PHANTOM GO

Joris Borsboom

Master Thesis MICC-IKAT 07-09

Thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science of Artificial Intelligence
in the Faculty of Humanities and Sciences
of the Universiteit Maastricht

Universiteit Maastricht
Maastricht ICT Competence Centre
Institute for Knowledge and Agent Technology
Maastricht, The Netherlands
August 2007

Thanks for advice, support and patience:
Peter Brouwers
Guillaume Chaslot
Jaap van den Herik
Jahn-Takeshii Saito
Maarten Schadd
Ben Torben-Nielsen
Jos Uiterwijk
Mark Winands

Summary

In the last decade, Monte-Carlo methods have considerably increased the playing strength of computer Go programs. In particular, Monte-Carlo Tree Search algorithms, such as UCT, have enabled significant advances in this domain. Also the combination of Monte-Carlo methods with domain-specific knowledge has produced good results. While much research has been done on Go, so far not much work has focussed on Phantom Go. We asked ourselves how we could apply the existing knowledge about Monte-Carlo methods to computer Phantom Go in order to produce a strong playing program.

The game of Phantom Go is a variant of Go complicated by the presence of hidden information. The opponent moves are not given. However, if an illegal move is attempted this is announced and the player can try to move again. Thus, a player can try to guess the precise location of the unknown opponent stones. Phantom Go is usually played on a 9×9 board.

The only work on Phantom Go known to us has been published by Tristan Cazenave in 2006. He used a very basic Monte-Carlo method. As a sequel to this research we applied three Monte-Carlo approaches which have been successful in regular Go, viz. (1) all-as first MC, which was used by Cazenave, (2) simple MC evaluation of moves, and (3) UCT. Moreover, different ways of dealing with the unknown stones on the board and combinations of algorithms were devised. Two ways of scoring games were tested. Furthermore, we tried to enhance UCT by integrating domain-specific knowledge. This led to a total of eight Phantom-Go players.

To our surprise none of the MC methods tried outperformed all-as-first. Simple MC evaluation needed too much simulated games to produce good play. UCT plays at a comparable level to all-as first. Tree search does bring an improvement in later stages of the game, but it suffers from a number of drawbacks. Below we list three of them.

- UCT tends to overestimate the value of a position. This leads to careless play.
- Using the values produced by UCT to account for unknown opponent stones influences the construction of the search tree in such a way that bad moves with an incorrect positive evaluation can persist.
- UCT plays disconnected moves in the opening stages of a game, thus giving the player a starting position worse than all-as-first methods, which play strongly connected.

Especially the last problem is a big drawback. We got around this problem by defining a hybrid player, which starts by using all-as-first evaluation and switches to UCT later. This does produce better play than pure all-as-first.

When trying to integrate domain-specific knowledge within MC methods, our experiments using the total gain in influence as a heuristic for Phantom Go were unsuccessful. While our approaches to combining knowledge with UCT are valid, the heuristic was not good enough to produce better results.

A big improvement over earlier methods was made by exchanging territory scoring of simulated games by 0/1 scoring. When a game is either valued as a win or a loss, the algorithms tend to focus more on the moves important to winning the game. This leads to better strategic play. However, it takes longer for the algorithms to focus on important moves, as there is no clear indication of how good a move is. When using territory scoring this is much more clearly defined. Thus algorithms using territory scoring exhibit better tactical play.

In conclusion we may state that while some of our approaches have produced automated players with a reasonable playing strength, we have not been able to improve the basic method used by Cazenave significantly. More research will be needed to overcome the difficulties we encountered and produce stronger Phantom Go-playing programs.

Samenvatting

In de afgelopen jaren is er door het gebruik van Monte-Carlo methoden veel vooruitgang geboekt in computer Go programma's. In het bijzonder noemen we de opkomst van Monte-Carlo algoritmen die gebruik maken van zoekbomen, zoals UCT. Ook de combinatie van Monte-Carlo methoden met domeinspecifieke kennis heeft tot goede resultaten geleid. Hoewel er veel onderzoek is gedaan naar computer Go, is er tot nog toe maar weinig aandacht besteed aan Phantom Go. Wij hebben onszelf tot doel gesteld de bestaande kennis over Monte-Carlo methoden toe te passen op Phantom Go, en een sterk spelend programma te produceren.

