STRUCTURAL REPRESENTATION OF THE GAME OF GO

by

James Ian Scrimger

B.Sc., Mount Allison University, 2004

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Computer Science

in the Graduate Academic Unit of Computer Science

This thesis is accepted by the Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

September, 2007

Abstract

Go, a popular game of strategy, is not amenable to the traditional AI techniques for game playing. Because of the game's hierarchical and temporal nature, the use of *structural representation* is a logical approach to try. This thesis presents a preliminary representation of the game of Go in the structural language of the Evolving Transformation System (ETS).

An ETS class hierarchy designed to capture progressively complex aspects of Go positions is presented. Individual Go stones are represented as elements of a few low-level classes, these classes are the constituent pieces of higher-level classes corresponding to blocks of stones, which are in turn assembled into classes corresponding to the evolution of larger shapes.

The presented class hierarchy could form the basis of the first entirely class-based Go-player, meaning that such a program's move selection would be directed by *generative* class descriptions.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and background

Games of strategy have long been a favorite domain for AI research:

> The complexity of games introduces a completely new kind of uncertainty ... [which] arises not because there is missing information, but because one does not have time to calculate the exact consequences of any move. Instead, one has to make one's best guess based on past experience, and act before one is sure of what action to take. In this respect, *games are much more like the real world than in the standard search problems [of AI]* [1, p. 123].

Traditionally, the flagship game for such research has been chess, but as computer chess play improved to the level of world-class human players, more attention has been focused on Go, which is not amenable to the search techniques that are so successful in chess programs.

> A common misconception drawn from decades of chess research is that brute force techniques, utilising good search and evaluation algorithms, is sufficient

> to solve any problem once it has been formally specified. Go is a domain that contradicts this common misconception. It is easy to formally specify the rules of Go, however, all current programs fall short of human performance even to the level of a beginner-intermediate player. [2, p. 1]

The above was written in 1995 and, while programs have improved slightly over the last decade, and significantly on small boards in the last year, there is still no program that can challenge a professional player on a full-sized board [3].

The problems that Go presents are an opportunity to explore the use of *structural representation*: because the search complexity of Go is very large, it is not possible to use any ad hoc representation scheme and then perform heavy operations on it algorithmically. In light of this, a structural approach, which seeks to maintain and enhance as much of the available information as possible, appears to be a natural candidate for building representations of the game. Besides helping to systematically organize a Go program's information, structural representation also supports *inductive classification*, the ubiquitously sought-after machine learning goal of discovering class representation from a few training objects to facilitate the identification of the classes of subsequently-encountered objects, in a new way.

There have been previous attempts at "richer representation" (e.g., [4]) and classification in Go (e.g., [5, 6, 7, 8, 9, 10, 11]), but none have led to programs that play as successfully as those using brute-force techniques. At the time of writing, some of the best Go programs use Monte Carlo simulation methods and a minimal amount of structured Go knowledge [3]. If such approaches ultimately succeed (though whether they will is very much an open question), it would put us in the unfortunate position of having "solved" Go without learning anything new *about* Go.

The structural representation of Go I have constructed for this thesis uses the language of the Evolving Transformation System (ETS), which is a new representational formalism designed with inductive classification in mind. The latest version, [12], was completed only recently, but the essential ideas have been in development by Goldfarb and other members of the ETS group for more than 20 years.

The original motivation for the ETS was to unify two separate approaches in pattern recognition: the statistical approach, and the structural (syntactic) approach [13]. The hope was that the strengths of these two models combined could overcome their individual shortcomings: objects could be modeled syntactically and partitioned into classes statistically. However, investigation in this area led Goldfarb and associates to see that the much more fundamental concept of *class representation* had been overlooked.

It is this concept of class representation that is at the heart of ETS. In most machine learning (or pattern recognition) schemes, the primary focus is on representing individual objects, usually as data points in some abstract space, then *grouping* them according to a boolean class variable. Goldfarb et al. thought that a class was much more than just a collection of elements: the class *itself* must have its own structure. ETS is meant to be formalism that can describe both structured objects and structured classes *in the same language* [14].

1.2 Scope of the thesis

The main aim of this thesis is to construct a representation of the game of Go in the language of the ETS formalism, focusing on the use of *generative class descriptions* (as distinct from the current machine learning paradigms) to model the game in a rich and systematic way. The aim of this work is *not* to develop a computer program that plays Go, or even to conduct learning experiments on game data. Rather, it is to create a foundational model of the game that could be used to support those two related projects.

The logic of the ETS formalism suggests that basic representation of data is perhaps the most important concern, and should not be treated casually. Because representations of Go are always developed with playing the game in mind (as opposed to merely describing how to make legal moves), such representations must extend beyond the basics of game rules and playing pieces, and into the realm of tactics.

In support of the above aim, the current approaches in computer Go will be reviewed, and the concepts of ETS will be introduced. Once my Go model has been presented, some of its desirable properties will be highlighted, and its use in constructing a Go playing program will be discussed.

1.3 Organization of the thesis

Chapter 2 begins with an overview of the game of Go, discussing some of the game's properties and rules. This leads into a discussion of why Go is difficult for computers to play compared to other strategy games (Section 2.2). Section 2.3 provides an overview of the standard techniques used in computer Go systems and their shortcomings. Finally, Section 2.4 discusses some of the differences between how computers play Go and how the game is played by humans.

Chapter 3 presents a pictorial (non-formal) overview of the Evolving Transformation System. The first half of the chapter discusses the formalism's key assumptions, so that the reader will get a sense of the purpose of the various formal components. The second part of the chapter describes the elements of the formalism themselves, first introducing the basic representational units (Section 3.3), and then describing the workings of ETS class representations (Section 3.4).

Chapter 4 covers the primary work of this thesis: the construction of an ETS representation of the main aspects of Go. The chapter begins with an overview of the entire class hierarchy, with each level examined in greater detail in Sections 4.3–4.7. Sections 4.1 discusses the particular aspects of Go that I have sought to capture, and Section 4.2 presents the thinking behind the the general approach taken, in the hopes that an elucidation of how one begins modeling a phenomenon in ETS will be useful to others working with structural representations.

Once my representation of Go has been recounted, Chapter 5 discusses its properties and implications. Section 5.1 elucidates several qualities of the Go representation that seem to be desirable. The chapter's middle (Section 5.2) revisits some of the information about human Go players gleaned from work in Psychology that was introduced in Chapter 2, this time with a view to how my Go representation fits with these observations. Finally, Section 5.3 proposes an outline for a new type of Go-playing program constructed on the basis of the classes described in Chapter 4.

Chapter 6 concludes this thesis, and discusses future research directions, as well as some of the implications of this work for the future development of the ETS formalism.

Chapter 2

Computer Go

2.1 The game of Go

Go is a classic game of strategy for two players, who take turns placing coloured stones (black for the first player and white for the second) on the vertices of a 19×19 grid, traditionally etched into a wooden board called a *goban*. The aim of each player is to control "territory" (a number of unoccupied vertices) by surrounding it with the stones of his own colour. It is also possible for each player to capture his opponent's stones by surrounding them in a similar way. The analogy used to describe Go is two armies competing to control an unoccupied land. Go is very popular in Asia, where tournaments draw attention on the same level as major sporting events. The game is seen as the king of strategy contests, in the same way chess is viewed in the west [15].

More formally, Go is a zero sum, perfect knowledge, perfect strategy game. Other games in this category include chess, checkers, and Othello. "Zero sum" means that one player's gain directly corresponds to his opponent's loss—there is no room for cooperation or for causing mutual harm. "Perfect knowledge" means there are no hidden resources—each player has complete information about the other player's pieces and capabilities. "Perfect strategy" means that, unlike games involving cards or dice, there is nothing left to chance. Such games are perhaps the most straightforward, since the only unknown factor is the opposing player. For humans, they are also considered to be those games that best test the intellect.

The roots of Go are in China, and it is believed to have been played in its present form for 3000 years [16], though it has only been played in the west for the past century [17]. There is speculation that the Go board was originally used as a divination tool, with seers scattering stones on its surface and then interpreting the meanings, or as a calendar, with a full-sized goban's 361 intersections relating to the days of the year [16]. The rules of the game are so simple that it is easy to imagine how they could have naturally evolved from playing with stones on a grid.

Go has its own rich culture and language. It features prominently in many Asian myths and serious players often approach the game as something almost spiritual—Go is not just a puzzle to be solved. In his discussion of the origins of the game, popular Go author Peter Shotwell writes:

> As illustrated by the development of Daoism, from those early times, 'surrounding' formed the basis of war aims and tactics, and was also was the principle of not only what many Chinese regard life is, but also how it should be conducted. [16, p. 8]

2.1.1 Some rules of Go in detail

Go has a simple set of rules governing where stones can be played. In general, a player may place his stone on any unoccupied vertex, but there are a few restrictions

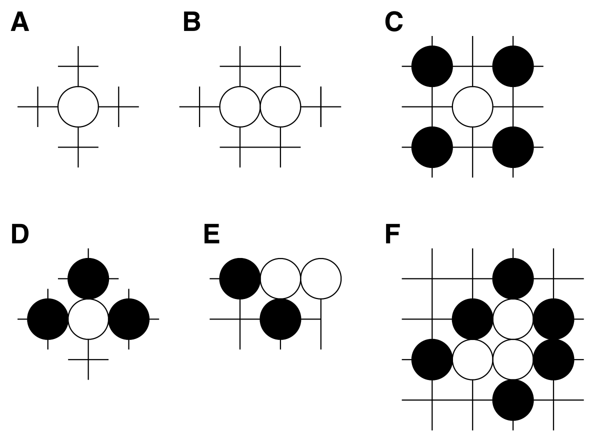


Figure 2.1: Several Go positions.

handling various special cases. Stones do not move around the board: once placed, they remain at a fixed location unless captured.

Life and death is perhaps the most important concept in Go. Any stone that is surrounded on four sides by enemy stones is "killed" (also called "captured"), and removed from the board. A stone's adjacent empty vertices are called its *liberties*. When a stone has no liberties left, it "dies". It is important to realize that only the four cardinal directions matter in this case: diagonally-adjacent stones don not directly affect the life and death of their neighbours.

Stones of the same colour that are immediate neighbours are considered to be *connected*, and live and die together. A block of connected stones is captured only when all of the liberties in common across all stones are filled. Figure 2.1 shows the following examples of liberties and captures. A: a single stone has four liberties. B: two stones together have six liberties. C: this white stone still has four liberties. D: the white stone has only one remaining liberty; black can capture in one move. E: The two white stones on the edge of the board also have only one remaining liberty. F: this group of white stones has one remaining liberty.

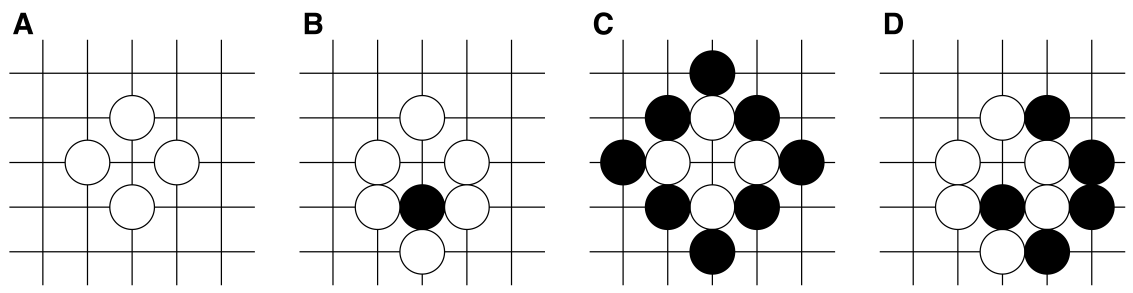


Figure 2.2: Suicide and gaining liberties.

A related rule is that it is illegal to *suicide*—that is, to place a stone where it would have no liberties. However, a player may place a stone in a completely surrounded location if doing so causes the capture of one or more of the adjacent stones, instantly providing the newly-played stone with at least one liberty. Figure 2.2 illustrates this concept. A: black cannot play at the center of the white group, as this would be suicide. B: black cannot play in the center of this group either, since it would mean suicide for both stones. C: black can play at the center of this group, since doing so captures all four white stones, but a white play here is suicide. D: black can play at the center of the white group, capturing the two right-most white stones.

As a consequence of these rules, the usual way to save a group of stones under threat is to make two *eyes*. Eyes are small areas of territory, often a single vertex, enclosed by a connected block of stones. If a single block has two eyes it is un-assailable, as neither liberty can be taken away by the opposing player, since playing inside an eye would be suicide.

Figure 2.3 shows several examples of eyes. A: black's group has two eyes and cannot be captured, despite being totally surrounded. B: by playing the marked stone, black makes two eyes, saving this group. C: this group also has two eyes. D: white's attack on this larger eye will fail. Black can either capture the interloping white stone, or just ignore it, since a second white play inside the eye would be a suicide play. E,

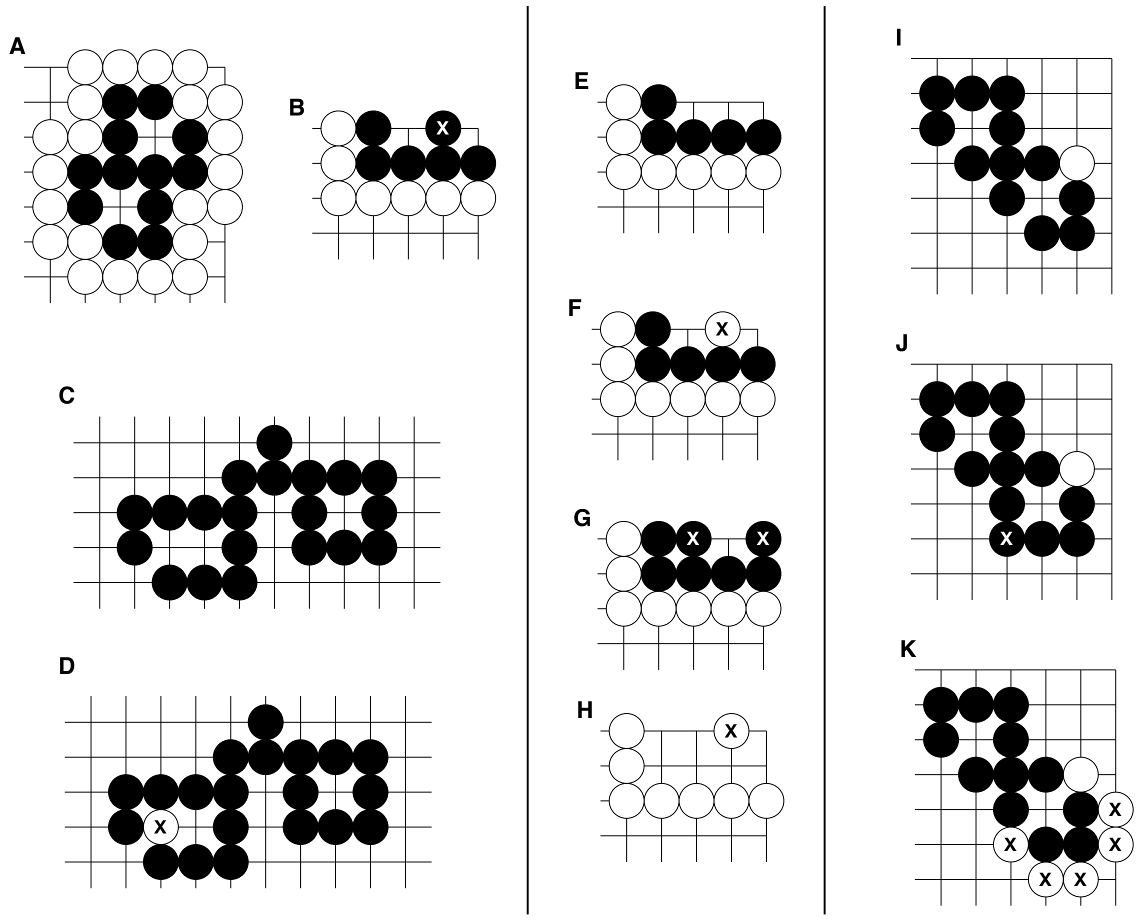


Figure 2.3: The principle of two eyes.

F, G, H: if white is allowed to play behind black's line, black's group is doomed. I: One of this black group's eyes is a "false eye"... J: if black cannot connect with the marked stone... K: white can divide this eye, isolating the lower three stones and forcing black to fill the failed eye to prevent their capture. If this happens, the entire group is threatened and must find another way to make a second eye.

Go games end by mutual agreement. When a player's turn comes, she may elect to *pass*, rather than play a stone. If both players pass in succession, the game is over and the score is determined. Each player gets one point for each empty vertex they have successfully surrounded (*territory*), as well as one point for each captured

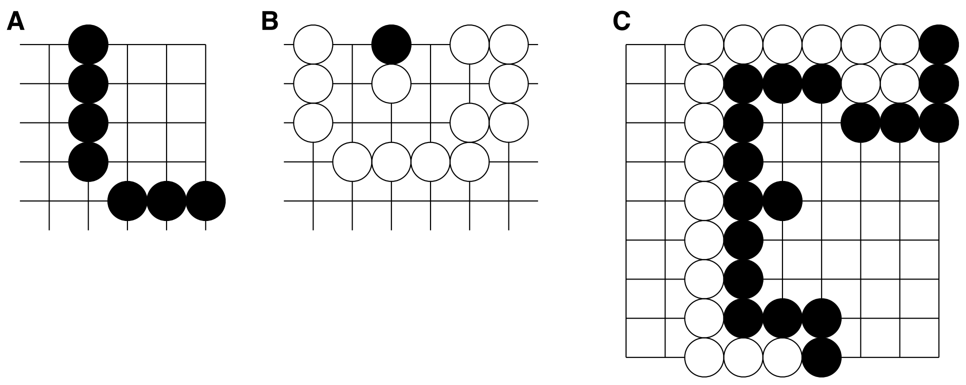


Figure 2.4: Examples of territory.

enemy stone. Figure 2.4 shows the following examples of territory. A: black controls 12 points of territory at the corner of the board. B: white controls nine points of territory, plus one point for the black stone. C: A 9×9 game between the author (black) and GNU Go (white). Black has 27 points of territory, while white has 18.

Because the game can end at any moment, stones can be dead without actually being captured—in fact, once it becomes clear that a group of stones cannot be saved, it is often wasteful to actually complete the capture of it. Dead groups are removed from the board at the end of the game and scored the same as stones captured during play. It is not uncommon for a game to end with some of one player's stones left "behind enemy lines" and unable to make two eyes.

The *ko* rule is an additional rule governing stone-play. It covers a special case situation that occasionally arises, and states that an identical full-board position cannot occur twice in the same game. As Figure 2.5 shows, were it not for the ko rule, players could become stuck in a situation where the same play is repeated indefinitely. Instead, when a "ko fight" arises, players will alternate plays directly challenging the ko situation with plays elsewhere on the board. Depending on what else is at stake on the board when a ko situation arises, ko fights can be far-reaching

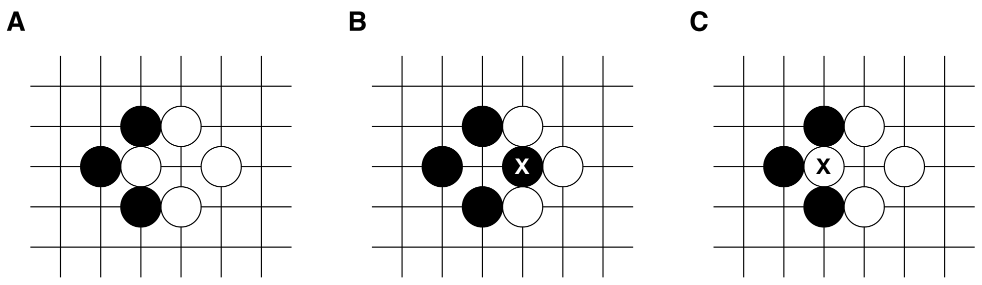


Figure 2.5: The ko rule.

and exciting. From Figure 2.5: A: the classic ko situation. B: black's play here captures the white stone. C: since capturing black's stone creates the same pattern as in A, white must first play elsewhere before this move can be made.

Go players are ranked, and Go includes a handicap system to allow competitive games between players of different ability levels. Weakest to strongest, amateur players are ranked from 20 kyu to 1 kyu, and expert armatures are ranked from 1 dan to 6 dan. Additionally, there are professional dan ranks of 1 to 9.

When players of the same rank play, the white player gets a few (usually 6.5) points of *komi*, or compensation for black's first-play advantage. If one player is a single rank stronger than the other, the weaker player typically takes black. For greater differences in rank, the weaker player gets some number of stones (usually between one and nine) pre-placed in specific locations on the board. This gives the weaker player a head-start in claiming territory.

While the ranking system is a useful way to quantify the relative strengths of (human and AI) players, researcher and dan-level Go player Martin Müller cautions that it is difficult to measure the playing strength of Go programs directly because they tend to have non-human strengths and weaknesses [18].

2.2 Why Go is difficult for computers to play

At first glance, Go appears to be a simple game. It has few rules and only two varieties of playing pieces: a white stone and a black stone. However, a great deal of complexity arises from these simple elements. A book by the Japanese Go association contains the following insight [19, p. 10]:

> The game of Go may be looked upon either as very easy or as very difficult. In one sense it is easy because the rules are simple. Since every player is free to put his stones wherever he likes, except for a few forbidden moves, it takes only an hour or so to remember all the rules. It is exactly for the same reason, however, that Go is considered to be a difficult game. If our moves were restricted by a number of rules, the game would become the easier for the limitations on our range of thinking. As it is, we can rely on nothing beyond our own ability, just as in the case of drawing a picture freely on a large canvas. Therefore, although each game results in victory or defeat, the process must involve such creative talent as is required in producing a work of art.

Traditionally, computers have avoided the need for "creative talent" in playing games by relying on their own strength: the ability to perform rapid calculations. However, in the case of Go, that strength has failed, and the above hints at the reason why: it is the lack of restrictions that makes Go difficult for a computer to play. At any juncture the number of available moves is much larger than in similar strategy games, and evaluating the merits of each move is more complex and nuanced. In practice, the standard computer game playing technique of generating a (possibly exhaustive) set of candidate moves and then comparing them via some score function [20] becomes doubly unwieldy.

2.2.1 Large search space for candidate moves

> If you want to understand intelligence, the game of Go is much more demanding ... It doesn't have the silver bullet: deep search. Chess has somewhat outlived its usefulness. It turned out to be easier than we thought.
>
> Jonathan Schaeffer, quoted in [21]

Game programmers have successfully applied various search techniques to games such as chess and checkers. The least sophisticated of these techniques was outlined by Claude Shannon in his 1950 paper proposing a computer chess program [20]. Shannon's program considers all legal moves that it can make, then considers all possible replies by its opponent to each of these moves, then all possible moves it could make after all such replies, etc. The result is called a *game tree*, where each node represents a board position, each node's children are all possible moves replying to that position, and each node's depth corresponds to the number of moves beyond the present. Each level of the tree represents the program's consideration of the game at a fixed point in the future, and is called a *ply*, which is a convenient measure of the amount of look-ahead that a program performs. Usually programs that use these techniques search until they find a *quiescence* [20]: a stable position in which the strength of the players' positions can be easily compared.

Such an approach can be unwieldy, since the complexity increases exponentially with each ply, but various techniques to remove bad lines of play and redundant moves have been developed, and chess programs that rely on deep searches (such as IBM's famous Deep Blue [22]), have been very successful. However, such full-scale searches are not practical for Go.

Go is typically played on a 19×19 grid (361 total vertices, compared with 64 squares in Chess), and each grid position can either contain a black stone, a white stone, or be empty. Thus, there are 3^{361} possible board positions, which is approximately 1.74×10^{172}, although only about 1.2% of them are legal [18]. There are too many possibilities for even the fastest computer to cope with. For comparison: it is thought that there are 10^{80} atoms in the universe, and strong 256 bit cryptography has a key space of about 1.16×10^{77}. Even a smaller 9×9 variation of the game still has a position space of approximately 4.43×10^{38}.

A Go program need not consider all possible board positions, only the one that it is presently faced with, and all possible moves to be made. Even this adds up quickly: there are 361 possible first moves, 360 possible second moves, etc. However, the search space of a full game is not exactly 361! because vertices that have already been filled can become available for play again via capture. A full game played to its conclusion will often last about 300 moves (150 for each player) [2], meaning that the board is not completely full at the end.

Obviously an exhaustive search of the game-space for full-board Go is impossible given current technology. What is interesting is that on a 9×9 board, computer play is not significantly better. In his overview of Computer Go, Müller writes,

> The large search space caused by the great number of possible moves and by the length of the game is often cited as the main reason for the the difficulty of Go. However, as Ken Chen points out ... 9×9 Go, with a branching factor comparable to chess, is just as difficult as full 19×19 Go ...
>
> The biggest difference between Go and other games is that static of evaluation is orders of magnitude slower and more complicated. Moreover, a good static full-board evaluation depends on performing many auxiliary local tactical searches. [18, p. 6]

2.2.2 Complicated move evaluation

> No simple yet reasonable evaluation function will ever be found for Go. This is evident to serious students of the game... [18, p. 7]
>
> Martin Müller, 2002

It is not always apparent to even master human players whether a given move is better or worse than another. By contrast, in chess, even a simple count of the number of pieces each player has remaining (since each piece has a point-value that captures its relative worth—pawns are worth 1, bishops and knights 3, rooks 5, and so on) can be a strong indicator of which player is stronger. Go positions are much more nuanced, however. They depend on the likelihood of a group's survival, or the amount of territory that can potentially be controlled. Complexity emerges in unexpected ways when neighbouring groups interact.

There are heuristics for evaluating Go positions: for example, an introduction to Go for beginners [23] suggests that a group with four liberties is safe from capture. However, this kind of rule is not absolute, and although there are many handy rules to remember, each player must ultimately depend on his ability to "see" the consequences of each move.

The fact that Go evaluation functions are both slow and unreliable further reduces the effectiveness of search. Most notably, *min-max* (for example"alpha-beta") pruning techniques cannot be employed. These techniques, which were an integral part of the famous chess program Deep Blue [22], reduce the size of the search-tree by limiting the search to each player's most promising moves at each stage. Obviously, to identify such promising moves depends on an effective full-board position evaluation.

2.3 Current computer Go techniques

The previous section outlined the problems of search scale and position evaluation faced by computer Go researchers. Current programs typically avoid tackling these issues head-on by employing a group of specialized modules. However, this modular approach to the independent facets of the game has not managed to make up for the overall difficulty. Müller writes,

> Most competitive programs have required 5-15 person-years of effort, and contain 50-100 modules dealing with different aspects of the game. Still, the overall performance of a program is limited by the weakest of all these components. The best programs usually play good, master level moves. However, as every games player knows, just one bad move can ruin a good game. Program performance over a full game can be much lower than master level. [18, p. 4]

Many programs, such as GNU Go 3.6 [24], which has a modular framework, use a programmatic approach[1]: rules and heuristics designed with input from expert players are used to search for and evaluate candidate moves. Other programs use machine learning techniques to generate playing strategies without expert intervention, though the programmatic approach has thus far been stronger. There are also programs that use novel techniques outside these two broad categories; Section 2.3.5 discusses one such technique that is becoming popular: the use of Monte Carlo simulations.

Additionally, research into Go has been directed towards goals besides successful play. Some research concentrates on solving local life-and-death problems (for example, [25]), while others (for example, [26]) use machine learning to identify human players by their style of play.

[1]Since the time of writing, GNU Go has incorporated Monte Carlo techniques into its play.

Besides play against humans, Go programmers test their creations against other programs. One such recent test came at the 11th Computer Olympiad [27]. The best program at this 2006 event on a 19×19 board was the Open Source GNU Go, with Ken Chen's Go Intellect taking silver and Bouzy's Indigo taking Bronze.

On a 9x9 board, a program called Crazy Stone, which uses Monte Carlo techniques, placed first, followed by Aya and Go Intellect.

2.3.1 Representation and knowledge

Some Go programmers have responded to the challenges of Go's complexity by constructing programs that represent the game in more nuanced ways than is typical of game-playing in general. Current programs tend to base their move decisions on a detailed analysis of the present game state, and often contain libraries of patterns implemented by expert Go players. A common starting place is to analyze the relationships between individual stones on the board. GNU Go, for example, identifies *worms*—groups of connected stones, and "dragons"—groups that cannot be disconnected by the opposing player [24].

In his summary of Computer Go research [28], Keh-Hsun Chen describes a typical hierarchical view of the board (bottom up): stones, blocks, chains, groups, territory. The atomic elements of Go are individual stones. Alone, they are not that interesting: they do not move around the board, they are simply placed, and sometimes removed. It is the interaction between stones that gives rise to the game's interest. The most straight-forward of these relationships is what Chen calls a *block* (also called a *string* and in the case of GNU Go, a *worm*): two or more connected stones of the same colour. Blocks are easy to identify, and it is natural to locate them on the Go board, since the safety of each member stone depends on all the others. Many programs conduct tactical searches on each block on the board to determine the degree of danger they face (ranging from *unconditionally safe* to *dead*) [28].

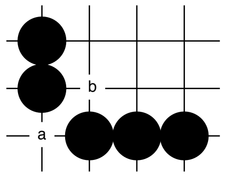


Figure 2.6: A *chain* (or *dragon*). If white attempts to divide these stones by playing at ‘a’, black can connect at ‘b’, and vice versa.

Chen calls a collection of “inseparable” (but not immediately connected) blocks a *chain* (or *dragon*, in GNU Go parlance). Chains are not as straightforward to identify, but programs can reliably detect them via heuristics, via matching with a pattern library, or via tactical search [28]. Stones in a chain are almost as closely linked as those in a block, since one block in a chain can often escape from danger by connecting to another block that has more liberties.

Chen calls a *group* “a strategic unit of an army of stones”, defined as one or more *chains* plus any *prisoners*, or captured (but not totally surrounded and thus still on the board) enemy stones. Groups are detected in a manner similar to *chains*, but are a more complete representation of a local region of the game board. Stones in a group also tend to live and die together.

Because groups of stones are often cut off from friendly forces and must fight to live locally, another way to organize information in a Go game is to decompose the game board into independent or semi-independent “subgames” [29]. This technique comes from combinatorial game theory and has been successfully applied to Go end-games, and other situations. The reason that this type of decomposition is desirable is because it serves as an attack on the complexity of search: a global search for the best move can be treated as the sum of many local searches [30].

Territory is the most abstract element of Chen's five, though it becomes concrete at the end of a game when the score is calculated. While the game is in progress, estimating the territory that can be finally won is an essential part of any playing program. Often territory is predicted via *influence*. Chen's Go Intellect program uses a radiating influence function: each live stone increases the numeric "influence" level of neighbouring empty vertices, with the amount dropping off as the distance increases. Areas of the board with several nearby stones of one colour are strongly influenced by that player—areas far from any stones or surrounded by both players are weakly influenced. The expectation is that areas of high influence will eventually become the territory of the influencing player.

A richer approach to determining the status of groups and territory was used by Bouzy in his original Indigo program [4]. His program had a library of small (approximately 25 vertex) shapes, which the program identified on the board and then iteratively assembled to find larger groups. This database was used to determine the urgency of a particular move, since information about the strength of the position associated with each pattern was included by hand [31]. However, since the roughly 300 patterns included the most common situations, adding more patterns to the set brought diminishing returns [4]. Because of this, in more recent versions of Indigo, Bouzy actually abandoned this approach in favour of Monte Carlo techniques [32].

2.3.2 Search

While it is true that a full-board search is prohibitively complex, Müller [18] notes that local (or "tactical") searches directed towards one or a few specific goals are commonly employed by modern Go programs. This kind of limited search avoids the two problems of full board search: for a given goal, such as the capture or rescue of one or more stones, the evaluation criteria are simple ("has the stone been captured?" etc.), and usually, only the relevant portion of the board need be examined.

A few programs, such as Bouzy's Indigo, use shallow full-board search, but it is not usually the program's primary search mechanism.

Often, tactical searches are not used to find moves directly, but rather to determine the state of the board. Local searches can be used to analyze the level of safety (conversely, the ease of capture) of various groups of stones [18].

2.3.3 Move generation and evaluation

The way candidate moves are generated is closely linked to the way they are evaluated. Because full-board searches are prohibitively complex, Go programs typically consider only a limited subset of all legal moves. In order for this approach to work, a good deal of position evaluation must take place before any moves are proposed, as any failure to properly identify the current game situation can lead to important moves being overlooked. Once such information has been collected, programs employ various heuristic techniques to "suggest" candidate moves [18]. How these moves are proposed and how many are considered depends largely on a given program's evaluation function.

It is hard to give a description of a standard evaluation function (EF) as implemented in a Go program. As Bruno Bouzy writes,

> Instead of tree search optimizations, it is the discovery of the EF for the game of Go that is the main task of Go programmers. Of course, each Go programmer

> has their own EF. Every EF results from intensive modeling, programming, and testing activities. Consequently, each EF is different from every other one, and no agreed model has clearly emerged in the community. [33, p. 11]

Müller [18] describes two broad approaches: *position evaluation* and *move evaluation*. Most programs use some combination of the two.

Position evaluation is an approach that is similar to that of traditional game-playing evaluation functions. To evaluate a candidate move, the program assumes that the move has been played and then does a full-board score estimate based on the resulting position. Müller notes that this approach, while somewhat effective, tends to be slow, and often overlooks defensive moves whose value is not apparent until much later in the game. However, Monte Carlo methods are gaining popularity as position evaluators because they allow for reasonably accurate prediction of final territory. Bouzy's Indigo program employs these methods, though in order to make them feasible, the program also has a move generation system that produces only a small number of moves to test [32].

Move evaluation is more peculiar to Go. Programs employ several specialized, purpose-driven move generators to propose moves and score these moves according to various (fast) heuristics. If the same move is proposed by different generators, the scores are summed. Because of its speed, it allows for a larger number of candidate moves to be considered, though Müller notes that this faster approach is less reliable [18]. GNU Go, a top-rated program, relies heavily on this technique.

Ultimately, any decision to play a particular move hinges on the amount of territory that can be won at the end of the game, and even programs that employ heuristic move generators tend to analyze projected territory. However, there is still no accepted best method for this (besides playing the rest of the game). Chen notes in

[28] that two top-level Go programs will sometimes differ by as much as 30 points of territory in their analysis of the same position.

2.3.4 Machine Learning and Go

> [M]ost current programs rely on a carefully crafted combination of pattern matchers, expert rules, and selective search. Unfortunately, the engineering effort involved suggests that making significant progress by simply fine-tuning the individual components will become increasingly difficult and that additional approaches should be explored.
>
> Huang et al, 2004 [5, p. 1]

Applying machine learning techniques to Go is a popular research area, though ML-based Go programs have yet to reach the level of their contemporaries. Artificial Neural Networks (ANNs) are popular, and the usual goal of learning is to determine whether a given group will live or die (or, put another way, whether a particular board region will be friendly or hostile at the end of the game), [5], [6], [7] [9]. This kind of territory prediction can be readily incorporated into an evaluation function, and has formed the basis of a few Go programs, included Enzenberger's NeuroGo [8]. Other uses for machine learning in Go exist: for example, Jacobs [26] describes using a Neural Network to classify human playing styles.

The typical reasons for choosing ANNs are their pattern recognition capability [5], and that they allow for at least a little structural information to be presented to the network, though not in any systematic way. To take three examples:

- Early work in Go-playing ANNs used each stone and its local neighbourhood as inputs to the network [9]. Enzenberger's NeuroGo was created to address a limitation of this technique: it is difficult for such a program to represent blocks of connected stones [8]. NeuroGo actually treated blocks as inputs to its neural network, and represented the adjacency relationships among blocks via connections inside the network. It has been further refined to use soft segmentation decomposition techniques to view the board as local areas of influence, allowing for higher-level groups of related stones to be presented as inputs to the network [8].

  Enzenberger notes that "...most parts of the network see only a portion of the board..." [9, p. 7]. In my mind, this points to another significant flaw: the network cannot learn generalized Go knowledge that is shared across the whole board. It is possible that some useful principle "learned" by one part of the network will be applied only to one part of the board, and not generalized. This is a weakness in artificial neural networks in general

- Huang et al.'s opening game experiments tested several kinds of inputs to the ANN. All of their tests involved feeding the raw board information directly to the network's input layer. In some experiments they provided higher-level information about the candidate move as inputs, including the distance of the vertex from the edge of the board, the number of liberties of the block that the move would connect to, the number of stones in the block, and the number of friendly stones nearby [5].

- In their experiments on predicting whether a group of stones would live or die, van der Werk et al. actually provided no direct information about the stones' configuration to their neural network. Instead, input is in the form of several higher-level features, such as the number of stones, the number of liberties, and the number of immediately adjacent enemy stones [7].

The common problem with these three approaches is that the network is not employed to inductively determine high-level information about the board. In the case of Huang et al. [5], certain higher-level “features” are detected algorithmically and then given to the ANN as additional inputs, as if they were somehow independent of basic board information.

NeuroGo takes a more interesting approach by allowing the board’s topology to shape the neural network, but, as is also the case with van der Werk et. al’s experiments, most or all of the *classification* of the board is done *before* it is provided to the neural network classifier—why do this unless the network is not up the the classification task?

The reality remains: Go is a conceptually multi-leveled game, and neural networks are not designed to deal with such hierarchical information. A classifier based on ETS would not have such limitations.

2.3.5 Monte Carlo techniques in Go

As the name implies, Monte Carlo (MC) techniques depend on randomly generated information. The first Go program to use this technique was called Gobble [34]. It was inspired by simulation techniques in physics, and its designer estimated its strength at about 25 kyu, which is very weak. It is actually a very simple program, with no built-in Go knowledge beyond the basic rules for legal moves, and its play was also limited to a 9×9 board. Its basic structure is this: each turn, play out the remainder of the current game making random moves. Do this several thousand times. Record the final scores correlated with each move that is made. Sum the final scores for every game a particular move appears in. Do this for all moves, then choose the one with the best score.

The advantages are twofold. One, there is no search through an exponentially-growing game tree, since a fixed number of random games are played at each juncture (though this can still be slow if the number of random games is high enough). Two, move evaluation is easy to perform, since the only positions to evaluate are end-game positions, and all that needs to be done is to compute the score. Note that in this system, the game ends when each vertex is occupied by either a stone or an eye [32], so the final positions are much simpler than in a typical human game.

In [35], Bouzy and Helmstetter outline experiments done to improve the play of a program using MC techniques. The most relevant improvement was the addition of some minimal Go knowledge beyond the rules of the game: the definition of an eye. Bouzy later incorporated MC methods into his knowledge-rich Indigo program's move generation function. Other researchers have started using MC techniques as well; a new program called MoGo [36] that uses MC techniques is currently ranked highly among programs that play on a 9×9 board. Recently, Kocsis and Szepesvári have proposed a planning algorithm that combines MC simulation with bandit based decision pruning, and it promises further improvements for Go programs [37].

