrastructure in place to be able to work comfortably with the system took a couple of years. We had to decide on an appropriate scheme for knowledge representation, build a knowledge editor with a GUI front-end, a truth-maintenance system, revertable [2] class library, board representation and display, pattern matcher, and many other bits and pieces necessary for a working system.

It wasn't until 1997 after several rewrites and platform changes that we had a work-

[2] We call a data-structure *revertable* if it supports an "undo" operation which reverts it to its state prior to the last recorded change.

ing problem solver. At that point we turned our attention to improving its knowledge and testing it on problem sets. We began our project with the idea that it would be a full go-playing program but quickly became focused on the problem of life and death. Life and death knowledge is central to accurate positional evaluation so we were determined to do a good job of it. We imagined we could build a good quality life and death problem solver and still have time to incorporate it into a playing program. As it turned out we have only been able to implement the problem solver, a full-playing program infrastructure, and some pieces of a full-playing program.

We determined that there were three basic kinds of knowledge we would need. First was *state* knowledge – the important facts about a position that go into making a move decision. This includes the standard repertoire of computer go entities such as stones, blocks, groups, cores (or strings) etc. These entities are computed and stored in data structures when the position changes. The code for their computation is quite static – once written it is unlikely to change much – so it makes sense that this kind of knowledge is essentially procedural rather than declarative.

The second kind of knowledge required was about which moves are *reasonable* in a particular situation. This knowledge changes fairly frequently as the rules are refined in response to testing so recording it in declarative form was important. We built the knowledge editor PET to make this job easier.

We trained the system on Kano's *Graded go problems for beginners*, a series of four volumes of increasing difficulty. When the program had enough knowledge to solve many of the training set problems, we tested it on a fresh set of untried problems. The next chapter discusses our results in more detail.

Chapter 4

Results

4.1 Testing and Results

To test the quality of our program's life and death analysis, we first trained it on the odd numbered problems in Kano's [13, 14] Graded Go Problems for Beginners: Volumes 1, 2, and 3, entering knowledge when it failed to see a solution path and filtering generally good moves that were bad in a particular context. Problems that asked for the best move or were otherwise not suitable for our use we excluded. Problems that required the solver to both find the right object and solve the problem were given the object. For example, if the problem stated "How can Black capture three white stones?" we would identify the block of three stones that was intended and provide this information to the problem solver. Guessing the correct object was almost always a trivial part of the problem.

After training we tested the program on the even numbered problems in volumes 1 and 2. Our hope was that the even problems would cover the same range of material but would be different enough to test whether we had trained the program with general go knowledge or just specifically how to solve the odd numbered problems in this book. The results were encouraging, especially for volume 1; they are summarized below.

It is somewhat difficult to compare results with other programs because of two factors. First, we are interested in life and death, not middle-game problems, so we focused on these in our testing. But, more importantly, we required our program to completely solve the problem by reading it out until a static analysis could be made. The program hypothesized moves then solved the sub-problems that resulted from playing the moves at each ply. Mueller [18] and Fotland [11] give their program's performance on some of the same problems but define a solution as finding the correct first move, not the correct

first move at each ply until the problem is solved.

Table 4.2 gives our results on a series of graded go problems of varying difficulty. Table 4.5 is a summary. It would be helpful to have the volumes handy as a reference when examining this data. We obviously cannot reproduce the problems here.

The program was not trained on any of the test problems, so these results – especially for volume 1 – indicate some degree of true competency at beginner life and death, not just knowledge of the problem set.

4.2 Two problems: A success and a failure

In this section we briefly examine two problems from the sample set – book 1, problem 24, which the program successfully solves, and book 2, problem 42, which the program fails to solve.

Figure 4.1 requires black to capture the stone marked P. The goal graph for this problem suggests 6 initial moves for black (labeled in the order they will be explored). The correct move is move 2. In this case moves 1 and 2 both have the same rank, but move 1 is chosen randomly to be first.

Figure 4.2 shows the position after black plays at 1, white responds at 2, and black connects at 3. At this point, the white block on point P is statically safe and black gives up backtracking to the choice of move 1 as her first move.

Next, black considers playing at 2. Figure 4.3 shows the position after black plays at 2, white responds at 5, and black ataris the block on point P. At this point, all the black stones are statically safe and the block on point P is statically dead. Black has succeeded and reading backtracks to white's last choice point. None of white's other

Figure 4.1: Correctly solved problem 1-124.

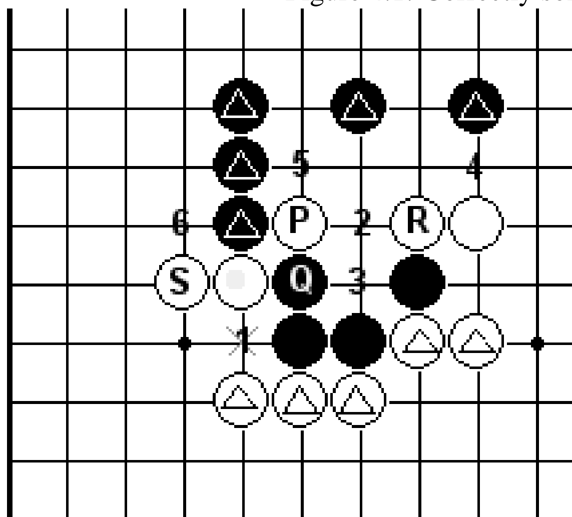
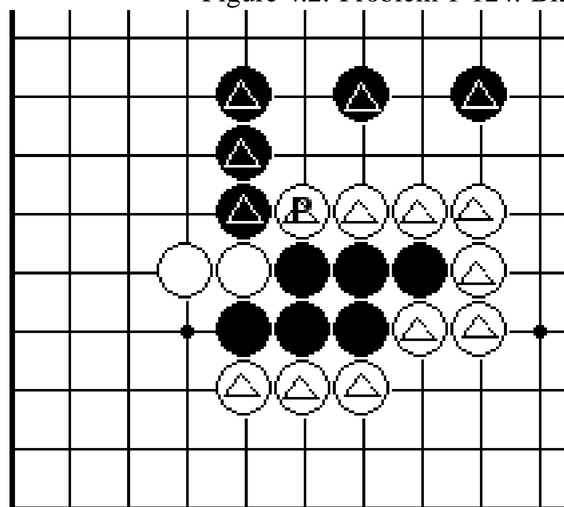


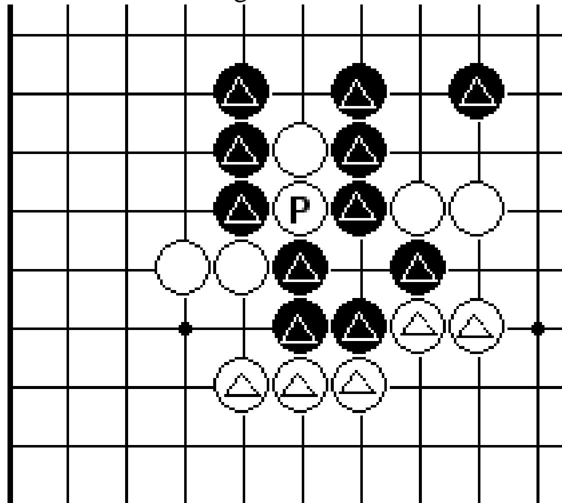
Figure 4.2: Problem 1-124. Black chooses the wrong path.



moves succeed and the problem is declared a success for black.