Het spel Phantom Go is een gecompliceerdere variant van Go die met verborgen informatie werkt. De zetten van de tegenstander worden niet openbaar gemaakt. Echter, wanneer een speler een zet probeert te doen die niet mogelijk is, wordt dit aan beide spelers kenbaar gemaakt en mag de speler een nieuwe zet proberen te doen. Zo kan een speler proberen af te leiden waar de stenen van zijn tegenstander zich bevinden. Phantom Go wordt meestal gespeeld op een bord van 9×9.

Het enige werk dat wij kennen over Phantom Go is gepubliceerd door Tristan Cazenave in 2006. Hij gebruikt een vrij algemene Monte-Carlo methode. Als een vervolg op dit onderzoek hebben wij drie Monte-Carlo benaderingen die succes gehad hebben in Go toegepast op Phantom Go, te weten (1) 'all-as-first' MC evaluatie, gebruikt door Cazenave, (2) MC evaluatie van alle mogelijke zetten, en (3) UCT. Verschillende manieren om rekening te houden met de onbekende stenen van de tegenstander zijn hiermee gecombineerd. Ook zijn twee verschillende methoden getest om de waarde van een uitgespeelde positie te bepalen. Verder hebben we geprobeerd enige van deze methoden te verbeteren door ze te combineren met domeinspecifieke kennis. Dit leidde tot een totaal van acht Phantom-Go spelers.

Tot onze verassing behaalden geen van de MC methoden een beter resultaat dan all-as-first. MC evaluatie van alle mogelijke zetten heeft te veel gesimuleerde spellen nodig om tot werkelijk goed spel te komen. UCT speelt op een vergelijkbaar niveau als all-as-first. Het gebruik van zoekbomen leidt tot betere zetten later in een partij, maar leidt aan een aantal tekortkomingen. We noemen er drie.

- Zoekboom methoden zoals UCT hebben de neiging de waarde van een positie te hoog in te schatten. Dit leidt tot een onvoorzichtige stijl van spelen.
- Het gebruik van de evaluatiewaarden van UCT om rekening te houden met de onbekende stenen beïnvloedt de manier waarop de zoekboom uitgebreid wordt. Dit leidt tot een zoekgedrag waarin slechte zetten met een, toevallige, goede evaluatie deze onterecht kunnen behouden.
- UCT opent niet sterk. Omdat het uitgaat van een zoekboom waarin stenen altijd verbonden kunnen worden, speelt het veel losse stenen verspreid over het bord. Dit is geen goede tactiek voor Phantom Go.

Vooral het laatste probleem leidt tot slechter spel. We hebben dit omzeild door een hybride speler te introduceren, die eerst all-as-first evaluatie gebruikt en later overgaat op UCT. Dit levert wel betere resultaten op dan enkel all-as-first.

Bij onze pogingen domeinspecifieke kennis te combineren met MC methoden, zijn onze experimenten met een heuristiek gebaseerd op 'influence' niet succesvol geweest. Hoewel onze benadering wel deed wat wij verwachtten, was de heuristiek niet goed genoeg om betere resultaten te geven.

Een grote verbetering ten opzichte van eerdere methoden kwam wel toen wij de waarde van een gesimuleerd spel niet langer lieten afhangen van het aantal bezette punten, maar slechts keken of de speler gewonnen of verloren had. In dit geval richten de algoritmen zich meer op de zetten die belangrijk zijn om het spel te winnen. Dit leidt tot beter strategisch spel. Echter, het duurt ook langer voor het algoritme doorkrijgt welke zetten belangrijker zijn dan andere. Dit leidt tot enigszins slechter tactisch spel.