There are several reasons why Monte Carlo Go is relevant for this thesis. First, this approach to solving a problem, while fairly typical of AI techniques, is completely unlike the way any real Go player operates. Even if a real player had the superhuman ability to play out 10,000 random games per second, doing so would not help them nearly as much as playing normally. Not only is this infeasible for a human player, it is unnatural.

Second, from one point of view, such programs do not actually play Go. Bouzy and Helmstetter [35] contains the record of a game between the Monte Carlo program OLEG and a human, in which OLEG doesn't seem to be playing Go so much as another game called “don’t get captured”. The authors note that the program tends to make tight groups that can’t be captured and do not claim much territory. This makes sense given that random games are played. One cannot make much territory when one is randomly placing stones inside one’s own borders, and the only prohibition against placing stones in OLEG is against filling eyes, which are the smallest units of territory, and this leads to a simplified “model” of Go that is not a good approximation for the game itself.

Third, it highlights the fact that *non-terminal* positions in Go are difficult to evaluate, while terminal positions are easy to evaluate (simply score the game and count the points). While the technique (playing random games) is questionable, trying to determine how easily a given position leads to a favourable outcome is a useful way to approach Go. This is a key reason to apply ETS: the concept of *generativity* is fundamental in the formalism, and is a natural way to describe the evolution of Go positions. If (as ETS allows), a Go program can create a generative class description for the kind of group being formed, that program can tell *at the early stages* whether this group can successfully develop.

Interestingly, Bouzy has incorporated MC techniques into a more sophisticated knowledge-based Go program [32]: the new version of Indigo uses MC as part of its move evaluation function, and this has improved the program’s overall abilities [32]. This success is due to what I have described above: the MC aspects of the program form a crude system for predicting final territory.

2.3.6 Solving sub-problems

2.3.6.1 Opening game

There are several standard opening plays in Go, typically involving play near the intersections of the third lines from the corners and then expanding along the edges on the second or third line. The purpose of these opening moves, or *joseki* is to make the first tentative territorial claims, and many Go programs make use of a fixed library of such plays [18]. There is little room for even local searches in the early game because of the complete lack of constraints when the board is open.

As mentioned above, there have also been attempts to use learning techniques to develop opening game strategies; [5] describes the use of an ANN to learn opening plays. The network played the first 10 moves of the game and then GNU Go self-played the remainder to determine the final score and provide feedback.

2.3.6.2 Endgame

The endgame is one of a very few aspects of Go where computers sometimes out-perform humans [33]. As the game nears completion, a large portion of the board is filled with stones, some territory has been settled, and the remaining disputed areas can often be partitioned into several independent “sub-games”. These facts make many (but not all) endgame situations amenable to techniques from combinatorial game theory, such as decomposition search [30].

Decomposition search is a divide-and-conquer approach to standard game-tree searching. In the Go endgame, heuristic Go knowledge can be used to find the regions of the board that are independently disputed. Optimal solutions to each region can often be found via a complete search to all terminal positions, and the complexity of these individual searches is much less than a combined full-board search would be. In [30], Müller presents Go examples where decomposition search has been used to solve the endgame as much as 60 moves before the game's completion. Even more impressively, work by Berlekamp and Wolfe [38] has shown that combinatorial game theory techniques can sometimes yield end-game solutions *better* than those played by human Go masters [33].

2.3.6.3 Life, death, and races to capture

In the course of a Go game, the question arises: can this group of stones be saved? The player who sees a definite answer to this question of *tsumego* has a clear advantage, either way. If the group can be saved, it is important to realize this quickly before it is too late. Likewise, if a group cannot be saved, it is best to abandon it rather than placing more stones only to be captured.

While Müller notes that the ability of Go programs to solve these problems does not surpass the level of their play in general, in certain restricted cases, such as when the group cannot escape to friendly stones and must make two eyes to live, computers can solve these problems very well [18]; the GoTools program [25] is known for its capabilities in this area. This class of *tsumego* problems are typically solved by local tactical searches [39]. As with the endgame, for an individual threatened group the search space is often restricted and the goal is clearly defined.

However, the more general (and common) *tsumego* problems that occur when a group can potentially live by escaping are much more difficult for computers to solve [33].

2.3.6.4 Repeating patterns

Go games include certain kinds of repetition, such as ko fights (see Section 2.1.1), which Go programs often have a specialized module to deal with [18].

Another repeating pattern commonly seen in Go is a *ladder*: a group of stones is threatened with capture and has only one escape route, and the player attempting to capture the group repeatedly limits it to just one escaping move. Lines of black and white stones grow across the board until an encounter with other stones or the board edge either allows the group to escape or seals its fate.

Because the move choices available to extend or constrain the ladder are very limited, local search techniques are effective. The formerly top-ranked program Goliath optimized its ladder searches by playing out the entire ladder as if it was one move and then evaluating the resulting position [33]. In [18], Müller describes a game between human professional-level players in which one player misjudged the outcome of a ladder, and this costly mistake eventually forced him to resign. Müller notes that a computer would never make such a blunder.

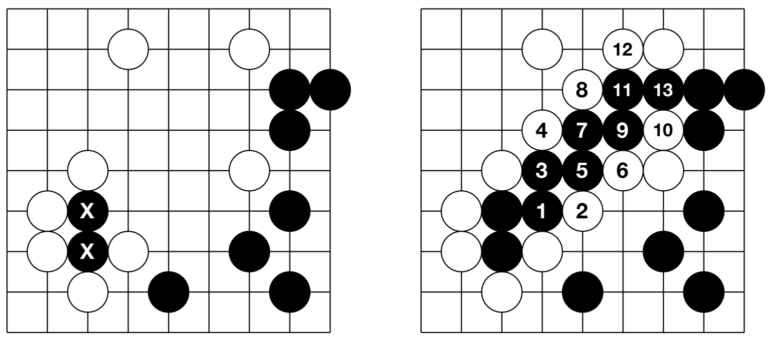


Figure 2.7: A Ladder. The two stones marked ‘X’ will be captured unless black plays 1. However, white’s response of 2 forces black to play 3, and 4 forces 5, etc. until black can connect with friendly stones on the other side of the board.

2.4 Aspects of human play

> Human professional players are amazingly good at Go. They are able to recognize subtle differences in Go positions that will have a decisive effect many moves later, and can reliably judge at an early stage whether a large, loose group of stones can be captured or not. Such judgment is essential for good position evaluation in Go. In contrast, obtaining an equivalent proof by a computer search seems completely out of reach.
>
> Martin Müller, [18, p. 7]

Judith Reitman's study [40] examined how Go players remember information about game positions, using the same methodology as earlier studies on chess players' perception and memory (see [41] for discussion of de Groot's key study [42] and others). She showed that Go players, like chess players, conceptually divide the board into *chunks*—groups of stones that are recalled as a single unit—and suggested that this grouping depends on the player's understanding of Go: the expert player outperformed the beginner in recalling positions taken from real games, but fared no better than the beginner in recalling random patterns of stones[2]. In other words, there is a distinct difference between real Go patterns and "nonsense" patterns, and expert players are good at remembering and thinking about the *real* patterns.

Perhaps the most interesting aspect of [40] is where it differs from the earlier studies on chess: in the chess studies the remembered chunks were assumed to be hierarchical, but the lowest-level set of chunks formed a strict partition of the pieces on the board [41]. Thus, any hierarchy stored in the memory of the player would be built of non-overlapping components. This assumption (while perhaps merely dubious when applied to chess) was shown not to bear out for Go: there was good evidence

[2]Interestingly, this was the only task set by de Groot in his Chess study in which master players consistently out-performed weaker players [41].

that the expert player organized the stones on the board into *overlapping* collections (examples are presented in Chapters 4 and 5).

The most likely explanation for why the Go master stored information about the board in overlapping chunks is also the simplest: stones on the board influence each other in complex and overlapping ways. This would seem to be at odds with Go programs that segregate the board into non-overlapping patterns of stones, as, for example, a pre-Monte Carlo version of Indigo did [4].

When Chen implemented decomposition search in his program Go Intellect, he quickly discovered that decomposing the *game board* into "independent" *regions* had a "disasterous" effect on the program's play [29, p. 1]:

> The decomposition of the board into sub-regions do not really produce independent sub-games. The development of a region may have significant impact to some other regions, which is not measured in the local sub-game score.

This led him to the technique of soft decomposition, where the game is partitioned into subgames that are not strictly limited to one region of the board (that is, two or more sub-games can be played out on overlapping portions of the board), lending credence to the importance of treating Go shapes as overlapping.

2.4.1 Where computers out-perform humans

Playing end-games, calculating ladders, and determining if an enclosed group can be captured are all areas where computers better human experts [18, 38]. These are all well-defined sub-problems that lend themselves to min-max search and other forms of brute-force calculation, and there are usually optimal solutions (sometimes only one). Recalling the passage from the Nihon-Kiin I quoted in Section 2.2: these sub-problems are the ones where there is no room for creativity, and as such are not representative of what makes the game such a challenge. Computer success here should not be mistaken for progress towards solving Go in general.

2.4.2 Where humans out-perform computers

Computer play still lags far behind that of dedicated humans. Besides failing to choose the best move, many programs have particular "blind spots" or other significant weaknesses, and can be made to look foolish if a human player knows how to exploit their shortcomings. [18] includes the record of a game Müller played against The Many Faces of Go [43], in which he gave the program a 29-stone handicap and then defeated it. Ironically, Many Faces is considered to be a strong "fighting" program (a 29-stone handicap means the board is crowded and the players are in constant conflict), but it was fighting that was the program's undoing: a human player with a 29-stone lead would have played conservatively and easily surrounded his opponent. Many Faces could not adapt and instead opted for a direct confrontation at every juncture, allowing its superior human opponent to slowly turn the tide.

It is worth noting that at the time of writing, some MC-based programs are beginning to challenge high-level human players on small boards [3].

Chapter 3

Evolving Transformation System

3.1 ETS is a representational formalism

A *formalism* is a mathematical framework or language, well defined axiomatically. The prefixed "*representational*" indicates the purpose of the formalism: it is designed to model any arbitrary phenomenon. In order to be an effective tool, a representational formalism must be able to account for the *structure* of objects [44, p. 38]:

> Many artificial intelligence problem domains require large amounts of highly structured interrelated knowledge. It is not sufficient to describe a car by listing its component parts; a valid description must also describe the ways in which those parts are combined and the interactions between them. This view of structure is essential to a range of situations including taxonomic information, such as the classification of plants by genus and species, or the description of complex objects such as a diesel engine or human body in terms of their component parts.

Currently, the representational formalisms most commonly used in AI are vector spaces (and more broadly, numeric formalism) and logic (especially first order predicate logic) [14]. The former are usually seen in statistics-based machine learning, and the latter in “good old fashioned” AI (expert systems, theorem proving, etc). The principal difference between these formalisms and ETS is that ETS was *designed* with representation and inductive classification *in mind.*

Numeric formalisms are amenable to statistical approaches to classification but fail to adequately account for object structure. The main problem is that the vector space formalism *itself* contains only two operations (addition and scalar multiplication). In order to create “interesting” objects, mathematical constructs that are external to the formalism itself get imported[1]. Because objects (read: points in an abstract space) lack any kind of structure, it is likewise impossible to create structured class representation out of them.

First order logic was created to codify “the laws of thought”, but does a poor job of describing the objects being thought about. Predicates merely *name* things without describing their structure, and so it is difficult to build sensors that directly output predicates: some classification is inevitably needed. The predicates of logic are closely related to natural languages (and do roughly the same job), but when someone uses a word, for example, “dog”, to name a particular animal, the audience depends on their own internal conception of what a dog *is.* Predicates (and language) turn out to be very useful tools of abstraction for humans, but they depend on our underlying classification abilities. Any effort to build an AI system based on predicates that has to interact with real phenomena is going to have to come to grips with this eventually.

[1]See [45] Section 8 for a discussion of these issues

3.2 Classes and inductive learning

3.2.1 ETS class definition: classes are generative

The ETS formalism is strict on the meaning of "class". What a group of objects from the *same class* have in common is their shared generative mechanism: that is, *all elements of a given class are assembled/constructed in the same way.*

Formally, ETS borrows this concept of generativity from Chomsky grammars [46]: an ETS class description is a system of "rules" that can generate all members of the class[2]. This is (almost) analogous to using a set of production rules to generate all strings in a language, but with a few key differences. First, constructing a string according to a set of production rules is a purely syntactic enterprise. While it is true that the primitive units of ETS representation (henceforth "primitives", see Section 3.3.1), like the characters that make up a string, are also "idealized" formal objects, they have a much more "representational" flavour than string characters: ETS primitives stand for and mirror the structure of some observed real event. As such, connecting ETS primitives together to form a *struct* (a "chunk" of representation - more in Section 3.3.2) has semantics as well as syntax.

A second difference is that ETS primitives are *temporally ordered*—in effect, events that occur in an object's past must be described before events in the present. In the case of strings, there is no such limitation: new characters can be added in any position (as opposed to, for example, strictly writing from left to right).

Lastly, and perhaps most significantly, ETS has no "non-terminal" symbols like those that are used in the process of generating strings via a grammar's production rules.

[2]The *real* process that this formal mechanism seeks to model is real-world object formation. Biological objects have the most elaborate formative processes (see: embryology), though all objects must have formed somehow. See Section 3.2.2 for a more detailed discussion of this.

Because the steps used in the construction of a string are discarded and not stored in the string itself, a given string could have been constructed in infinitely many ways, and as a result inductive recovery of a grammar is not feasible from individual strings.

Following from the above, it is clear that not any arbitrary grouping of objects can be considered a class: for example, one could not construct a class that contains both apples and wax apples, since they are formed in completely different ways.

Furthermore, while it is true that a *set* containing all members of a class could be defined, this set alone would not adequately capture the nature of that class, because the class itself has its own representation independent of its members. To return to the analogy with strings, the entire set of strings in a language is not the same as the rules used to generate them. However, in order to support inductive learning, the relationship between class description and class members should be very tight: having even a few members of a class *should be sufficient to inductively discover the class representation.* This is emphatically not true of strings and their corresponding generative grammars.

If one insists on the generative definition of a class, one discovers that most existing "classes" in ML fail to meet it. Indeed, it is often the case that the class variable does not really represent a class at all, but rather some additional property (or feature) of the various objects. Groups that are often treated as classes but do not meet our stricter definition might include: "things that are blue" (having a colour is a property or feature of several objects, not a class description), or "photographs that contain faces" (though the faces themselves could be construed as a class).

Alternatively, some examples of classes that fit the ETS definition might include: human beings (sharing common evolutionary and developmental histories), stars

(stars are formed by the agglomeration of gas and dust), water molecules (formed by the bonding of two hydrogen atoms to an oxygen atom), the process of a human taking a walking step [47], the process of making a purchase [48].

3.2.2 Formative history

The concept of *formative history* is essential for ETS classification in general and important for modeling Go in particular. The ETS formalism allows for (and insists on) the representation of the formation of objects, and this is unusual in machine learning, where thus far the focus has been on the features of finished objects.

In fact, as the next subsection will make clear, an "object" is represented in ETS *as* its history, not as a set of properties.

> One critical feature of biological "objects" stands out: *any* organism is not built from scratch but rather its instantiation requires following some kind of stored "formative history". It appears quite reasonable to extend this form of instantiation to *all objects in the universe*, including man-made objects, where the "formative" history should be interpreted reasonably broadly. Indeed, stones, pencils, web pages all have their formative histories, albeit of different "kinds". For example a web page has a quite complex formative history related to its conception and execution. [14, p. 4]

Formative history is the series of steps taken to *create* a given object. For a water molecule, formative history is the process of combining an oxygen atom with two hydrogen atoms. For a human infant, formative history is the complex embryological development process (which in turn is linked to the long-term evolutionary history of the species). For a Go "shape", formative history consists of the moves that were played to construct it.

3.2.3 Objects as processes

On those who step into the same rivers ever different waters are flowing.

Heraclitus [49, p. 41]

In AI and related fields, objects are usually treated as non-temporal entities, often represented as points in some static space. By contrast, in the ETS view, what I have thus far been referring to as "objects" are more properly called "processes". The basic representational primitives of the ETS formalism represent events, and a process (be it a Go shape, or a bumble bee, or a water molecule) is a collection of these events, arranged temporally.

This *process view* is based on the principle that nothing in the world exists outside of time: objects undergo continuous change. Even seemingly "stable" objects are in a state of flux: biological organisms undergo metabolic processes, stars consume their fuel, rocks are eroded by wind and water, electrons zip endlessly through the space around nuclei. In ETS terminology, such "stable" processes are considered to be "mature". These processes have completed their formation and behave in more regular ways, but they are not static: "objects", as understood in the traditional sense, are epiphenomena, roughly corresponding to the "state" of an unfolding process at some frozen moment in time.

Because ETS treats everything as a process, it is possible to model real-world *dynamic* phenomena very naturally. For example, one of the classes presented in [48] is called "Material Acquisition", and it produces elements that represent various purchases being made. It is difficult to treat such real processes as static objects. The same is true of the process of human walking, which is described in [47].

3.3 Modeling in ETS

3.3.1 ETS primitives and primal classes

The two interrelated basic units of the ETS formalism are *primal classes* and *primitive transformations* [12, Definition 1]. Primal classes represent classes of similar *processes* and primitive transformations represent *events* that transform several such processes into several other processes. The internal structure of the event and associated processes is suppressed.

More formally, an ETS *concrete* primitive is defined in terms of two tuples: its initial primal processes and its terminal primal processes (or *initials* and *terminals*). Each primal process is an *element* of some primal classes. For example, "Bob" and "Jim" might be names of primal processes belonging to the class of persons depicted in Figures 3.1 and 3.2. Typically, an ETS representation of data will include many primitives that have the same structure, i.e., that capture the same kind of event happening to different elements of the same primal classes. The set of all such "same events" is called an *abstract* primitive, and a typical ETS representation of data will include several different abstract primitives.

Figure 3.1 shows a pictorial representation of three abstract primitives and their associated primal classes from three different "domains", [47], [48], and this thesis. The lines attached to the top of each primitive represent its *initials*, the lines attached to the bottom represent its *terminals*. Each initial and terminal has an associated line-style and small shape to denote to which primal class it belongs. It is convenient to refer to a primitive as having, for example, two initial *sites* and three terminal *sites*, meaning that the primitive accepts two initial processes and transforms them into three terminal processes.

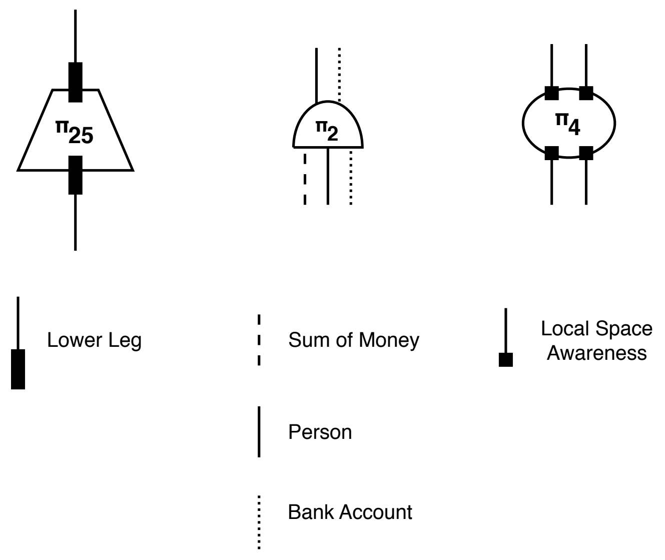


Figure 3.1: Three ETS primitive events, labelled π_{25}, π_2, and π_4, along with their associated primal classes, from three very different domains.

The primitives from Fig. 3.1 sit at very different "conceptual" levels: π_{25} is a physical event that happens to part of a human (the lower leg); π_2 occurs when a person and a bank account interact in a particular way, and results in three new processes corresponding to the same person, the same bank account with a reduced balance, and the sum of money that has been withdrawn; π_4 is an "informational" event that operates on the "mental" processes of a Go player.

It should be clear from Figure 3.1 that ETS primitives can capture wildly different kinds of events. The primitives that are needed to model a particular domain are dictated by the types of events the domain's primal classes undergo. With this in mind, the first thing one needs to choose when modelling a domain is the set of primal classes. This set should be small in size, and the individual primal classes ought to be of comparatively low complexity. Thus, for example, if one wanted to model the construction of houses, one would be well served to choose primal classes corresponding to nails and boards, and allow the floor, wall and room classes to be

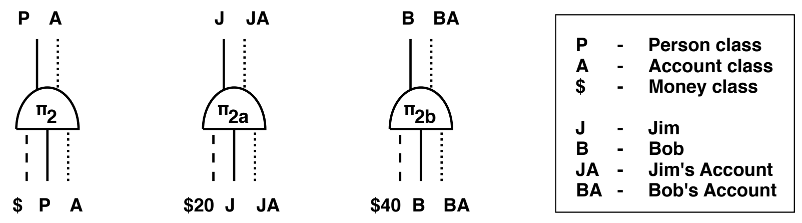


Figure 3.2: An abstract primitive (left), and two concrete primitives.

assembled out of them.

A related fact about primitives and primal classes is that in order to be useful in modelling real phenomena, they must be easily detectable. Although a primal class is rightly called a *class*, one does not want to have to *learn* that class inductively, since it serves as an atomic unit in the learning of other, more complex classes. Likewise, one wants events that can be easily recorded via sensors, perhaps with some additional software processing. Obviously, these concerns inform the choice of primitives and primal classes: if your events are difficult to detect and your primal classes require learning, you should search for simpler building-blocks to begin with.

Figure 3.2 shows *abstract* and *concrete* primitives. An abstract primitive is a postulated event that is only defined to the level of which primal classes make up its initials and terminals (e.g., “persons”). A concrete primitive can be viewed as the instantiation of an abstract primitive: it is an actual, observed event, defined by the specific primal processes involved (e.g., “Jim”, “Bob”). Formally, we denote a concrete event with a double subscript, e.g., π_{2a} and π_{2b} are both concrete instances of the abstract primitive π_2.

The first concrete primitive in Fig. 3.2 corresponds to Jim withdrawing \$20 from his account and the second corresponds to Bob withdrawing \$40 from his account. Strictly speaking, while some of the initials and terminals have the same label (e.g., ‘J’), the ETS formalism insists that the terminal process must have undergone some change as a result of the event. In the case of the bank accounts, this is easy to see: though not expressly indicated, the terminal JA is the same account but with a balance \$20 lower than the initial JA. In the case of Jim himself, the difference is harder to see, but conceptually, the “new” Jim has slightly more *history* than the “old” Jim, on account of having participated in one more event.

Unlike logic, where a predicate is a kind of referential place-holder, or formal grammars, where any meaning attached to the characters is purely external, ETS primitives’ syntax and semantics are tightly linked.

> The novel nature of ETS representation ... allows the introduction of structural primitives whose syntax and semantics are inseparable, and it also makes such primitives fundamentally different from similar concepts in previous formalisms. [45, p. 5]

In logic, the rules that govern where a particular predicate can appear in the course of a proof are purely formal; there are no rules derived from the predicates themselves. $P_x \Rightarrow Q_x$ is valid construction, regardless of whether P and Q stand for “tiger” and “cat” or “alive” and “dead”.

In ETS, the attachment of a particular primitive to primal processes is governed by the primitive’s structure: you cannot, for example, have a “car crash” primitive modify a process corresponding to a grapefruit. Not only is it nonsensical (car crashes happen to cars, not grapefruit), but it is also un-syntactic, since the “car crash” primitive does not have “grapefruit” classes as its initials and terminals.

3.3.2 Structs

As primitives and primal processes are observed and recorded, they can be *attached* to one-another to form a "structural history", or *struct.* The word "recorded" in the previous sentence is important: a struct is a transcription of gathered data about one or more objects/processes. Figure 3.3 shows a pictorial representation of an example struct.

The most immediate difference between a struct thus depicted and an ordinary directed graph is that a struct contains *temporal* information, because the initial processes of one primitive can only be connected (or *attached*) to the terminal processes of a temporally preceding primitive; cycles are not permitted.

When a struct is presented pictorially, the temporal information is inherent in the relationships between the primitives, not in the exact positions of the primitives on

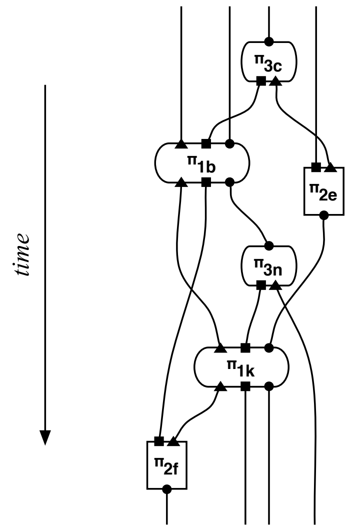


Figure 3.3: A struct, composed of six primitives belonging to three abstract primitive types. The primitives are connected by primal processes, and the corresponding primal classes are denoted by the shapes on the top and bottom of each primitive. Primitive events are temporally ordered, from the top of the struct to the bottom.

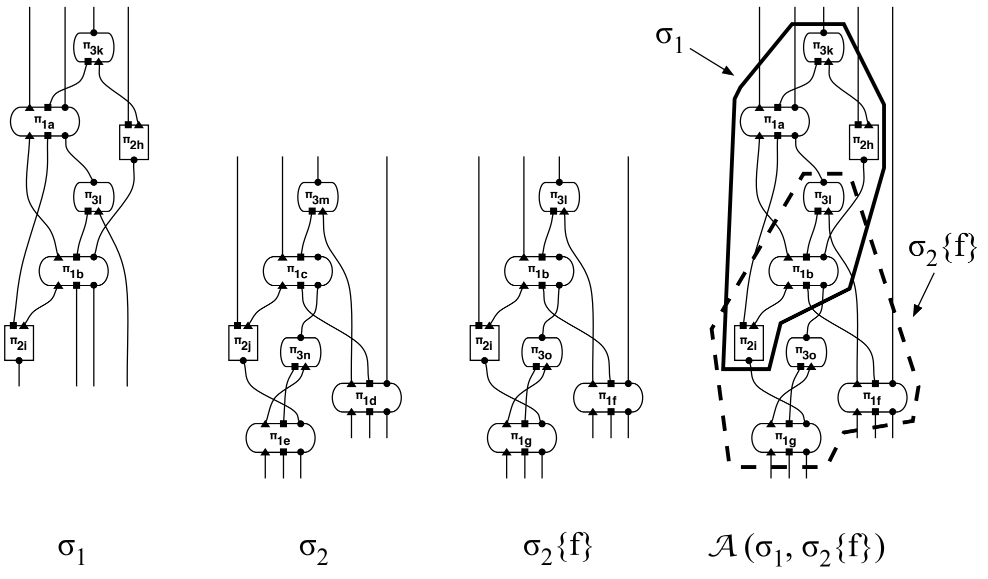


Figure 3.4: Struct σ_1 is assembled with $\sigma_2\{\mathrm{f}\}$, which is a *relabeling* of (and hence *structurally identical* to) σ_2. The assembly is on their common primitives π_{3l}, π_{1b}, and π_{2i}.

the page. For example, in Fig. 3.3, π_{1b} takes place before π_{3n} because π_{1b} initiates one of the processes that participate in π_{3n}. π_{2e} takes place before π_{1k}, but whether it takes place at the same time as π_{1b} or π_{3n}, or before them, or between them, or after them, cannot be determined from this struct, since there are no processes that depend on all of them.

As Figure 3.4 shows, two structs, perhaps corresponding to two separate observations, that share some common structural history (i.e., connected primitives) can be *assembled* to form a new, larger struct. This assembly depends on the concept of *relabeling*: concrete primitives from different structs can be systematically renamed so that they coincide.

The structs shown in Figures 3.3 and 3.4 are more properly called *level 0 structs.* Higher level structs, which depend on *classes*, are created on top of these basic structs.

3.4 Classes in ETS

3.4.1 Level 0 classes

A (sufficiently large) struct is actually a recording of several *processes*, each of which is composed of sequences of events. Each of these new processes is *an element of some level 0 class.* The goal of learning under the ETS paradigm is to inductively construct representations of each class, so that subsequently observed elements of each class can be recognized, where this recognition takes the form of an overlapping partition of the underlying struct (Figure 3.5).

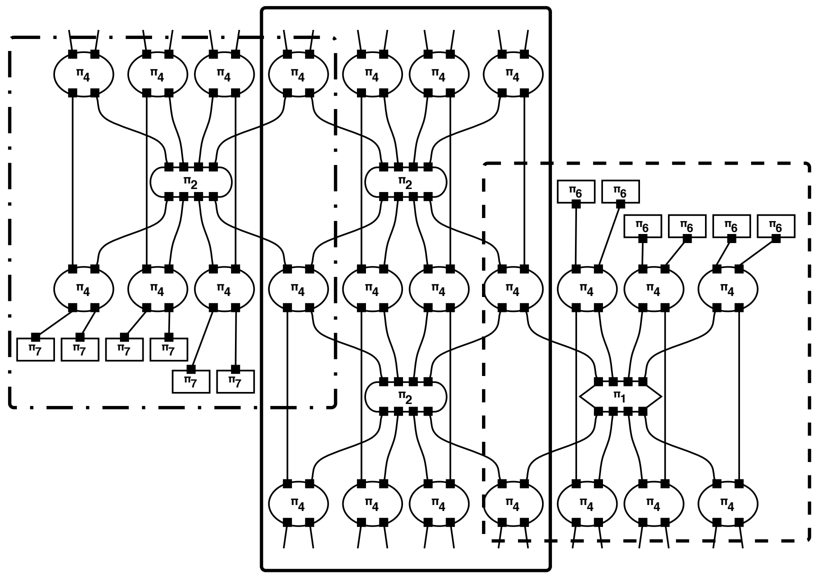

π_4 π_4 π_4 π_4 π_4 π_4 π_4
π_2 π_2
π_6 π_6
π_6 π_6 π_6 π_6
π_4 π_4 π_4 π_4 π_4 π_4 π_4 π_4 π_4 π_4
π_7 π_7 π_7 π_7
π_7 π_7
π_2 π_1
π_4 π_4 π_4 π_4 π_4 π_4 π_4

Figure 3.5: A struct containing three level 0 class elements, delineated by boxes.

3.4.1.1 Level 0 constraints

Central to the concept of a class in ETS is the concept of a *structural constraint* (or informally, "a constraint"), which is an abstract specification of a family of structs [12, Def. 9]. Denoted $\text{Con}(\Pi_1)$, a constraint has a set of concrete primitives (Π_1), and

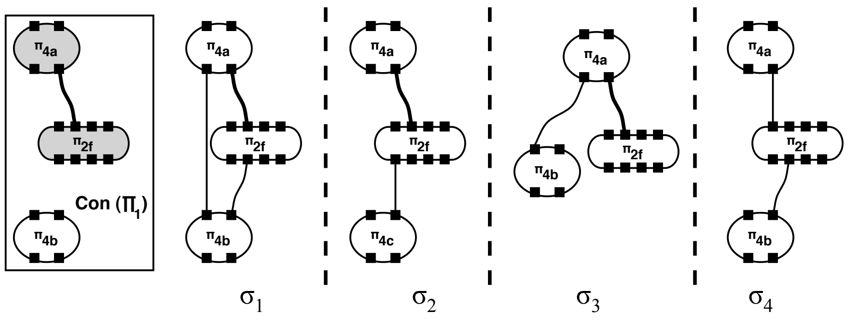


Figure 3.6: In the box: a structural constraint. Structs σ_1, σ_2, and σ_3 satisfy this constraint, while σ_4 does not. The context part of the constraint is shown in grey (see Fig. 3.7).

a set of primal processes connecting some of these primitives. A struct that *satisfies* the constraint is a struct whose primitives are Π_1 (perhaps under relabelling) and whose set of primal processes includes (but is not limited to) those specified by the constraint, see Figure 3.6.

A constraint may be *applied* to an existing ("working") struct. In this case, a struct that satisfies the constraint is non-deterministically chosen, and then assembled with the working struct to form a new struct. In order for the application to be valid, there must be at least one primitive in common between the working struct and the added struct (otherwise the assembly is not valid), and there must be at least one primitive that is not already present in the working struct. Constraints may optionally have a *context part*, which is a set of primitives that *must* already be present in the working struct in order for the constraint to be applicable. Conceptually, the context might be those events that "cause" the later (newly added) events.

A useful addition to the concept of a constraint in [12] is the addition of a *null constraint*, denoted Θ. Θ is applicable to any struct, and does not change the struct when it is applied.

3.4.1.2 Level 0 class generating systems

The relationship between the class description and the class elements is that, when put into action, the class description becomes a generating system that can produce all members of the class.

The actual formation of a class element proceeds stepwise, with new pieces being added to the struct corresponding the class element as it has thus-far formed (the *working struct*). These new pieces take the form of a small struct, satisfying a constraint that is *applied* by the *class generating system*, meaning that the new struct is assembled with the working struct, implying that at least one primitive in the new struct already exists in the working struct.

An ETS *class generating system* contains several sets of constraints, and has rules describing the order in which to use them. At each step in the process of forming a class element, the generating system non-deterministically chooses one constraint from the corresponding set, and applies it.

An additional factor in the class generating process is the environment. No object exists in isolation: the formation and existence of most things depends on other nearby things. In ETS terms, processes can affect each other. Thus, at each step in the action of the generating system, external processes (collectively known as "the environment") are given a turn to attach primitives to the working struct. This allows for the formation of class elements to be influenced by other processes running in the same setting, in a way that is much less drastic than the complete restructuring.

The environment may add "noise" to a forming class element, or it may add essential structure. In the latter case, two class elements *overlap* on key events.

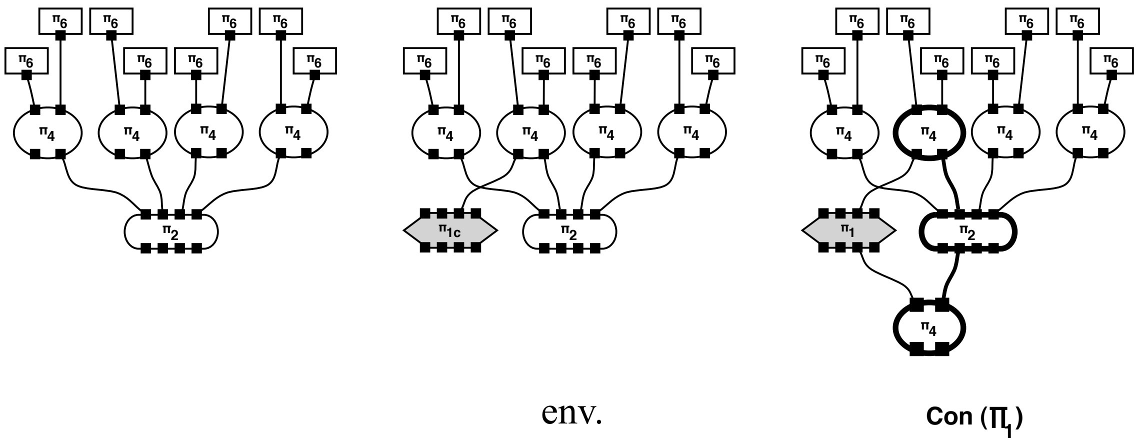


Figure 3.7: One step in the formation of a class element. Left: the working struct for a forming class element. Center: the same struct with a new primitive (grey) added by the environment. Right: the middle struct after the constraint from Fig. 3.6 is applied. Note that the struct satisfying this constraint is shown in bold, and that the constraint's context part was already present in the working struct.

Level 0 classes are denoted with a $\mathfrak{C}$ and an index corresponding to the particular class; for example $\mathfrak{C}_2$ and $\mathfrak{C}_5$ are names of two level 0 classes. Elements of these classes are denoted with a lower-case c, the class index, and a sub-index identifying the particular class element; for example, $c2_1$ and $c2_6$ are elements of the class $\mathfrak{C}_2$.

3.4.2 Higher level classes

ETS supports a hierarchy of class levels, allowing for natural modelling of multilevel phenomena. Each class level organizes the level below, just as level 0 classes dictate the arrangement of primitives. In the case of Go, level 0 classes correspond to individual stones, and level 1 classes organize these stones into small Go shapes.

Level 0 classes are so-called because they have no lower levels—they are simply partitions of the underlying struct. Level 1 classes have 1 lower level (the 0^{th}) and are composed of level 0 class elements. Generalizing, level n classes have n lower levels and are composed out of level $n-1$ class elements.

Level 1 classes are associated with the concept of *level 1 structs*, which are composed of level 0 class elements. The connection between primitives in a level 0 struct is properly called a *class link*; in a level 1 struct, level 0 class elements are "connected" by a *level 1 class element link*, or CEL^1. A CEL^1 takes the form of a set of one or more shared constraints, on which two level 0 class elements *overlap*.

A convenient (though incomplete[3]) notation that I have adopted for depicting level 1 classes is an overlap graph, an example of which is shown in Figure 3.8. Overlap graphs have level 0 class elements as nodes and edges for each level 0 constraint that two elements overlap on. I have similarly used overlap graphs to depict level 1 and 2 classes overlapping on the corresponding level 1 and 2 constraints. The two nodes and connecting edge shown in Figure 3.8's overlap graph could be a CEL^1($\{\mathfrak{C}_1, \mathfrak{C}_2\}$,Con($\Pi_1$)) relating those two primitives, though it is not always possible to directly "see" CEL^{1}s in an overlap graph because CEL^1 relations can be more than pairwise.

[3]A more involved way to draw level 1 structs is presented in [12], Part III.