Black's other moves at 3-6 are not explored, but it is interesting to see why they were suggested. The move at 3 is suggested because a subgoal of $Capture(P)$ in the goal graph, is the goal $Save(Q)$ and one way to save Q directly is to try to extend its

Figure 4.3: Problem 1-124. Black chooses the right path.



liberties.

Move 4 is suggested to capture the white block R, which is a subgoal of $Save(Q)$ (which in turn is a subgoal of the root goal, $Capture(P)$).

Move 5 is suggested for $Capture(P)$.

Move 6 is suggested for $Capture(S)$, which is also a subgoal of $Save(Q)$.

Book 2, problem 42, shown in figure 4.4, is a problem that the program solved incorrectly. The goal is for black to capture the white block P. The only move generated for this purpose is move 1, which fails. The correct move is at X, since this forces white into a shape that can only make one eye. Unfortunately, the program was missing an eye rule. The program failed to recognize that White has two disjoint non-overlapping eyes (if she gets to play at X herself). No eye attacks were generated, and hence the correct move was missed. Luckily, this problem required very little time to get wrong.

Missing rules, such as the eye rule for problem 42, were often the culprits in incorrectly solved problems. Another major source of error, which often led to lengthy

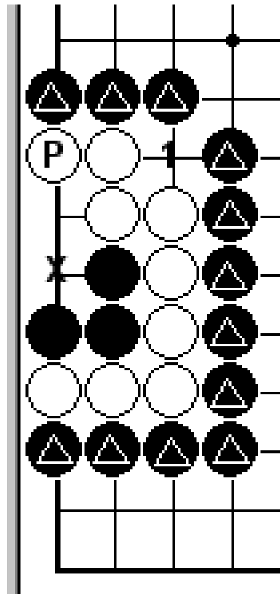
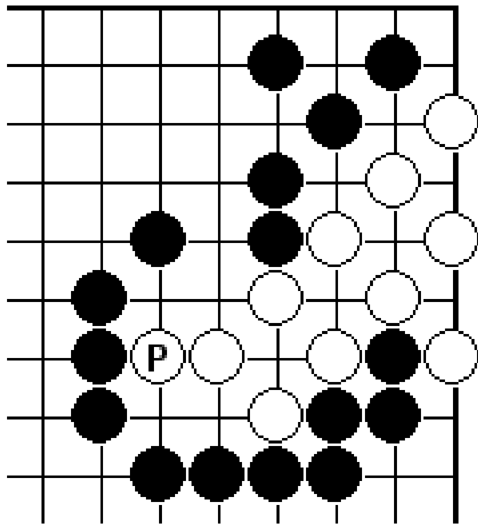


Figure 4.4: Incorrectly solved problem 2-42.

reading and, ultimately caused the program to run out of time, was overly general patterns in problems with many objects involved (problems with lots of cutting points, for example). An early mistake in such a problem can mean a tremendous amount of wasted reading. Figure 4.5 shows an example of a problem which the program was not able to finish.

In every case though, the program did very well in its analysis of the problem into goals and the goal relations. Missing rules can be added, overly general rules filtered and more static knowledge added, but the goal theory is right, the program has the ability to determine which stones are logically relevant to the problem and that is a big step forward in reasoning about life and death.

Figure 4.5: Incorrectly solved problem 2-108 (gets lost).



Book	Problem	Result	Solution	Correct?	Num nodes	Time (secs)
1	2	S	S	Y	2	1
1	4	S	S	Y	2	2
1	6	S	S	Y	2	1
1	12	S	S	Y	2	1
1	14	S	S	Y	2	2
1	16	S	S	Y	2	2
1	18	S	S	Y	2	1
1	20	S	S	Y	2	1
1	24	S	S	Y	12	16
1	26	S	S	Y	4	5
1	28	S	S	Y	14	14
1	32	S	S	Y	2	1
1	42	S	S	Y	2	1
1	44	S	S	Y	2	1
1	46	S	S	Y	2	1
1	62	S	S	Y	2	2
1	64	S	S	Y	2	1
1	66	S	S	Y	2	2
1	70	S	S	Y	2	2
1	74	S	S	Y	4	4

Table 4.1: Results on Kano's Graded Go Problems for Beginners: vols. 1-2

Book	Problem	Result	Solution	Correct?	Num nodes	Time (secs)
1	78	S	S	Y	4	2
1	80	S	S	Y	6	5
1	80	S	S	Y	2	1
1	82	S	S	Y	2	1
1	84	S	S	Y	14	12
1	90	S	S	Y	2	2
1	94	S	S	Y	2	0
1	96	S	S	Y	4	5
1	102	U	S	N	1	0
1	104	S	S	Y	2	1
1	106	S	S	Y	4	4
1	108	U	S	N	1	0
1	112	S	S	Y	2	1
1	116	U	S	N	2	2
1	122	S	S	Y	2	1
1	126	S	S	Y	6	8
1	128	S	S	Y	4	3
2	2	S	S	Y	24	23
2	4	S	S	Y	74	105
2	6	S	S	Y	2	11

Table 4.2: Results on Kano's Graded Go Problems for Beginners: vols. 1-2

Book	Problem	Result	Solution	Correct?	Num nodes	Time (secs)
2	10	S	S	Y	20	29
2	14	U	S	N	1	1
2	14	U	S	N	1	1
2	16	S	S	Y	2	1
2	18	S	S	Y	100	125
2	24	S	S	Y	10	8
2	30	S	S	Y	2	0
2	34	S	S	Y	2	1
2	36	U	S	N	1	0
2	38	U	S	N	1	1
2	40	U	S	N	1	1
2	40	U	S	N	1	1
2	42	F	S	N	3	1
2	44	F	S	N	11	16
2	46	S	S	Y	4	3
2	66	S	S	Y	2	2
2	70	S	S	Y	18	24
2	76	U	S	N	1	0
2	80	S	S	Y	2	2
2	82	S	S	Y	2	2

Table 4.3: Results on Kano's Graded Go Problems for Beginners: vols. 1-2 cont.

Book	Problem	Result	Solution	Correct?	Num nodes	Time (secs)
2	84	U	S	N	1	0
2	86	S	S	Y	56	73
2	88	F	S	N	249	305
2	108	F	S	N	1	0
2	110	U	S	N	15	29
2	112	F	S	N	1	1
2	116	S	S	Y	2	1
2	118	S	S	Y	2	0
2	122	F	S	N	49	36
2	134	F	S	N	9	9
2	134	F	S	N	9	9

Table 4.4: Results on Kano's Graded Go Problems for Beginners: vols. 1-2 cont.

Text	# probs	% correct	avg. tree size
Kano (vol. 1)	37	90	3
Kano (vol. 2)	34	59	18

Table 4.5: Summary of results on Kano's Graded Go Problems for Beginners: vols. 1-2

Chapter 5

Game Logic

5.1 Introduction

Until very recently, games have been considered almost exclusively for interpretation of other kinds of logical systems. In this chapter we go the other way and use logic to analyze games. This idea was first proposed in a paper by Parikh [19] and more recently considered by Pauly [2], however both of these papers consider games as atomic entities; our interest lies in the analysis of the *internal* structure and strategies of games. To accommodate this added expressivity requirement, we have chosen the first-order modal μ-calculus as the basis for our logic.