Ter conclusie kunnen we stellen dat hoewel sommige van deze benaderingen programma's van een redelijke sterkte hebben opgeleverd, het ons niet gelukt is een veel betere speler te produceren dan met de methoden die gebruikt werden door Cazenave. Meer onderzoek zal nodig zijn om deze problemen op te lossen of te omzeilen, en sterkere Phantom Go spelende programma's te ontwerpen.

Contents

Chapter 1

Introduction

In this chapter we introduce the field of computer game-playing, and give an introduction to methods relevant for this project. We explain why we chose to do research on Phantom Go, and give an overview of the previous research relevant to this thesis. Finally, we give a problem statement and five research questions.

Chapter contents: Games, Phantom Go, Monte Carlo, Related Work, Problem Statement, Research Questions

1.1 Games and AI

For many years now, board games have been used as a testbed for Artificial Intelligence (Van den Herik, Uiterwijk, and Van Rijswijck, 2002). Because of the structured and deterministic nature of these games they are ideal for experiments about reasoning behaviour in machines while they are still complex and difficult enough to pose continuously new problems. Since in most games a human may challenge a computer in a one-to-one contest, games have become a test of intelligence between man and machine.

As AI techniques progressed we saw a shift in the kind of games that were studied. No sensible person will play tic-tac-toe against a computer more than a few times. The kind of games focussed on seems to increase in complexity. Minimax algorithms combined with good evaluation functions have long been the dominant method of computer game-playing. The results have been impressive. Chess has been the holy grail of AI researchers since the 1950's. With Kasparov's defeat by DEEP BLUE, that point seems passed (Schaeffer and Plaat, 1997). The focus shifted then to games in which minimax algorithms will not yield very good results. There are two main causes for minimax to fail: complexity and uncertainty.

Complexity comes into play if games have a high branching factor, resulting in large game trees, and if there is no simple way of determining whether a position reached after a number of moves is a 'good' position. In this situation minimax is at a loss because it needs an incredible amount of time before being able to decide on any 'good' move. Go is a notorious example, and even though much research has been done, and is being done, computers have still not progressed beyond the level of a good amateur. Humans are much better at this game, as it seems mainly because they recognize patterns in the game instead of analyzing moves.

Uncertainty is a factor in games in which some information is not disclosed to all players. Card games such as Bridge and Poker are examples. Kriegspiel, a Chess variant in which the player is not able to see what the opponent does, is another example. Minimax fails here because if it considers a move, it will have to consider it for all possible situations, instead of only the 'real' situation. Humans think about these possibilities at a more abstract level, which might not yield perfect play, but provides solutions which are usually quite good.

New techniques have to be tried to overcome these difficulties. Hence we arrive the topic of this thesis: Go with uncertainty.

1.2 Go Rules and Terminology

Go is an old Chinese board game. It has been played in China, Japan, and Korea for over 3000 years. Two players, Black and White, take turns in placing one stone on a cross of lines (an *intersection*).

These stones will not be moved again, unless they are *captured.* It is allowed to pass. Stones need to be connected, horizontally or vertically, to at least one free position to survive. The stone is then *alive.* Completely surrounded stones are taken off the board. They are captured, or *dead.* Stones of the same colour that are in adjacent positions, horizontally or vertically, form a *chain.* The stones in a chain live and die together. They are alive as long as one of them is adjacent to an empty position. The empty intersections a chain is connected to are called *liberties.* When a chain has only one liberty left it is in *atari.* This means it could be captured on the next move. It is somewhat similar to 'check' in chess.

Suicide is illegal in Go. This means that it is not allowed to play a move if as a result of this move the stone, or a chain of the player's own colour, has no liberties left. However, if a move captures opponent stones they are taken off the board first. The stone will have liberties then, and therefore the move is legal.

If a move would result in the same board pattern as existed before the opponent made his[1] move, a player has to abstain from that move and should make a move elsewhere. This is called *ko* and is necessary to avoid endless repetitions and stalemates.