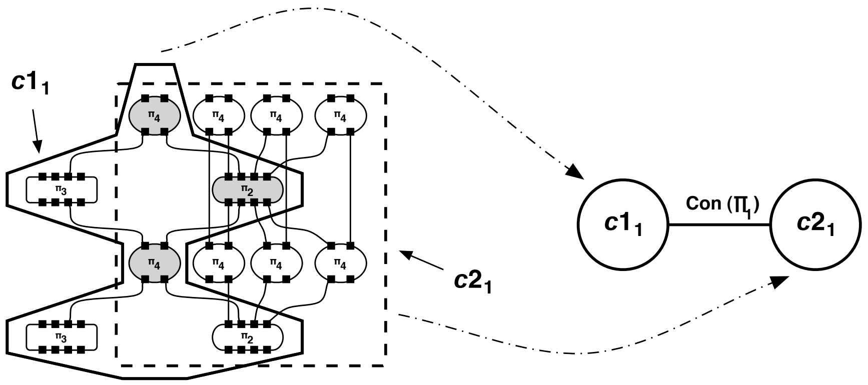


Figure 3.8: Left: a struct containing two class elements, $c1_1$ and $c2_1$, that overlap on a shared constraint Con(Π_1) (shown in grey). Right: an overlap graph showing the class elements as nodes and their shared constraint as an edge. *Temporal information is absent from the overlap graph.*

Level 1 classes use level 1 constraints in the process of generating level 1 class elements. The structure of the ETS definition of a level 1 constraint[4] is:

$$\mathrm{Con}^1(\mathscr{C}) \;=\; \left\langle \mathscr{C}\,,\, \left\{ \mathrm{CEL}^1\big(\mathscr{C}_j, \mathrm{Con}(\Pi_j)\big) \right\}_{1\le j\le J} \right\rangle \;,\; \mathscr{C}_j \subseteq \mathscr{C}$$

In other words, it is two sets, the set $\mathscr{C}$ of level 0 class elements, and the set containing J level 1 class element links (CEL^1's) on particular constraints between one or more of the level 0 class elements.

Following from this, the steps of a level 1 class generating system add *class elements* to the level 1 working struct, and attaches these elements using the specified class links. Higher level class generating systems operate in the same manner, where classes at level n use CEL^n's composed of level $n-1$ constraints to attach level $n-1$ class elements to the level n working struct.

These classes and their elements are denoted in the same manner as level 0 classes, but carry an additional superscript indicating their level. For example, $c3^2_1$ and $c3^2_4$ are both elements of the level 2 class $\mathfrak{C}^2_3$.

It is possible to have arbitrarily many class levels; exactly how many levels there should be is dictated by the phenomenon being modeled, but as a general rule, high-level classes should not become too conceptually distant from those at the lowest level. As conceptual distance increases, ETS allows for the transition to a new *stage* of representation, where the old high-level classes become the new primal classes, and their overlapping areas become the new primitives. As the model of Go outlined in the next chapter operates at a single stage, details of stage ascension are not presented here.

[4]Refer to p. 45 of [12] for the complete definition.

Chapter 4

ETS Go Model

4.1 Overview

The first question to be settled when applying ETS to a given domain is which aspects of the to model. In the case of Go, my main goal is to represent the evolution of shapes as they play out across the board.

In Go, a great deal of complexity is created out of simple elements. Such emergent complexity is captured in a natural way in ETS thanks to the formalism's multilevel class hierarchy. The main approach is to begin by modelling the simplest elements of the game (stones) at the lowest level, and introduce more complex structures at each new class level. Classes at each level are composed out of the previous level classes, mirroring the way that complex shapes in Go are composed out of simpler shapes, and ultimately out of stones. By handling this formally via levels, one can manage the emergence of complex structures in a systematic way.

This structural representation of Go has four levels: a struct level, and three class levels corresponding to stones, blocks, and groups. The standard ETS view is that the struct level should be produced via sensing some real phenomenon (the game board) and the class levels should be produced via a learning system. However, since ETS learning algorithms have not yet been devised, and since developing one is outside the scope of this thesis, the classes that will be described in this chapter were "learned" analytically by me. In fact, the whole system was designed with certain lower level classes in mind, to allow for the natural formation of higher-level classes.

4.1.1 Primitives and primal classes

For reasons explored in Section 4.2, the struct level is actually the representational level *below* stones. Primitives correspond to the Go player's awareness of various (small!) aspects of the Go board, and structs, which correspond to the player's awareness of board positions, are built out of these simple awareness components.

It is important to clarify "awareness" in this context. I have sought to make the (very necessary) distinction between the player's *idea* of the board and the board itself, but do not want to make any strong statements about the actual psychological/neurological mechanisms that support this idea. My "player" is an abstract Go-playing agent (possibly a program implemented using this representational approach). Why have a player at all? Go is a mental phenomenon—the things that the player knows (groups, territory, influence, etc.) are not *on the board*; they depend on the meaning ascribed to the board by the player, so to claim that such aspects were present *in a representation of the board* would be inaccurate.

Although structs are not meant to represent the board *itself*, they are closely linked to it since they represent the player's direct awareness of some part of the board.

Which parts of the board the player should be aware of at any given time is outside the scope of how structs are formed, but struct formation is robust enough to allow many scenarios, from taking the whole board in one “snapshot”, to examining it one stone at a time.

4.1.2 Class level 0: stones and vertices

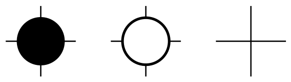


Figure 4.1: A black stone, white stone, and empty vertex, all captured as level 0 classes.

Level 0 classes partition structs into the basic elements of Go. There are three families of level 0 classes: the first correspond to the player’s awareness of stones and empty vertices, the second to the placement and removal of stones, and the third to the player’s awareness of the proximity of stones and vertices to each other. The second and third families of classes are discussed in detail in Section 4.5, but some aspects of the first bear exploring here.

Stones and empty vertices are treated in exactly the same way and the classes that represent each are structurally quite similar: essentially, each vertex is treated as having three states (black, white, empty). Go is a game of empty space just as much as it is a game of stones, and this is highlighted at the beginning and at the end of the game. At the beginning, available space is vast so the eye is naturally drawn to the few “interesting features” of the board: the initially-placed stones. At the end of the game the board is crowded with stones and as a result the few remaining free vertices that have not yet been absorbed into one player’s territory are the focus of attention.

I have specified a level 0 *class setting* of 16 classes. Higher-level class elements will be composed out of elements of these 16.

4.1.3 Class level 1: blocks and spaces

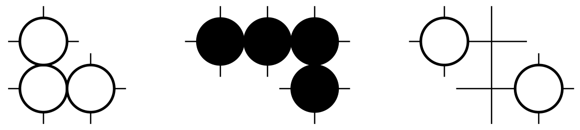


Figure 4.2: Two blocks and a "knight's jump" space, all captured as classes at level one.

Level 1 classes group the level 0 stones and vertices into *blocks* and *spaces*, where a block is a set of connected same-coloured stones, and a space is a collection of "connected" vertices. A block is a well-defined concept in Go: it is a group of stones that live and die together. A space, however, is a useful invention arrived at in the course of developing the model. It is not directly analogous to a block in that it must be delimited by two or more stones (see the knight's jump in Figure 4.2), and, most importantly, need not contain all "connected" empty vertices. Classes of spaces are used to describe the gaps between stones and blocks on the board.

4.1.4 Class level 2: groups

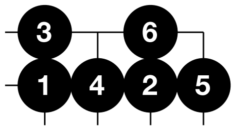


Figure 4.3: The formation of a black group that could be treated as a level 3 class.

Level 2 classes compose blocks and spaces to form *groups* of stones. A group, like a block, is part of standard Go nomenclature, meaning any related collection of nearby stones. A group is often more restrictively defined as some stones of one colour plus any enemy stones that they have effectively captured [28], but my level 2 classes fit the looser definition.

Level 2 is where the representational flexibility becomes apparent. I have only sketched a few examples in Section 4.7, but in practice any kind of group can be learned and represented. What is also essential about group classes is that they embody the formation of the group as well as its shape, and learning useful methods of forming good groups is the key task for any Go program using this system.

4.2 Representing spatial relations in Go

In keeping with the ETS tenet that "formative history is essential", I will discuss the thinking that led to the final set of primitives that I chose, after having tried and discarded several others.

My intuition was that the most basic and process-like element of Go that a player need be aware of is an individual stone. A stone, the most important "object" in Go, is a good candidate to be modelled as a (regular) process because of its stable behaviour: it is placed at some time, sits on the board in a (fixed) position, and (possibly) is removed from the board at a later time.

However, it turns out that stones are not usefully represented as primal processes, because it is difficult to relate primal processes spatially (i.e.: to depict where stones are with respect to one another on the board), as the only way primal processes interact is when they are transformed by a common primitive. By treating a stone as a level 0 class element instead, such spatial relationships can be handled much more naturally via the representational mechanism of overlap.

Since the spaces between stones are just as important as the stones themselves, it makes sense to treat empty vertices as processes as well: in some sense, “empty” is a third stone colour, besides black and white.

4.2.1 What are spatial and temporal relations?

There are two kinds of relationships between events and processes: spatial and temporal. Temporal relationships describe the order in which events occur and processes unfold *in time*. Spatial relationships describe *where in space* processes and events are with respect to each other. Spatial relations are more complex, not only because the number of dimensions involved is greater (three versus one), but because time only “flows” one way, while objects can move freely in space.

Go only has two spatial dimensions, since an idealized goban is completely “flat”, but even so, it is the spatial relationships between stones that pose the greatest representational challenge, since the ETS formalism includes temporal relations automatically: primitives can be explicitly temporally ordered. A given primitive’s initial processes finish before terminal processes begin (or, more properly, one set is transformed into the other). ETS does not, however, include spatial relationships in such an immediate way. Rather, they emerge, built out of the temporal relationships.

4.2.2 The emergence of spatial relations

Setting ETS aside for a moment, here is an example of another “representation scheme” in which temporal relations are immediate but spatial ones are not:

> He saw the two men in blanket capes and steel helmets come around the corner of the road walking toward the bridge, their rifles slung over their shoulders.

> One stopped at the far end of the bridge and was out of sight in the sentry box. The other came on across the bridge, walking slowly and heavily. He stopped on the bridge and spat in the gorge, then came on slowly to the near end of the bridge where the other sentry spoke to him and then started off back over the bridge. The sentry who was relieved walked faster than the other had done (because he's going to coffee, Robert Jordan thought) but he too spat down into the gorge.
>
> Ernest Hemingway, For Whom the Bell Tolls [50, p. 482]

The temporal relationships among the described (fictional) events are implicit in the basic structure of the narrative quoted above. The order in which the author tells us about the events maps directly to the actual order of the events themselves. This order is not simply based on where they appear on the page: the written narrative is most properly *a representation* when it is activated by being read—and as reading takes time, the reporting of the events really is temporally ordered. By contrast, the spatial relationship between those participants in the various events are not explicitly part of the relationship scheme.

When we read "the other sentry spoke to him", however, the words the author uses to describe the event (the brief conversation) allow us to infer something about the spatial relationship between the two sentries. The use of "spoke" instead of another phrase (for example, "called out") implies that they are standing close together. We are not provided with any numeric distance information (though we could probably infer some reasonable numbers).

This state of affairs is much the same as in ETS—processes that interact via events are clearly spatially related, but exactly how they are related depends on the particular event in question.

The naive approach to modelling space in ETS would be to follow this logic: the most obvious and explicit way to demonstrate that two processes are adjacent to each

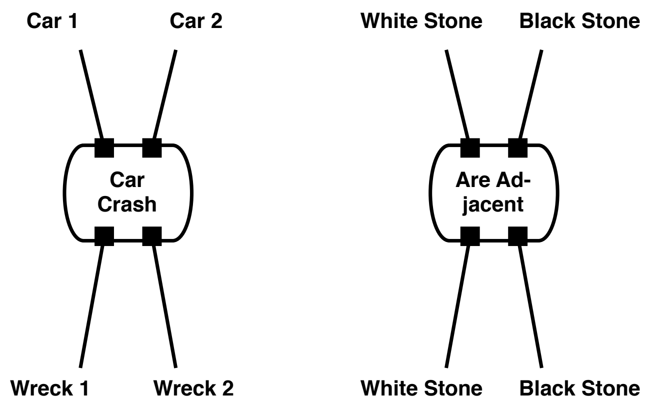


Figure 4.4: Two ETS primitives that imply spatial interaction.

other is to have them interact via a common event: Figure 4.4. While the situation on the left in Figure 4.4 is quite natural, the situation on the right is not, as the event itself, which might be construed as the perception of adjacency between two Go stones by the Go player, is a little contrived (though a few "unnatural" events are to be expected because Go is an artificial domain), and ultimately leads to some representational difficulties, as Figure 4.5 illustrates.

The ETS formalism suggests that an event completely restructures its initial processes en route to generating its terminal processes. The terminal processes may be similar to the initials—they may be elements of the same class—but they cannot be the *same* process: at best they are the previous process after undergoing one more event. An event like the right-hand primitive in Figure 4.4 appears to have terminals that are the same processes as its initials, when in fact the identity of each individual stone process is broken by the event: you cannot, according to the logic of ETS, consider the terminal white stone to be the same instance as the initial one.

This failure of identity is problematic if, for example, later in the struct, it is necessary to determine whether the original stone is also adjacent to some other stone as well. This cannot be handled reliably because the *same* process corresponding to the

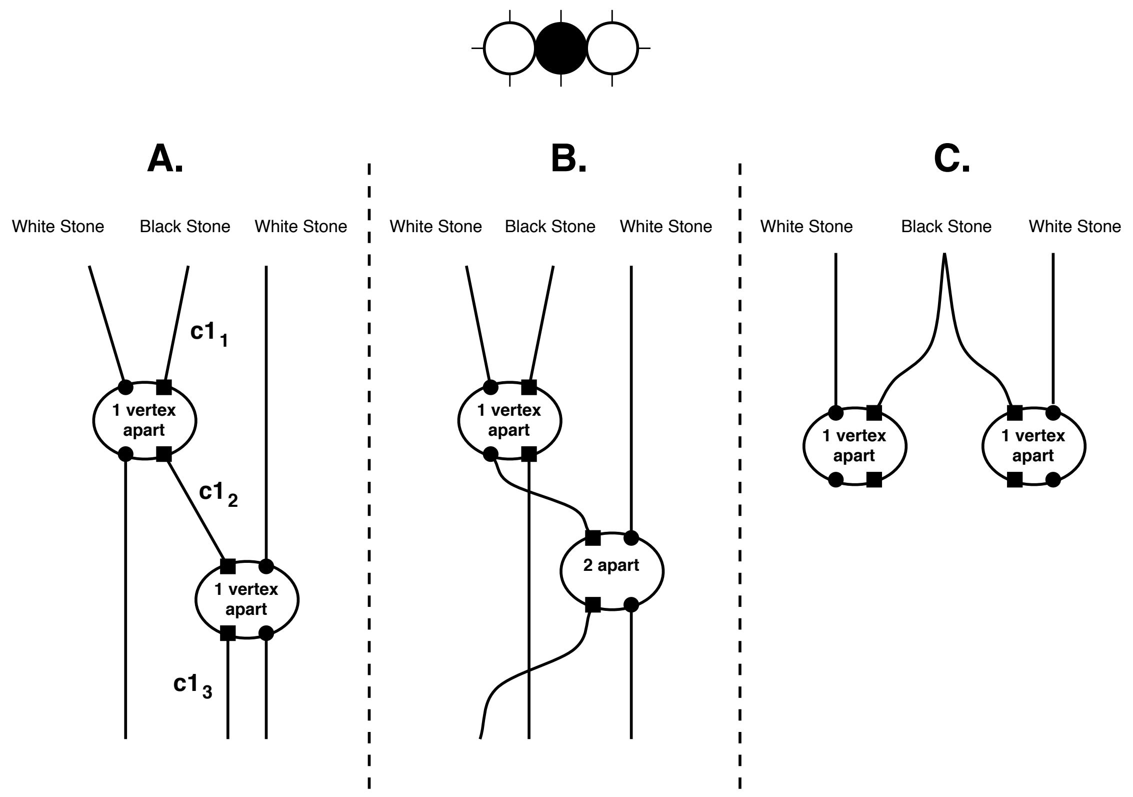


Figure 4.5: Three dubious attempts to represent the spatial aspect of the pictured Go position. A: Because each event generates *new primal class elements*, it is difficult to say formally whether $c1_3$ represents the *same* black stone as $c1_1$. B: This approach is highly numeric, and also suffers from the same problem as A. C: The same primal class cannot simultaneously interact with two events; this is forbidden by the model.

original stone cannot interact with two other stones at once.

Besides the above, this approach to modeling space has a very numeric flavour: it becomes tempting to introduce primitives like those in part B. of Figure 4.5 that actually *quantify* space. While such primitives are not explicitly forbidden by the model, the reintroduction of numbers via the back door into a non-numeric formalism is dubious at best. One should not struggle against one's model, but rather let its logic guide one's thinking.

4.2.3 Representing spatial relations via overlapping classes

ETS version 5 contains a powerful capability to overlap classes. When single-level classes have events (primitives) in common, it means they are influencing each other and sharing some formative history, without being completely transformed by the interaction. This is a very natural way to handle physical adjacency: proximate processes affect each other in certain limited ways.

In modelling Go, I have chosen to describe the physical relationship between immediately adjacent stones in this way. My model does not, however, depict stones that are not immediate neighbours as overlapping. Why? In Go it is natural to transform physical distance into something akin to Levenshtein distance[1]. "How far apart are two stones in actual distance?" is a less important question than "How many moves does it take to connect these stones?"

4.2.4 From stones to structs

With the previous considerations in mind, and with a desire to keep things simple[2], I decided that the components (primitives and primal classes) of the level 0 classes should be the simple elements of the player's awareness of a stone or empty vertex.

Each vertex has a "colour" (black, white, empty, edge) which is fixed regardless the status of adjoining vertices, and as such the colour of each vertex is inaccessible to the adjacent vertex class elements[3]. Formally, this means that each stone and

[1] "Levenshtein", or "edit" distance refers to the number of operations it takes to transform one object (usually strings) into another

[2] An earlier version of my model had 60 different primitives and 8 different primal classes

[3] The case in which a stone being laid leads to neighbouring stones being captured should not properly be thought of as the new stone interacting with the existing stones and changing their colour. Rather, the newly-placed stone causes a transformative event to occur: the old stone processes ends and new ones (corresponding to the empty vertices) begin.

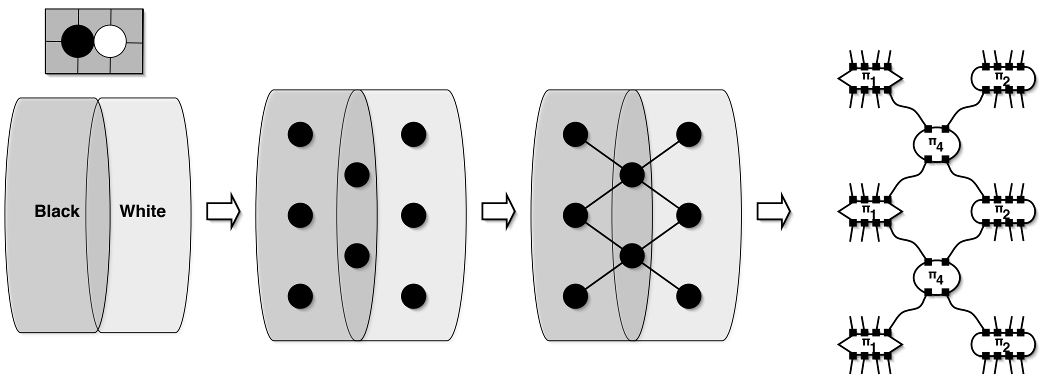


Figure 4.6: Logical progression. From left to right: 1. Go stones are class elements, and neighbouring stones overlap. 2. Each stone must have some shared primitives and some only belonging to itself. 3. These primitives need to be structurally arranged. 4. The resulting struct.

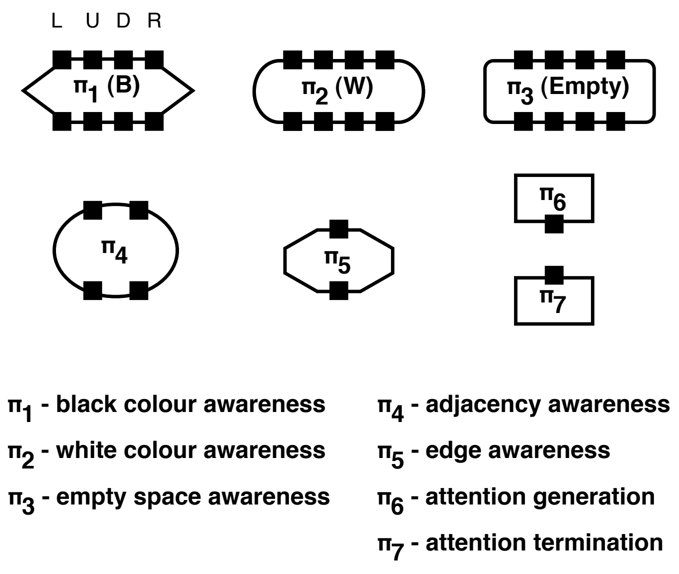
vertex class element should contain primitives related to the player's awareness of a particular vertex's colour, and these primitives should not be involved in overlap with adjacent vertex processes.

In what sense is a stone "overlapping" with its neighbor? Conceptually, two adjacent stones share a "sense of proximity". Go players think of nearby opposing stones as putting "pressure" on each other. Nearby allied stones are "working together". These qualitative phrases describe some kind of interaction between stones. From a purely literal point of view, two adjacent vertices "share" the line that connects them.

These two concerns motivated the creation of two main types of primitives. One type (of which there are four) corresponds to the player being aware of the *colour* of a particular stone (or vertex). The other main type represents the player's awareness of the physical proximity of two stones/vertices. Combinations of these two types of primitives are assembled to form stone and vertex classes, and each class element shares some adjacency primitives with its neighbours.

4.3 Primitives and the primal class

This section presents the basic units of ETS as they are employed to model Go. Defined here are one primal class and seven primitives, which together capture the very basic elements of the game. Figure 4.7 shows the seven primitives I have chosen; each primitive is discussed in one of the following subsections.



L U D R
π_1 (B)
π_2 (W)
π_3 (Empty)
π_4
π_5
π_6
π_7
π_1 - black colour awareness
π_2 - white colour awareness
π_3 - empty space awareness
π_4 - adjacency awareness
π_5 - edge awareness
π_6 - attention generation
π_7 - attention termination

Figure 4.7: The seven Go primitives.

4.3.1 The primal class

The only primal class in this model of Go is the class of *local space awareness processes*. A single such process represents a player's continued attention to *one half* of one of the many short "lines" that connect two vertices. Why divide a line connecting two adjacent vertices into two parts? So that the player may be aware of only once vertex at a time, independent of what is adjacent. This also simplifies handling the edge of the board (discussed in Section 4.3.4).

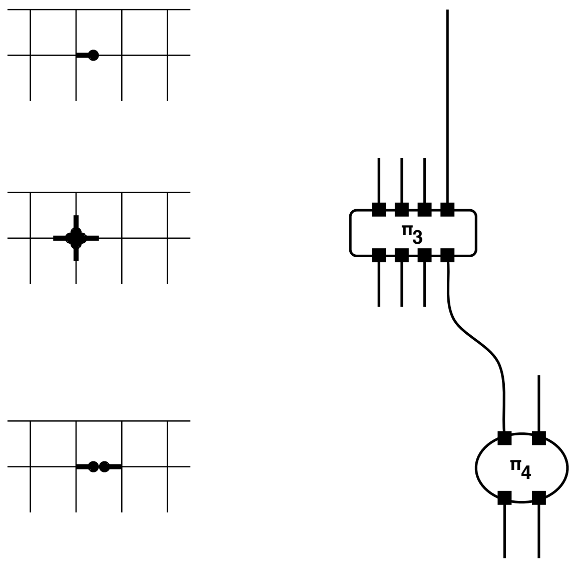


Figure 4.8: The primal class corresponds to the player's awareness of the part of one line nearest a given vertex.

Local space awareness processes are short-lived, fast-running processes that interact frequently with neighbouring processes. Because the Go board is organized into a regular grid, each local space awareness process only has a few immediate neighbours with which to interact.

When four local space awareness processes interact, the resulting event corresponds to the player's awareness of the colour-state (black, white, or empty) of that *part of the board*—this should not be confused with the player's awareness of a coloured *stone* or empty *vertex* because the concept of an entire vertex or stone is represented as a first-level class, and as such is not present as a single element in the struct.

When two local space awareness processes interact, the resulting event is the player's recognition that the lines these two processes represent are physically connected to each other in the space between two vertices.

4.3.2 Colour primitives

The three colour-state awareness primitives, numbered π_1, π_2, and π_3, represent the player noticing the current state of a vertex: for example, π_1 represents the player's awareness that a particular location contains something black. It is not proper to call this primitive the detection of a black *stone*, as colour is only one aspect of a stone.

A colour-state awareness primitive's four initial processes correspond to four local space awareness processes coming together, meaning that the player's awareness is momentarily focused on the state-of-affairs at the location where these four lines meet. This event also generates four new local space awareness processes and sends them off "in search" of others to interact with.

In keeping with my overall treatment of "emptiness" as simply the third state that a particular location might contain, π_3 represents the player noticing that a particular location is "empty", that is, it contains no black or white. Just as "blackness" or "whiteness" is only one property of a stone, "emptiness" is only one property of an unoccupied vertex.

These primitives' four initial and four terminal sites are associated with the four cardinal directions: *left, up, down, right.* This ordering is important because a consistent scheme is required for relating vertices in a way that reflects the rigid geometry of the board.

4.3.3 Adjacency primitive

The adjacency awareness primitive, π_4, represents the player noticing that two short lines are connected to one another. This is an important step in the larger recognition

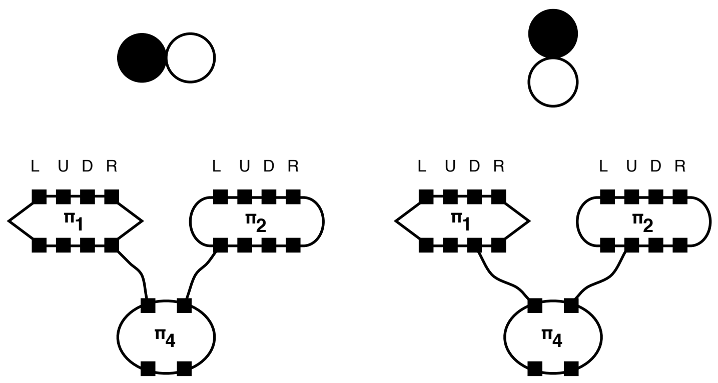


Figure 4.9: *Left-right* vs. *up-down* adjacency depends on how the colour primitives are attached to the adjacency primitive.

of the adjacency relationship between vertices.

The grid arrangement of the Go board dictates how adjacency primitives may be attached to colour primitives. The same primitive is used in *left-right* and *up-down* connections, since all the primitive registers is simple adjacency of two points. The left-hand sites always match with *right* or *down*, and the right-hand sites always mach with *left* or *up*.

Like the colour primitives, the adjacency primitive restarts two primal classes and sends them "in search of" a nearby vertex.

When these first four primitives are combined to form structs, the result is a lattice, potentially connecting all vertices on the board. As such, these four are the most important primitives. The three remaining primitives deal with setting spatial and temporal boundaries on this lattice.

4.3.4 Edge primitive

Primitive π_5 is a special *edge* primitive. This primitive plays a role similar to that of the colour awareness primitives, except that the "state" it captures is that of

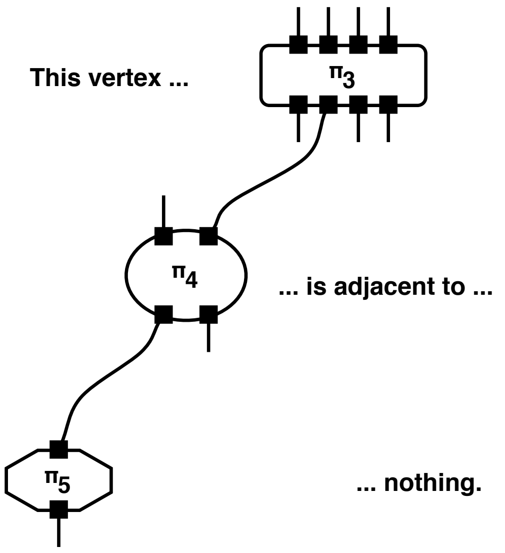


Figure 4.10: This vertex is adjacent to nothing (i.e. it is on the edge of the board).

being "off the board". Figure 4.10 depicts this. Unlike the three colour awareness primitives, the edge primitive has only one initial and one terminal. Semantically, an adjacency primitive connected to an edge primitive says "this vertex is adjacent to... nothing".

An earlier version of this Go model treated the edge of the board as if it was a fourth vertex-state and represented it using primitives similar to the colour awareness primitives, treating the board as if it was surrounded by grey stones. However, this proved undesirable because it merely displaced the edge: the question of what was adjacent to the grey stones still needed to be answered. I also considered a three-site version of the grey stone primitive (as it was only adjacent to one board vertex and two other grey stones), but in order for this to be meaningful there would have had to have been 4 slightly different primitives: one for each side of the board. Instead I chose the simpler π_5.

4.3.5 Attention generation primitives

The last two primitives, designated π_6 and π_7, were introduced to increase representational flexibility by allowing only part of the Go board to be represented at any time. Primitive π_6 creates a new local space awareness process. Broadly speaking, this corresponds to the player "turning his attention" to a particular part of the board. Primitive π_7 has the opposite effect: it simply absorbs a local space awareness process, and corresponds to the player's attention departing from this part of the board. A more complete view of these primitives might also include initials and terminals corresponding to "abstract attentional resources", but I have omitted them for simplicity. As it makes more sense to discuss this in the context of complete structs, I will return to this in more detail in Section 4.4.2.

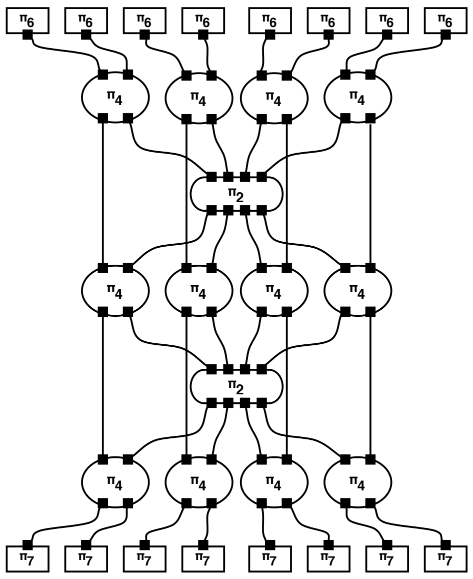


Figure 4.11: The player briefly considers a white stone. This is a complete (if small) example of a Go struct.

4.4 Structs

This section presents Go structs, which are composed of the seven primitives outlined in the previous section. Figures 4.11 and 4.12 each depict a complete struct, and it is apparent that these structs have regular alternating layers of colour and adjacency primitives. This configuration is a product of the rules for how structs are generated from board positions, which in turn depends on the assumptions made about the meaning of the various primitives. The following subsections explore these and other properties of Go structs.

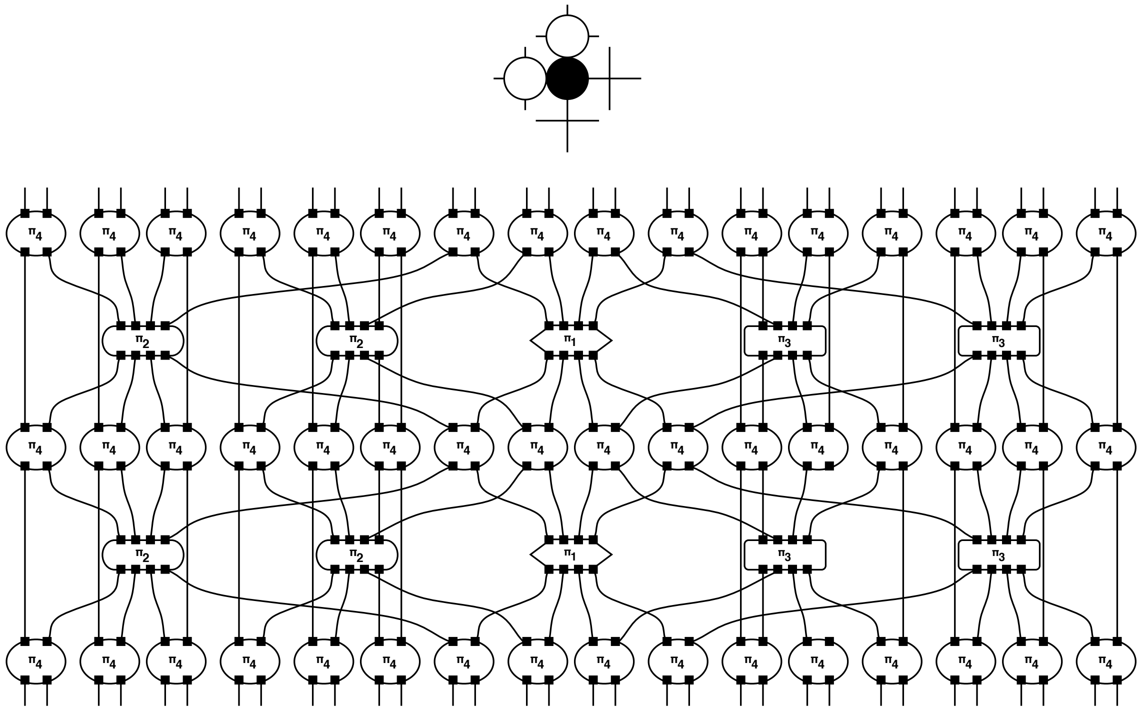


Figure 4.12: An (abstract) struct corresponding to the pictured board position.

4.4.1 How structs are recorded

As we recall from Section 3.3.2, structs are empirical, but, as stressed in Section 4.1, Go is largely a *mental* phenomenon. As a result, sensors cannot be used to record events directly (as in for example [47]); measurements have to be postulated. The approach used here is similar to how the same issue is handled in modeling Fairy Tales for document retrieval [51]: the events used to build structs are mental, but as measuring "inside the listener's head" is impossible, it is postulated that the listener will react in certain predictable ways to the story as it is read. This means that the story itself can be parsed and used to build a struct based on the expected corresponding mental events. In the case of Go, it is possible via some simple assumptions to build structs based on the recorded moves of a game.

The Go primitives and primal class all represent some aspect of the player's *awareness*. For purposes of this model, I have made this assumption about how awareness functions: each type of observation (adjacency, colour, etc) needs to be refreshed often or it will be "forgotten". The result is structs with a lattice-like structure produced by the continuously-updated awareness. The strict alternation of layers of colour and adjacency primitives represents a continuous oscillation of the local awareness processes between the center of their respective vertices and the frontier between vertices as the player focuses on that part of the board.

Because there is only one primal class in this representation scheme it is true that any primitive could legally be connected to any other, giving rise to a struct like the ones in Figure 4.13. However, simply because a struct is possible does not mean that it should be created: structs must correspond to reality since they represent actual observed sequences of events, and so structs are limited by the kinds of observations

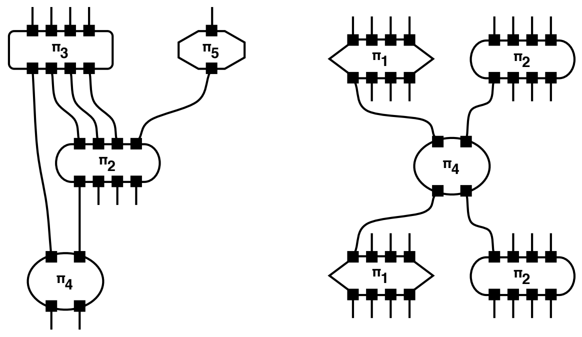


Figure 4.13: Left: This struct does not violate any purely formal rules, but it corresponds to a nonsense state-of-affairs, i.e., a situation that can never be observed on a Go board. Right: a more subtle nonsense struct showing two stones to the left of each other, a physical impossibility.

that can be made, which in turn are limited by the actual configuration of the Go board.

If, in the future, a Go-playing engine were implemented that used ETS to model the game, training structs could be generated by automatically parsing game records, most likely in the form of the ubiquitous Smart Game Format (.sgf) file [52]. Since the model allows for different structs to be created depending on what parts of the board the player chooses to pay attention to, some rules for how this would be accomplished need to be devised. Although it would be possible to ask Go players to report on what they are paying attention to, in much the same way that the study [40] operated, some simple initial heuristics could be defined. For example, one could specify that the player always focuses on the most recently placed stone and its neighbourhood, that it has an upper limit of 50 local state awareness processes, and that it remains focused on the same location for 3 iterations before the next stone is played. Longer term, the program should learn where attention is needed as more sophisticated classes are assembled.

4.4.2 Temporal aspects of Go structs

The structs given in Figures 4.11 and 4.12 are both “snapshots” of part of the board between moves, and so the only obvious temporal aspect is the continuous refresh of the attention processes and primitives: time is passing while the player considers the position on the board[4]. This is one of three temporal aspects of the game that I have modelled.

The second and most obvious temporal aspect of Go is a sequence of moves, resulting in stones being added to or removed from the board. At the level of structs, there is no single event corresponding to the placing or removal of a stone, since *stones* are not a struct-level phenomenon, but rather a level 0 class. When a stone is played, the struct registers a change in the colour of a particular location (Figure 4.14). Likewise, when stones are removed, the struct registers a corresponding change to the empty state. Is it plausible to have the stone simply “appear” without the player seeing his opponent place it? I chose to model it this way for simplicity, but this kind of sudden appearance of a stone is exactly what a player experiences when playing a game on computer-generated board instead of a real one, so it does not seem problematic.

The third temporal aspect is the player’s shifting attention: this model of Go does not require that the entire board be represented at once. There are two reasons for this: first, the observation that a player need not remember the location of every stone on the board simultaneously as a “quick glance” at the board is sufficient to refresh his memory. Second, the ability to limit which parts of the board are under scrutiny and retain the same representational capabilities could be useful in creating a Go-playing engine for which resources (memory, processing power) are scarce.

[4]I have deliberately not stated that each process runs for some specific (numeric) time interval: to do so is against the non-numeric nature of ETS.

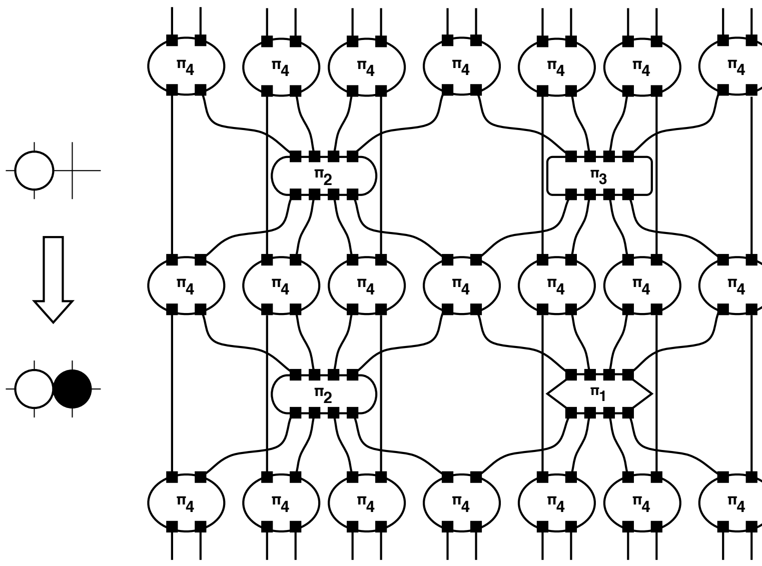


Figure 4.14: An (abstract) struct showing a black stone being placed next to the white stone.