In the next section we briefly introduce modal logic and then consider the models, axioms, and semantics for our game logic. In the next chapter we will apply this logic to stating rules for *Go*.

5.2 Introduction to Modal Logic

Modal logic is a branch of logic originally formulated to characterize the notions of possibility and necessity but more recently applied to formal reasoning in a variety of domains including discrete time, computer program execution, action, proof theory, circuit verification and many others. Any domain in which it is desirable to make statements that are true or false relative to a particular state of affairs (or time, or game position etc.) is potentially a candidate for description in modal logic. Many of these applications require only propositional modal logic but some – especially those involving knowledge representation – need the extra expressivity offered by predicate modal logic. The standard introductory reference for propositional modal logic is [12]. A very good recent text is [5]. For first-order modal logics [6] and [10] are good references.

The μ-calculus is a logic of operators on modal sentences which includes μ – the least fixed point – and ν – the greatest fixed point operators. Any modal language can be augmented with axioms for the μ-calculus to allow construction of more complex operators from the primitives provided by the basic language. We use the modal operators to define *adversarial operators*.

A standard reference paper on the modal μ-calculus is [15]. Completeness results for the propositional modal μ-calculus are given in [25].

5.3 Models

The models for our logic are game trees whose nodes represent positions and whose edges are labeled with the physically possible moves in the game. The fact that our models are trees and not just rooted graphs means that identical positions reached by different move sequences have different nodes in the model. The physically possible moves may include moves that are not legal, such as moving a knight forward one square in Chess. We could restrict ourselves to only legal moves, with respect to some atomic definition of legality, but the rules of the game are themselves interesting and to present these formally requires being able to refer to all the possible moves, not just the legal ones. For example, in *Go* the concept of suicide move m is easiest to define in terms of what happens after a player plays m, regardless of whether m is legal.

Our models specify two opposing players, a predicate to determine which moves are legal for each player, a turn structure, and a function to determine who has won. We assume a set of moves from which the players may choose. In *Go* the set of moves available to both players is the set of the 361 available board points. Obviously, not all

of these are legal moves in any given position, but they define the universe from which the players may choose.

The nodes of the game tree are Tarskian (first-order) models. They have a set of domains corresponding to the important entities in the game, as well as functions and predicates over those domains. We assume that the domains are the *same* for each node in the tree.

More formally, a *game structure* is a labeled transition system with the following components:

- A set of Tarskian models $\mathcal{S}$.
- A finite set of directed labeled transitions (moves) $\mathcal{M} = \{m_1, m_2, \ldots, m_n\}$.
- A set of two players $\mathcal{I}$.
- A set of (partial) "reachability" functions $\mathcal{R}$. For each $m \in \mathcal{M}$, there is a partial function $\mathcal{R}_m \in R$,

  $$\mathcal{R}_m : \mathcal{S} \longrightarrow \mathcal{S}$$

  $\mathcal{R}_m(s)$ is undefined iff m is not a possible move from s.
- Each reachability function $\mathcal{R}_m$ has an inverse function, $\mathcal{R}_m^{-1}$. $\mathcal{R}_m^{-1}(s)$ is undefined iff m is not a move that leads to s.
- A function $\mathbf{ToPlay} : \mathcal{I} \longrightarrow 2^{\mathcal{S}}$. For each player this function gives the states in which it is that player's turn to play.
- A function $\mathbf{Legal} : \mathcal{M} \times \mathcal{I} \longrightarrow 2^{\mathcal{S}}$. For a given move m and player i, this function gives the states in which it is legal for i to play move m.

- A function $\mathbf{Win} : \mathcal{I} \longrightarrow 2^{\mathcal{S}}$. For a given player, this function gives the states which are winning for that player.

We require that $\mathcal{R}_m$ and $\mathcal{R}_m^{-1}$ be inverse in the following way:

$$\text{If } \mathcal{R}_m(s) \text{ is defined then } \mathcal{R}_m^{-1} \text{ is defined on } \mathcal{R}_m(s) \text{ and } \mathcal{R}_m^{-1}(\mathcal{R}_m(s)) = s \tag{5.3.1}$$

$$\text{If } \mathcal{R}_m^{-1}(s) \text{ is defined then } \mathcal{R}_m \text{ is defined on } \mathcal{R}_m^{-1}(s) \text{ and } \mathcal{R}_m(\mathcal{R}_m^{-1}(s)) = s \tag{5.3.2}$$

We require that it is exactly one player's turn to play in any given state: For all states $s \in S$, for players $i, i' \in \mathcal{I}$

$$[s \in \mathbf{ToPlay}(i) \cap \mathbf{ToPlay}(i')] \text{ implies } i = i' \tag{5.3.3}$$

We require that if it is not a player's turn in a state s, then that player has no legal moves in s: For all states $s \in \mathcal{S}$, for all players $i \in \mathcal{I}$, for all moves $m \in \mathcal{M}$

$$\mathbf{Legal}(m, i) \subseteq \mathbf{ToPlay}(i) \tag{5.3.4}$$

To ensure that the structure is a tree, we stipulate the following conditions:

$$\mathcal{R}_m^{-1}(s) \text{ defined and } \mathcal{R}_n^{-1}(s) \text{ defined, implies } m = n \tag{5.3.5}$$

and

$$\text{There is a unique state, } s_0 \text{ , the \textit{root} state, for which } \mathcal{R}_m(s_0) \text{ is undefined for all } m \tag{5.3.6}$$

We do not make any assumption about whether or not a state can be winning for more than one player. In games such as *Go* or chess, the winning states for each player are disjoint.

5.4 The first-order modal μ-calculus

The predicate modal μ-calculus is a predicate modal logic with a least fixpoint operator, "μ." Propositional modal μ-calculus originated with Scott and De Bakker [22] and was further developed by Hitchcock, Park, De Bakker and De Roever [20], [4], [7]. Kozen [15] provides the axiomatization we use here.

The variables "X" and "ϕ", in the examples below are *predicate* variables not a first-order domain variables. Predicate variables can be bound or free like first-order variables. If a predicate variable X occurs within the scope of a μX or νX operator it is *bound*, otherwise it is *free*. Typically bound predicate variables will be denoted by upper-case letters from the end of the alphabet such as X, Y, or Z. Free variables will be denoted by lower-case Greek letters such as ϕ or ψ. If all the predicate variables in a μ-calculus formula are bound then the formula is *closed* otherwise it is *open*.

Before we proceed further, it will be useful to define some abbreviations:

- $start \equiv [u]\bot$
- $term \equiv \Box\bot$

As usual, $\bot$ denotes "falsum" – the proposition that is always false. Observe that $start$ is only true at the root of the tree and $term$ at its leaves. Also, note that $start$ is true only in the root state. If there is a sequence of moves that returns the position to the root position, this node will be a descendant of the root and $start$ will not be true there.