Chains can also be called *dead* while they are still on the board. They may technically still be alive, but it is used to indicate that they will never survive. Chains can also be *unconditionally alive.* This means that they can never be taken. To understand this we need to introduce the concept of *eyes.* While different definitions of what exactly an eye is are being used, it is essentially an empty intersection that is surrounded by stones of one colour. An eye that can be threatened by capturing one or more of the surrounding stones is called a *false eye.* Correspondingly, an eye that cannot be threatened is called a *real eye.* When speaking of eyes, usually real eyes are meant. A chain needs two eyes to be unconditionally alive. A chain with one eye can be caught by surrounding it completely, and finally playing in the eye. A chain with two eyes cannot be captured, because to play in one of the eyes would be suicide. The chain always has the other liberty left.

The game is over when both players pass. The winner is the player that has surrounded and occupied the most intersections. There are different ways of counting this; we will be using *Chinese rules.* For the complete Chinese rules see Davies (1992). According to these rules, all intersections that are either occupied or completely surrounded by alive chains are counted. Since Black plays first, to White's score a certain number of points are added. This is called *komi.*

1.3 Phantom Go

Phantom Go is a two-player game often played as an entertaining variation by regular Go players. It is the equivalent of Kriegspiel for Chess.

Phantom Go is a variant of classic Go in which the players do not see the opponent's moves. Two players each have their own board, on which only their own stones are visible. A referee is needed. He has a reference board on which the actual state of the game is maintained. The players communicate their move decisions to the referee. The referee has the following replies to a suggested move:

- illegal (already occupied, suicide, ko); after which a new move may be suggested.
- legal; the move is played.
- stones taken; the stones taken are removed from the opponent's board, and the number of taken stones is announced to the player.

After both players pass, the game ends and the final score is computed on the reference board, according to Chinese rules. These rules are according to Cazenave (2006). In practice, other sets of rules are used as well, for instance the reply 'suicide' or 'ko' in contrast to a 'normal' illegal move, or the announcement 'atari' when stones are in danger of being captured. Such rules will not be used at this stage.

While the described game may seem absurdly difficult, in practice it is not. Usually the game is played on smaller boards, 9×9 for instance, and much information about the opponent's position will turn up during a match. We have also decided to play on a 9×9 board. If a move is declared 'illegal' by the referee, a player knows something about the position of opponent stones: most probably one of the opponent stones is on the position he tried. However, blunders will be made, much to the benefit of spectators of course, who can see the reference board.

[1]For brevity we use 'he' and 'his' instead of 'he or she' and 'his or her'.

1.4 Monte-Carlo Methods

Monte-Carlo (MC) methods are a widely used class of algorithms for simulating the behaviour of various physical and mathematical systems. They can also be used for other computations. MC methods are basically a form of statistical sampling. In MC games moves are evaluated by playing games of random or pseudo-random moves and assigning values to moves according to the results of these games. This value is usually some sort of mean result. How exactly these values are computed, and which moves to consider at a particular point of evaluation, are the main differences between different algorithms.

The major advantages of this approach are the fact that very little domain knowledge is needed for these algorithms, the rules of the game will usually do, and that it is well equipped to deal with uncertainty. Also, complex games are less of a problem, because not every branch in the game tree has to be explored in order to evaluate a move. Monte-Carlo methods have also proved very effective in 'normal' (complete-information) Go. Therefore, they seem especially suited to the problem of Phantom Go.

1.5 Related Work

In this section we describe existing work on Kriegspiel (Subsection 1.5.1), Phantom Go (Subsection 1.5.2), and on related subjects such as Monte-Carlo Go (Subsection 1.5.3), knowledge-based approaches to Go (Subsection 1.5.4), and combining Monte-Carlo and knowledge-based approaches (Subsection 1.5.5).