There are two immediate consequences of this property that must be taken into account should this representation scheme be incorporated into a Go-playing engine. First, obviously, that engine must keep the entire board encoded in some non-ETS format in memory, and this must be easily accessible to the program, just as the real board is easily accessible to the player. Second, it must be the case that the order in which the player's attention sweeps over the various vertices must not affect classification. This property of the model will be discussed in Section 5.1.3.

This principle of short-run awareness processes and continually refreshing observational events also dictates how the structs are shaped when the player's attention shifts to a different part of the board. When this occurs, the attention focused on the previous area is quickly terminated and the ongoing processes are absorbed by several attention termination primitives. Figure 4.15 gives an example of the player's attention sweeping across a group of stones. This figure is meant to be illustrative and not realistic, as the player's attention is tightly focused on one stone at a time.

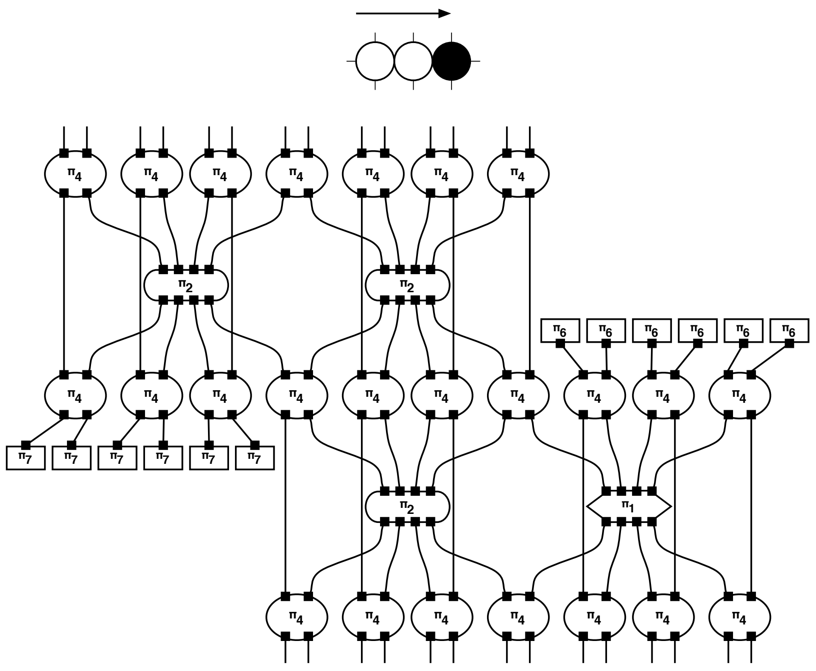


Figure 4.15: Attention shifts across the board.

Although this bears some resemblance to how a beginner approaches the game, a Go-playing engine would benefit from viewing a larger region at once. This figure also illustrates something that has been implicit in previous figures that show structs: at the edge of the player's attention, the second site on the adjacency awareness primitive is connected directly to the next such primitive in the struct, and not to an intervening colour awareness primitive.

4.4.3 Spatial aspects of Go structs

A consequence of introducing temporal information when modeling the Go board (which itself is two-dimensional) is that the resulting representation has three dimensions. An unfortunate side effect of this is that drawing structs in two dimensions becomes cumbersome. This provides no extra challenge for software designed to manipulate structs, but does make visualization more difficult for humans[5], and is the principal reason why most of the examples in this chapter are small. Fortunately, when drawing higher-level classes that span larger structs, some notational simplification is possible (see Sections 4.6 and 4.7).

4.5 Level 0 classes

Section 4.2 described the main aim of this representation of Go: to model the player's awareness of Go stones and board locations as overlapping classes. This section describes those classes.

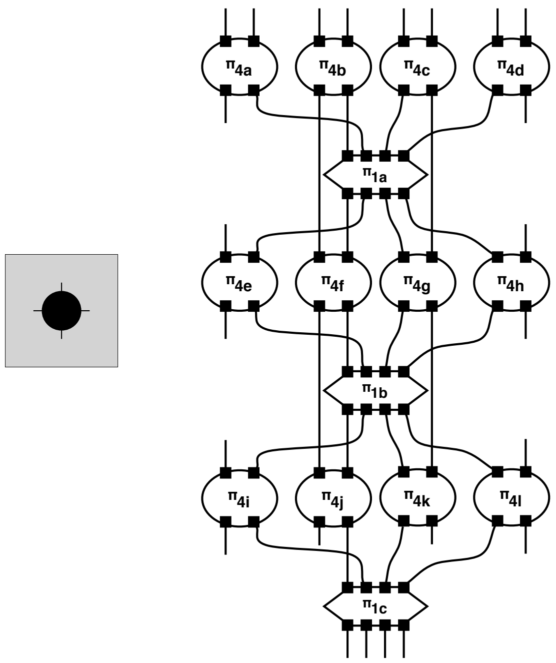


Figure 4.16: This struct, representing the player's awareness of a single black stone, is an element of the level 0 black stone class.

[5]This also highlights the need for new ETS visualization tools to be developed. If a 3-dimensional phenomenon were to be modelled, the situation would be even worse: 4-dimensional structs.

There are sixteen one-level classes, divided into three broad categories. The first category contains three classes corresponding to black stones, white stones, and empty vertices. The second category contains four classes of short-running processes corresponding to the placement or removal of stones from the board. The third category, which is the largest and most complex, contains nine *proximity classes* that capture the spatial relationships between pairs of vertices in a detailed way.

Table 4.1: The level 0 class setting

Num	Name	Symbol	Num	Name	Symbol
$\mathfrak{C}_1$	Black stone	[●]	$\mathfrak{C}_2$	White stone	[○]
$\mathfrak{C}_3$	Empty vertex	[+]			
$\mathfrak{C}_4$	Black play	[↓●]	$\mathfrak{C}_5$	Black removal	[ⓧ]
$\mathfrak{C}_6$	White play	[↓○]	$\mathfrak{C}_7$	White removal	[⊗]
$\mathfrak{C}_8$	Adjacent blacks	[●●]	$\mathfrak{C}_9$	Adjacent whites	[○○]
$\mathfrak{C}_{10}$	Adjacent empties	[++]	$\mathfrak{C}_{11}$	Black beside white	[●○]
$\mathfrak{C}_{12}$	Black beside empty	[●+]	$\mathfrak{C}_{13}$	White beside empty	[○+]
$\mathfrak{C}_{14}$	Black beside edge	[●|]	$\mathfrak{C}_{15}$	White beside edge	[○|]
$\mathfrak{C}_{16}$	Empty beside edge	[⊣]			

The constraints that generate each class element operate within the context of the rules for how structs are assembled, as outlined in the previous sections. Because structs are predictable, class constraints are allowed to be more natural—any system for learning these classes would not need to introduce a set of special case rules to carefully rule out a number of impossible circumstances.

For example, the constraint $\mathrm{Con}_6^{\mathrm{C2}}$ (Figure 4.18) which describes one way that adjacency primitive may be attached to a colour primitive, only specifies one of the two initials on the adjacency primitive. The possible ways that the second primal class might be attached are limited by the ways in which the struct can be validly constructed (Figure 4.22).

4.5.1 Stones and vertices

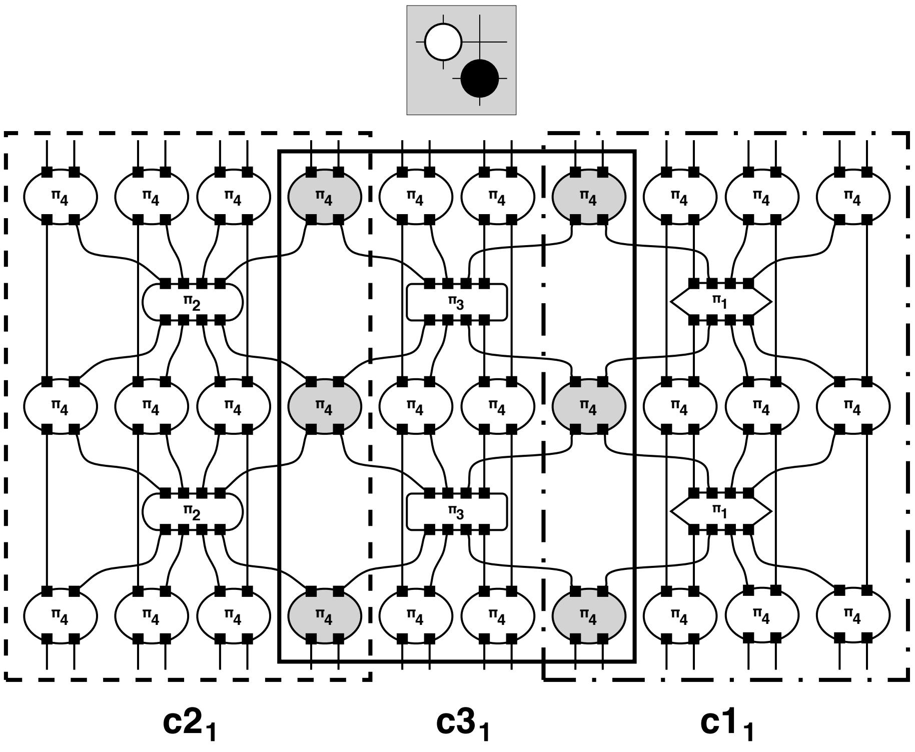


Figure 4.17: An example element of one of each of the stone/vertex classes, representing a player's awareness of the above pictured board state. Primitives shared between two overlapping class elements are shown in grey.

Figure 4.17 shows elements of the black stone class, the white stone class, and the empty-vertex class. These three classes, denoted $\mathfrak{C}_1$, $\mathfrak{C}_2$, $\mathfrak{C}_3$, are structurally very similar. This is appropriate in light of the elements being represented: black and white stones differ only in colour, so it is natural that they have similar class descriptions. Formally, empty vertices are treated as having a third, "empty" colour.

Figure 4.18 shows the constraint set for the white stone class. The black stone and empty vertex class constraints are structurally the same: simply substitute a π_1 or

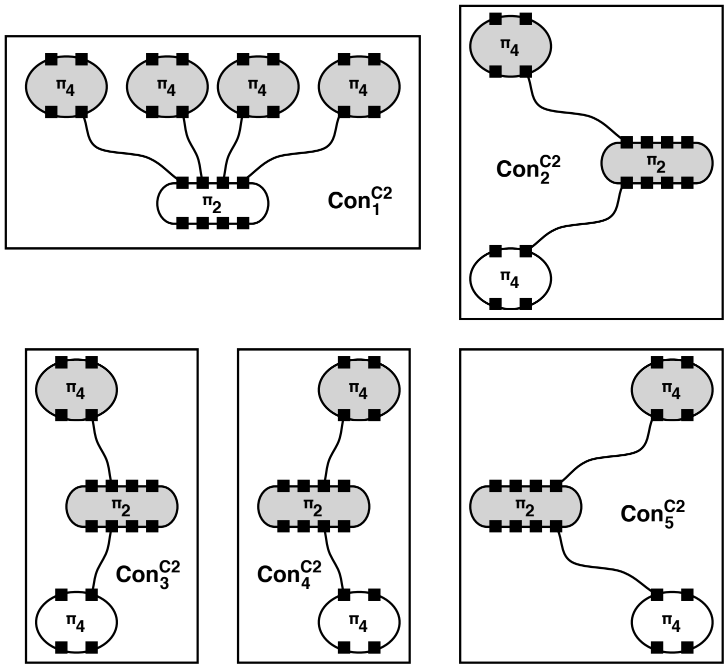


Figure 4.18: Constraints for the white stone class. The primitives in grey are the context: these primitives must be present in the working struct in order for the constraint to be applicable.

a π_3 for the π_2 primitives. The five constraints for this class have a context part, shown in grey on the figure, and the primitives in the context must be present in the working struct in order for the constraint to be applicable. I chose to include such contexts because they allow for finer control in specifying the situations in which each constraint can be applied; because of the small number of primitives and single primal class, this kind of fine control is desirable. By contrast, in [47], the variety of primitives and primal classes limited which constraints could be applied at any point in a class element's formation, so this kind of finer control was not needed.

The rules governing the application of these constraints are very simple: at each step, apply any constraint that is applicable to the working struct. The next section is an extended example of how this works in practice.

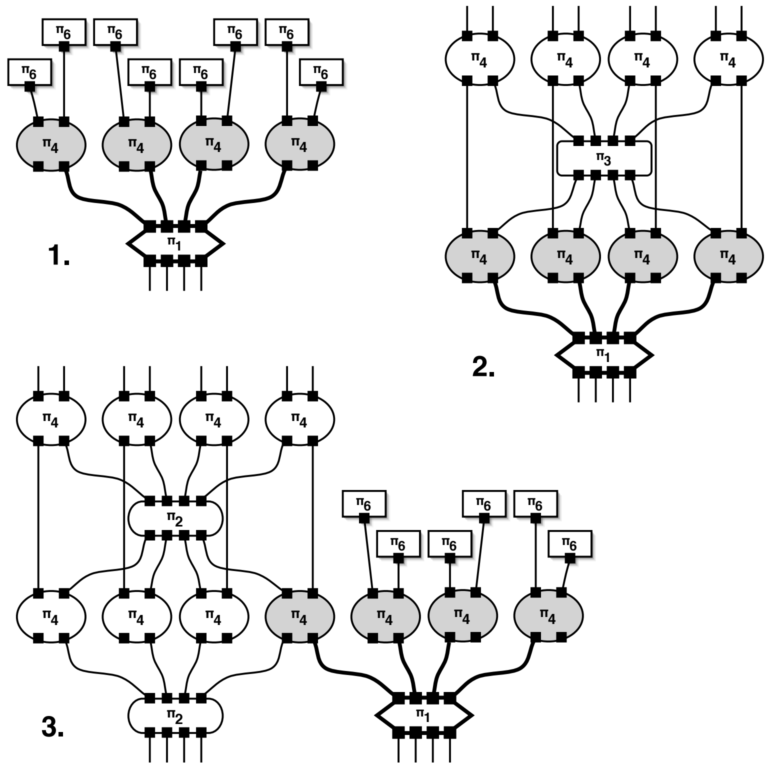


Figure 4.19: Three examples of a class element's initial formation. The primitive attached by the application of Con_1^{C1} is shown in bold, and the context is shown in grey, and is provided by the environment. All other primitives are not part of this new class element. 1. attention jumps to the black stone. 2. the black stone is placed on the board. 3. attention shifts to the black stone from a neighbouring stone.

4.5.2 Example: the generation of a class element

Since the application of a constraint represents the first step in the generation of a new class element, and since all of the constraints have contexts, it is clear that the context used by this first constraint must be supplied by the *environment.* There are a few different ways in which this initial context can be produced. Figure 4.19 shows three of them.

Once a new class element has been created, the generating system operates indefinitely until the player's attention shifts away from the vertex, or the state of the

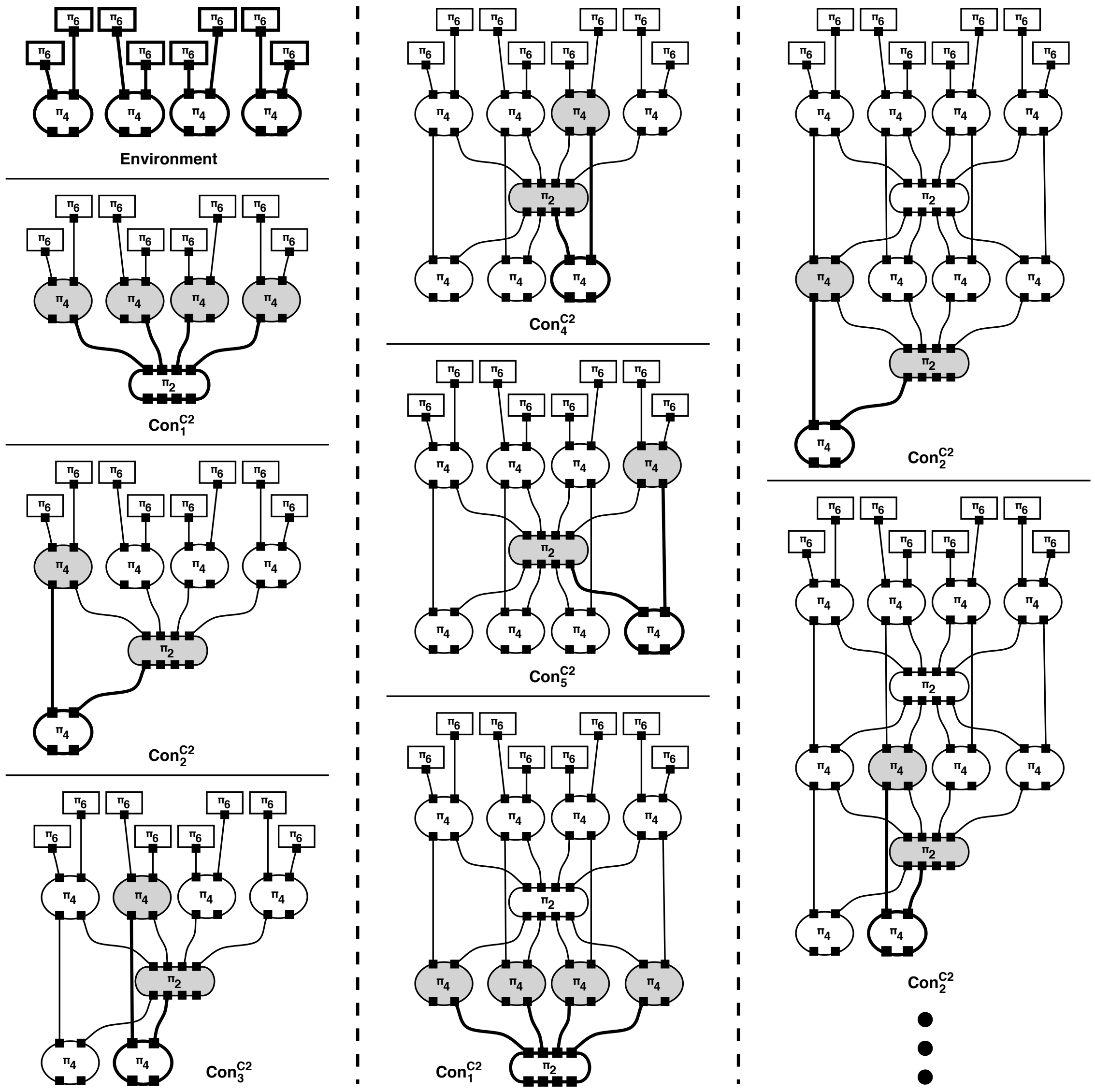


Figure 4.20: Assembly of an element of class $\mathfrak{C}_2$, a white stone. Newly added primitives and primal classes are shown in bold, contexts are in grey, and the particular constraint applied is indicated at each step.

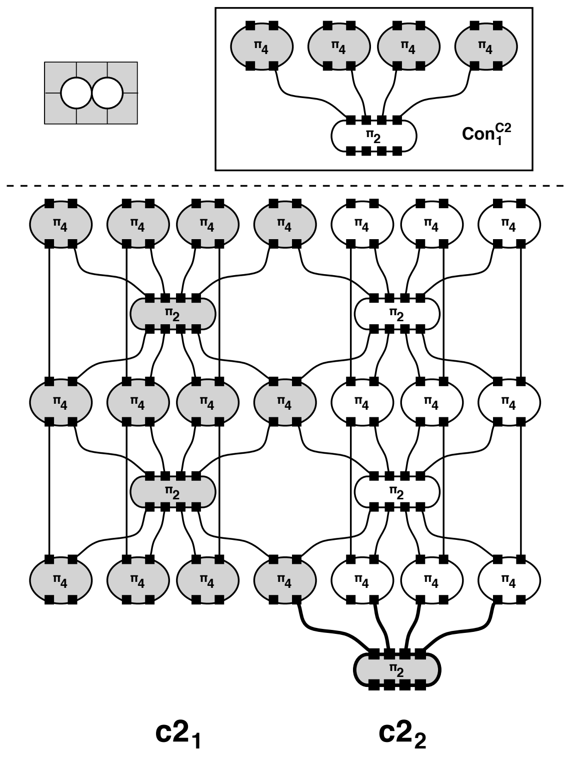


Figure 4.21: Struct continuation must attach to the working struct. Above the dashed line: the board position represented here (two adjacent white stones), and the constraint being applied. Below: the two connected class elements, with one of them shown in grey. Note that the middle primitives are shared between the two classes. The newly-added primitive, which would be a valid continuation of the right-hand class, is not a valid continuation of the class on the left, as the context of the constraint being used is not within the working struct.

vertex changes via a stone play or removal.

The rule governing the constraints at each step is simple: apply any that can be applied. However, in practice, the following will be the case. When $\mathrm{Con}_1^{\mathrm{C2}}$ can be applied, none of the others will be applicable. When any of the others are applicable, $\mathrm{Con}_1^{\mathrm{C2}}$ will not be. Thus, the propagation of a class element will alternate between the application of a $\mathrm{Con}_1^{\mathrm{C2}}$, and then applications of the other four constraints (in any order), followed by another $\mathrm{Con}_1^{\mathrm{C2}}$, followed by the other four, etc. Figure 4.20 shows the complete assembly of a white stone class element. Figures 4.21 and 4.22

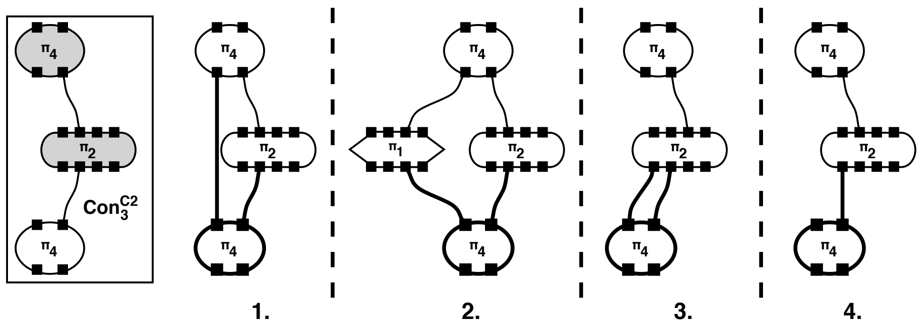


Figure 4.22: The constraint on the left only specifies one of π_4's initials. 1. and 2. are valid ways of *applying* the constraint. 3. is not valid, not because the constraint forbids it, but because this struct is never observed. 4. is structurally valid but does not satisfy the constraint.

illustrate the applications of certain constraints in more detail.

What is the environment that provides the initial pieces of the new class element? In some cases, as in part 2. of Figure 4.19, the environment is in fact another class element. In other cases, such as part 1. of the same figure, the environment is an external component, in this case governing the player's attention to the specific part of the board. Part 3. of the figure is a combination of the two.

4.5.3 Play and capture

Table 4.2: The level 0 play and capture classes.

Num	Name	Symbol	Num	Name	Symbol
$\mathfrak{C}_4$	Black play	[●]	$\mathfrak{C}_5$	Black removal	[⨂]
$\mathfrak{C}_6$	White play	[○]	$\mathfrak{C}_7$	White removal	[⊗]

There are four level 0 classes describing placement and removal of stones, denoted $\mathfrak{C}_4$, $\mathfrak{C}_5$, $\mathfrak{C}_6$, and $\mathfrak{C}_7$. Each class produces short-running processes that terminate once information about the colour-state of its associated location has been transformed.

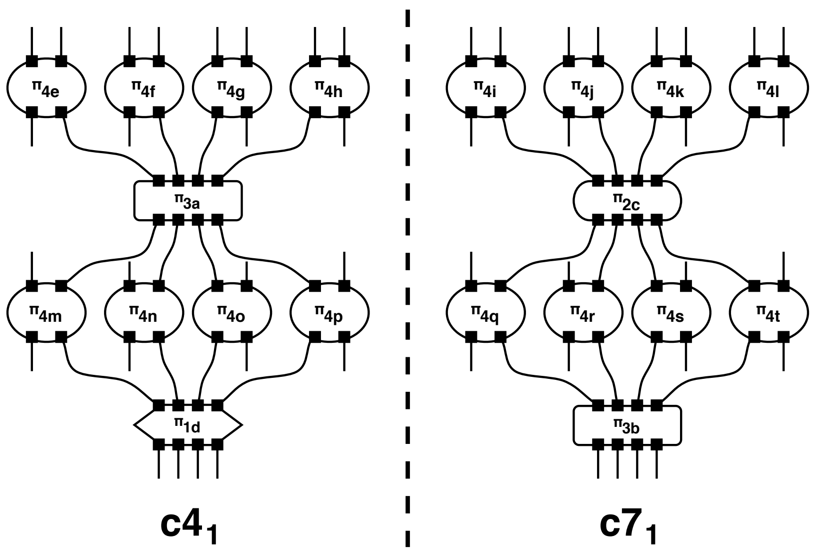


Figure 4.23: Left: an element from $\mathfrak{C}_4$, the black play class. Right: an element from $\mathfrak{C}_7$, the white stone removal class.

Figure 4.23 shows an example of the black play class, and Figure 4.24 shows its constraints. As was the case in the previous section, the only difference between this class and the corresponding class for a white stone is use of π_1 primitives in the place of π_2's.

Figures 4.23 shows an example of a white stone being captured. This class is similar to the black stone capture class (replace π_2 with π_1). It is also very similar to the stone play class, except that the order of the colour awareness primitives is reversed. This reflects the symmetry of placing and removing stones.

Because of this symmetry, the generating systems for all four classes are analogous to one another. Figure 4.24 shows the constraints for $\mathfrak{C}_4$. The generating system for this class is as follows: at the first step, apply Con_1^{C4}, at the next four steps, apply one of Con_2^{C4}, Con_3^{C4}, Con_4^{C4}, Con_5^{C4}, and finally at the last step apply Con_6^{C4}. At this point, the class element is complete and the generating system terminates.

Figure 4.25 shows how an element of the white capture class overlaps the white stone

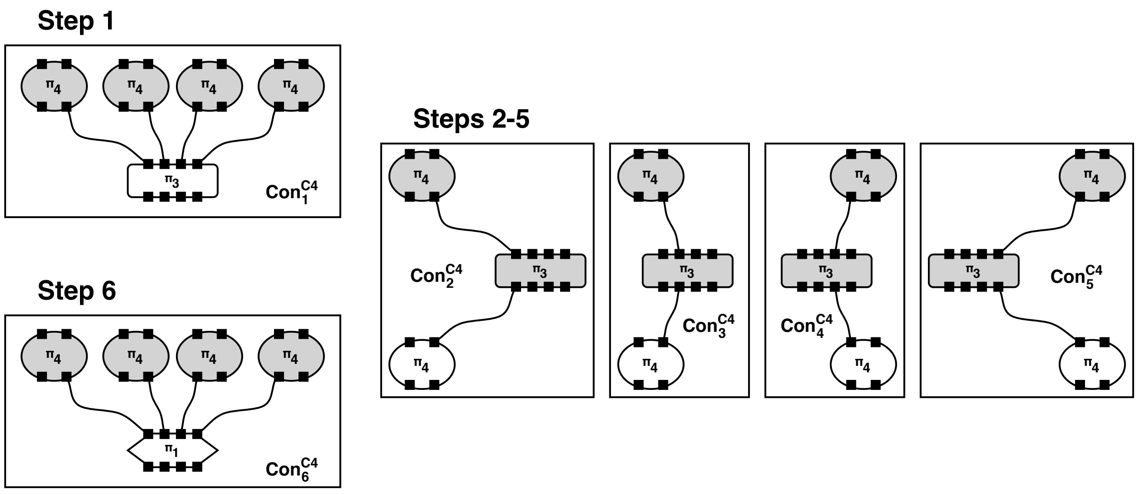


Figure 4.24: The constraints for $\mathfrak{C}_4$, the black play class. Context primitives (i.e., those that must be present in the working struct before the constraint may be applied) are shown in grey.

class element that precedes it, and the empty vertex class that follows. Because of its short and transformational nature, $\mathfrak{C}_7$ could conceivably be treated at the *next representational stage* as a primitive transformation instead of a primal class. In other words, it is possible that $\mathfrak{C}_4$, $\mathfrak{C}_5$, $\mathfrak{C}_6$, and $\mathfrak{C}_7$ are classes not of ordinary processes but rather of *transforms*.

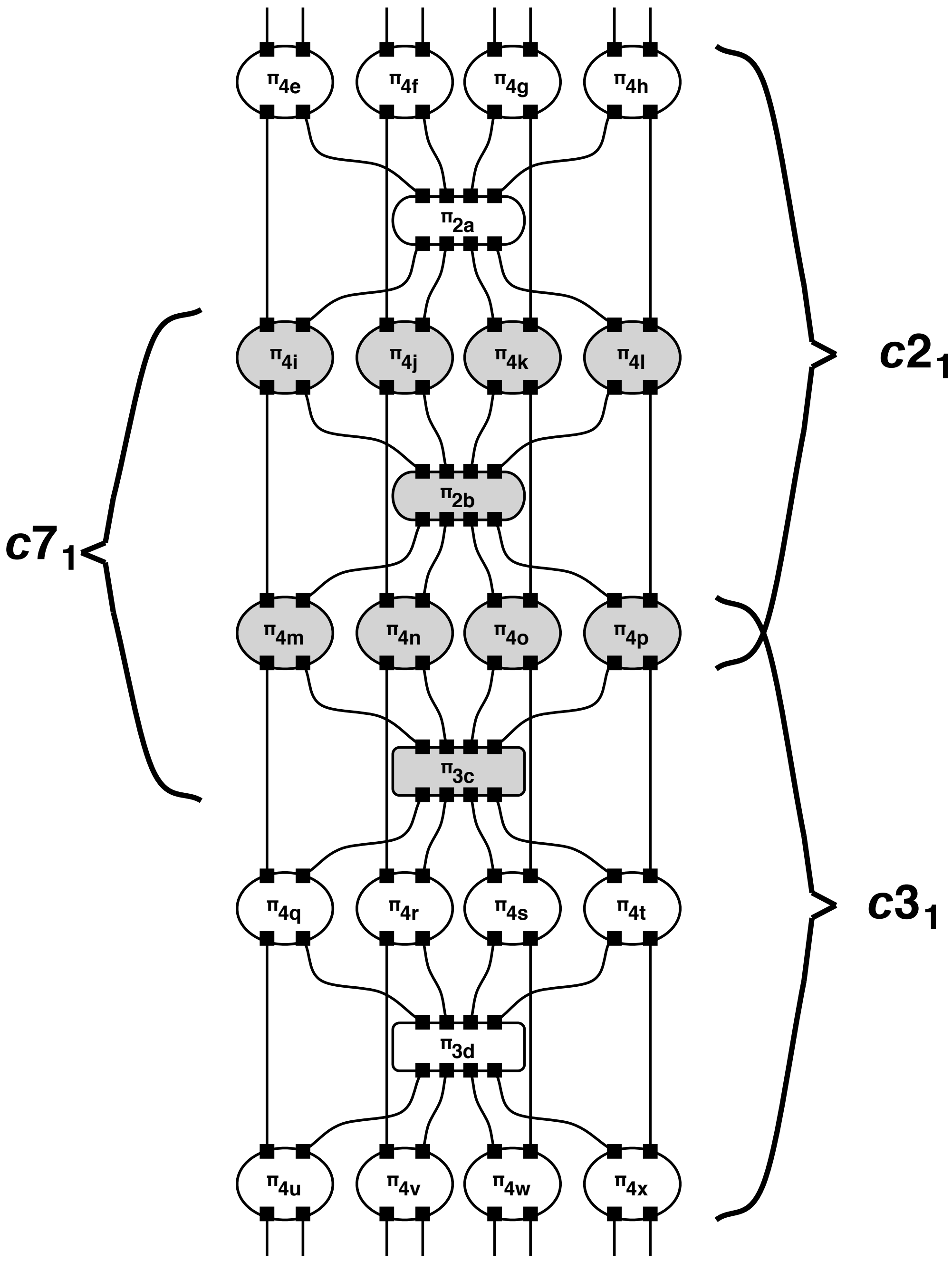


Figure 4.25: The capture of a white stone: $c7_1$, shown in grey, overlaps with $c2_1$, an element of the white stone class, and $c3_1$, an element of the empty vertex class.

4.5.4 Proximity

Level 0 proximity classes describe the spatial relationships between vertices in a more detailed way than the struct-level adjacency primal classes because they carry information about which kinds of stones are "touching" each other. There are nine of these classes, corresponding to all of the pairwise relationships between stones, empty vertices, and edges.

Table 4.3: The level 0 proximity classes

Num	Name	Symbol	Num	Name	Symbol
$\mathfrak{C}_8$	Adjacent blacks	[●●]	$\mathfrak{C}_9$	Adjacent whites	[○○]
$\mathfrak{C}_{10}$	Adjacent empties	[++]	$\mathfrak{C}_{11}$	Black beside white	[●○]
$\mathfrak{C}_{12}$	Black beside empty	[●+]	$\mathfrak{C}_{13}$	White beside empty	[○+]
$\mathfrak{C}_{14}$	Black beside edge	[●|]	$\mathfrak{C}_{15}$	White beside edge	[○|]
$\mathfrak{C}_{16}$	Empty beside edge	[⊣]			

Figure 4.26 shows a struct containing two of these classes. It is easy to see in the figure that the elements of these classes are a pair of colour awareness primitives alternating with a single adjacency awareness primitive. Indeed, all of the seven classes in this category produce elements of this form. The largest deviation from this pattern occurs in $\mathfrak{C}_{14}$, $\mathfrak{C}_{15}$ and $\mathfrak{C}_{16}$, in which one of the colour awareness primitives is replaced by (naturally) an edge awareness primitive.

Before discussing the generating systems for these classes it is worth noting that the relationships between all pairs of, for example, adjacent black and white stones, are all members of class $\mathfrak{C}_{11}$. A black stone to the left of a white stone and a black stone to the right of a white stone are different elements of $\mathfrak{C}_{11}$ (see Figure 4.28).

Figures 4.27 and 4.28 show the constraints for $\mathfrak{C}_{10}$ and $\mathfrak{C}_{11}$. $\mathfrak{C}_{10}$, the adjacent empty vertices class, has a simpler specification than $\mathfrak{C}_{11}$, because there are only two ways

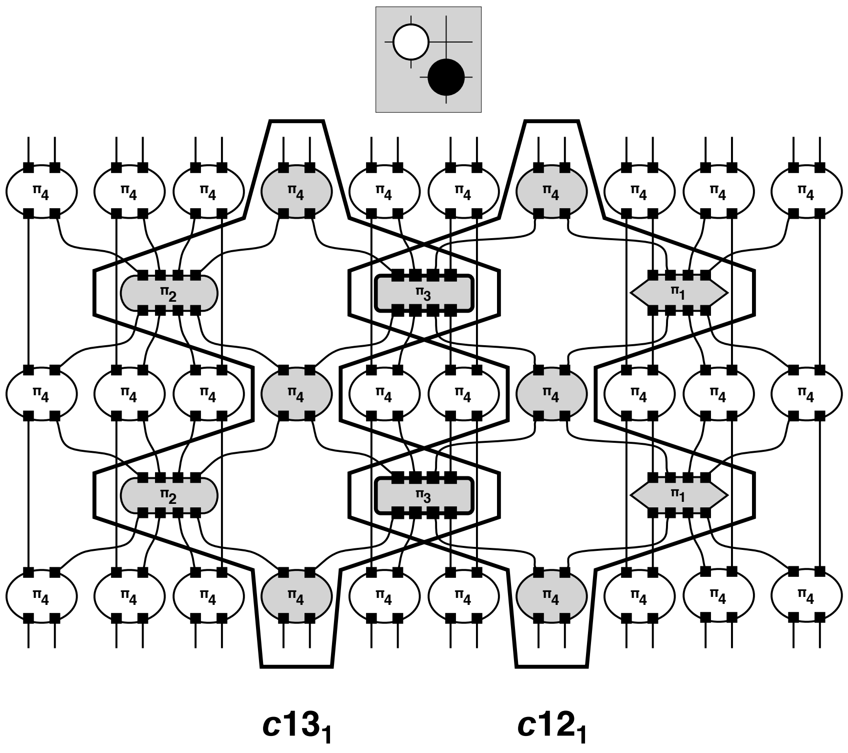


Figure 4.26: The struct from Figure 4.17, with two proximity class elements outlined. Grey primitives are those within either element. The primitives on which the class elements overlap are bold.

two stones/vertices of the same type can be spatially related: *left-right* and *up-down* (the same is the case for $\mathfrak{C}_8$ and $\mathfrak{C}_9$). Figure 4.27 shows the *left-right* constraints on the left and the *up-down* on the right. The generating system for this class is quite simple: at the first step, select either Con_1^{C10} or Con_2^{C10}; in subsequent steps, choose one of $\text{Con}_3^{C10}, \text{Con}_4^{C10}, \text{Con}_5^{C10}, \text{Con}_6^{C10}, \text{Con}_7^{C10}, \text{Con}_8^{C10}$. Note that for a given class element, only 3 of these last 6 will ever be applicable.