In a finite model, you may imagine the formula $\mu X.F(X)$ as standing for the (finite)

disjunction:

$$F(\bot) \vee F(F(\bot)) \vee F(F(F(\bot))) \vee \cdots F^n(\bot)$$

This intuition is only valid for finite models. If the models are infinite then there is no guarantee that the least-fixpoint formula will "converge" in ω steps. Since we are primarily concerned with finite models, it does no harm to think about it this way.

Here are some more examples to give the intuition behind the μ operator:

1. The statement,

   $$s \models \mu X.\phi \wedge \Box X$$

   means, for some n,

   $$s \models \phi \wedge \Box(\phi) \wedge \Box(\Box(\phi)) \wedge \cdots \wedge \Box^n(\phi)$$

   which in English means that ϕ is true in state s and in all descendants of s.

2. The statement

   $$s \models \mu X.\phi \vee \diamond X$$

   means, for some n,

   $$s \models \phi \vee \diamond(\phi) \vee \diamond(\diamond(\phi)) \vee \cdots \vee \diamond^n(\phi)$$

   which means that ϕ is eventually true along some path from s.

3. The statement,

   $$s \models \mu X.\Box X$$

means, for some n

$$s \models term \vee \Box(term) \vee \Box(\Box term)) \vee \cdots \vee \Box^n(term)$$

which means that the tree rooted at s is finite.

4. The statement,

$$s \models \mu X.[u]X$$

means, for some n

$$s \models start \vee [u]start \vee [u]([u]start) \vee \cdots \vee [u]^n(start)$$

which means that every upward path reaches a root.

5.5 First-Order Modal Game Logic

Our modal language consists of the usual first-order logical symbols (with equality) and the following additions.

- A finite set of constants denoting moves in the game.
- A set of two constants denoting the two players in the game.
- A predicate on player terms, $ToPlay(i)$, intended to mean that it is player i's turn to play.
- A predicate on move and player terms, $Legal(m, i)$, intended to mean that move m is legal for player i.

- A predicate on player terms, $Win(i)$, intended to mean that player i has won the game.

We add a family of modal operators – one for each move term in the language – $\diamond_m$ and its dual $\Box_m$. We read "$s \models \diamond_m \phi$" as meaning "move m, made from state s, leads to a state in which ϕ is true". Dually, "$s \models \Box_m \phi$" is taken to mean that "Either, move m taken in state s, leads to a state in which ϕ is true or m is not a possible move from state s". The operators $\diamond_m$ and $\Box_m$ are equivalent except at leaf states where $\Box_m$ is true and $\diamond_m$ is false.

The operator $\diamond_m$ is a "downward" operator in the sense that it examines the children of a node in the game tree. It is also useful to have an "upward" operator which checks the parent of a node. We define an upward version of diamond, $\langle u \rangle_m$, which is true for a formula ϕ at a state s when ϕ is true in a parent of s with a transition labeled m to s. The angle brackets around the u in $\langle u \rangle_m$ are meant to evoke a $\diamond_m$ operator (split down the middle). Like $\diamond_m$, $\langle u \rangle_m$ has a dual $[u]_m$ which is identical to $\langle u \rangle$ except at the root where $[u]_m \phi$ is always true and $\langle u \rangle_m \phi$ is always false.

These operators are more primitive than the $\diamond$ and $\Box$ operators usually considered primitive in basic modal languages. In fact, we can define $\diamond$ and $\Box$ in terms of $\diamond_m$ and $\Box_m$ as follows:

$$\diamond(\phi) \equiv \exists m \;\; \diamond_m (\phi)$$

and

$$\Box(\phi) \equiv \neg \diamond \neg \phi$$

In this definition, $s \models \diamond(\phi)$ iff there is some move from s to a state in which ϕ is true. Similarly, $s \models \Box(\phi)$ holds when *all* moves from s lead to states in which ϕ is true

(it is vacuously true if there are no moves from s, i.e. s is a terminal state in the game graph).

5.6 Game Logic

We are now ready to consider extensions to first-order μ-calculus that lend it its adversarial flavor.

We propose two new primitive adversarial operators, $\mathbf{est}_i(\phi)$, and $\mathbf{irr}_i(\phi)$. Intuitively, $\mathbf{est}_i(\phi)$ means that player i has a strategy to make ϕ true in some game and $\mathbf{irr}_i(\phi)$ means ϕ is true now and player i has a strategy to maintain its truth until the end of the game. If $\bar{i}$ denotes player i's opponent, then the operators $\mathbf{est}_i$ and $\mathbf{irr}_{\bar{i}}$ are duals. In English we say a formula ϕ can be established by player i iff $\neg\phi$ is not irrefutably maintained by player $\bar{i}$.

From these two primitive operators, we define a derived operator $\mathbf{ach}_i(\phi) \equiv \mathbf{est}_i(\mathbf{irr}_i(\phi))$ which means intuitively that player i has a strategy to *achieve* ϕ, or *establish irrefutably* the goal ϕ. Note that $\mathbf{est}_i(\mathbf{irr}_i(\phi))$ and $\mathbf{irr}_i(\mathbf{est}_i(\phi))$ are not equivalent in all models. The former says that there is a strategy to make ϕ true "forever." The later says that there is a strategy to perpetually establish ϕ, which admits the possibility that the opponent can also perpetually deny ϕ, leading to an indeterminate game. In finite or, more generally, determined, games, the two are equivalent.

In the next section we present a axioms and definitions for our game logic.

5.7 Axioms and Definitions

If i denotes a player, then $\bar{i}$ denotes the opponent.

Define,

$$termLegal \equiv \forall m \neg Legal(m, i) \wedge \neg Legal(m, \bar{i})$$

$termLegal$ is true in any state in which there are no legal moves for either player.

1. $ToPlay(i) \longleftrightarrow \neg ToPlay(\bar{i})$

   It is exactly one player's turn to play in any state.

2. $Legal(i, m) \longrightarrow ToPlay(i)$

   If move m is legal for player i, then it must be player i's turn to play.

3. $$\begin{aligned} Causes(i, \psi) \longleftrightarrow \quad & [ToPlay(i) \longrightarrow \exists m\ Legal(m, i) \wedge \diamond_m(\psi)] \\ & \wedge [ToPlay(\bar{i}) \longrightarrow (\forall m\ Legal(m, \bar{i}) \longrightarrow \Box_m(\psi))] \end{aligned}$$

   Player i has a legal move to make ψ true if it is her turn to play, otherwise player $\bar{i}$ makes ψ true no matter where she plays.

4. $\mathbf{est}_i(\phi) \leftrightarrow \mu X.\phi \vee ((\neg termLegal) \wedge Causes(i, X))$

   A formula ϕ can be established by a player i iff ϕ is true or player i has a move to establish ϕ or all player $\bar{i}$'s moves establish ϕ.

5. $\mathbf{irr}_i(\phi) \longleftrightarrow \nu X.\phi \wedge (termLegal \vee Causes(i, X))$

   A formula ϕ is irrefutable by a player i iff ϕ is true and player i has a move to maintain its truth, or all player $\bar{i}$'s moves maintain its truth.

6. $\mathbf{ach}_i(\phi) \longleftrightarrow \mathbf{est}_i(\mathbf{irr}_i(\phi))$

   A formula ϕ is achievable if it can be established irrefutably.