1.5.1 Kriegspiel

Kriegspiel is a game closely related to Phantom Go. Much research in this area has been about taking into account all possibilities and still trying to generate optimal play. Working with *Metapositions* has been studied for Kriegspiel by Sakuta and Iida (2000), Bolognesi and Ciancarini (2006), and Bolognesi and Ciancarini (2007). This would be a possible approach for Phantom Go also. However, the drawbacks of this are substantial: because a node is kept for every possible state of the board, the complexity explodes. Therefore, work has been focussed on endgame computations. Attempts have been made to keep the complexity under control, as described in the context of the DARKBOARD program (Ciancarini and Favini, 2007) or by the work by Parker, Nau, and Subrahmanian (2005). DARKBOARD uses many heuristics and limitations of information that are specific to Kriegspiel. While the approach of Parker *et al.* is more easily applicable to other fields, it relies heavily on the use of an evaluation function. While such approaches are imaginable, it remains to be seen whether this would lead to reasonable play. This especially holds for Phantom Go, since even for regular Go, good general evaluation functions are still an open question (Bouzy, 2002).

In essence, most of the work in Kriegspiel has focussed on creating an *optimal* player in the sense used by Van Rijswijck and Müller (2001). They make a distinction between an optimal and a maximal player. The idea here is that maximal players attempt to exploit the opponents' perceived weaknesses, but in doing so, become vulnerable to exploitation themselves. In essence a maximal player is the ultimate greedy player. A game-theoretically optimal player is immune to exploitation, but in turn, also incapable of inflicting it. In games with much uncertainty this may lead to very careful play, because it always assumes a worst-case scenario. As shown by Parker, Subrahmanian, and Nau (2006) for Kriegspiel this may not be the most desirable approach.

1.5.2 Phantom Go

The only work on Phantom Go known to us is done by Cazenave (2006). Also employing MC techniques, this work proposes a number of approaches to make the game susceptible to the standard ways in which MC methods are used in games. Most important amongst these are the following.

- If the referee replies 'illegal' to a proposed move, put an opponent stone in that position. While this is not always correct, it is what all human players do, and a reasonably good heuristic.
- Before performing MC evaluation, compute how many of the opponent stones you do not know about. This is done by subtracting the number of opponent's stones on your own board from the number of moves played. These 'unknown' stones are then distributed randomly over the board.

Cazenave (2006) uses an all-as-first MC algorithm. This heuristic means that the value of any move played during the course of a sample game is updated with the same value. While this approach yields a large number of values for moves very quickly, which is desirable in MC evaluations, it tends to lead to selecting moves that are generally a good idea, but not necessarily in the current state of the game. According to Bouzy and Helmstetter (2004) the reason for this is that the evaluation of a move from a random game in which it was played at a late stage is less reliable than the evaluation of a move which is played at an early stage.

1.5.3 Monte-Carlo Go

In this subsection we outline some improvements to Monte-Carlo methods applied to Go. A global improvement, used in nearly all Go programs, is that during MC evaluation a move is never played in one's own eyes. While this is not technically an illegal move (except when it is also a suicide move), it is usually a very weak move, possibly damaging a player's own position. One exception to this is the situation where one or more of the stones surrounding the group are in atari.

To improve on the all-as-first algorithm we could turn to other MC techniques already being employed in regular Go. Since MC techniques offer a way to perform a global search with very little domain knowledge, we focus on techniques that do not use any, except what is needed to play and score random games. How we can improve play by using domain knowledge will be the focus of the next subsection.

One way of dealing with the drawbacks of the all-as-first algorithm would be to use standard Monte-Carlo evaluation. In this, only the first move of every random game is evaluated. While this does eliminate the problem that evaluations later in the game are less reliable, it has its own drawback: many more random games are needed to evaluate a position reliably.

The obvious first approach to deal with the large number of games needed for evaluation is to try to make the evaluation more effective: spend less time on evaluation of positions that will not end up as the winning candidate. One way of doing this would be by using pruning techniques as described by Bouzy (2006a). The idea of these techniques is that as evaluation progresses, certain moves will perform worse than others. Instead of evaluating these moves until the end, they will get discarded as soon as they prove suboptimal according to some statistical method. Especially Progressive Pruning (PP) seems to be yielding good results. There is a trade off here between the speed gained and the chance of deleting a good move.