Classes $\mathfrak{C}_{11}$, $\mathfrak{C}_{12}$, $\mathfrak{C}_{13}$, $\mathfrak{C}_{14}$,$\mathfrak{C}_{15}$ and $\mathfrak{C}_{16}$ have more complex generating systems, with constraints analogous to those pictured in Figure 4.28. The dotted lines in the figure separate $\mathfrak{C}_{11}$'s constraints into two sets, and for convenience, the constraints for the four possible orientations are divided into four columns. The generating system first chooses one of the constraints for step one, and then in each subsequent step chooses

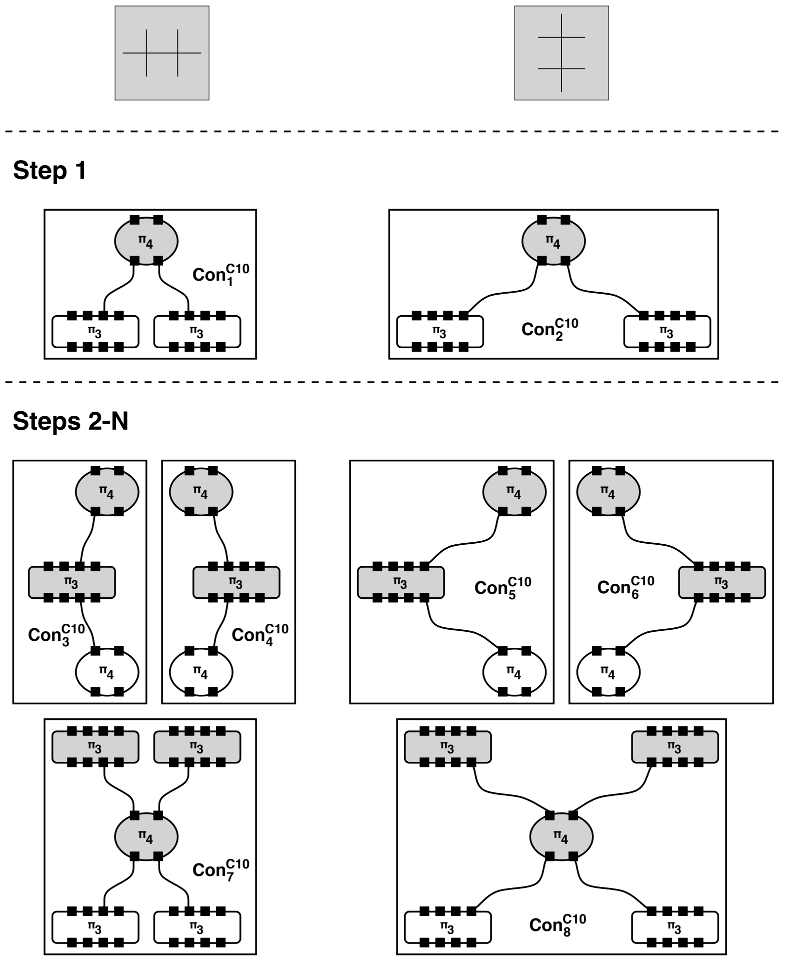


Figure 4.27: Constraints for $\mathfrak{C}_{10}$, the adjacent empty vertices class. Context primitives are shown in grey.

any constraint from the larger set that applies. As before, the choice at step one will dictate which constraints are applicable in the latter steps.

An important aspect of these class elements is that they contain constraints that are structurally identical to some of the constraints from other classes in the level 0 class setting. These shared constraints allow elements of different classes to *overlap* in more sophisticated ways than merely sharing one or two primitives. Overlap on constraints is an essential feature of any class setting, because the next-level classes will depend on such shared constraints. Figure 4.29 illustrates the constraint-overlap between elements of three level 0 classes. The simple struct in this example is part of the struct used in Figures 4.17 and 4.26, with, for the first time, all present class elements identified.

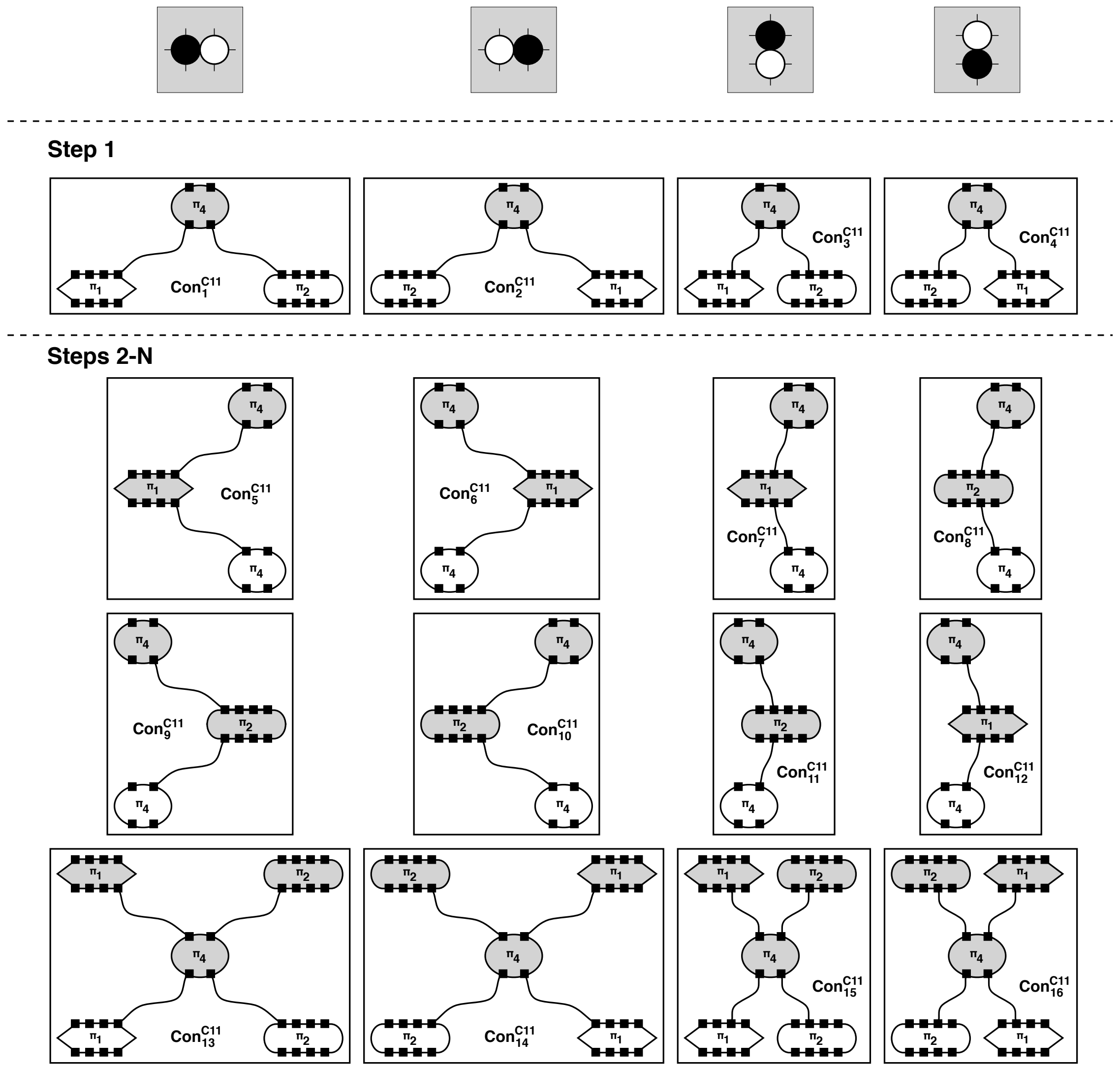


Figure 4.28: Constraints for $\mathfrak{C}_{11}$, the black adjacent to white class. Context primitives are shown in grey.

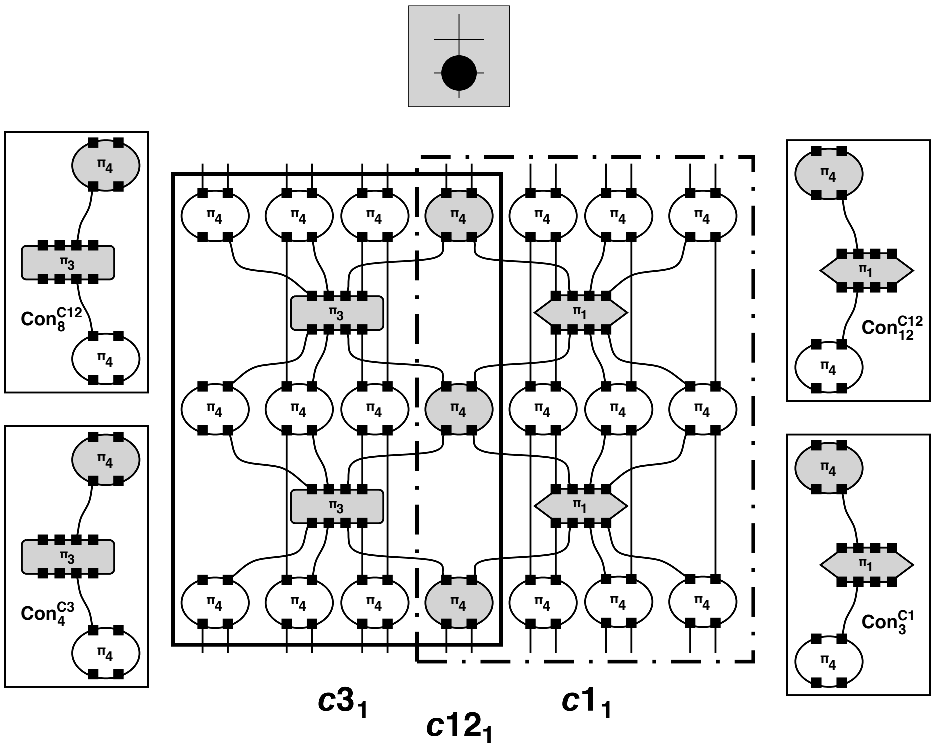


Figure 4.29: A struct containing three class elements: an empty vertex (solid box), a black stone (dashed box), and the proximity relationship between them (grey primitives). Element $c3_1$ shares constraints with $c12_1$; for reference, the constraints from the respective classes are shown on the left. Likewise, $c12_1$ and $c1_1$ have common constraints, depicted on the right side.

4.5.5 Example: a fully-classified struct

Figures 4.30, 4.31 and 4.32 show a larger example of the level 0 classes in action. Figure 4.31 also illustrates the difficulty of drawing (but not creating) structs to capture board positions for more than two or three vertices. There is no good way to draw this effectively three-dimensional struct in two dimensions, especially if one wants to indicate all of the overlapping class elements that it contains. Figure 4.32 gives a stylized depiction of the class elements in this struct, in the form of an overlap graph, which has class elements as its nodes. Two nodes are connected by an edge if they overlap on a shared constraint. This short-hand drawing of several ETS classes has temporal and other *information removed* should not be mistaken for an actual ETS struct: its only purpose is to aid in visualization.

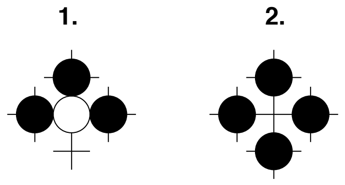


Figure 4.30: An *atari* position and subsequent capture. The representation of this board position and stone placement is shown in Figure 4.31.

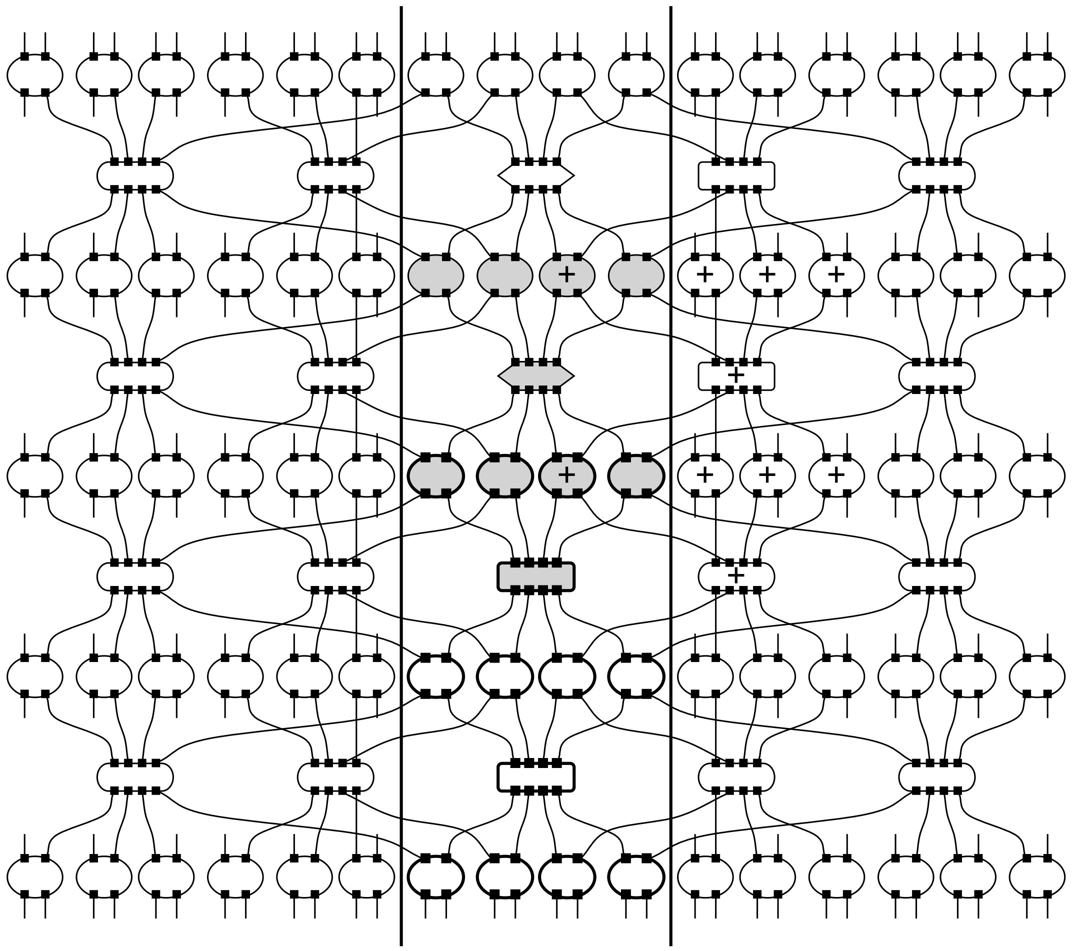


Figure 4.31: A struct corresponding to the board positions in Figure 4.30. Note that for simplicity, primitive numbers and some primal classes are suppressed. Only a few of the 17 class elements within this struct are identified. The vertical bars isolate the three class elements that correspond to the center vertex. The $\mathfrak{C}_7$ (white capture) element is shown in grey, the resulting $\mathfrak{C}_3$ (empty vertex class) element is bold, and primitives marked with a '+' are those that compose the element of $\mathfrak{C}_4$ (black play class) on the lowest vertex. Figure 4.32 shows an overlap graph of all 17 class elements.

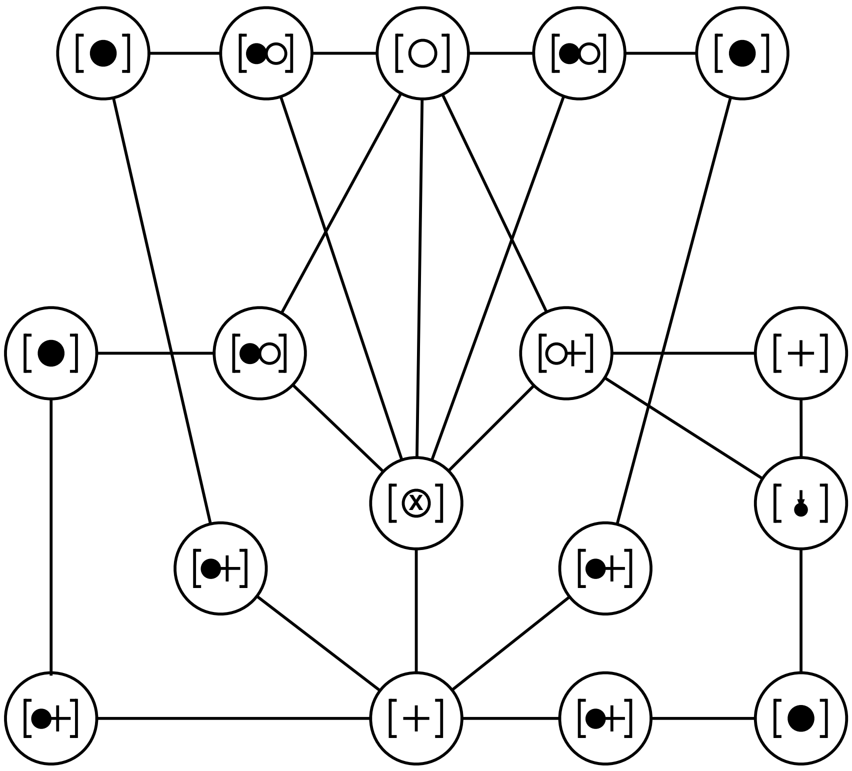


Figure 4.32: An overlap graph corresponding to the struct in the previous figure. Each node corresponds to one class element, and each edge between elements indicates that these two elements overlap on a constraint.

4.6 Level 1 classes

This section describes level 1 classes, which are assembled from the classes in the level 0 setting. In the previous section it was possible to present the entire level 0 class setting; however, there are many more classes in the level 1 setting, and this section presents only a few examples. A further limitation on discussing level 1 classes is that exactly which Go structures should be treated as classes at this level in order to facilitate play by an ETS-based Go program is a question to be settled empirically, i.e., via learning.

The level 0 classes depicted individual stones and the immediate relationships between them. Level 1 classes each combine several level 0 classes to form *blocks* of connected stones, and, in a similar way, the *spaces* between stones.

Table 4.4: Level 1 classes in this section

Number	Name	Colour
$\mathfrak{C}_1^1$	1-stone block	Black
$\mathfrak{C}_2^1$	2-stone block	Black
$\mathfrak{C}_3^1$	n-stone block	Black
$\mathfrak{C}_4^1$	black/white block interface	Black and White
$\mathfrak{C}_5^1$	1-point jump	Black
$\mathfrak{C}_6^1$	knight's jump	White
$\mathfrak{C}_7^1$	diagonal jump	White
$\mathfrak{C}_8^1$	long knight's jump	White
$\mathfrak{C}_9^1$	3 points from edge	Black

4.6.1 Previous level constraints

For simplicity, I will present the level 1 class constraints in this section as a pair of sets. The first element of this pair is the set of all level 0 class elements that participate in that level 1 constraint. The second element of this pair is a set of all the class element links (CEL^{1}s) that relate the various classes involved. The level 0 constraints named in these CEL^{1}s are all depicted in Figure 4.33

Figure 4.33 contains all of the level 0 constraints that will be used in the level 1 class examples in this section. Potentially, any constraint from any level 0 class can participate in a level 1 class; however, I have only depicted those constraints used in the few level 1 class examples presented here. All of these constraints have analogs in various level 0 class definitions, though that is not strictly necessary: a level 0 constraint may be used in a level 1 class definition if it specifies some part of the struct corresponding to a particular element from the necessary level 0 class.

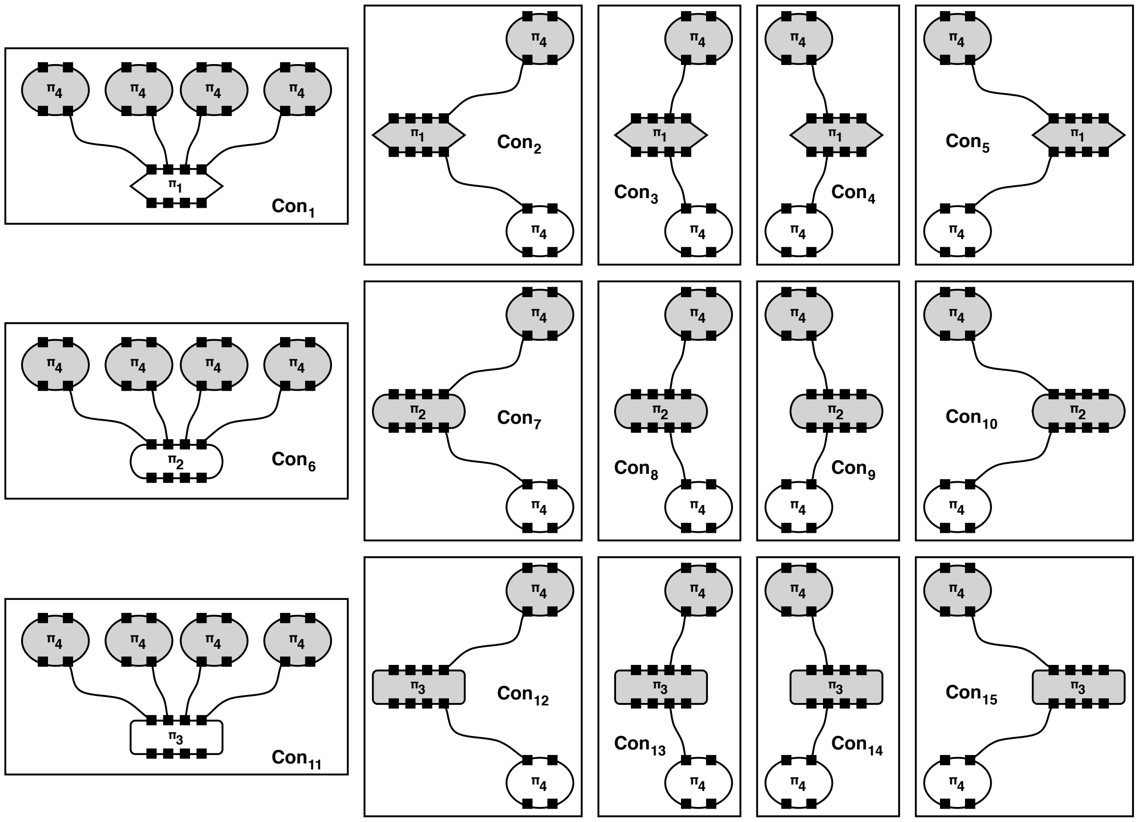


Figure 4.33: A selection of constraints present in various level 0 classes.

4.6.2 Blocks of stones

Following the terminology from [28] (as was discussed in Section 2.3.1), a block is a collection of *connected*, same-colour stones that share liberties. As a result, the block classes presented here are composed of not only level 0 stone classes, but also level 0 adjacency classes, both for describing the connectedness of the stones in the block, and describing the opposing stones/edges/empty vertices that delimit the block. The example block classes presented here are all blocks of black stones, but it is easy to see that blocks of white stones will follow the same pattern.

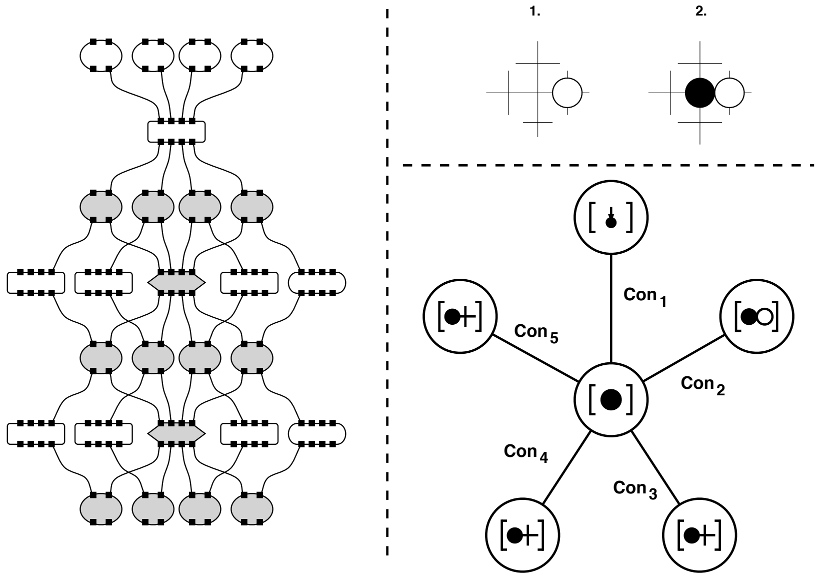


Figure 4.34: An element of $\mathfrak{C}_1^1$, the black one-stone block class. Upper right: the board position that resulted in this class element. Left: the complete struct, with the contained black stone class shown in grey. Lower right: the associated overlap graph with the particular overlapping constraints identified.

The simplest 2-level class is the one-stone block. The difference between a one-stone block and the previous-level single stone is that a one-stone block class contains information about the surrounding area as well.

<u>**Class Definition**</u> **1.** Class $\mathfrak{C}_1^1$, black one-stone block

An element of this class is pictured in Figure 4.34.

Class constraints:

- $\text{Con}_1^{\mathfrak{C}_1^1} = \big\langle \,\{\, c1_1, c12_1 \,\}, \{\, \langle \{c1_1, c12_1\}, \text{Con}_2 \rangle \,\}\, \big\rangle$
- $\text{Con}_2^{\mathfrak{C}_1^1} = \big\langle \,\{\, c1_1, c11_1 \,\}, \{\, \langle \{c1_1, c11_1\}, \text{Con}_2 \rangle \,\}\, \big\rangle$
- $\text{Con}_3^{\mathfrak{C}_1^1} = \big\langle \,\{\, c1_1, c14_1 \,\}, \{\, \langle \{c1_1, c14_1\}, \text{Con}_2 \rangle \,\}\, \big\rangle$
- $\text{Con}_4^{\mathfrak{C}_1^1} = \big\langle \,\{\, c1_1, c12_2 \,\}, \{\, \langle \{c1_1, c12_2\}, \text{Con}_3 \rangle \,\}\, \big\rangle$

- $\mathrm{Con}_5^{\mathfrak{C}_1^1} = \big\langle \big\{ c1_1, c11_2 \big\}, \big\{ \langle \{c1_1, c11_2\}, \mathrm{Con}_3 \rangle \big\} \big\rangle$
- $\mathrm{Con}_6^{\mathfrak{C}_1^1} = \big\langle \big\{ c1_1, c14_2 \big\}, \big\{ \langle \{c1_1, c14_2\}, \mathrm{Con}_3 \rangle \big\} \big\rangle$
- $\mathrm{Con}_7^{\mathfrak{C}_1^1} = \big\langle \big\{ c1_1, c12_3 \big\}, \big\{ \langle \{c1_1, c12_3\}, \mathrm{Con}_4 \rangle \big\} \big\rangle$
- $\mathrm{Con}_8^{\mathfrak{C}_1^1} = \big\langle \big\{ c1_1, c11_3 \big\}, \big\{ \langle \{c1_1, c11_3\}, \mathrm{Con}_4 \rangle \big\} \big\rangle$
- $\mathrm{Con}_9^{\mathfrak{C}_1^1} = \big\langle \big\{ c1_1, c14_3 \big\}, \big\{ \langle \{c1_1, c14_3\}, \mathrm{Con}_4 \rangle \big\} \big\rangle$
- $\mathrm{Con}_{10}^{\mathfrak{C}_1^1} = \big\langle \big\{ c1_1, c12_4 \big\}, \big\{ \langle \{c1_1, c12_4\}, \mathrm{Con}_5 \rangle \big\} \big\rangle$
- $\mathrm{Con}_{11}^{\mathfrak{C}_1^1} = \big\langle \big\{ c1_1, c11_4 \big\}, \big\{ \langle \{c1_1, c11_4\}, \mathrm{Con}_5 \rangle \big\} \big\rangle$
- $\mathrm{Con}_{12}^{\mathfrak{C}_1^1} = \big\langle \big\{ c1_1, c14_4 \big\}, \big\{ \langle \{c1_1, c14_4\}, \mathrm{Con}_5 \rangle \big\} \big\rangle$
- $\mathrm{Con}_{13}^{\mathfrak{C}_1^1} = \big\langle \big\{ c1_1, c4_1 \big\}, \big\{ \langle \{c1_1, c4_1\}, \mathrm{Con}_1 \rangle \big\} \big\rangle$
- $\mathrm{Con}_{14}^{\mathfrak{C}_1^1} = \big\langle \big\{ c1_1, c5_1 \big\}, \big\{ \langle \{c1_1, c5_1\}, \mathrm{Con}_1 \rangle \big\} \big\rangle$

Sets of constraints used by the generating system:

- $\mathrm{A} = \big\{ \mathrm{Con}_{13}^{\mathfrak{C}_1^1}, \Theta \big\}$
- $\mathrm{B} = \big\{ \mathrm{Con}_2^{\mathfrak{C}_1^1}, \mathrm{Con}_3^{\mathfrak{C}_1^1}, \mathrm{Con}_4^{\mathfrak{C}_1^1}, \mathrm{Con}_5^{\mathfrak{C}_1^1}, \mathrm{Con}_6^{\mathfrak{C}_1^1}, \mathrm{Con}_7^{\mathfrak{C}_1^1}, \mathrm{Con}_8^{\mathfrak{C}_1^1}, \mathrm{Con}_9^{\mathfrak{C}_1^1}, \mathrm{Con}_{10}^{\mathfrak{C}_1^1},$
  $\mathrm{Con}_{11}^{\mathfrak{C}_1^1}\ \mathrm{Con}_{12}^{\mathfrak{C}_1^1}, \Theta \big\}$
- $\mathrm{C} = \big\{ \mathrm{Con}_{14}^{\mathfrak{C}_1^1}, \Theta \big\}$

The generating system for this class chooses a constraint from set A at the first step, optionally beginning with $\mathrm{Con}_{13}^{\mathfrak{C}_1^1}$, which reflects the play of the stone that forms this single-stone "block", as is the case in the pictured example. At steps 2 to $n-1$, the system chooses a constraint from set B, assembling the block's surrounding environment and handling situations in which the block's neighbouring white stones are played or removed. Finally, the class element may optionally terminate with the application of $\mathrm{Con}_{14}^{\mathfrak{C}_1^1}$, should the stone be captured (it is also possible for the class generating system to terminate because a second black stone is added the the block). ▶

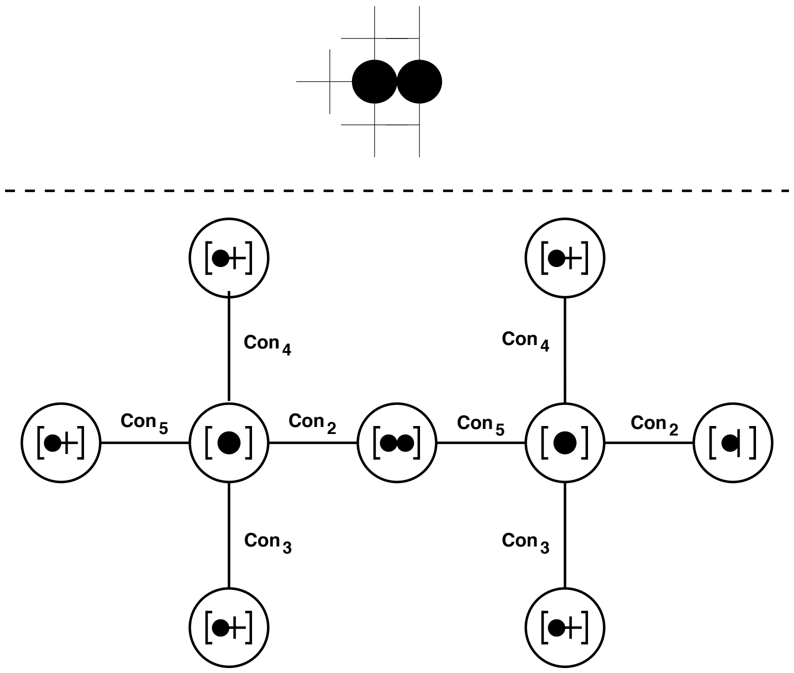


Figure 4.35: An element of $\mathfrak{C}_2^1$, the two-stone black block class. This particular class element is sitting at the edge of the board.

Class Definition 2. Class $\mathfrak{C}_2^1$, black two-stone block

An element of this class is pictured in Figure 4.35. This class contains all of the constraints from class $\mathfrak{C}_1^1$, which I have not duplicated here. The following constraints are unique to this particular class.

- $\mathrm{Con}_{15}^{\mathfrak{C}_2^1} = \left\langle \left\{ c1_1, c8_1, c1_2 \right\}, \left\{ \langle\{c1_1, c8_1\}, \mathrm{Con}_2\rangle, \langle\{c8_1, c1_2\}, \mathrm{Con}_5\rangle \right\} \right\rangle$
- $\mathrm{Con}_{16}^{\mathfrak{C}_2^1} = \left\langle \left\{ c1_1, c8_1, c1_2 \right\}, \left\{ \langle\{c1_1, c8_1\}, \mathrm{Con}_3\rangle, \langle\{c8_1, c1_2\}, \mathrm{Con}_4\rangle \right\} \right\rangle$

The class generating system follows the same pattern as that of $\mathfrak{C}_1^1$. It begins with a constraint from the same set A, but before progressing to set B, the generating system chooses one of $\mathrm{Con}_{15}^{\mathfrak{C}_2^1}$ or $\mathrm{Con}_{16}^{\mathfrak{C}_2^1}$ (depending on the orientation of the block).

After that, the generating system proceeds for many steps using set B, before finally choosing constraints from set C in the two final steps (since the block might be captured). ▶

It is possible to create class descriptions at this level with varying degrees of restrictiveness. The following class definition illustrates this: it is a class for any arbitrary block of black stones. Such a class could be useful in a situation where a Go program needs only a broad overview of the blocks on the board without examining them in great detail.

Class Definition 3. Class $\mathfrak{C}^1_3$, arbitrary black block

This class also contains all of the constraints from class $\mathfrak{C}^1_1$, but replaces the constraints unique to $\mathfrak{C}^1_2$ with a set of less restrictive constraints:

- $\text{Con}^{\mathfrak{C}^1_3}_{15} = \big\langle \big\{ c1_1, c8_1 \big\}, \big\{ \langle \{c1_1, c8_1\}, \text{Con}_2 \rangle \big\} \big\rangle$
- $\text{Con}^{\mathfrak{C}^1_3}_{16} = \big\langle \big\{ c1_1, c8_1 \big\}, \big\{ \langle \{c1_1, c8_1\}, \text{Con}_3 \rangle \big\} \big\rangle$
- $\text{Con}^{\mathfrak{C}^1_3}_{17} = \big\langle \big\{ c1_1, c8_1 \big\}, \big\{ \langle \{c1_1, c8_1\}, \text{Con}_4 \rangle \big\} \big\rangle$
- $\text{Con}^{\mathfrak{C}^1_3}_{18} = \big\langle \big\{ c1_1, c8_1 \big\}, \big\{ \langle \{c1_1, c8_1\}, \text{Con}_5 \rangle \big\} \big\rangle$

The class generating system is more involved, so it is not presented here, but it follows the same pattern as the previous two classes: first, constraints that describe the internal structure of the block are applied during the first m steps, and then constraints describing the block's immediate surroundings are applied from step $m+1$ to n. Finally, the block can optionally be captured. ▶

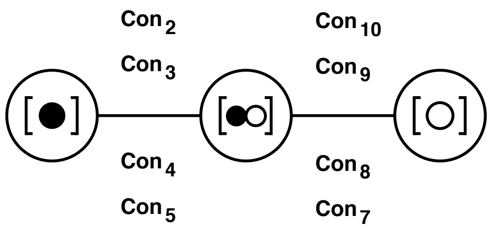


Figure 4.36: An element of $\mathfrak{C}_4^1$, the block-interface class. The eight level 0 constraints shown in the figure are those used in the formation of the eight level 1 constraints: see Def. 4

.

In order to describe the relationship between different-coloured blocks at the next level, it is necessary to capture at this level the interface between two immediately adjacent blocks. The result is $\mathfrak{C}_4^1$.

<u>**Class Definition**</u> **4.** Class $\mathfrak{C}_4^1$, the block-interface class.

This simple class has eight constraints:

- $\mathrm{Con}_1^{\mathfrak{C}_4^1} = \Big\langle \Big\{ c1_1, c11_1, \Big\}, \Big\{ \langle \{c1_1, c11_1\}, \mathrm{Con}_2 \rangle \Big\} \Big\rangle$
- $\mathrm{Con}_2^{\mathfrak{C}_4^1} = \Big\langle \Big\{ c11_1, c2_1, \Big\}, \Big\{ \langle \{c11_1, c2_1\}, \mathrm{Con}_{10} \rangle \Big\} \Big\rangle$
- $\mathrm{Con}_3^{\mathfrak{C}_4^1} = \Big\langle \Big\{ c1_1, c11_1, \Big\}, \Big\{ \langle \{c1_1, c11_1\}, \mathrm{Con}_3 \rangle \Big\} \Big\rangle$
- $\mathrm{Con}_4^{\mathfrak{C}_4^1} = \Big\langle \Big\{ c11_1, c2_1, \Big\}, \Big\{ \langle \{c11_1, c2_1\}, \mathrm{Con}_9 \rangle \Big\} \Big\rangle$
- $\mathrm{Con}_5^{\mathfrak{C}_4^1} = \Big\langle \Big\{ c1_1, c11_1, \Big\}, \Big\{ \langle \{c1_1, c11_1\}, \mathrm{Con}_4 \rangle \Big\} \Big\rangle$
- $\mathrm{Con}_6^{\mathfrak{C}_4^1} = \Big\langle \Big\{ c11_1, c2_1, \Big\}, \Big\{ \langle \{c11_1, c2_1\}, \mathrm{Con}_8 \rangle \Big\} \Big\rangle$
- $\mathrm{Con}_3^{\mathfrak{C}_4^1} = \Big\langle \Big\{ c1_1, c11_1, \Big\}, \Big\{ \langle \{c1_1, c11_1\}, \mathrm{Con}_5 \rangle \Big\} \Big\rangle$
- $\mathrm{Con}_4^{\mathfrak{C}_4^1} = \Big\langle \Big\{ c11_1, c2_1, \Big\}, \Big\{ \langle \{c11_1, c2_1\}, \mathrm{Con}_7 \rangle \Big\} \Big\rangle$

Generation is a 2-step process; the class generating system first chooses one constraint from this set:

- A = $\{ \mathrm{Con}_1^{\mathfrak{C}_4^1}, \mathrm{Con}_3^{\mathfrak{C}_4^1}, \mathrm{Con}_5^{\mathfrak{C}_4^1}, \mathrm{Con}_7^{\mathfrak{C}_4^1} \}$

and then chooses one from this set:

- B = $\{ \mathrm{Con}_2^{\mathfrak{C}_4^1}, \mathrm{Con}_4^{\mathfrak{C}_4^1}, \mathrm{Con}_6^{\mathfrak{C}_4^1}, \mathrm{Con}_8^{\mathfrak{C}_4^1} \}$

The choices dictate the orientation of the black and white stones. ▶

4.6.3 Spaces

In addition to classes of blocks, the level 1 class setting includes classes of spaces. Spaces are "chains" of connected empty vertices, but they are not directly analogous to blocks as I have defined them above. The purpose of a space class is to describe the spatial relationship between two or more nearby but not touching stones, and as such, the space classes include the particular stones that they relate. Similarly, a space may also exist between a single stone and the edge of the board.

Another respect in which space classes differ from block classes is that a space class need not contain all connected empty vertices, just those that are of interest. In other words, while it is not possible to have two immediately adjacent blocks of the same colour (since they would actually be one large block), it is possible to have two immediately adjacent (or even overlapping) spaces. The reason for this is that empty vertices do not behave in the same way as blocks of stones: they do not "live and die" together. Spaces may be incrementally divided and filled.