   The next two axioms are the standard axioms for the first-order modal μ-calculus.

7. $F(\mu X.F(X)) \longrightarrow \mu X.F(X)$, where $\mu X.F(X)$ is free for X in $F(X)$.

8. $[F(Y) \rightarrow Y] \longrightarrow \mu X.F(X) \rightarrow Y$, where Y does not occur in $F(Y)$.

   These next two axioms correspond to the converse Barcan axiom.

9. $(\mu X.\forall A\ F(X)) \longrightarrow \forall A\ \mu X.F(X)$

10. $(\exists A\ \mu X.F(X)) \longrightarrow \mu X.\exists A\ F(X)$

    The next two axioms state that $\diamond_m$ and $\langle u \rangle_m$ are inverse:

11. $\diamond_m(\langle u \rangle_m(\phi)) \longrightarrow \phi$

12. $\langle u \rangle_m(\diamond_m(\phi)) \longrightarrow \phi$

    Finally, we have the tree axioms:

13. $(\langle u \rangle_m(\phi) \wedge \langle u \rangle_n(\phi)) \longrightarrow m = n$

14. $\mu X.[u]X$

Axiom 1 corresponds to model condition 5.3.3. Axiom 2 corresponds to model condition 5.3.4. Axioms 7 and 8 come from Kozen (axiom 8 is originally Park's induction axiom) who gives them in equational form. We adapt them to our style of presentation. Axioms 9 and 10 together are correspond to the 'converse Barcan" axiom. The Barcan axiom does not hold in general for tree structured models.

Axiom 11 says that $\langle u \rangle_m$ is a right-inverse for $\diamond_m$. This corresponds to model condition 5.3.2. Axiom 12 says that $\diamond_m$ is a right-inverse for $\langle u \rangle_m$; this corresponds to model condition 5.3.1. Axiom 13 says that each node has exactly one parent; this corresponds to model condition 5.3.5. Axiom 14 says that all paths up through the tree lead to a root; this corresponds to model condition 5.3.6.

5.8 Semantics

The semantics for FOL (first-order logic) and the operators of the modal μ-calculus are standard and not repeated here. In this section we will just give the semantics for our modal operators and game-specific extensions.

We associate the same set of domains (one for each sort in the language) with each state. Semantics for non-modal formulas are given as usual for first-order logic with equality.

Given a game structure with components

- $\mathcal{M}$, a set of moves
- $\mathcal{I}$, a set of two players
- $\mathbf{ToPlay}(i)$, a function determining the set of states in which it is player i's turn to play ($i \in \mathcal{I}$).
- $\mathbf{Legal}(m, i)$, a function determining the set of states in which move m is legal for player i ($m \in \mathcal{M}$, $i \in \mathcal{I}$).
- $\mathbf{Win}(i)$, a function determining the set of states in which player i is the winner ($i \in \mathcal{I}$).

- A set of reachability functions $\mathcal{R}_m$, one for each move $m \in \mathcal{M}$.

we extend the notion of first-order valuations to map move terms to moves in $\mathcal{M}$, and player terms to $\mathcal{I}$. The interpretation of a term a under a valuation V is denoted $[a]_V$. For brevity, we will write "$[a]$" instead of "$[a]_V$", omitting the valuation subscript.

Let $s \in \mathcal{S}$. The semantics for the $ToPlay$, $Legal$, and Win predicates are given in the obvious way by:

- $s \models ToPlay(i)$ iff $s \in \mathbf{ToPlay}([i)$
- $s \models Legal(m, i)$ iff $s \in \mathbf{Legal}([m], [i])$
- $s \models Win(i)$ iff $s \in \mathbf{Win}([i])$

For a non-modal formula ϕ:

$$
\begin{aligned}
s \models \diamond_m(\phi) \quad & \text{iff} \quad \mathcal{R}_{[m]}(s) \text{ is defined and } \mathcal{R}_{[m]}(s) \models \phi \\
s \models \langle u \rangle_m (\phi) \quad & \text{iff} \quad \mathcal{R}^{-1}_{[m]}(s) \text{ is defined and } \mathcal{R}^{-1}_{[m]}(s) \models \phi
\end{aligned}
$$

5.9 Some useful theorems

In this section we state a few simple consequences of our definitions.

A formula is *positive* in variable X iff X within the scope of an even number of negations in that formula. For example, $X \vee \neg((\neg X) \vee Y)$ is positive in X, but not in Y.

Theorem 5.9.1. *The following is a restatement of proposition 5.7 from [15]. For* $\sigma \in \{\mu, \nu\}$*:*

1. $\mu X.F(X) \longleftrightarrow \mu Y.F(Y)$ *where* X *and* Y *free for* Z *in* $F(Z)$
2. $[F(X) \to G(X)] \longrightarrow [\sigma X.F(X) \to \sigma X.G(X)]$
3. $[X \to Y] \longrightarrow [F(X) \to F(Y)]$ *where* Z *is positive in* $F(Z)$
4. $F(\sigma X.F(X)) \longleftrightarrow \sigma X.F(X)$ *where* $\sigma X.F(X)$ *is free for* Z *in* $F(Z)$
5. $(\sigma X.F) \longleftrightarrow F$ *where* X *is not free in* F
6. $[G(\mu X.F \wedge G(X)) \to F] \longrightarrow [(\mu X.G(X)) \to F]$ *where* F *is free for* X *in* $G(X)$

The next theorem shows that $\mathbf{irr}_i$ is monotonic.

Theorem 5.9.2.

$$(\phi \longrightarrow \psi) \longrightarrow (\mathbf{irr}_i(\phi) \longrightarrow \mathbf{irr}_i(\psi))$$

Proof. Define

$$F_i(\phi, X) \longleftrightarrow \phi \wedge (termLegal \vee Causes(i, X))$$

So, by definition,

$$\mathbf{irr}_i(\phi) \longleftrightarrow \nu X.F_i(\phi, X)$$

By theorem 5.9.1 part 3,

$$(\phi \longrightarrow \psi) \longrightarrow (F_i(\phi, X) \longrightarrow F_i(\psi, X))$$

So, by theorem 5.9.1 part 2,

$$(\phi \longrightarrow \psi) \longrightarrow [(\nu X.F_i(\phi, X)) \longrightarrow (\nu X.F_i(\psi, X))]$$

Hence the result. ☐

Theorem 5.9.3.

$$(\phi \longrightarrow \psi) \longrightarrow (\mathbf{est}_i(\phi) \longrightarrow \mathbf{est}_i(\psi))$$

Proof. Similar to the proof for $\mathbf{irr}_i$. ☐

The $\mathbf{ach}_i$ operator is also monotonic as the next theorem shows.