A second way of improving MC algorithms is Situated Annealing. This method, adapted from physics, was first used in Go by Brügmann (1993). A move is played randomly, and kept with a certain probability. The probabilities change over time, and with the evaluation of the current move, eventually converging to a maximum. This maximum should then be the best move. However, there is the possibility that this algorithm converges to a local maximum instead of a global maximum.

As the speed of algorithms progresses, one drawback of all these MC methods still stands. Even if the iterative process would last for a very long time, the selected move does not necessarily converge to a game-theoretic optimum. A third approach, which would converge to an optimum, is to construct a game tree according to the partial results of the evaluation. The way this is usually achieved is by an algorithm adapted from the UCB1-algorithm originally designed for the Multi-Armed Bandit problem (Auer, Cesa-Bianchi, and Fischer, 2002a; Auer *et al.*, 2002b). Named UCT by Kocsis and Szepesvári (2006), this has recently become the dominant algorithm in Go-playing programs. It is being used by ManGo, MoGo, CrazyStone, and others.

1.5.4 Domain knowledge and heuristics

Many heuristics have been proposed to improve the performance of computer-Go programs. These can be divided into a few classes. Some try to limit the search space by preprocessing. Others try to improve results through a better guidance of the search. Evaluation functions can be used to assess the value of a position. Then local goals can be used instead of the single, global goal of winning a game.

A good overview of techniques is given in Bouzy and Cazenave (2001). Especially important in these approaches is the concept of a *group*. A group is not a chain. Instead it is a collection of stones and chains that live and die together. Two different ways are used to determine what could be a useful move. Highly specialized searches could be used, that only take into account moves that affect some direct goal, e.g., the life or death and size of a group or the possibility of connecting groups. The second way of dealing with groups are patterns. Patterns are templates of typical Go situations, for which good moves are already suggested. Patterns and rules can be generated automatically (Cazenave, 2003), or defined by the designer of the program.

Apart from these local evaluations, a global evaluation function would be a very powerful tool. If Chinese rules are being used, the evaluation at the end of the game is simple: how much territory has each player claimed? However, before the game has ended it is very difficult to determine which player has claimed which places on the board. Bouzy (2002) analyzes some possible functions for small boards but they cannot simply be extended to larger boards.

Instead of a formal evaluation function a characteristic called *influence* can be used, which can be seen as a measure of potential. Though a formal definition is hard to give, different algorithms have been proposed to compute a value. Bouzy (2003) offers a variation on the classic model of Zobrist (1969).

A more useful variation on influence might be *thickness*, which does not only take into account the potential of groups, but their strength as well. However, this has not been widely used in computer Go. Cazenave (1998) offers an algorithm, which uses the potential value groups could have, multiplied by the chance a group will survive to the end of the game.

1.5.5 Using domain knowledge in Monte-Carlo evaluation

Knowledge-based approaches to Go and MC methods suffer from different drawbacks. Most knowledge-based Go programs are quite strong tactically, but have a weak global sense that results from the breaking up of a whole problem into sub-problems (Bouzy, 2005). MC-based programs are quite strong on a global scale but are weak tactically (Bouzy and Helmstetter, 2004). Unsurprisingly, programs that are using both domain-specific knowledge and Monte-Carlo approaches have been developed.

Bouzy (2005) suggests two ways in which MC evaluation could be improved by knowledge. The first is to replace the random games in the evaluation with pseudo-random games. Instead of generating moves according to a uniform probability distribution, moves are selected with a certain probability based on knowledge-based evaluation of that move. Move urgencies are used here, based on chain captures and 3×3 patterns. With such knowledge, the games used in the evaluation of moves are more plausible, thus giving a better approximation of the value of a move. Gelly *et al.* (2006) also use 3×3 patterns to improve random games but combine it with UCT. They further try to make the search more efficient by reusing information about a node's parent node in the search tree.

Secondly, before MC evaluation even takes place, a preprocessor can be used. This limits the number of moves to be evaluated, thus speeding up the selection of a move. As shown by Van der Werf (2005) a relatively small number of high-ranked moves, selected at this stage, can be enough to play a reasonably strong game, at least against other computer programs. The program in which these improvements were made, OLGA, is significantly stronger than INDIGO2002, Bouzy's knowledge-based program.