The examples presented in this section are all *jumps*: positions that commonly arise in Go games, resulting when a player places a new stone near one of his previous stones. Figure 4.37 shows examples of three such patterns, along with the element

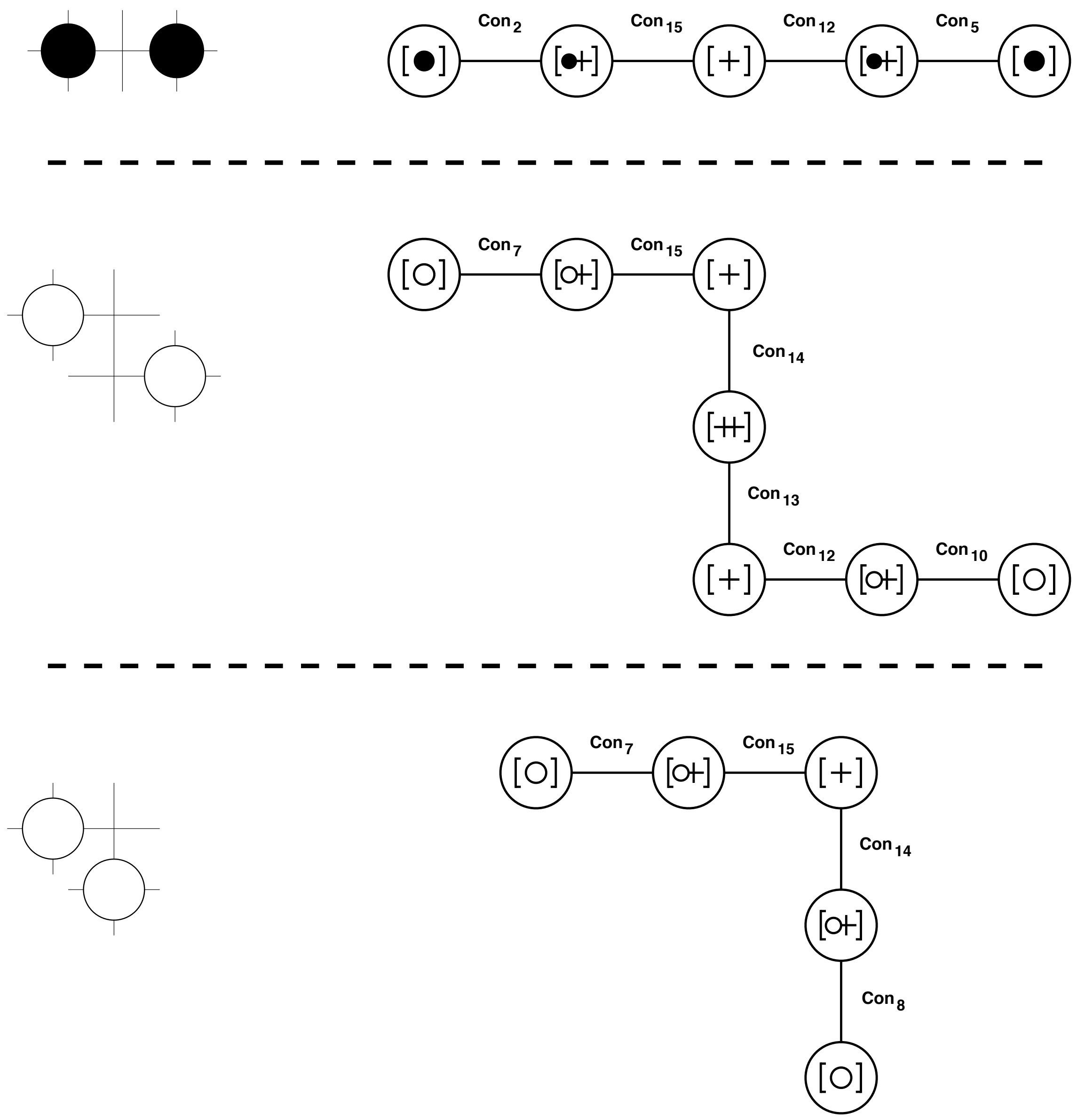


Figure 4.37: Examples of $\mathfrak{C}^1_5$, a one-point jump, $\mathfrak{C}^1_6$, a knight's jump, and $\mathfrak{C}^1_7$, a diagonal play.

of the associated class. A one-point jump is the most elementary jump play, in which a player adds a new stone on the same line as an existing stone, but with a single empty vertex between them. The knight's jump is so-called because of its resemblance to the movement of a knight's piece in chess: it is a combination of a lateral and a diagonal step. The diagonal play is a more defensive move that places two stones diagonally adjacent without actually connecting them.

A noteworthy feature of the following classes is that they have two groups of constraints: one group describes the particular configuration of empty vertices that define the space, and the other group describes the stones attached to the edges. This second group is important for overlapping these classes with the block classes to form level 2 classes.

<u>**Class Definition**</u> **5.** Class $\mathfrak{C}_5^1$, a one-point jump.

Constraints for this class:

- $\mathrm{Con}_1^{\mathfrak{C}_5^1} = \left\langle \left\{ c12_1, c3_1, c12_2 \right\}, \left\{ \langle \{c12_1, c3_1\}, \mathrm{Con}_{15} \rangle, \langle \{c3_1, c12_2\}, \mathrm{Con}_{12} \rangle \right\} \right\rangle$
- $\mathrm{Con}_2^{\mathfrak{C}_5^1} = \left\langle \left\{ c12_1, c3_1, c12_2 \right\}, \left\{ \langle \{c12_1, c3_1\}, \mathrm{Con}_{14} \rangle, \langle \{c3_1, c12_2\}, \mathrm{Con}_{13} \rangle \right\} \right\rangle$
- $\mathrm{Con}_3^{\mathfrak{C}_5^1} = \left\langle \left\{ c1_1, c12_1, \right\}, \left\{ \langle \{c1_1, c12_1\}, \mathrm{Con}_2 \rangle \right\} \right\rangle$
- $\mathrm{Con}_4^{\mathfrak{C}_5^1} = \left\langle \left\{ c1_1, c12_1, \right\}, \left\{ \langle \{c1_1, c12_1\}, \mathrm{Con}_3 \rangle \right\} \right\rangle$
- $\mathrm{Con}_5^{\mathfrak{C}_5^1} = \left\langle \left\{ c12_1, c1_2, \right\}, \left\{ \langle \{c12_1, c1_2\}, \mathrm{Con}_4 \rangle \right\} \right\rangle$
- $\mathrm{Con}_6^{\mathfrak{C}_5^1} = \left\langle \left\{ c12_1, c1_2, \right\}, \left\{ \langle \{c12_1, c1_2\}, \mathrm{Con}_5 \rangle \right\} \right\rangle$

Sets of constraints used by the generating system:

- $\mathrm{A} = \left\{ \mathrm{Con}_1^{\mathfrak{C}_5^1}, \mathrm{Con}_2^{\mathfrak{C}_5^1} \right\}$
- $\mathrm{B} = \left\{ \mathrm{Con}_3^{\mathfrak{C}_5^1}, \mathrm{Con}_4^{\mathfrak{C}_5^1} \right\}$
- $\mathrm{C} = \left\{ \mathrm{Con}_5^{\mathfrak{C}_5^1}, \mathrm{Con}_6^{\mathfrak{C}_5^1} \right\}$

The generating system for this class has three steps, choosing a constraint from set A at the first step, B at the second, and C at the third. ▶

<u>Class Definition</u> 6. Class $\mathfrak{C}^1_6$, a knight's jump.

Constraints for this class:

- $\mathrm{Con}^{\mathfrak{C}^1_6}_1 = \left\langle \left\{ c13_1, c3_1, c10_1, c3_2, c13_2 \right\}, \left\{ \langle\{c13_1, c3_1\}, \mathrm{Con}_{15}\rangle, \langle\{c3_1, c10_1\}, \mathrm{Con}_{13}\rangle, \langle\{c10_1, c3_2\}, \mathrm{Con}_{14}\rangle, \langle\{c3_2, c13_2\}, \mathrm{Con}_{12}\rangle \right\} \right\rangle$
- $\mathrm{Con}^{\mathfrak{C}^1_6}_2 = \left\langle \left\{ c13_1, c3_1, c10_1, c3_2, c13_2 \right\}, \left\{ \langle\{c13_1, c3_1\}, \mathrm{Con}_{12}\rangle, \langle\{c3_1, c10_1\}, \mathrm{Con}_{13}\rangle, \langle\{c10_1, c3_2\}, \mathrm{Con}_{14}\rangle, \langle\{c3_2, c13_2\}, \mathrm{Con}_{15}\rangle \right\} \right\rangle$
- $\mathrm{Con}^{\mathfrak{C}^1_6}_3 = \left\langle \left\{ c13_1, c3_1, c10_1, c3_2, c13_2 \right\}, \left\{ \langle\{c13_1, c3_1\}, \mathrm{Con}_{14}\rangle, \langle\{c3_1, c10_1\}, \mathrm{Con}_{12}\rangle, \langle\{c10_1, c3_2\}, \mathrm{Con}_{15}\rangle, \langle\{c3_2, c13_2\}, \mathrm{Con}_{13}\rangle \right\} \right\rangle$
- $\mathrm{Con}^{\mathfrak{C}^1_6}_4 = \left\langle \left\{ c13_1, c3_1, c10_1, c3_2, c13_2 \right\}, \left\{ \langle\{c13_1, c3_1\}, \mathrm{Con}_{13}\rangle, \langle\{c3_1, c10_1\}, \mathrm{Con}_{12}\rangle, \langle\{c10_1, c3_2\}, \mathrm{Con}_{15}\rangle, \langle\{c3_2, c13_2\}, \mathrm{Con}_{14}\rangle \right\} \right\rangle$
- $\mathrm{Con}^{\mathfrak{C}^1_6}_5 = \left\langle \left\{ c2_1, c13_1, \right\}, \left\{ \langle\{c2_2, c13_1\}, \mathrm{Con}_{7}\rangle \right\} \right\rangle$
- $\mathrm{Con}^{\mathfrak{C}^1_6}_6 = \left\langle \left\{ c2_1, c13_1, \right\}, \left\{ \langle\{c2_1, c13_1\}, \mathrm{Con}_{8}\rangle \right\} \right\rangle$
- $\mathrm{Con}^{\mathfrak{C}^1_6}_7 = \left\langle \left\{ c13_1, c2_1, \right\}, \left\{ \langle\{c13_1, c2_1\}, \mathrm{Con}_{9}\rangle \right\} \right\rangle$
- $\mathrm{Con}^{\mathfrak{C}^1_6}_8 = \left\langle \left\{ c13_1, c1_1, \right\}, \left\{ \langle\{c13_1, c2_1\}, \mathrm{Con}_{10}\rangle \right\} \right\rangle$

Sets of constraints used by the generating system:

- $\mathrm{A} = \left\{ \mathrm{Con}^{\mathfrak{C}^1_5}_1, \mathrm{Con}^{\mathfrak{C}^1_5}_2, \mathrm{Con}^{\mathfrak{C}^1_5}_3, \mathrm{Con}^{\mathfrak{C}^1_5}_4 \right\}$
- $\mathrm{B} = \left\{ \mathrm{Con}^{\mathfrak{C}^1_5}_5, \mathrm{Con}^{\mathfrak{C}^1_5}_6 \right\}$
- $\mathrm{C} = \left\{ \mathrm{Con}^{\mathfrak{C}^1_5}_7, \mathrm{Con}^{\mathfrak{C}^1_5}_8 \right\}$

The generating system for this class has three steps, choosing a constraint from set A at the first step, B at the second, and C at the third. ▶

<u>Class Definition</u> 7. Class $\mathfrak{C}^1_7$, a diagonal play.

This class follows the same pattern as the previous two. Constraints 1-4 are unique to this class, while constraints 5-8 (not listed here) are the same as those from $\mathfrak{C}^1_6$

- $\mathrm{Con}_1^{\mathfrak{C}^1_7} = \left\langle \left\{ c13_1, c3_1, c13_2 \right\}, \left\{ \langle \{c13_1, c3_1\}, \mathrm{Con}_{15}\rangle , \langle \{c3_1, c13_2\}, \mathrm{Con}_{13}\rangle \right\} \right\rangle$
- $\mathrm{Con}_2^{\mathfrak{C}^1_7} = \left\langle \left\{ c13_1, c3_1, c13_2 \right\}, \left\{ \langle \{c13_1, c3_1\}, \mathrm{Con}_{15}\rangle , \langle \{c3_1, c13_2\}, \mathrm{Con}_{14}\rangle \right\} \right\rangle$
- $\mathrm{Con}_3^{\mathfrak{C}^1_7} = \left\langle \left\{ c13_1, c3_1, c13_2 \right\}, \left\{ \langle \{c13_1, c3_1\}, \mathrm{Con}_{12}\rangle , \langle \{c3_1, c13_2\}, \mathrm{Con}_{13}\rangle \right\} \right\rangle$
- $\mathrm{Con}_4^{\mathfrak{C}^1_7} = \left\langle \left\{ c13_1, c3_1, c13_2 \right\}, \left\{ \langle \{c13_1, c3_1\}, \mathrm{Con}_{12}\rangle , \langle \{c3_1, c13_2\}, \mathrm{Con}_{14}\rangle \right\} \right\rangle$

The class generating system is the same as that of $\mathfrak{C}^1_6$ ▶

Figure 4.38 shows an element of the "long-knight" jump class, so-called because it follows the knight's jump pattern but extends the gap by an additional vertex. This class is the most complex space class depicted so far. Note that the center "box" of vertices is specified by a single constraint, but attaching the stones in the right locations requires an intermediate set of constraints.

<u>Class Definition</u> 8. Class $\mathfrak{C}^1_8$, long knight jump class.

Constraints for this class:

- $\mathrm{Con}_1^{\mathfrak{C}^1_8} = \left\langle \left\{ c3_1, c10_1, c3_2, c10_2, c3_3, c10_3, c3_4, c10_4 \right\}, \left\{ \langle \{c3_1, c10_1\}, \mathrm{Con}_{12}\rangle , \langle \{c10_1, c3_2\}, \mathrm{Con}_{15}\rangle , \langle \{c3_2, c10_3\}, \mathrm{Con}_{13}\rangle , \langle \{c10_3, c3_3\}, \mathrm{Con}_{14}\rangle , \langle \{c3_3, c10_3\}, \mathrm{Con}_{15}\rangle , \langle \{c10_3, c3_4\}, \mathrm{Con}_{12}\rangle , \langle \{c3_4, c10_4\}, \mathrm{Con}_{14}\rangle , \langle \{c10_4, c3_1\}, \mathrm{Con}_{13}\rangle \right\} \right\rangle$

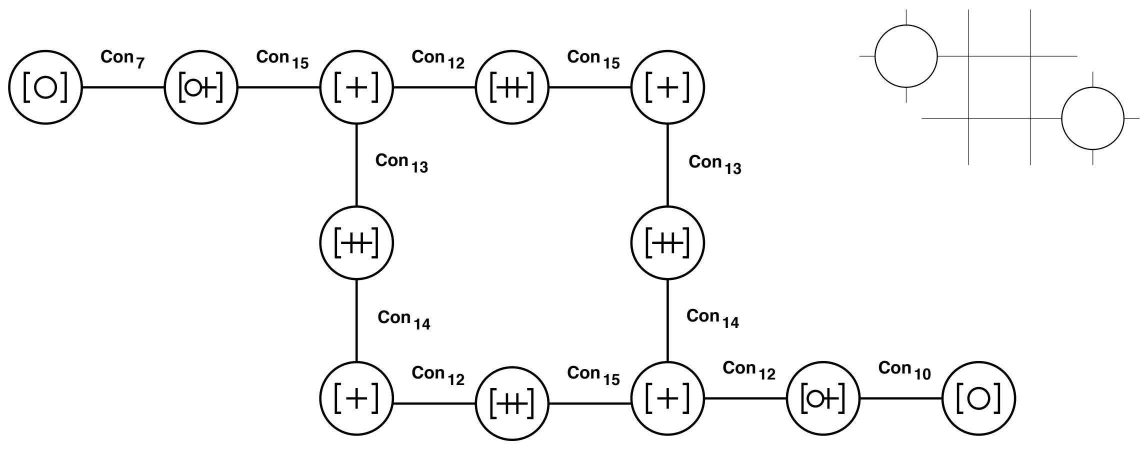
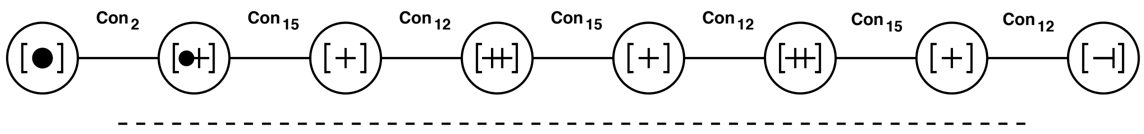


Figure 4.38: An element of $\mathfrak{C}_8^1$, the long-knight jump class.

- $\text{Con}_2^{\mathfrak{C}_8^1} = \big\langle \,\{\, c13_1, c3_1, c10_1, c10_2 \,\}, \{\, \langle\{c13_1, c3_1\}, \text{Con}_{15}\rangle\,, \langle\{c3_1, c10_1\}, \text{Con}_{12}\rangle\,, \langle\{c3_1, c10_2\}, \text{Con}_{13}\rangle\,, \} \,\big\rangle$
- $\text{Con}_3^{\mathfrak{C}_8^1} = \big\langle \,\{\, c13_1, c3_1, c10_1, c10_2 \,\}, \{\, \langle\{c13_1, c3_1\}, \text{Con}_{12}\rangle\,, \langle\{c3_1, c10_1\}, \text{Con}_{15}\rangle\,, \langle\{c3_1, c10_2\}, \text{Con}_{14}\rangle\,, \} \,\big\rangle$
- $\text{Con}_4^{\mathfrak{C}_8^1} = \big\langle \,\{\, c13_1, c3_1, c10_1, c10_2 \,\}, \{\, \langle\{c13_1, c3_1\}, \text{Con}_{15}\rangle\,, \langle\{c3_1, c10_1\}, \text{Con}_{12}\rangle\,, \langle\{c3_1, c10_2\}, \text{Con}_{14}\rangle\,, \} \,\big\rangle$
- $\text{Con}_5^{\mathfrak{C}_8^1} = \big\langle \,\{\, c13_1, c3_1, c10_1, c10_2 \,\}, \{\, \langle\{c13_1, c3_1\}, \text{Con}_{12}\rangle\,, \langle\{c3_1, c10_1\}, \text{Con}_{15}\rangle\,, \langle\{c3_1, c10_2\}, \text{Con}_{13}\rangle\,, \} \,\big\rangle$
- $\text{Con}_6^{\mathfrak{C}_8^1} = \big\langle \,\{\, c13_1, c3_1, c10_1, c10_2 \,\}, \{\, \langle\{c13_1, c3_1\}, \text{Con}_{14}\rangle\,, \langle\{c3_1, c10_1\}, \text{Con}_{13}\rangle\,, \langle\{c3_1, c10_2\}, \text{Con}_{12}\rangle\,, \} \,\big\rangle$
- $\text{Con}_7^{\mathfrak{C}_8^1} = \big\langle \,\{\, c13_1, c3_1, c10_1, c10_2 \,\}, \{\, \langle\{c13_1, c3_1\}, \text{Con}_{13}\rangle\,, \langle\{c3_1, c10_1\}, \text{Con}_{14}\rangle\,, \langle\{c3_1, c10_2\}, \text{Con}_{15}\rangle\,, \} \,\big\rangle$
- $\text{Con}_8^{\mathfrak{C}_8^1} = \big\langle \,\{\, c13_1, c3_1, c10_1, c10_2 \,\}, \{\, \langle\{c13_1, c3_1\}, \text{Con}_{13}\rangle\,, \langle\{c3_1, c10_1\}, \text{Con}_{14}\rangle\,, \langle\{c3_1, c10_2\}, \text{Con}_{15}\rangle\,, \} \,\big\rangle$
- $\text{Con}_9^{\mathfrak{C}_8^1} = \big\langle \,\{\, c13_1, c3_1, c10_1, c10_2 \,\}, \{\, \langle\{c13_1, c3_1\}, \text{Con}_{14}\rangle\,, \langle\{c3_1, c10_1\}, \text{Con}_{13}\rangle\,, \langle\{c3_1, c10_2\}, \text{Con}_{12}\rangle\,, \} \,\big\rangle$
- $\text{Con}_{10}^{\mathfrak{C}_8^1} = \big\langle \,\{\, c2_1, c13_1, \,\}, \{\, \langle\{c2_2, c13_1\}, \text{Con}_7\rangle \,\} \,\big\rangle$



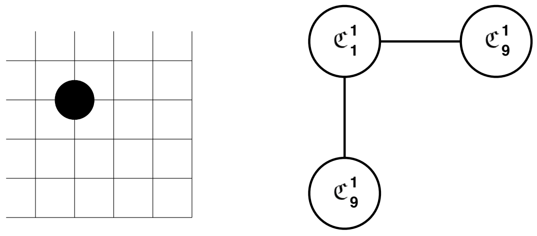


Figure 4.39: Top: an element of $\mathfrak{C}^1_9$, the three-points-from-edge class. Bottom left: a common opening play near the corner. Bottom right: the *level 2* overlap graph representing this position: two elements of $\mathfrak{C}^1_9$ and one of $\mathfrak{C}^1_1$ (1-stone black group).

- $\mathrm{Con}^{\mathfrak{C}^1_8}_{11} = \big\langle \big\{ c2_1, c13_1, \big\}, \big\{ \langle\{c2_1, c13_1\}, \mathrm{Con}_8\rangle \big\} \big\rangle$
- $\mathrm{Con}^{\mathfrak{C}^1_8}_{12} = \big\langle \big\{ c13_1, c2_1, \big\}, \big\{ \langle\{c13_1, c2_1\}, \mathrm{Con}_9\rangle \big\} \big\rangle$
- $\mathrm{Con}^{\mathfrak{C}^1_8}_{13} = \big\langle \big\{ c13_1, c1_1, \big\}, \big\{ \langle\{c13_1, c2_1\}, \mathrm{Con}_{10}\rangle \big\} \big\rangle$

Sets of constraints used by the class generating system:

- $\mathrm{A} = \big\{ \mathrm{Con}^{\mathfrak{C}^1_8}_1 \big\}$
- $\mathrm{B} = \big\{ \mathrm{Con}^{\mathfrak{C}^1_8}_2, \mathrm{Con}^{\mathfrak{C}^1_8}_4, \mathrm{Con}^{\mathfrak{C}^1_8}_6, \mathrm{Con}^{\mathfrak{C}^1_8}_8 \big\}$
- $\mathrm{C} = \big\{ \mathrm{Con}^{\mathfrak{C}^1_8}_3, \mathrm{Con}^{\mathfrak{C}^1_8}_5, \mathrm{Con}^{\mathfrak{C}^1_8}_7, \mathrm{Con}^{\mathfrak{C}^1_8}_9 \big\}$
- $\mathrm{D} = \big\{ \mathrm{Con}^{\mathfrak{C}^1_8}_{10}, \mathrm{Con}^{\mathfrak{C}^1_8}_{11} \big\}$
- $\mathrm{E} = \big\{ \mathrm{Con}^{\mathfrak{C}^1_8}_{12}, \mathrm{Con}^{\mathfrak{C}^1_8}_{13} \big\}$

The five-step generating system chooses a constraint from each of the above sets in order. ▶

Finally, Figure 4.39 shows an element of $\mathfrak{C}^1_9$, the "three-points-from-edge" class, and illustrates how two elements of that class can overlap with a 1-stone black group to form a common corner play. The definition of $\mathfrak{C}_9^1$ is not presented, but it is similar to the one-point jump definition.

In general, the classes at this level have many common constraints, and only a few “key” constraints that differentiate them. Obviously, detection of these key constraints is an important component of the recognition of a given class element. Successful recognition of a class element would also depend on its *maturity* (that is, how completely it has formed): the many shared constraints will make elements from different classes in the early stages of their formation difficult to distinguish.

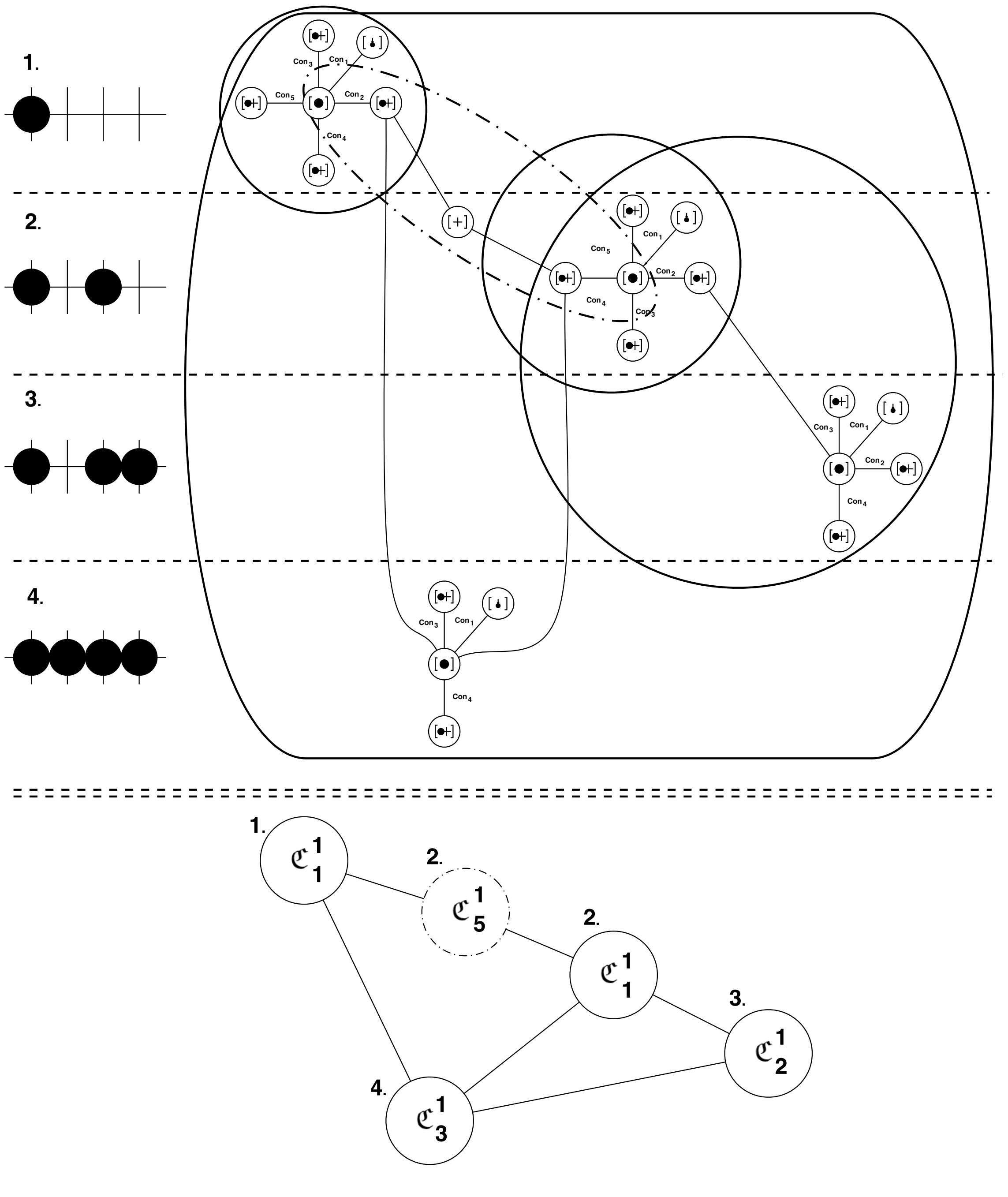


Figure 4.40: The evolution of class elements as a four-stone wall is built, with the resulting level 2 overlap graph shown below the double line. Each circled area is one level 1 class element, and overlap is shown. The element shown with a dotted circle does not overlap with the final four-stone class (the large outer loop).

4.7 Level 2 classes

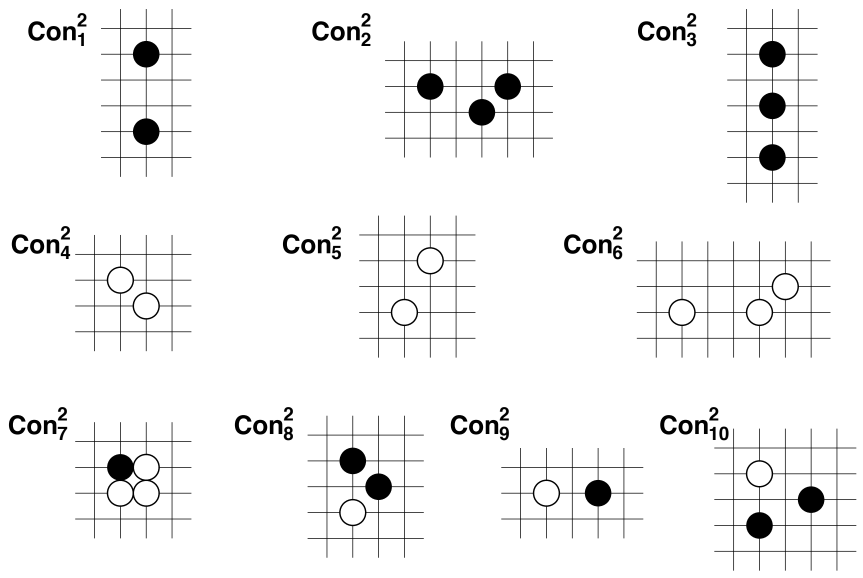


Figure 4.41: The board positions captured by some level 2 constraints

This section discusses level 2 classes, which are assembled from the classes in the level 1 setting. Figure 4.41 shows some shapes that can be represented as level 2 constraints arranging overlapping level 1 classes in various ways.

It is at this level that the full representational power of this model begins to emerge: level 2 classes combine the blocks and spaces described by level 1 classes to form arbitrary groups of stones. As with the previous levels, these classes can overlap, and can describe both the spatial and temporal organization of previous level blocks. A typical level 2 class describes the formation and shape of a particular kind of group. Such a class would describe the complete evolution of groups of stones that ultimately form a territory, including nearby and captured enemy stones, since these stones have a great effect on the formation of the group.

Unfortunately, the complexity of classes at this level makes them difficult for humans to design: this is where ETS learning algorithms should be employed instead. With such limitations in mind, this section presents a set of level 2 constraints which would commonly appear in the formation of various kinds of groups. Section 4.7.3 shows a tentative example of a level 2 class element.

4.7.1 Previous level classes and constraints

Table 4.5 lists the level 1 classes used to create the constraints used to describe the shapes in Figure 4.41. Classes $\mathfrak{C}^1_{10}$ throughd $\mathfrak{C}^1_{18}$ were not defined in the previous section, but all are very similar to previously described classes.

Table 4.5: Level 1 classes and constraints.

Number	Name	Colour
$\mathfrak{C}^1_1$	1-stone block	Black
$\mathfrak{C}^1_2$	2-stone block	Black
$\mathfrak{C}^1_3$	n-stone block	Black
$\mathfrak{C}^1_4$	black/white block interface	Black and White
$\mathfrak{C}^1_5$	1-point jump	Black
$\mathfrak{C}^1_6$	knight's jump	White
$\mathfrak{C}^1_7$	diagonal jump	White
$\mathfrak{C}^1_8$	long knight's jump	White
$\mathfrak{C}^1_9$	3 points from edge	Black
$\mathfrak{C}^1_{10}$	1-stone block	White
$\mathfrak{C}^1_{11}$	n-stone block	White
$\mathfrak{C}^1_{12}$	knight's jump	Black
$\mathfrak{C}^1_{13}$	diagonal jump	Black
$\mathfrak{C}^1_{14}$	2-point jump	Black
$\mathfrak{C}^1_{15}$	2-point jump	White
$\mathfrak{C}^1_{16}$	1-point jump	Black and White
$\mathfrak{C}^1_{17}$	knight's jump	Black and White
$\mathfrak{C}^1_{18}$	black/white diagonal interface	Black and White

The following are the level 1 constraints used in the examples in the remainder of this section. Each is present in one or more level 1 class. To simplify the notation, the superscripted class name has been replaced by a superscripted 1, indicating the appropriate level.

Black and empty:

- $\text{Con}^1_1 = \big\langle \big\{ c1_1, c12_1 \big\}, \big\{ \langle\{c1_1, c12_1\}, \text{Con}^0_2\rangle \big\} \big\rangle$
- $\text{Con}^1_2 = \big\langle \big\{ c1_1, c12_2 \big\}, \big\{ \langle\{c1_1, c12_2\}, \text{Con}^0_3\rangle \big\} \big\rangle$
- $\text{Con}^1_3 = \big\langle \big\{ c1_1, c12_3 \big\}, \big\{ \langle\{c1_1, c12_3\}, \text{Con}^0_4\rangle \big\} \big\rangle$
- $\text{Con}^1_4 = \big\langle \big\{ c1_1, c12_4 \big\}, \big\{ \langle\{c1_1, c12_4\}, \text{Con}^0_5\rangle \big\} \big\rangle$

White and empty:

- $\text{Con}^1_5 = \big\langle \big\{ c2_1, c13_1 \big\}, \big\{ \langle\{c2_1, c13_1\}, \text{Con}^0_7\rangle \big\} \big\rangle$
- $\text{Con}^1_6 = \big\langle \big\{ c2_1, c13_2 \big\}, \big\{ \langle\{c2_1, c13_2\}, \text{Con}^0_8\rangle \big\} \big\rangle$
- $\text{Con}^1_7 = \big\langle \big\{ c2_1, c13_3 \big\}, \big\{ \langle\{c2_1, c13_3\}, \text{Con}^0_9\rangle \big\} \big\rangle$
- $\text{Con}^1_8 = \big\langle \big\{ c2_1, c13_4 \big\}, \big\{ \langle\{c2_1, c13_4\}, \text{Con}^0_{10}\rangle \big\} \big\rangle$

Black and white:

- $\text{Con}^1_9 = \big\langle \big\{ c1_1, c11_1 \big\}, \big\{ \langle\{c1_1, c11_1\}, \text{Con}^0_2\rangle \big\} \big\rangle$
- $\text{Con}^1_{10} = \big\langle \big\{ c1_1, c11_2 \big\}, \big\{ \langle\{c1_1, c11_2\}, \text{Con}^0_3\rangle \big\} \big\rangle$
- $\text{Con}^1_{11} = \big\langle \big\{ c1_1, c11_3 \big\}, \big\{ \langle\{c1_1, c11_3\}, \text{Con}^0_4\rangle \big\} \big\rangle$
- $\text{Con}^1_{12} = \big\langle \big\{ c1_1, c11_4 \big\}, \big\{ \langle\{c1_1, c11_4\}, \text{Con}^0_5\rangle \big\} \big\rangle$

White and black:

- $\text{Con}^1_{13} = \big\langle \big\{ c2_1, c13_1 \big\}, \big\{ \langle\{c2_1, c13_1\}, \text{Con}^0_7\rangle \big\} \big\rangle$
- $\text{Con}^1_{14} = \big\langle \big\{ c2_1, c13_2 \big\}, \big\{ \langle\{c2_1, c13_2\}, \text{Con}^0_8\rangle \big\} \big\rangle$
- $\text{Con}^1_{15} = \big\langle \big\{ c2_1, c13_3 \big\}, \big\{ \langle\{c2_1, c13_3\}, \text{Con}^0_9\rangle \big\} \big\rangle$
- $\text{Con}^1_{16} = \big\langle \big\{ c2_1, c13_4 \big\}, \big\{ \langle\{c2_1, c13_4\}, \text{Con}^0_{10}\rangle \big\} \big\rangle$

Black and black:

- $\text{Con}^1_{17} = \big\langle \, \{ \, c1_1, c8_1 \, \}, \{ \, \langle \{c1_1, c8_1\}, \text{Con}^0_2 \rangle \, \} \, \big\rangle$
- $\text{Con}^1_{18} = \big\langle \, \{ \, c1_1, c8_1 \, \}, \{ \, \langle \{c1_1, c8_1\}, \text{Con}^0_3 \rangle \, \} \, \big\rangle$
- $\text{Con}^1_{19} = \big\langle \, \{ \, c1_1, c8_1 \, \}, \{ \, \langle \{c1_1, c8_1\}, \text{Con}^0_4 \rangle \, \} \, \big\rangle$
- $\text{Con}^1_{20} = \big\langle \, \{ \, c1_1, c8_1 \, \}, \{ \, \langle \{c1_1, c8_1\}, \text{Con}^0_5 \rangle \, \} \, \big\rangle$

4.7.2 Overlapping constraints: an example

Figure 4.42 shows the level 2 overlap graphs for Con^2_6 and Con^2_{10}. Formally, the definitions of these constraints are:

$$\begin{aligned}\text{Con}^2_6 = \big\langle \, \{ \, c10_1, c15_1, c10_2, c7_1, c7_2, c10_3 \, \}, \{ \, \langle \{c10_1, c15_1\}, \text{Con}^1_5 \rangle, \\ \langle \{c15_1, c10_2\}, \text{Con}^1_8 \rangle, \langle \{c10_2, c7_1\}, \text{Con}^1_7 \rangle, \langle \{c10_2, c7_2\}, \text{Con}^1_5 \rangle, \\ \langle \{c7_1, c10_3\}, \text{Con}^1_8 \rangle, \langle \{c7_2, c10_3\}, \text{Con}^1_6 \rangle \, \} \, \big\rangle\end{aligned}$$

$$\begin{aligned}\text{Con}^2_{10} = \big\langle \, \{ \, c10_1, c16_1, c1_1, c12_1, c1_2, c17_1 \, \}, \{ \, \langle \{c10_1, c16_1\}, \text{Con}^1_6 \rangle, \\ \langle \{c16_1, c1_1\}, \text{Con}^1_3 \rangle, \langle \{c1_1, c12_1\}, \text{Con}^1_1 \rangle, \langle \{c12_1, c1_2\}, \text{Con}^1_4 \rangle, \\ \langle \{c1_2, c17_1\}, \text{Con}^1_4 \rangle, \langle \{c17_1, c10_1\}, \text{Con}^1_5 \rangle \, \} \, \big\rangle\end{aligned}$$

Section 2.4 discussed Reitman's study [40] on how Go players remember board positions, which concluded that players organize the board as a series of chunks, consistent with earlier studies on chess players[6]. However, a major difference between these studies was that the Go player's chunks were thought to *overlap*.

As part of this study, Reitman had an expert Go player examine several board positions and circle the stones that he thought formed meaningful patterns. Figure

[6]Discussed in [41], p779.