Theorem 5.9.4.

$$(\phi \longrightarrow \psi) \longrightarrow (\mathbf{ach}_i(\phi) \longrightarrow \mathbf{ach}_i(\psi))$$

Proof. By definition,

$$\mathbf{ach}_i(\phi) \longleftrightarrow \mathbf{est}_i(\mathbf{irr}_i(\phi))$$

By theorem 5.9.2,

$$(\phi \longrightarrow \psi) \longrightarrow (\mathbf{irr}_i(\phi) \longrightarrow \mathbf{irr}_i(\psi))$$

And, by theorem 5.9.3

$$(\phi \longrightarrow \psi) \longrightarrow (\mathbf{est}_i(\mathbf{irr}_i(\phi)) \longrightarrow \mathbf{est}_i(\mathbf{irr}_i(\phi))$$

Hence, the result. ☐

Corollary 5.9.5.

$$\mathbf{ach}(\phi \wedge \psi) \longrightarrow \mathbf{ach}(\phi) \wedge \mathbf{ach}(\psi)$$

Proof. By FOL, $\phi \wedge \psi \longrightarrow \psi$ so, by the monotonicity of $\mathbf{est}_i$ and $\mathbf{irr}_i$,

$$\mathbf{ach}(\phi \wedge \psi) \longrightarrow \mathbf{ach}(\phi)$$

Similarly,

$$\mathbf{ach}(\phi \wedge \psi) \longrightarrow \mathbf{ach}(\psi)$$

The result then follows by FOL. □

Corollary 5.9.6.

$$[\mathbf{ach}(\phi) \vee \mathbf{ach}(\psi)] \longrightarrow \mathbf{ach}(\phi \vee \psi)$$

This theorem allows establishes the duality between $\mathbf{est}_i$ and $\mathbf{irr}_i$:

Theorem 5.9.7.

$$\mathbf{est}_i(\phi) \longleftrightarrow \neg\mathbf{irr}_{\bar{i}}(\neg\phi)$$

Proof. First observe that $Causes(\bar{i}, \phi) \longleftrightarrow \neg Causes(i, \neg\phi)$ since,

$$\begin{aligned}
\neg Causes(i, \neg\phi) \quad &\longleftrightarrow \quad \neg[[ToPlay(i) \longrightarrow \exists m\ Legal(m, i) \wedge \diamond_m(\neg\phi)] \wedge \\
&\qquad [ToPlay(\bar{i}) \longrightarrow (\forall m\ Legal(m, \bar{i}) \longrightarrow \Box_m(\neg\phi))]] \\
&\longleftrightarrow \quad [ToPlay(i) \wedge \forall m\ Legal(m, i) \longrightarrow \Box_m(\phi)] \vee \\
&\qquad [ToPlay(\bar{i}) \wedge \exists m\ Legal(m, \bar{i}) \wedge \diamond_m(\phi)] \\
&\longleftrightarrow \quad [ToPlay(i) \longrightarrow (\forall m\ Legal(m, i) \longrightarrow \Box_m(\phi))] \wedge \\
&\qquad [ToPlay(\bar{i}) \longrightarrow \exists m\ Legal(m, \bar{i}) \wedge \diamond_m(\phi)] \\
&\longleftrightarrow \quad Causes(\bar{i}, \phi)
\end{aligned}$$

Now, by the definition of ν as $\neg\mu\neg$ and theorem 5.9.1 part 2

$$\begin{aligned} \neg\mathbf{irr}_{\bar{i}}(\neg\phi) &\longleftrightarrow \mu X.\phi \vee [(\neg termLegal) \wedge \neg Causes(\bar{i}, \neg\phi)] \\ &\longleftrightarrow \mu X.\phi \vee [(\neg termLegal) \wedge Causes(i, \phi)] \\ &\longleftrightarrow \mathbf{est}_i(\phi) \end{aligned}$$

□

Chapter 6

Rules for *Go*

6.1 Introduction

In the previous chapter we presented a proof system and semantics for a modal game logic. In this section we formally state the rules of the game and provide a motivation for the life and death strategy presented in chapter 2.

Although we are using game logic with least and greatest fixpoint operators, the language and rules require not much more than first-order (sorted) modal logic. The fixpoint operators are required when we want to state strategic theories that require the adversarial operators. The only additional operator that is required is a kind of quantifier over points, $Count_q(\phi)$, which acts syntactically like a term of sort integer. The interpretation of the $Count_q$ operator is as the cardinality of the extension of its formula argument. That is, $Count_q(\phi)$ counts the number of points satisfying the formula ϕ (ϕ is assumed to have q as its only free variable).

All domains of interest to us are finite, so quantifiers can in principle be eliminated. We use them anyway for conciseness of expression.

Our definition of $Legal(m, i)$ corresponds most closely with Chinese rules. In fact, there are many different rule-sets for *Go* with many subtleties (see [1] for more discussion of the rules). We will present a rule-set which is the most amenable to axiomatization. The differences between our rules and standard rule-sets are relatively minor overall and do not affect the character of the game substantially.

6.2 Meta-language conventions

Our language has terms of the following types:

- board points (the 361 intersections on a 19×19 *Go* board)

- moves (the 361 intersections on a *Go* board where it might be legal to play including PASS_MOVE which switches players but otherwise leaves the state unchanged, and NULL_MOVE which changes nothing)
- stone colors (BLACK, WHITE, EMPTY)
- players (Black, White)
- integers in the range 1 to 19.

Point variables will typically be denoted p or q, colors c, players i, moves m, and integers n. Since the number of points on the board is finite (361 in the case of the standard 19×19 *Go* board) we assume we have a constant denoting each point, written in as (row, column) coordinate pairs: $(1, 1), (1, 2), \ldots, (19, 19)$. Board occupancy colors are $BLACK$, $WHITE$,or $EMPTY$. Players are $Black$, or $White$.

We will sometimes use a variable of sort Player where a variable of sort Color is required. In these cases what we mean is that if the player is $Black$ we intend $BLACK$ and if the player is $White$ we intend $WHITE$. [1] For convenience, if c is a variable of sort color we use the notation $\bar{c}$ to refer to the "opposite" color of c: $\overline{BLACK} = WHITE$, $\overline{WHITE} = BLACK$, and $\overline{EMPTY} = EMPTY$.

We will also use move and point variables interchangeably since every move in on some board point and every board point is possibly a legal move. There are two nullary functions, LastMove and PrevLastMove of type move that we will use to track the last and previous to last moves. This is necessary to rule out repeated positions and to determine when there have been two passes in a row (signaling the end of the game).

[1] Technically, we can introduce a Color-valued function on Players which performs this translation.

Table 2.1 gives the predicates of the language. When we gave the definitions in chapter 2 we had not yet discussed the μ-calculus, so our definition of $SameBlock$ did not use the least fixpoint operator. Now we can give a more precise definition:

$$\begin{aligned} \forall p, q \; SameBlock(p, q, i) \quad \longleftrightarrow \quad & \mu X.Occ(p, i) \wedge Occ(q, i) \wedge \\ & p = q \vee \exists r \; Adj(r, p) \wedge \exists p \; p = r \wedge X \end{aligned}$$

In essence, this says that p is in the same block as q iff both p and q are occupied with stones of color i and either $p = q$ or there is some point r adjacent to p which is itself in the same block as q. The μ operator is used to select the the least solution to this recursive definition.

The details are a little bit tricky, however. The first part,

$$Occ(p, i) \wedge Occ(q, i) \wedge p = q \vee \cdots$$

simply accounts for the case in which p and q are the same point. The next part handles the recursive case, but notice that $\exists p \; p = r$ occurs within the scope of the $\forall p$ in the beginning of the definition. This effectively renames the variable p to be the adjacent point r before recursively applying the definition. It is much the same as the following recursive algorithm:

```
SAME_BLOCK(p, q, i)
(1)   if Occ(p, i) AND p = q
(2)       return true
(3)   else
(4)       foreach r, such that Adj(p, r)
(5)           if SAME_BLOCK(r, q)
(6)               return true
(7)   return false
```

6.3 Rules of Go

In this section we present a basic set of rules for *Go*. Our rules correspond most closely with Chinese rules which count stones and empty space both as territory. Counting in this way makes the rules easier to state since we do not have to keep track of captures. The rules presented here are somewhat formal; for a more readable, intuitive presentation see [1].

The rules below fall into three main categories: rules describing the initial state of the game, rules describing how the state changes from move to move, and rules describing how the game ends and who wins. Since all facts of interest in the state can be described in terms of the primitive predicates $Occ(p, c)$, $Adj(p, q)$, and $ToPlay(c)$, it suffices to characterize the effects of moves on these three primitive predicates. The effects on non-atomic formulas involving these three are entirely determined by the extensionality of the basic move operator $\diamond_m$.

In our rules suicide is illegal.

Black plays first.

Rule 6.3.1. $start \longrightarrow ToPlay(Black)$

The players alternate turns.

Rule 6.3.2. $ToPlay(i) \longleftrightarrow \forall m\ \Box_m(ToPlay(\bar{i}))$

The intersection points of the board form a 19×19 grid. If we imagine the grid labeled with board points starting in the upper left corner and progressing left to right to the lower right corner then points to the North, South, East, and west of each other are adjacent. The rule below is just English shorthand for the 176 axioms stating which points are adjacent.

Rule 6.3.3.

for all $1 \leq i, j, i', j' \leq 19$

If $|j - i| = 1$ xor $|j' - i'| = 1$ then $Adj((i, j), (i', j'))$

The game begins with the board empty.

Rule 6.3.4. $start \longrightarrow \forall p\ Empty(p)$

Initially there is no last move or previous to last move.

Rule 6.3.5.

$$start \longrightarrow LastMove = NULL_MOVE \wedge$$

$$PrevLastMove = NULL_MOVE$$

After a move m, the last move is now m and the previous to last move is what the last move was.

Rule 6.3.6. $\exists X X = LastMove \wedge \diamond_m (LastMove = m \wedge PrevLastMove = X)$

The next three axioms characterize the effects of a move on the primitive predicates $ToPlay$, Adj, and the relational predicates.

The adjacency of points on the board is the same for all states.

Rule 6.3.7. $Adj(q, q') \longleftrightarrow \diamond_m Adj(q, q')$

The relational predicates are the same for all states.

Rule 6.3.8 (schema for each relational operator $\prec$). $x \prec y \longleftrightarrow \diamond_m (x \prec y)$

The next four axioms describe how moves affect the occupancy of board points. There are four cases to consider: empty to occupied, occupied to empty, empty to empty, and occupied to occupied for each of both friendly and enemy play. In each of the axiom descriptions "friendly" play means the color of the player to play and the object point are the same; "enemy" play means they are different.

If a point p is a legal move for player i, then playing on point p will cause it to become occupied with a stone of color i, if it is not suicidal.

Rule 6.3.9 (empty to occupied by friendly play).

$$
\begin{aligned}
&CausesOccFriendly(p, q, i) \longleftrightarrow \\
&\qquad Legal(p, i) \wedge p \neq PASS_MOVE \wedge p = q \wedge \neg Suicide(p, i)
\end{aligned}
$$

A friendly legal play preserves the occupancy of a point if whenever it is block-adjacent to the point it is not a suicide, or it is a pass move.

Rule 6.3.10 (occupied to occupied by friendly play).

$$PreservesOccFriendly(p, q, i) \longleftrightarrow$$
$$Legal(p, i) \wedge Occ(q, i) \wedge [p = PASS_MOVE \vee$$
$$[BlockAdj(p, q) \longrightarrow \neg Suicide(p, i)]]$$

A legal enemy play preserves the occupancy of a point q if it is either on a point not block-adjacent q, it is a pass move, or the point is not in atari.

Rule 6.3.11 (occupied to occupied by enemy play).

$$PreservesOccEnemy(p, q, i) \longleftrightarrow$$
$$Legal(p, \bar{i}) \wedge Occ(q, i) \wedge$$
$$[\neg BlockAdj(p, q) \vee p = PASS_MOVE \vee \neg Atari(q)]$$

These cases exhaust the ways in which q will be occupied after a move by player i on point p.

Rule 6.3.12.

$$[CausesOccFriendly(p, q, i) \vee$$
$$PreservesOccFriendly(p, q, i) \vee$$
$$PreservesOccEnemy(p, q, i)] \longleftrightarrow \diamond_p Occ(q, i)$$

A legal enemy play captures a point q if it is not a pass move, is block-adjacent to q and the q is in atari.

Rule 6.3.13 (occupied to empty by enemy play).

$$CausesEmptyEnemy(p, q, i) \longleftrightarrow$$
$$Legal(p, \bar{i}) \wedge p \neq PASS_MOVE \wedge$$
$$Occ(q, i) \wedge BlockAdj(p, q) \wedge Atari(q)$$

A legal friendly play on p captures a point q if it is not a pass move, and p is block-adjacent to q, and p is suicidal.

Rule 6.3.14 (occupied to empty by suicide).

$$CausesEmptyFriendly(p, q, i) \longleftrightarrow$$
$$Legal(p, i) \wedge p \neq PASS_MOVE \wedge BlockAdj(p, q) \wedge Suicide(p, i)$$

This axiom combines the cases for the different player's turns. Playing a legal move on point p different from an empty point q, or passing a turn, causes q to remain empty (regardless of which player plays).

Rule 6.3.15 (empty to empty by friendly or enemy).

$$PreservesEmpty(p, q, i) \longleftrightarrow$$
$$(Legal(p, i) \vee Legal(p, \bar{i})) \wedge Empty(q) \wedge$$
$$[p = PASS_MOVE \vee p \neq q \vee Suicide(p, i)]$$

These cases exhaust the ways in which a point will be empty after a move.

Rule 6.3.16.

$$CausesEmptyEnemy(p, q, i) \vee$$
$$CausesEmptyFriendly(p, q, i) \vee$$
$$PreservesEmpty(p, q, i) \longleftrightarrow \diamond_p Empty(q)$$

The next five axioms describe which moves are legal and how the game ends.

This rule defines the predicate $TwoPasses$ which is true if the current state is the result of two consecutive passes. Two passes in a row means the game is over.

Rule 6.3.17.

$$TwoPasses \longleftrightarrow LastMove = PASS_MOVE \land$$
$$PrevLastMove = PASS_MOVE$$

If player i plays on point p in the current position and the resulting board position is identical to the board position before current position then it is a *repeated* position.

Rule 6.3.18.

$$RepeatedPosition(p, i) \longleftrightarrow$$
$$ToPlay(i) \land \forall q \exists c \, (\diamond_p Occ(q, c)) \longleftrightarrow \langle u \rangle_{LastMove} Occ(q, c)$$

A move p by player i is legal iff it is i's turn to play, the point p is unoccupied, and playing at p would not lead to a repeated position. A "pass" move is legal so long as it is not the third pass in a row (the game is over after two consecutive passes). The requirement that the position not repeat is called the *ko rule*.