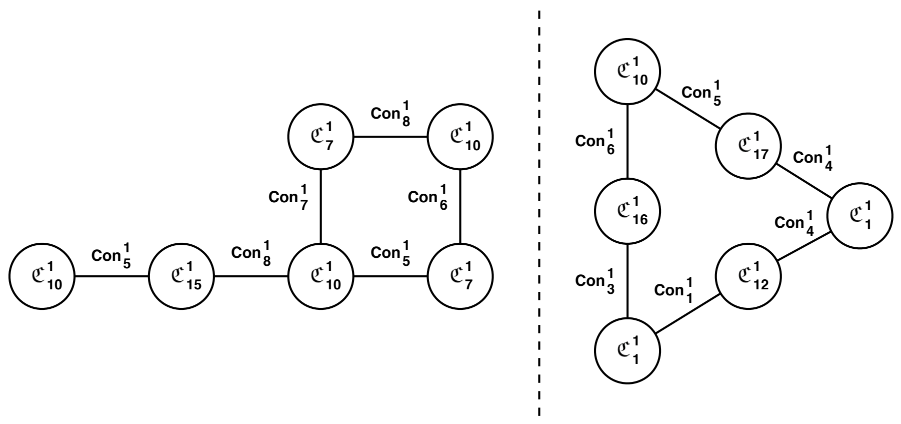


Figure 4.42: Overlap graphs for constraints Con^2_6 (left) and Con^2_{10} (right).

4.43 reproduces two of these positions along with the groups indicated by the Go expert. Each circled group can be treated as a level 2 constraint.

Four white stones in the left-hand position of Figure 4.43 are marked X: the Go expert identified this block as being “related”. I draw the reader’s attention to the fact that he also identified the black stone and three of the four stones in this block: this is a situation where the arbitrarily-shaped black and white block classes are useful, as the part of the block relevant to the identified shape can be specified by Con^2_7, and the rest of the block may have any configuration. This is a desirable situation where the formation of some level 2 class that uses Con^2_7 not only specifies which *classes* must be present at the previous level, but also partially determines which *class elements* are present, in this case by requiring some of the level 1 classes overlap on multiple constraints. Figure 4.44 illustrates this.

Figure 4.45 shows the overlap graph (with constraints indicated) for the large overlapping region of the right hand position in Figure 4.43. This combination of constraints could be produced at some step in the formation of two or three level 2 class elements. Only the relationships between various stones and groups that are necessary to form the constraints are depicted in the overlap graph.

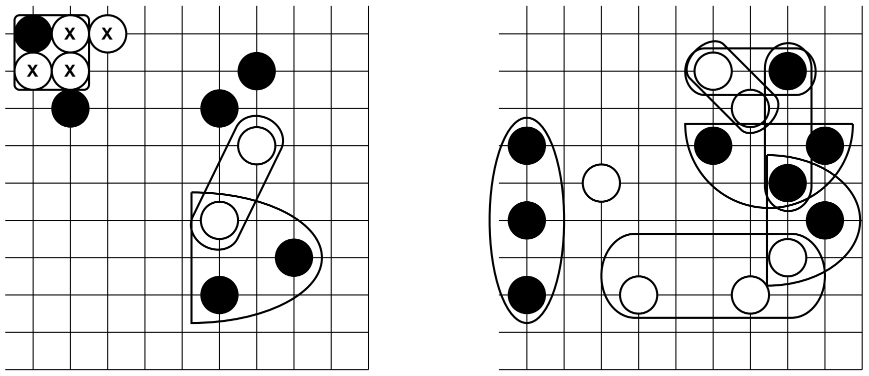


Figure 4.43: Two annotated Go positions taken from [40]. The circles were added by an expert Go player who was asked to indicate which stones formed a “meaningful pattern”.

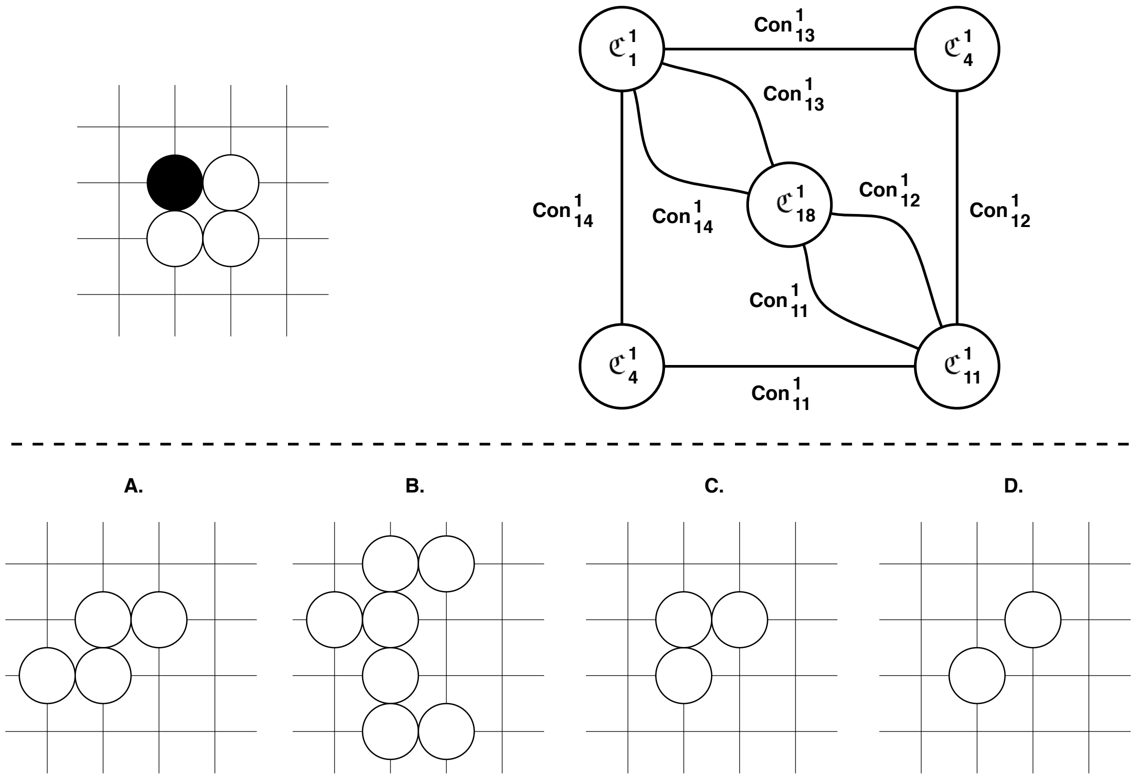


Figure 4.44: Top: a Go position and the overlap graph for the corresponding constraint: Con$_7^2$. Bottom: A and B are elements of $\mathfrak{C}_{11}^1$ that could be used to satisfy this constraint. C is an element of $\mathfrak{C}_{11}^1$ that could not satisfy the constraint, and D is neither a class element nor satisfies the constraint.

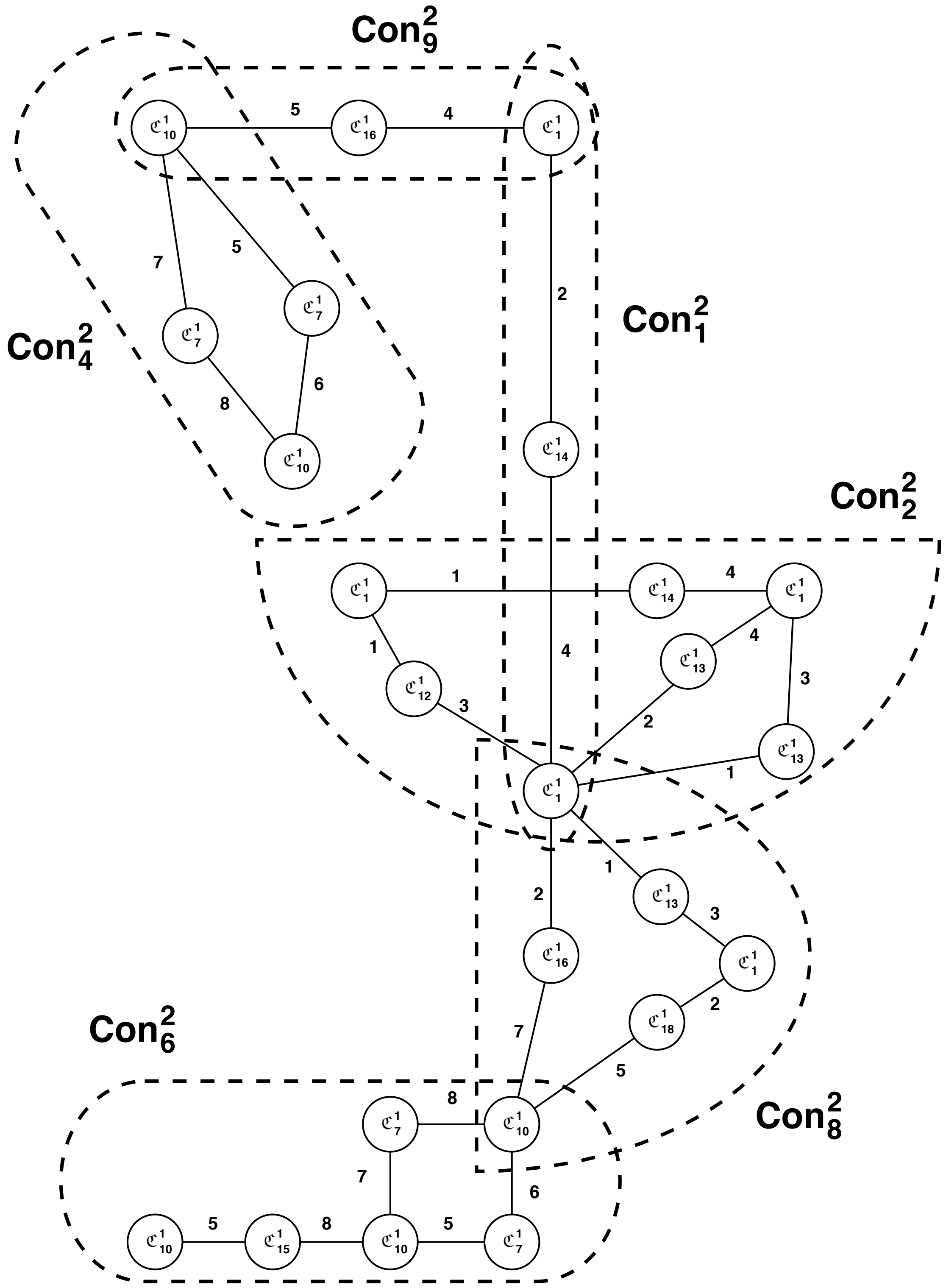


Figure 4.45: An overlap graph corresponding to the main set of stones on the right side of Figure 4.43. Individual level 1 constraints are indicated by their subscript.

4.7.3 A tentative level 2 class element

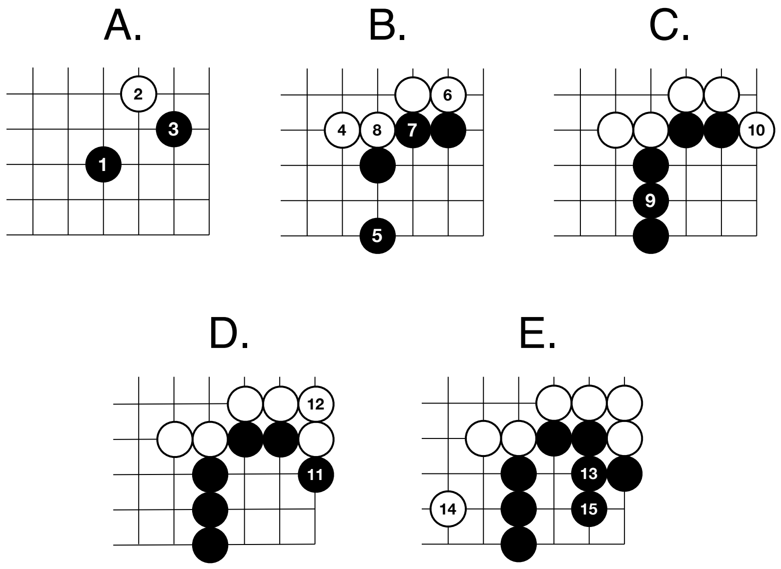


Figure 4.46: The formation of this shape can be treated as a level 2 class element.

This section presents a pictorial example of the formation of a level 2 class element representing a small area of black territory. Figure 4.46 shows the moves that resulted in this class element, broken into five steps of formation, labelled A through E. Figures 4.47 to 4.51 each show the class element at one of these steps. For clarity, level 1 classes that represent blocks are shown with bold circles. Each diagram after the first also shows the transitions between the two steps: dotted lines with a single black arrow indicate that the upper class's element overlaps with and terminates at the beginning of the lower class's element. Dashed lines with double white arrows indicate that the same class element (at a later stage of its development) appears in both steps. Some tentative constraints are circled at each step.

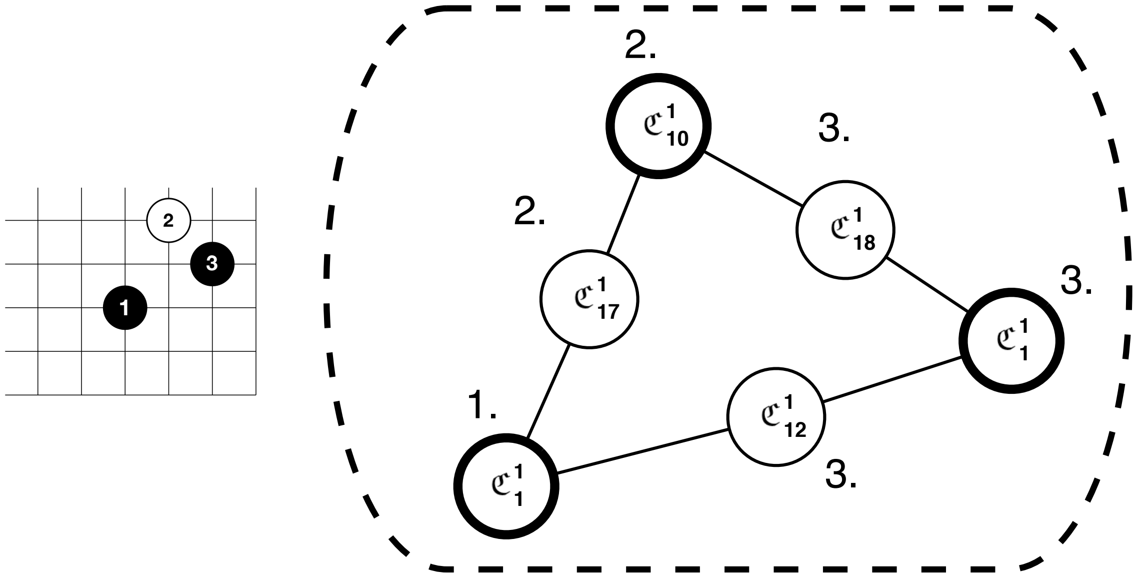


Figure 4.47: The level 2 class element at stage A. The external numbers correspond to the order in which the corresponding stones were played.

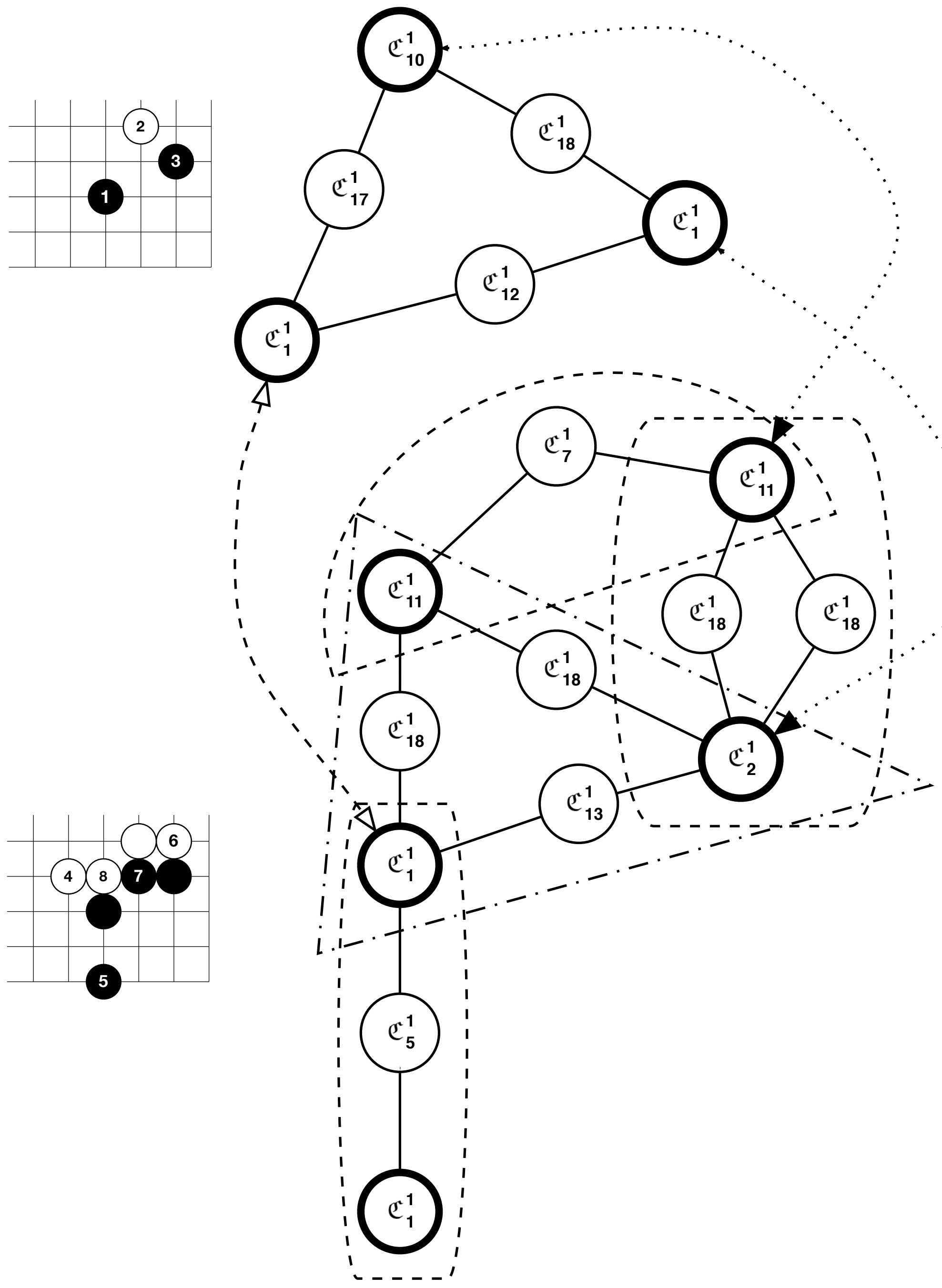


Figure 4.48: The level 2 class element transitioning from stage A to stage B.

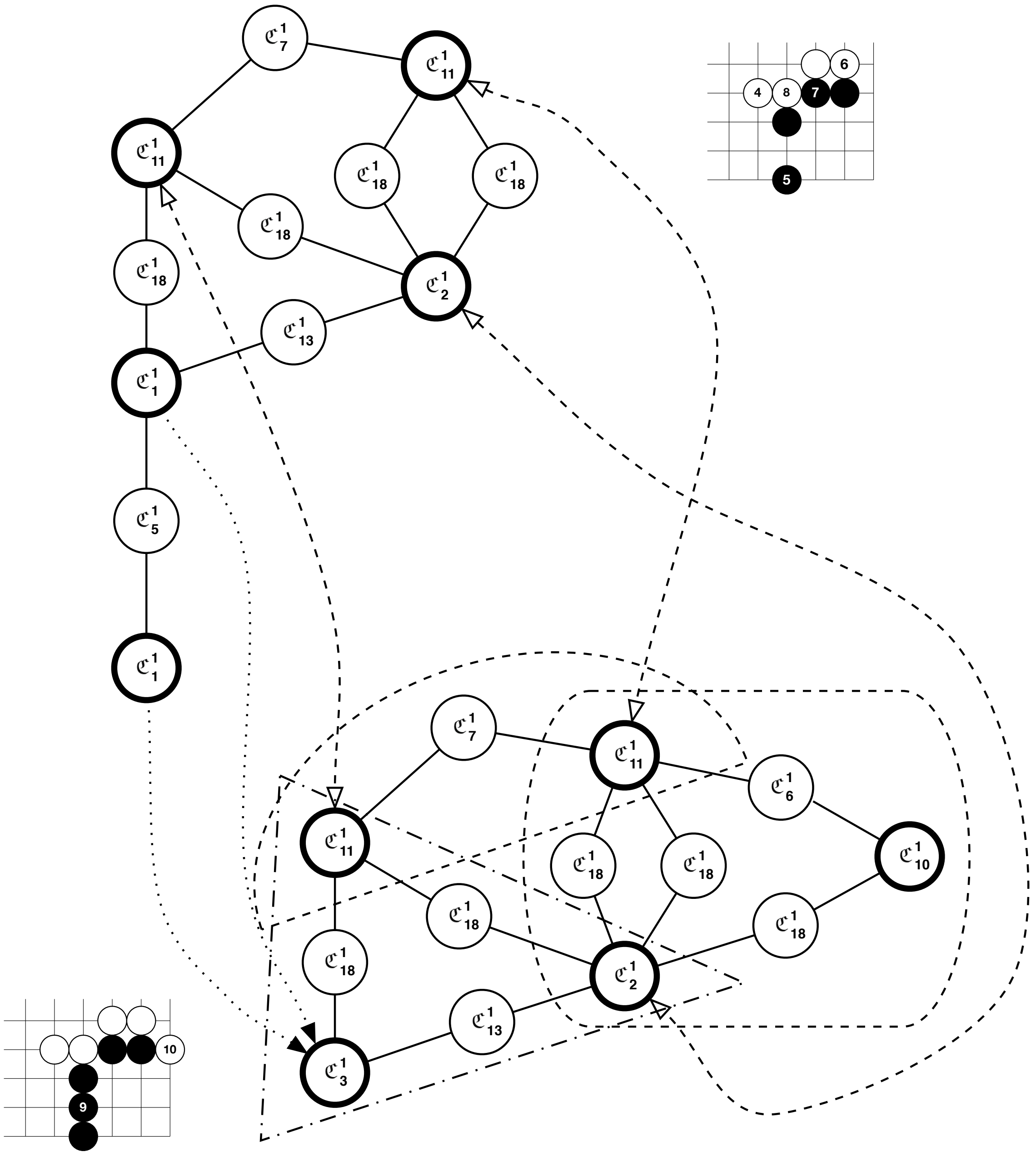


Figure 4.49: The level 2 class element transitioning from stage B to stage C.

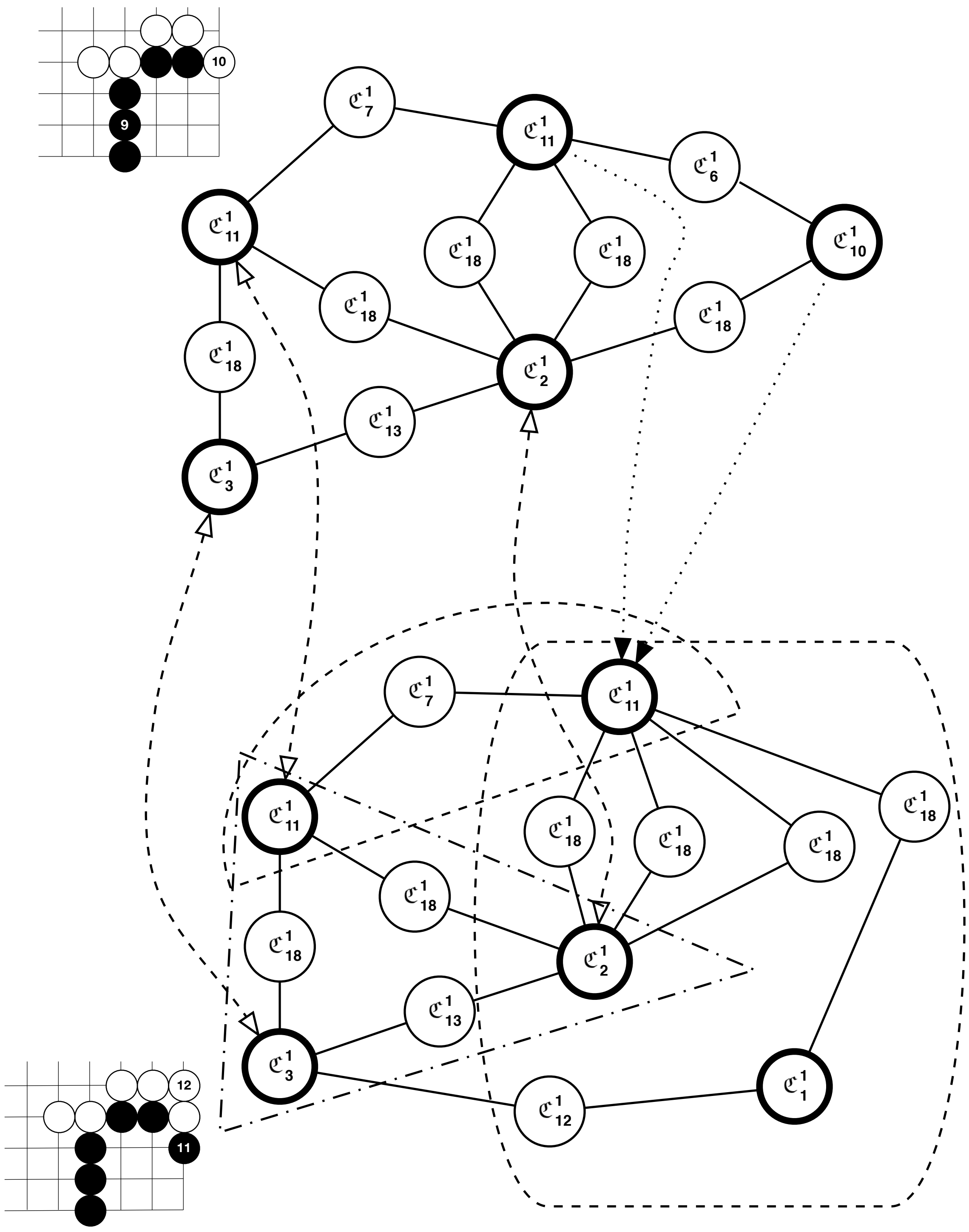


Figure 4.50: The level 2 class element transitioning from stage C to stage D.

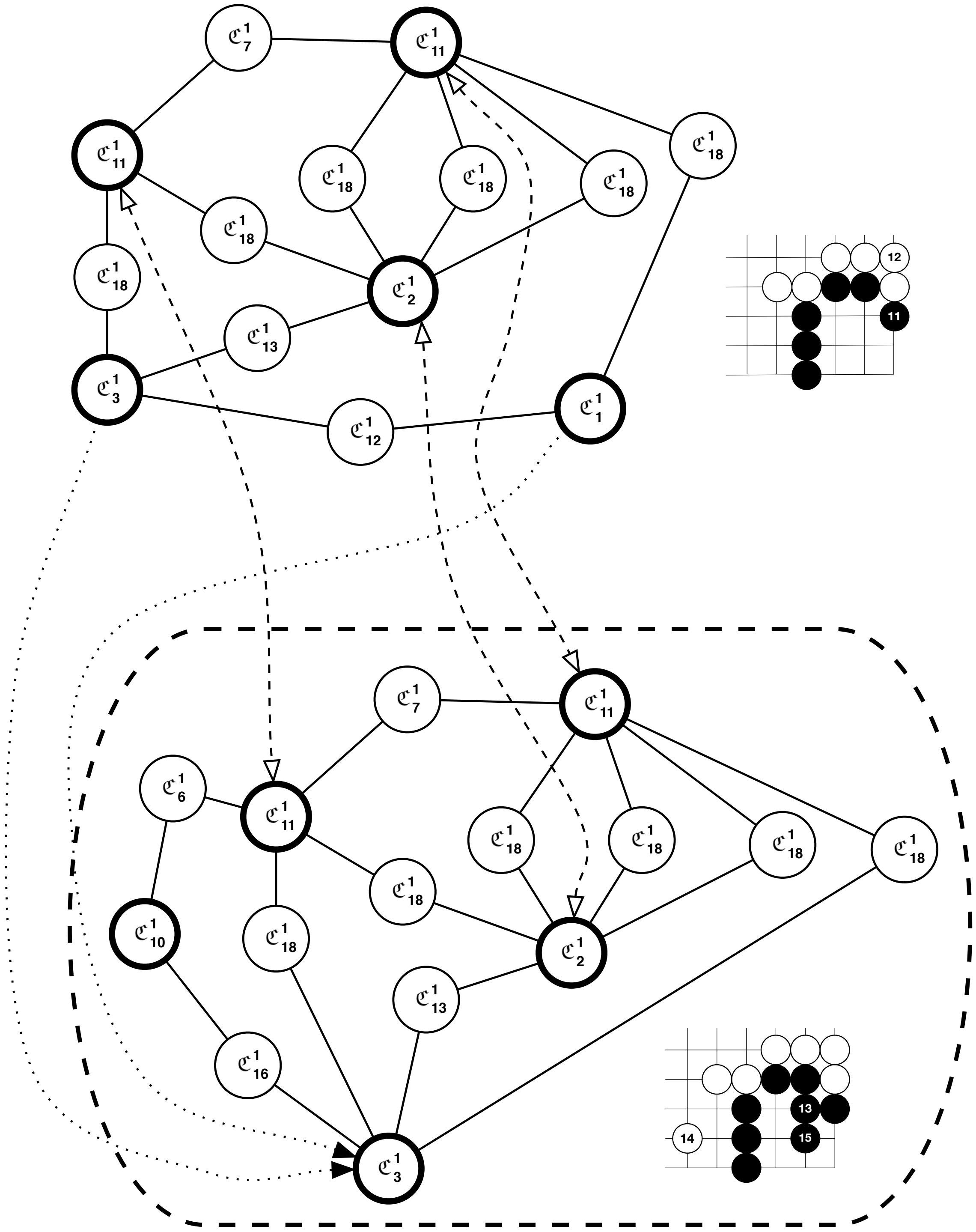


Figure 4.51: The level 2 class element transitioning from stage D to stage E, its final, stable state.

Chapter 5

Discussion

This chapter discusses several properties of the ETS model of Go presented in the previous chapter. Section 5.1 describes what this model offers in purely representational terms, exploring several aspects of the relationship between board positions and corresponding classified structs.

Section 5.2 discusses to what extent this model of Go is a cognitive model for how human Go players approach the game, mostly from the point of view of how ETS classes correspond to the overlapping "chunks" Go players are thought to rely on when they mentally manipulate board positions.

Finally, Section 5.3 sketches an answer to the very important question of how this model could be used in a system that plays Go. Such software should be entirely class-driven and must be developed from scratch based on the logic of the model.

5.1 Representational properties

5.1.1 Formative history

The level 0 and level 1 classes described in the previous chapter have a very spatial flavour in that they usually represent a combination of stones that are present on the board at the same time. However, as Figure 4.25 as well as the long example of Figures 4.47-4.51 illustrate, these classes can be combined temporally to describe the evolution of a board position. This kind of evolutionary representation, epitomizing the ETS tenet that *formative history* defines objects, is well suited to Go, since knowing how to make a shape is as important to the player as recognizing the shape itself. Figures 4.47-4.51 identify only constraints that delimit spatial structures on the Go board, but such a class would also contain constraints overlapping the classes connected by the dashed arrows, defining how the shape evolves over time.

5.1.2 Class variability

Classes at level 0 and level 1 generate only a few (structurally) different objects. A black stone, for example, differs from another only in which concrete primitives compose it, and for how long the corresponding process runs (i.e., how long between play and capture). However, the reader should not get the idea that ETS class generating systems must be so inflexible in their outputs. These level 0 and level 1 classes have a high degree of element homogeneity because that is the nature of Go; one stone is identical to another, but complexity arises at higher (conceptual) levels. Accordingly, at higher representational levels, classes become more flexible and expansive.

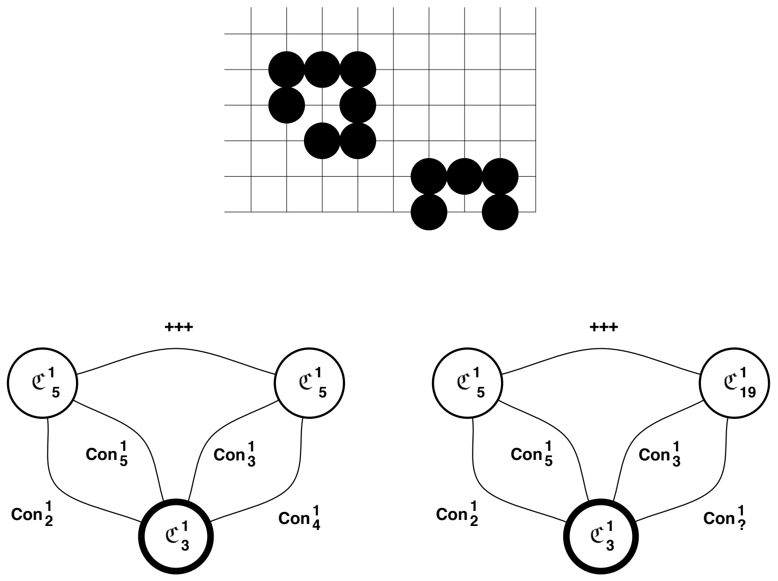


Figure 5.1: Two level 2 constraints associated with the two "eyes" in the above position. $\mathfrak{C}^1_3$ is the black block class, $\mathfrak{C}^1_5$ is the black one-point jump class, and $\mathfrak{C}^1_{19}$ is the not-previously-defined "black stone one point from the edge" class. The constraint labelled $\text{Con}^1_?$ is the not-previously-defined empty+edge analog to Con^1_4. The overlap-connection labelled '+++' is actually a whole set of constraints, since the classes overlap on the entire center vertex.

Figure 5.1 shows a pair of level 2 constraints corresponding to two different "eyes". It is quite clear from the diagram that the two different board positions are captured via two structurally very similar constraints. These two constraints might both be present in the generating system of a larger class that describes the evolution of blocks with two eyes, for example.

Many Go programs that manipulate patterns in a significant way (i.e., not Monte Carlo programs) operate at the conceptual level of blocks and above. Since such patterns are realized here as level 2 classes, the reader might wonder why one would bother with the lower class levels at all. To see the answer to this, consider how to represent level 2 classes without the framework of level 1 classes underneath them. Level 2 class constraints depend on level 1 constraints, which in turn depend on level 0 classes, meaning that level 0 and level 1 classes must be realized in a complete way. The question then becomes: can you compress all of the level 0 and 1 class information down into the structs? It is probably possible, but to do so would cost flexibility, since the new classes would each have to contain information about stones and small patterns directly, in addition to the information about their own structure. Instead, it is better to take advantage of the savings inherent in hierarchical organization.

The ETS approach to modelling suggests that one should think carefully about data. In the case of Go, designing appropriate primitives and level 0 classes forced me to think about how to represent the spatial aspects of the Go board, the difference between stones and empty vertices, and the play and removal of stones. The multi-leveled structure of ETS forces one to solve these problems systematically, and prevents one from trying to model the more complex aspects of the phenomenon without addressing the simple ones first.

5.1.3 Translational and rotational independence

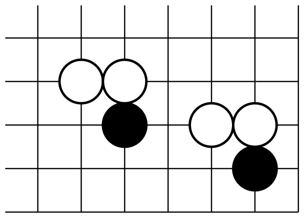


Figure 5.2: The same configuration of Go stones under *translation*, i.e., in two different locations on the board.

An important capability of any Go-playing system is the ability to generalize knowledge. A basic element of this is the ability to recognize the same Go position or move sequence under a simple transformation, such as translation (move to a new location on the board), mirroring (flip about one axis) and rotation (pivot by 90 degrees).

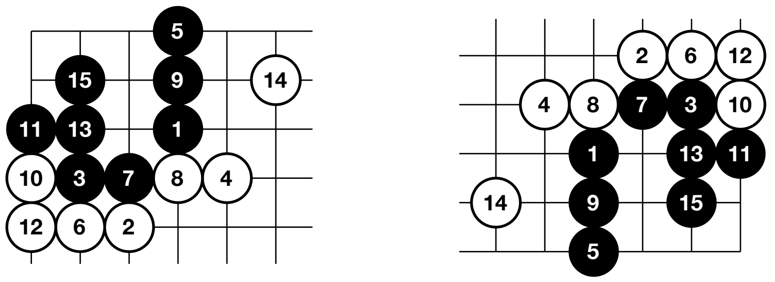


Figure 5.3: These two Go positions are identical: they merely appear in opposite corners; i.e., the first position can be transformed into the second by *rotating* the board 180 degrees.

That the same Go stone configuration is represented in the same way regardless of where it is on the board is obvious, since the entire representational hierarchy from primitives to high-level classes is based on the relative position of the various elements. The only case in which translating a position results in different structs is when stones are moved to the very edge of the board, but the slight variation this introduces is desirable because stones on the edge of the board have fewer liberties and as such are not exactly the same as stones in the middle.

Rotation and mirroring are closely related (since both deal with various symmetric transformations of a given position), and are handled more subtly in this Go model: changing the orientation of a Go position *does* result in different structs (Figure 5.5). However, the *classification* of these structs remains the same: the rotated position is a different element of the same class (Figure 5.6).

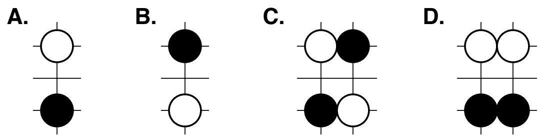


Figure 5.4: In isolation, A and B are the same pattern under mirroring. However, as C and D illustrate, A and B cannot be treated as strictly equivalent, since their relative position matter greatly when they are combined to form a larger position.

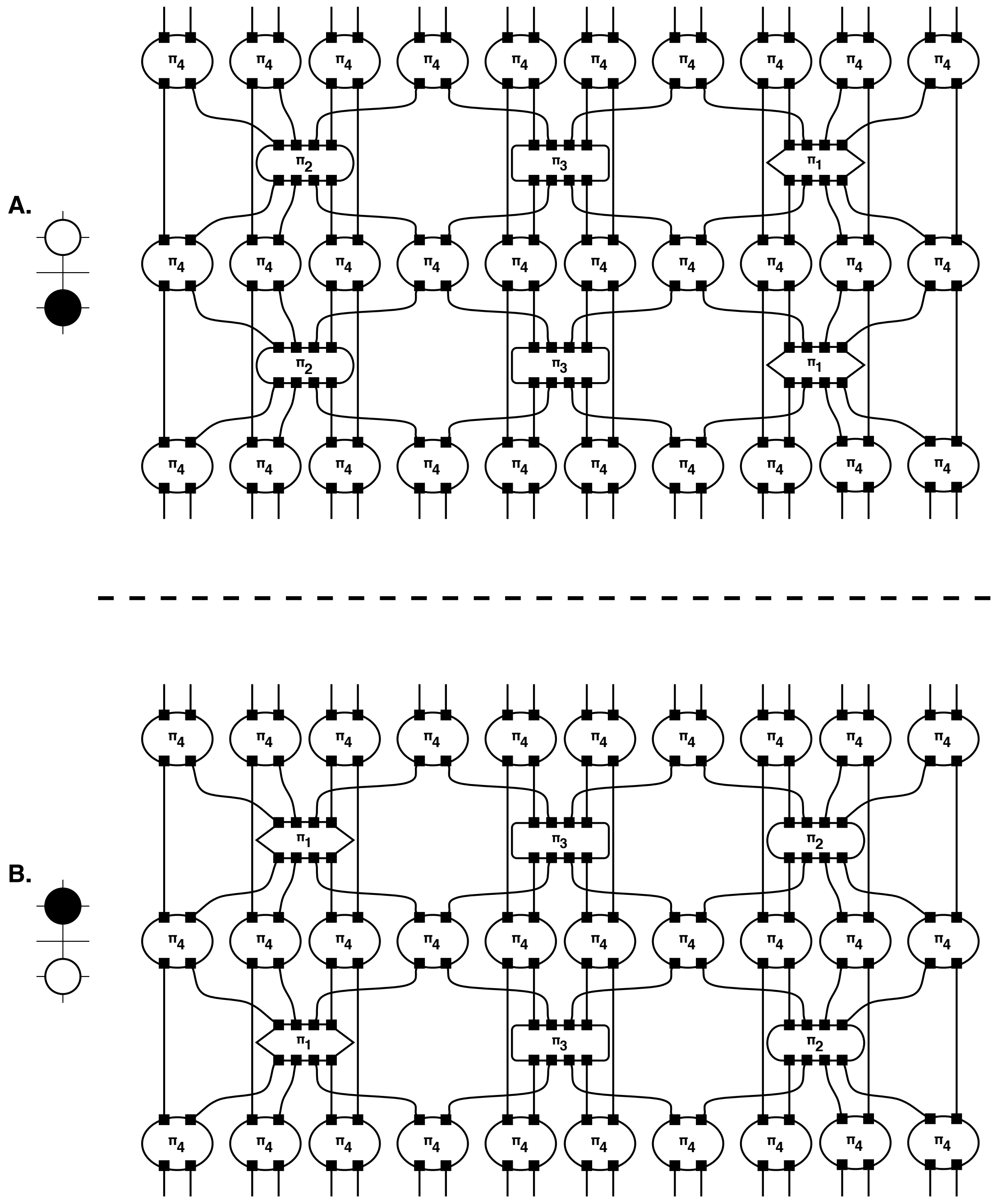


Figure 5.5: Two structs corresponding to the two opposite board positions shown at left.

To understand why this is desirable, recall what ETS class elements have in common: objects in the same class are generated in the same way. This is a natural way to

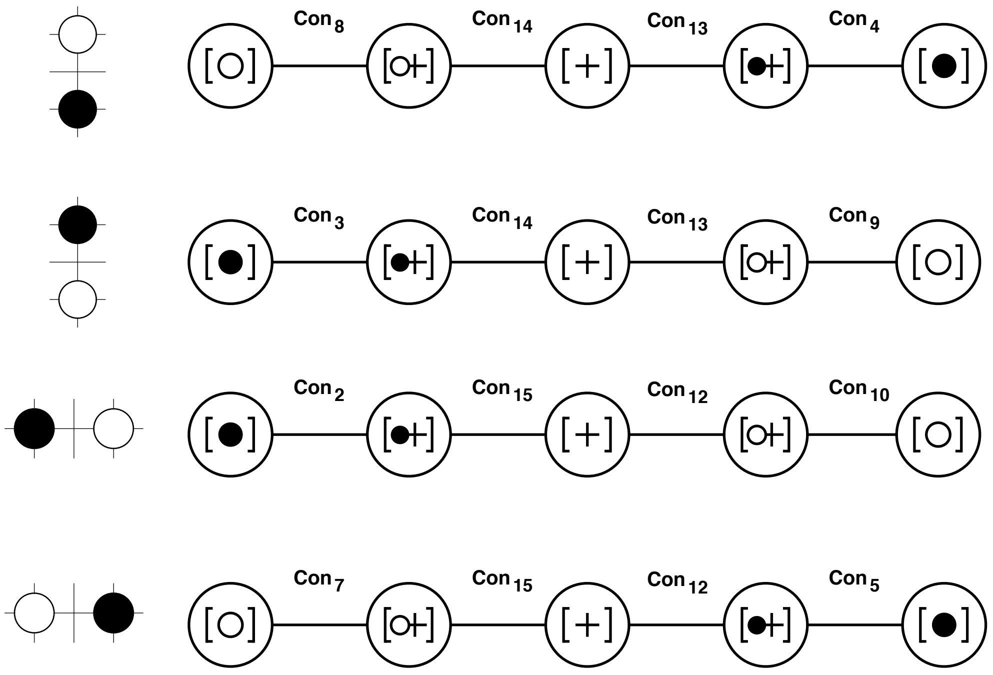


Figure 5.6: Four elements of class $\mathfrak{C}^1_{16}$, representing the four possible orientations of a black and white stone with one space between them.

describe two mirrored positions, since the same (re-oriented) plays were made to create them (Figure 5.3). A further benefit of using classes in this manner instead of simply making the structs of two mirrored positions the same is that the situation depicted in Figure 5.4 can be avoided. A robust representation of the Go board must be able to recognize the same game as seen from a different board orientation, as well as tell the difference between the same shape appearing multiple times under multiple mirroring in the same game.

5.1.4 Attention shift independence

In Section 4.4.2, I noted that classification of a Go position operated independently of the order in which the particular stones were "considered" by the Go-playing agent, but deferred an explanation of why until later, as such an explanation depends on

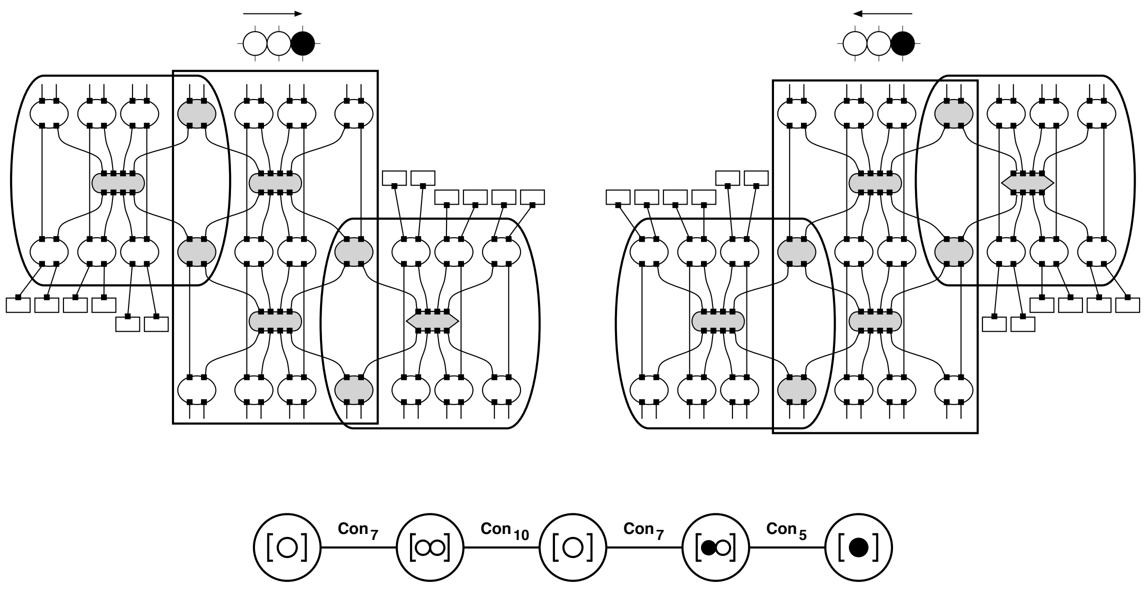


Figure 5.7: Two structs corresponding to attention shifting across the same position in opposite directions. Stone classes are indicated with boxes, and primitives that are part of proximity classes are shown in grey. The order the stones are added to the struct does not affect the overlap graph of resulting classes: both the struct on the left and the struct on the right result in the same overlapping level 0 classes.

the structure of classes that had not yet been presented.

The key to attention shift independence is that different (but similar) structs are elements of the same class: Figure 5.7 illustrates this. However, it is obvious that all of a stone's neighbours must be added to the struct before awareness of that stone is dropped, or certain relationships will be left out. In practice, there are several ways of scanning the board region that are acceptable, the simplest of them being a simple pan from left to right. Any scan that progressively adds information about immediate neighbours without "skipping around" will lead to the present classes overlapping on the same constraints.

5.1.5 Overlapping hierarchies

Another aspect of ETS that makes it well-suited to modelling Go is its support for a hierarchy of *overlapping* classes. Stones on the Go board influence each other in complex ways, and allowing the same stone to be part of multiple groups (high-level classes) is a natural way to account for these multifaceted relationships. To get a sense of this, look at the extended example at the end of Chapter 4: the "wall" constructed by white to keep black pinned in the corner is also likely to be part of some piece of white's territory.

In [4], Bouzy discusses the techniques used by his program INDIGO to represent spatial patterns. He writes,

> Human Go players are much stronger than the computer. The human skill in Go is mainly due to the intuitive knowledge about space. Therefore, Go programmers must observe human Go players and mimic them. In this meaning, Go computational models are cognitive models. [Experiments on Chess and Go have shown that]... [human] experts use "chunks". [4, p. 2]

The spatial model presented in [4], which, formally, is based in set theory, bears some resemblance to the one presented here in Chapter 4. Bouzy's *groups* (analogous to my level 2 classes) are formed iteratively out of small "patterns" of nearby (same coloured) stones (similar in scope to my level 1 classes). However, there are some key differences, primarily to do with overlap[1]. Bouzy's groups do not overlap one another: two proximate groups don't share one small pattern and instead each contain their own (colour-specific) instance of it. The relationships between groups are calculated externally and simply determine if one group is "stronger" than another. Essentially, at a high level the board is decomposed into a graph with each node representing a

[1]Another important difference is that my model includes temporal information, while Bouzy's does not.

group (which can be either black or white but not mixed) and each edge indicating that the two groups are adjacent, and which is stronger. Contrast that with level 2 classes that specify *how* two groups are interrelated, based on the sharing of (i.e., overlap on) particular level 1 classes elements.

Bouzy has moved away from spatial reasoning and towards Monte Carlo methods in his more recent work on Indigo [32]. Despite this, my intuition is that his statement that Go programmers can learn from the superiority of human spatial reasoning is accurate. However, my model of Go suggests a different approach that relies on "chunks" realized as overlapping hierarchies.

It is important to note that ETS classes contain temporal overlap as well as spatial; that is, classes overlap each other at different times as well as in different places. For an illustration, again refer to Figures 4.47-4.51, in which the dark arrows denote a class element that overlaps a new, temporally subsequent class element. Also, Figure 4.25 shows an overlap graph where some of the overlap is between classes that turn into one another as time passes: the white stone class overlaps the stone capture class, which in turn overlaps the resulting empty vertex class.

5.2 Psychological plausibility

We turn again to [40], Reitman's key study on Go players' memory[2]. Reitman's results were consistent with the hypothesis that the expert Go player she tested remembered Go positions as a collection of *overlapping* "chunks" (groupings of "related" stones).

[2]Some details of the study are described in Sections 2.4 and 4.7.2

A memory “chunk”, first described in Miller’s classic paper on “The magical number seven”, is the name given to a single unit held within short-term memory. The experimentally determined limit of human short term memory appears to be “seven plus or minus two” chunks [53]. A key property of these chunks is that they are of variable size: if a person can encode some amount of information into a single cohesive unit, that whole unit becomes one of the seven (plus or minus two) elements. The expert players in [40] and in the earlier Chess studies described in [41] vastly out-perform the beginners at the task of game position recall not because they can remember more chunks, but because their chunks are bigger, that is, they encompass more of the board. A rank beginner, on the other hand, might have chunks the size of a single stone or playing piece.

Newell and Simon neatly define a chunk as “any configuration that is familiar to the subject and can be recognized by him”. [41, p. 781] This definition is of particular interest here as it includes the concept of recognition. In my Go model, the natural candidates for “chunks” are *class elements* (and recognizing that a particular element is part of a given class is precisely the main task of an ETS classifier). Level 1 classes encompass precisely the kinds of board positions Reitman found her subject used in his chunks[3]. Consistent with her findings that chunks overlap, level 1 classes overlap each other and can be composed to form larger positions.

That a player’s chunks increase in size as his skill progresses echos the multi-leveled structure of my model. At the start of the “learning task”, an ETS-based learner classifies structs according to (small) level 0 classes. As learning progresses, new levels are added, each one with classes that correspond to more information covering larger parts of the board. ETS also has a built-in mechanism for “decoding” chunks

[3]In fact, Figure 4.43 and the subsequent examples were based on a position from Reitman’s paper, so as to demonstrate precisely this property of the model.

and retrieving the contained information: because class descriptions are *generative*, recalling which class a shape came from allows you to *reproduce* the lower-level class elements and project all the way from one high-level chunk to a low-level vertex-by-vertex realization.

Another study on the ability of Go players to "replay" games that they are shown also demonstrated that expert players have very accurate recall abilities. In [54], one expert player was asked about how he remembered the game he was asked to re-create (emphasis mine):

> The subject reported that... he created a dialog of the 'story' as the moves were added to the board. Meaningful moves were remembered because they made sense in the context of the story. A few moves did not make sense with respect to the story and therefore stood out, making it easier to remember them. Although the subject was not explicitly asked, it would seem that the dialog he created indicates that there was an *element of prediction associated with the next move.* [54, p. 6]

One interpretation of this explanation is that the "story" the player identified corresponds to a high-level class that he has learned. The expected moves are those that are consistent with the way that class generates elements, which also accounts for the important observation that his way of storing the game seemed to be predictive.

5.3 Towards an ETS Go engine

This section discusses how the model presented in this thesis might be used in a system that plays Go. Developing such a system is a two-part process: first, *learning* must be performed in order to populate the outlined class hierarchy with many more classes. Then, the resulting class system information can be incorporated into a *class-driven* Go program. The first task, learning, is a bottom-up process, as new classes are incrementally assembled based on previous-level components. The second task, choosing moves, is top-down, in that the unfolding of high-level classes in a particular game drives the formation of lower-level classes, down to the level of stones being placed.

5.3.1 Learning

This section does not describe how an ETS-based learner actually works, as the development of algorithms and techniques for learning ETS classes from data is a general problem that is separate from how such algorithms will be put to use. It is sufficient for the present purpose to assume that learning can be done in an efficient manner, and confine my discussion to what kinds of classes need to be learned from what kind of training data[4].

As the goal is to create a Go engine that depends entirely on classes, clearly a lot of Go experience must be collected, in the form of many classes (mostly at level 2 but some at level 1) describing the complete evolution of territory shapes and dead groups. The outlined class framework should be a sufficient scaffolding to support learning many such classes. Learning must also be symmetric: that is, classes should be either stored for both colours, or an efficient mechanism of "colour-swapping" a given class should be included.

Such classes could be discovered via unsupervised learning based on the records of expert-level games. It is quite trivial to produce structs from game records, and "paving" a struct with level 1 classes and associated level 2 constraints should be

[4]See [12] Section 10 for a preliminary discussion of learning.

fairly straight-forward, given some general ETS structure-matching algorithms. In order to determine which level 2 class elements are present in a given game, it would be best to start from the endgame and generate the struct “in reverse”, assuming that each territory, dead group, and captured group corresponds to one class element.

Although only classes up to level two have been presented in this thesis, it is possible that still higher class levels might be learned, even to the point of stage ascension. That this Go model has not explicitly defined level 3 classes does not rule them out.

5.3.2 Playing Go

At a high level, the proposed Go-playing methodology is the same as that traditionally used to play games in that, for each play, the current board position must be evaluated, and then a move selected. The main difference is that this entire process is *class-driven.*

The Go-playing program first inductively recognizes the various class elements present in the current position. The program should try to identify the level two classes in various stages of formation that are present. Candidate moves are then supplied by the generating systems of these classes, and the chosen move is one that does the best to further the evolution of favourable class elements while disrupting the evolution of unfavourable ones. This is the advantage of having a temporal representation: identification of a class element at an early stage of its evolution allows the anticipation of future circumstances.

5.3.2.1 Position evaluation

As is the case with human players, the process of choosing a move to play begins with an examination of the current state-of-affairs, in this case by identifying all present class elements. For most interesting (that is, non-quiescent) positions, the classification process does not just identify a single class for each group of stones. Instead, several (overlapping) classes in the process of being formed should be identified. Exactly how many should be discovered is an important question that must be resolved for optimization purposes: too few and the program will fail to "see" certain eventualities and make mistakes, and too many and it will be paralyzed under the weight of complexity (an empty board is, after all, the initial phase in the formation of *every class*). One possible solution might be to sort known classes based on the similarity of their constraints, and only detect those possible classes whose early stages of formation are highly similar to the position already on the board. Another mechanism for narrowing the range of classes to consider might be the use of even higher-level classes to control the unfolding of level 2 classes: these classes would produce level 2 elements from level 2 classes that are in some way "more important".

Figures 5.8 and 5.9 illustrate how a particular board position can provide the "context" for two different results, one that favours black, and one that favours white. This is a situation where the constraints that capture the left-most position in Figure 5.8 are present in the generating systems of (at least) two very different classes: the upper graph, representing the initial position, serves as predecessor for both lower graphs, meaning that the lower graphs can be created from it via one operation. From a class-generating point of view, the upper graph might represent the context part of a constraint that exists in two (or more) level 2 classes, and the lower graphs

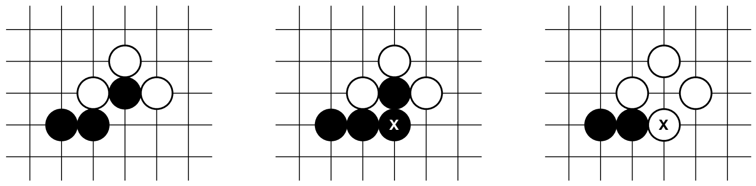


Figure 5.8: The position at left can be resolved favorably for black (middle position) or for white (right position), depending on who plays first.

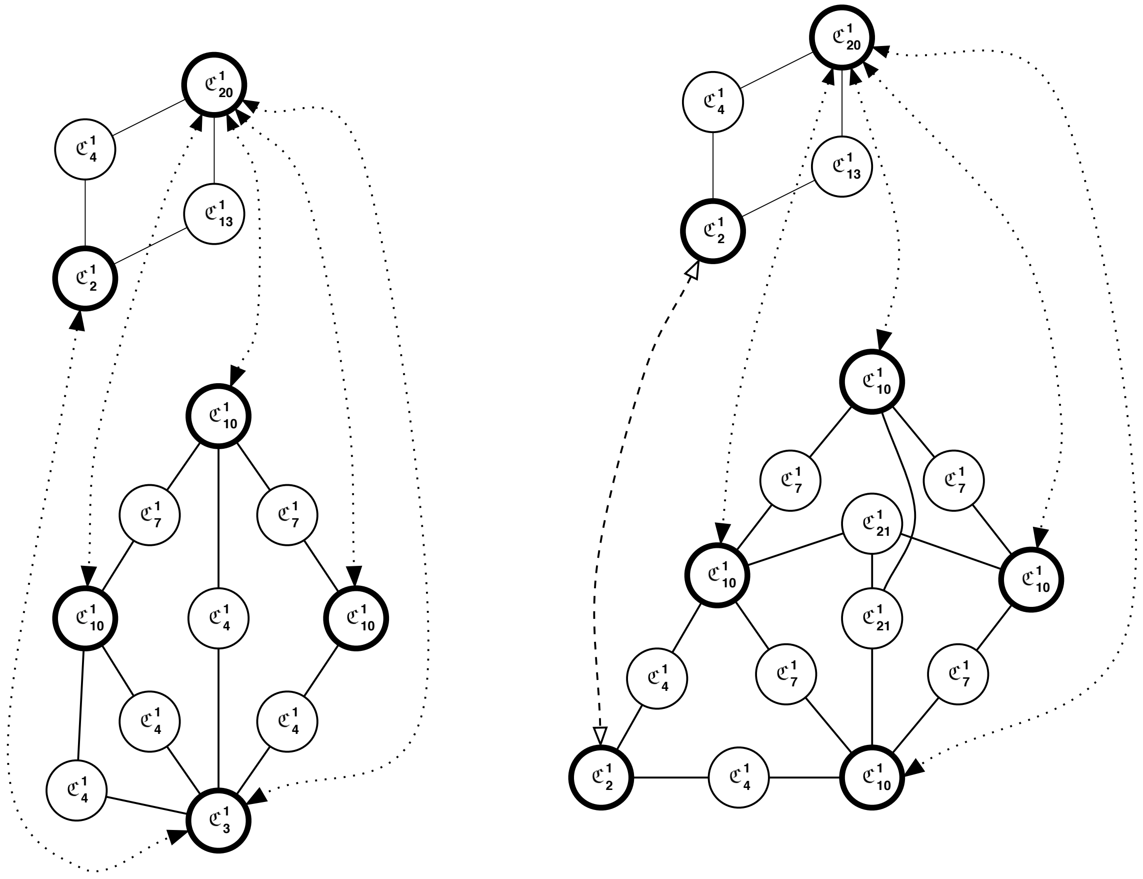


Figure 5.9: Overlap graphs corresponding to the two ways the position in Figure 5.8 can be resolved. Arrow notation is the same as in Section 4.7.3.

are the bodies of those two constraints. Note that previously unidentified level 1 classes pictured here are $\mathfrak{C}^1_{20}$: one-stone in atari, and $\mathfrak{C}^1_{21}$: white one-point jump.

In general, the process of recognition need not be done "from scratch" at every turn, since each play by the opponent is likely to affect only some part of the board. Indeed, if the play represents the next step in the formation of some *anticipated* group (that is, the corresponding level 2 class that the new play perpetuates has already been identified at some previous stage), then the present class elements might change very little from move to move. Thus, while fully classifying a position is likely to be computationally intensive, a lot of work is saved thanks to the temporal nature of a representation. Contrast this with traditional machine-learning approaches: you cannot feed a position into a neural network and save any time in producing the output because that position is very similar to the one that was previously processed.

It is obvious that human players maintain information on already classified positions from move to move. While it is true that a human player can "take over" in the middle of a game already in progress, to do so requires that the player study the board for longer than they would if they had been involved in the game up until that point. No human player purposefully forgets everything and approaches the board afresh every move!

5.3.2.2 Move selection

Once classification of the current position is complete, the program must choose a move to play. Candidate moves are those that push the or more of the present classes towards the next stage of their evolution. In other words, candidate moves are "proposed" by *the class generating system*: this is the key element that makes the proposed framework a *class-driven* Go player. This is a natural way to select moves because the goal of each move is to further develop some piece of the program's territory and/or hinder its opponent. Towards the second goal, the program might choose a disruptive move that interferes with the formation of a class that is favourable to its opponent: adding a stone in the right place might push the nascent position from one class ("black's territory") to another ("white captures").

Naturally, this kind of move selection depends heavily on the ability of the program to determine which classes are favourable, since it must know which classes to attempt to perpetuate. A simple solution is to examine what each class ultimately produces, and select classes that lead to large territories or large enemy captures over less productive classes. Classes that result in enemy territories for the program's own stones being captured should be disrupted.

More subtly, higher-level classes could be employed to make these selections. These classes would begin to encompass the concept of strategy, since they would seek to produce favourable positions on a board-wide scale. I will not conjecture as to the structure of these strategy classes, but to what extent they can be learned automatically should be examined.

This kind of move selection does include a kind of "look ahead", since it seeks to influence the future development of territories, but because it operates on high-level classes, it is more abstract than the traditional game-tree search that must treat every individual play as a separate entity to be considered.

5.3.3 Opening game

The opening game is often treated as a special case by both human and computer players. Opening moves are typically played *joseki*, or "from a book". For human players, this means that they memorize certain opening combinations. Computers play much more literally "by the book" in that they typically employ a library of opening plays.

A class-based Go program would also employ a kind of book to handle the opening, but it is important to recognize what this book contains: opening plays should reflect the initial constraints used to form desirable class elements. The opening game is a kind of environment process that triggers the formation of several stone and group processes.

Even special cases are heavily influenced by the workings of the class hierarchy.

5.3.4 Evaluating play

Clearly, the main goal of any Go program is competent play, and obviously the strength of the proposed program should be tested, both against human players other programs. More subtly, a second important goal of the proposed program is human-like play. As was discussed in Chapter 2, Go programs, and game playing programs in general, often play “strange” moves and have very inhuman weaknesses that human players can eventually learn to exploit.

I propose that the “human-like” properties of a program’s play could be evaluated in a kind of Turing test [55]: have a human player play several games against unseen opponents, some of them human, the others various Go programs, and see how accurately that player can tell the former from the latter. The human opponents would have to include a range of skill levels on par with the various programs, so as not to give their identity away with either too strong or too weak play.

Chapter 6

Conclusion and future work

6.1 Conclusion

In this thesis, I have presented a simplified overview of the ETS formalism and developed a preliminary ETS representation of the game of Go. I designed seven primitives and a single primal class, which together form various structs corresponding to an idealized Go player's awareness of the game board. On the basis of these structs, I designed three class levels dealing with increasingly complex elements of the game. The aim of this work was twofold: first, to apply structural representation to the game of Go and examine what benefits it brings, and second, to test the use of ETS in a new domain.

The application of ETS to any new area necessitates rethinking that area's basic data, because event-based representation requires that data be packaged differently than it is conventionally. ETS is meant to capture the formation of objects, and in the case of Go, this representational requirement recasts the ubiquitously discussed

Go *shape* as a process of playing stones that evolve into a familiar pattern. This view of shapes as evolving processes, taken together with the generative nature of ETS class representation, seems to be a natural mechanism to use for move selection in a Go-playing program.

The non-opaque structure of ETS classes means that class definitions can both be learned automatically and designed by an expert player, allowing for flexibility that is not available in other inductive learning paradigms (a Go expert could not improve a neural network by manually editing the edge weights, for example).

The ETS formalism appears to be a natural tool for representing Go, largely because the formalism's hierarchical structure is well equipped to capture the game's hierarchical structure. Lowest-level classes correspond to stones, i.e., elements that are *immediate* in their presence on the board. Elements of higher-level classes correspond to more subjective structures, because they attempt to capture patterns and groups that are judged by players to be meaningful/significant. Elements at each level naturally overlap one another, which is consistent with how human players think about the board.

Go also appears to be a useful test domain for ETS. This work helped to clarify that the natural way to model spatially-related patterns is to compose them of overlapping class elements. During the development of this work, a few refinements have been made to the ETS formalism, including a generalization on the definition of a struct-level constraint that allows for more robust generation of class elements that are heavily affected by their local environment. Had I been able to make use of that generalization in this thesis, it may have saved some trouble in designing primitives and level 0 classes. Essentially, to compensate for the older definition, which did not allow the insertion of "noise" primitives into a class element's working struct, I had to construct a rigid (and fairly uninteresting) struct level in order to support classes of stones that gave sufficient flexibility. Still, because the change to the definition is a generalization, the work in this thesis is not incompatible with it, but merely uses a subset of the available flexibility.

During the course of developing my Go representation, several classes of short-running and dynamic processes, e.g., those corresponding to the play and removal of stones, emerged. The ETS formalism as it currently stands does not specify much about the body of a *transform* [12, Part IV], but the above class elements seem to perform transformational functions. This lends credence to the notion that ETS transforms are simply classes of processes that happen to produce more short-lived and unstable elements.

6.2 Future work

As this thesis is a preliminary work on the uses of ETS for playing Go, it is clear that there is much to do, both with respect to refining the presented representation, and to using it in a Go-playing program. I sketched the workings of such a program in the previous chapter, so I will not discuss it in great detail here.

Some additions and improvements to the presented Go representation ought to be investigated:

- Reconsider the chosen primitives and resulting structs and level 0 classes on the basis of the new constraint definition. Perhaps less rigid and more organic structs could be created.

- Investigate the use and nature of class levels beyond the three presented in this thesis.
- Investigate the possibility of ascending to a new *stage* of representation on top of the high-level classes.

Once the representation scheme is finalized, it should be possible to construct a Go-playing program on its basis. Such an effort would include these tasks:

- Conduct learning to populate the class hierarchy. To do so might require the development of new ETS-based learning algorithms, which is a significant endeavour in its own right.
- Experiment with unsupervised learning based on game records, and with designing classes in cooperation with an expert Go player.
- Compare the performance of the developed program with existing Go programs, both in terms of playing strength and playing style.

More generally, one might take this work as a foundation for the following:

- Investigate how well the outlined framework agrees with how human players think about the Go board.
- Investigate the application of ETS to other games. Perhaps in other stone games, e.g, Othello, some of the low-level representation presented here could be re-used.

The development of a full Go-playing program on the basis of this thesis would be a very good test of ETS: it would transform the work presented here from a preliminary example into a full-fledged application of the formalism.

Bibliography

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 1st ed., Prentice Hall, 1995.

[2] J. Burmeister and J. Wiles, "The challenge of Go as a domain for AI research: a comparison between Go and chess," *Proc. 3rd Australian and New Zealand Conf. on Intelligent Information Systems*, IEEE Western Australia Section, Nov. 1995, pp. 181–186.

[3] American Go Association, "Mueller on computer Go's "revolutionary" advances," *American Go E-Journal*, vol. 8, no. 23, July 23, 2007.

[4] B. Bouzy, "Spatial Reasoning in the game of Go," 1996; http://www.math-info.univ-paris5.fr/∼bouzy/publications/SRGo.article.pdf.

[5] T. Huang, G. Connell, and B. McQuade, "Experiments with learning opening strategy in the game of Go," *Int'l Journal of Artificial Intelligence Tools*, vol. 13, no. 1, 2004, pp. 101–114.

[6] E.C.D. van der Werk, H.J van den Herik, and J.W.H.M Uiterwijk, "Learning to score final positions in the game of Go," *Theoretical Computer Science*, vol. 349, no. 2, 2005, pp. 168–183.

[7] E.C.D. van der Werk et al., "Learning to predict life and death from Go game records," *Information Sciences*, vol. 175, no. 4, 2005, pp. 258–272.

[8] M. Enzenberger, "The Integration of a Priori of Knowledge into a Go Playing Neural Network," 1996; http://www.cgl.ucsf.edu/go/Programs/NeuroGo-PS.html.

[9] M. Enzenberger, "Evaluation in Go by a Neural Network Using Soft Segmentation," *10th Advances in Computer Games Conf.* (ACG-10), Kluwer, 2003, pp. 97–108.

[10] N. Richards, D.E. Moriarty, and R. Miikkulainen, "Evolving Neural Networks to Play Go," *Applied Intelligence*, vol. 8, no. 1, 1998, pp. 85–96.

[11] B. Bouzy and G. Chaslot,"Bayesian generation and integration of K-nearest-neighbor patterns for 19x19 go," *IEEE 2005 Symposium on Computational Intelligence in Games*, G. Kendall and S. Lucas, eds., 2005, pp. 176–181.

[12] Lev Goldfarb, David Gay, "What is a structural representation? Fifth variation," *Faculty of Computer Science, U.N.B., Technical Report TR05–175*, December 2005; http://www.cs.unb.ca/∼goldfarb/ets5/index.html.

[13] L. Goldfarb, "On the foundations of intelligent processes I: An evolving model for pattern learning," *Pattern Recognition*, vol. 23, no. 6, 1990, pp. 595–616.

[14] L. Goldfarb, "On the concept of class and its role in the future of machine learning," *What is a Structural Representation*, L. Goldfarb, ed., in preparation, 2007; http://www.cs.unb.ca/∼goldfarb/ETSbook/Class.pdf.

[15] The British Go Association, 2004, "Introduction to the game of Go," 2004; http://www.britGo.org/intro/intro1.html.

[16] P. Shotwell, "The Origins of Go," *American Go Association*, 2002; http://www.usgo.org/resources/downloads/originsofgo.pdf.

[17] The British Go Association, "Go: The Most Challenging Board Game in the World," 1999; http://www.britgo.org/intro/booklet.pdf.

[18] M. Müller, "Computer Go," *Artificial Intelligence*, vol. 134, 2002, pp. 145–179.

[19] Nihon-Kiin, *Go: The Worlds Most Fascinating Game*, vol. 1, R. Kajiki and T. Konami, Trans., Tokyo, 1973.

[20] C.E. Shannon, "Programming a computer for playing chess," *Philosophical Magazine*, vol. 41, 1950, pp. 256–275.

[21] T. Hershman, "Chess: Man vs. Machine Plays Out," *Wired News*, Oct. 2002; http://www.wired.com/news/culture/0,1284,55839,00.html.

[22] F. Hsu, M.S. Campbell, A.J. Hoane, Jr., "Deep Blue system overview," *Proc. 9th Int'l Conf. Supercomputing*, ACM, 1995, pp. 240–244.

[23] C. Matthews, *Teach Yourself Go*, McGraw-Hill, 1999.

[24] Free Software Foundation, "GNU Go Documentation," 2004; http://www.gnu.org/software/gnugo/gnugo_toc.html.

[25] T. Wolf, "GoTools – the Tsume-Go program," 2001; http://alpha.qmul.ac.uk/∼ugah006/gotools/.

[26] N. Jacobs, "Relational Sequence Learning and User Modeling," Ph.D. thesis, Departement Computerwetenschappen, Katholieke Universiteit Leuven, 2004.

[27] G. Chaslot, "11th Annual Computer Olympiad," 2006; http://www.cs.unimaas.nl/Olympiad2006/.

[28] K. Chen, "Computer Go: Knowledge, Search, and Move Decision," *ICGA Journal*, vol. 24, no. 4, 2001, pp. 203–215.

[29] K. Chen, "Soft decomposition search and binary game forest model for move decision in Go," *Information Sciences*, vol. 154, no. 3–4, 2003, pp. 157–172.

[30] M. Müller. "Decomposition search: A combinatorial games approach to game tree search, with applications to solving Go endgames," *IJCAI-99*, vol. 1, 1999, pp. 578–583.

[31] B. Bouzy, "The move decision strategy of Indigo," *ICGA Journal*, vol. 26, no. 1, 2003, pp. 14–27

[32] B. Bouzy, "History and Territory Heuristics for Monte-Carlo Go," *New Mathematics and Natural Computation*, vol. 2, no. 2, 2006, pp. 1–8.

[33] B. Bouzy and T. Cazenave "Computer Go: an AI-Oriented Survey," *Artificial Intelligence Journal*, 2001, pp. 39–103.

[34] B. Bruegmann, "Monte Carlo Go," 1993; ftp://ftp.cgl.ucsf.edu/pub/pett/go/ladder/mcgo.ps.

[35] B. Bouzy and B. Helmstetter, "Monte Carlo Go Developments", *10th Advances in Computer Games Conf.* (ACG-10), Kluwer, 2003, pp. 159–174.

[36] S. Gelly and Y. Wang, "Exploration exploitation in Go: UCT for Monte-Carlo Go," *NIPS-2006, Online trading between exploration and exploitation*, December 2006.

[37] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo Planning," *Lecture Notes in Artificial Intelligence*, vol. 4212, 2006, pp. 282–293

[38] E. Berlekamp, D. Wolfe, *Mathematical Go Endgames, Nightmares for the Professional Go Player*, Ishi Press Int'l, 1994.

[39] J. Burmeister and J. Wiles, "AI Techniques Used in Computer Go," *Proc. of the 4th Conf. of the Australasian Cognitive Science Society*, Univ. of Newcastle, 1999.

[40] J.S. Reitman, "Skilled perception in Go: deducing memory structures from inter-response times," *Cognitive Psychology*, vol. 8, no. 3, 1976.

[41] A. Newell and H.A. Simon, *Human Problem Solving*, Prentice-Hall, 1972.

[42] A. de Groot, *Thought and Choice in Chess*, Mouton, 1965.

[43] D. Fotland, "David Fotland's Many Faces of Go", 2002; http://www.smart-games.com/manyfaces.html.

[44] G.F. Luger, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, 4th ed., Addison Wesley, 2002.

[45] L. Goldfarb, "Representational formalisms: what they are and why we havent had any," *What is a Structural Representation*, L. Goldfarb, ed., in preparation, 2007; http://www.cs.unb.ca/∼goldfarb/ETSbook/ReprFormalisms.pdf.

[46] M. Sipser, *Introduction to the Theory of Computation*, PWS Publishing Company, 1997.

[47] L. Goldfarb and I. Scrimger, "On ETS Representation of human movement," *What is a Structural Representation*, L. Goldfarb, ed., in preparation, 2007; http://www.cs.unb.ca/∼goldfarb/ETSbook/Walking.pdf.

[48] L. Goldfarb, I. Scrimger, B.R. Peter-Paul, "ETS as a tool for decision modeling and analysis: planning, anticipation, and monitoring", *2007 Decision and Risk Analysis Conf.*, 2007. Also submitted to the journal *Risk and Decision Analysis.*

[49] R. Waterfield, trans., *The First Philosophers: The Presocratics and the Sophists*, Oxford University Press, 2000.

[50] E. Hemingway, *For Whom the Bell Tolls*, Simon & Schuster, 1995.

[51] S. Falconer, "On the Evolving Transformation System Model Representation of Fairy Tales," master's thesis, Faculty of Computer Science, UNB, 2005.

[52] A. Kierulf, "Sensei's Library: Smart Game Format," 2007; http://senseis.xmp.net/?SmartGameFormat.

[53] G.A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *The Psychological Review*, vol. 63, 1956. pp. 81–97.

[54] J. Burmeister et al., "Memory performance of master Go players," *Games in AI Research*, H.J. van den Herik and H. Iida, eds., Universiteit Maastricht, 2000, pp. 271–286.

[55] A.M. Turing, "Computing machinery and intelligence," *Mind*, vol. 59, 1950, pp. 433–460.

Vita

Candidate's full name:	James Ian Scrimger
Universities attended:	University of New Brunswick New Brunswick, Canada 2004-2007
	Mount Allison University New Brunswick, Canada 2000-2004 B.Sc. (Computer Science and Philosophy)

Conference presentations and poster sessions:

L. Goldfarb, I. Scrimger, B.R. Peter-Paul, "ETS as a tool for decision modeling and analysis: planning, anticipation, and monitoring," *Decision and Risk Analysis Conf.*, 2007. Also submitted to the journal *Risk and Decision Analysis*.

L. Goldfarb, B.R. Peter-Paul, I. Scrimger, "ETS Representation of Human Movement," *UNB Computer Science Research Expo*, April 4, 2007.

Tech reports:

L. Goldfarb and I. Scrimger, On ETS Representation of human movement, Technical Report TR07-184, Faculty of Computer Science, UNB, 2007