Rule 6.3.19.

$$Legal(p, i) \longleftrightarrow ToPlay(i) \land \neg TwoPasses \land$$
$$Empty(p) \land (p \neq PASS_MOVE \longrightarrow \neg RepeatedPosition(p, i))$$

The game is over when a player has no legal moves.

Rule 6.3.20.

$$EndOfGame \longleftrightarrow \exists i\ ToPlay(i) \wedge \forall m\ \neg Legal(m, i)$$

The player with the most territory at the end of the game wins.

Rule 6.3.21.

$$Win(i) \longleftrightarrow EndOfGame \wedge Count_p(Point(p, i)) > Count_p(Point(p, \bar{i}))$$

If neither player wins at the end of the game, the game is a draw.

Rule 6.3.22. $Draw \longleftrightarrow \neg Win(Black) \wedge \neg Win(White)$

Chapter 7

Conclusion

7.1 Conclusions and Directions for Future Work

Computer *Go* has long been considered an extremely challenging automation problem because of its resistance to solution by standard game tree search techniques like minimax. In fact, closer examination of the problem shows that not only is the game space huge, it is also difficult to derive simple static evaluation functions for the game, such as are used in computer chess. No simple tally of positional features suffices to guide a search engine sufficiently well to make big-search a viable option for computer *Go*. Our view is that a successful go program must focus on emulating some of the reasoning applied by human beings in their game analysis and couple this with knowledge of which moves are reasonable in given position.

Initially, we set out to write a full-playing go program that could compete in competition. After pursuing this goal for a number of years we realized that it was overly ambitious – since we didn't really understand how to write a program to solve the necessary sub-problem of life and death – and at the same time, emphasized aspects of tournament game programming (like short-time limits for moves) that were less interesting to us than research on basic algorithms for the problem. We decided to focus our energies on building a knowledge-based life and death problem solver and trying to formalize some of the logic of human adversarial reasoning. This dissertation is the culmination of our work in both of these areas.

In a knowledge-based approach, the traditional view of game search with an integer-valued evaluation function is somewhat restrictive. In other domains in which we want to understand human reasoning we have at our disposal the full power of formal logic in which to develop languages and state theories. Why not in the domain of games as

well?

The language we developed for the purpose of describing games is a modal game logic. Using this logic we were able to formally state the rules of go (see chapter 6) and to develop a strategic theory of life and death (see chapter 2) that makes clear the relationships between the goals in the theory. Later sections of that chapter showed how a formal theory of life and death could be put to use in solving life and death problems by building an appropriate model and using expert heuristic knowledge about which moves are reasonable. Chapter 5 addressed some of the logical issues of this language developing formal syntax and semantics and proving some of the basic facts of the system.

Chapter 3 described how the implementation of a computer program to solve real life and death problems based on these ideas. Our tests on Kano's series of graded go problems yielded very encouraging results. When a problem was solved, the tree of moves considered was very small in general (on the same order of magnitude as human solutions) and very much smaller than big-search computer solutions. Furthermore, when our program was given a problem to solve, we required that it solve that problem completely, generating moves to try, hypothesizing them on the board, and solving the sub-problems that resulted, recursively.

There are a number of different directions which would be promising for future research. One clear path would be to take the life and death solver we have created and either extend it to a full-playing program or incorporate it as a module of an existing program. Another interesting project – one which we have looked at a bit – is to study how the solver could be made to learn, both within the exploration of a given problem, and from problem to problem.

Outside of the realm of computer *Go* there are also a number of potential applications for our work. The clearest short-term application would be an analysis of another game, such as chess using our game logic. Some parts of the program are quite specific to *Go*, but others, such as the revertable class library, knowledge editor, and truth maintenance systems are generally useful tools.

Bibliography

[1] American go association. Web page: http://www.usgo.org/computer.

[2] Formalizing the dynamics of information. Web page: http://www.cwi.nl/ pauly/.

[3] J. Nievergelt A. Kierulf, K. Chen. Smart game board: A workbench for game-playing programs, with go and othello as case studies. Technical report, ETH Zürich, 1990.

[4] J. W. De Bakker and W. De Roever. A calculus for recursive program schemes. *Proc. 1st Intl. Coll. on Automata, Languages, and Programming*, pages 167–196, 1972.

[5] Patrick Blackburn, Wilfried Meyer-Viol, and Maarten de Rijke. A proof system for finite trees. *Lecture Notes in Computer Science*, pages 86–104, 1996.

[6] Brian F. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, 1980.

[7] W. P. DeRoever. *Recursive program schemes: Semantics and proof theory*. PhD thesis, Free University, Amsterdam, 1974.

[8] R. Bozulich (ed.). *The Go Player's Almanac*. Ishi Press, 1992.

[9] D. W. Erbach. Computers and go. *In [8]*, 1992.

[10] Melvin Fitting and Richard Mendelsohn. *First-order Modal Logic*. Kluwer Academic Publishers, 1999.

[11] David Fotland. Personal communication.

[12] Robert Goldblatt. *Logics of Time and Computation*. CSLI, 1992.

[13] Yoshinori Kano. *Graded Go Problems for Beginners, Vol. 3*. Japanese Go Association, 1987.

[14] Yoshinori Kano. *Graded Go Problems for Beginners, Vol. 4*. Japanese Go Association, 1990.

[15] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[16] D. Lichtenstein and M. Sipser. Go is pspace hard. *Proc. 19th Ann. Symp. on Foundations of Computer Science, IEEE Computer Society, Long Beach, CA.*, pages 48–54, 1978.

[17] David A. Mechner. All systems go. *The Sciences*, 38:1, 1998.

[18] Martin Mueller. *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory*. PhD thesis, Swiss Federal Institute of Technology Zurich, 1995.

[19] Rohit Parikh. The logic of games and its applications. *Topics in the Theory of Computation, Karpinski and van Leeuwen, eds. Anals of Discrete Mathematics*, 24:111–140, 1985.

[20] D. M. R. Park. Fixpoint induction and proof of program semantics. *Machine Intelligence*, 5:59–78, 1970.

[21] J. L. Ryder. *Heuristic Analysis of Large Trees as Generated in the Game of Go*. PhD thesis, Stanford University, 1971.

[22] D. Scott and J. W. De Bakker. A theory of programs. *Unpublished manuscript, IBM, Vienna*, 1969.

[23] Claude Shannon. Automatic chess player. *Scientific American*, 182, 1950.

[24] Alan Turing. *Faster Than Thought*, chapter 25. Pitman, London, 1953.

[25] Igor Walukiewicz. Notes on the propositional μ-calculus: Completeness and related results. *BRICS*, 1995.

[26] David Wilkins. Using patterns and plans in chess. *Artificial Intelligence*, 1980.

[27] Willmott. Adversarial planning techniques and the game of go. Master's thesis, Department of Artificial Intelligence Edinburgh, 1997.

[28] T. Wolf. The program gotools and its computer-generated tsumego database. *Proceedings of the Game Programming Workshop in Japan*, 1994.

[29] A. L. Zobrist. *Feature Extraction and Representation for Pattern Recognition and the Game of Go*. PhD thesis, University of Wisconsin, 1970.