In Bouzy (2006b) these two improvements are used as well, but with different functions to define the probabilities that a move is selected. A territory heuristic is suggested, setting the urgency of playing a move according to the probability a position is controlled by the player at the end of the game. The probability of controlling a position is computed on the outcomes of the random games in MC evaluation. If the probability is small, the urgency will be high. Bouzy also uses a history heuristic, originally proposed by Schaeffer (1989), which tries to avoid reanalyzing moves that were also bad moves in previous evaluations.

Cazenave and Helmstetter (2005) suggest a way of integrating tactical goals within Monte Carlo. In all-as-first MC Go the means of the random games where an intersection has been played first by a player are computed for each intersection. This same evaluation can also be done for tactical goals instead of positions. Thus, the mean value of the games where the goal has been accomplished and the games where the goal has not been accomplished are compared. Typical goals here are capturing or saving a chain, connecting or disconnecting two chains, or making an eye. Once a goal has been selected, search algorithms are used to determine how to accomplish this goal.

Since simply 'playing first on an intersection' has also been defined as a goal, it is not always necessary to pursue a complicated goal. If occupying a position would be the move with the highest expected utility, it does just that. Therefore it does not lose the way the standard algorithm plays, it simply enhances it by giving it more choices. One drawback of this approach is that in the process of MC evaluation moves that could threaten tactical goals are not explicitly taken into account. Because of this it is not always the case that selected goals can actually be accomplished.

1.6 Problem Statement and Research Questions

In conclusion of the first chapter, we may state that although much work has been done on Monte-Carlo methods for computer Go, it is not apparent how to adjust these methods to deal with uncertainty. So far, only one approach to playing Phantom Go has been published. Thus, we come to our problem statement.

> How can we apply the existing knowledge about Monte-Carlo methods to computer Phantom Go in order to produce a strong playing program?

So, the research objective is to implement a strong playing program. By a strong playing program we mean at this moment a program that outperforms the program GoLois by Tristan Cazenave. To achieve this objective and to answer the problem statement we have formulated five research questions (RQs).

RQ1 Will other Monte-Carlo methods perform better than all-as-first?

While the all-as-first algorithm used by Cazenave (2006) certainly has some advantages we feel it also has some weak points. One definite advantage is that it generates reasonable values for many moves very fast. However, there is also the implicit assumption that the order in which these moves are played does not matter. We feel that in most games the order in which moves are played are certainly of some value. We will try to use the most simple Monte-Carlo algorithm, in which only the first moves of the different games are valued, to assess this point. However, this does have the drawback that many games are necessary to yield reasonable values for every move.

RQ2 How much improvement will tree search produce?

We also use a variation of the UCT algorithm (Kocsis, Szepesvári, and Willemson, 2006) which has proven to be successful in regular Go, and needs much less simulated games. While we expect this to perform better than standard MC methods, we want to measure how big this improvement is.

RQ3 Can we improve on standard Monte-Carlo algorithms by integrating domain-specific knowledge?

Some positions are of greater use to the opponent than others. Since there are a number of unknown stones on the board, these positions may already have been played. This assumption leads to other questions. What heuristics can be used to assign probabilities to a certain move? If we have some way of determining these probabilities, what would be the effect of distributing the number of 'unknown' enemy stones according to these probabilities? And what would be the effect of allowing these probabilities to influence the choice of moves while building a search tree?

RQ4 How big will the drawbacks of our (possibly faulty) evaluation of referee feedback be?

We intend to follow Cazenave (2006), as well as most human players, in the way to deal with the reply 'illegal'. Since this reply can have more than one meaning, this may mean that the position we create on our own board will be faulty. Will this prove to be a big problem or will the effects it has on the game be minimal?

RQ5 How much information about the opponent's position is typically uncovered during the game, and can this information be maximised?

Phantom Go is not only about building your own position. The more information is uncovered about the opponent's position, the better you will be able to make your decisions. If more information can be uncovered during a game, the chances of winning improve. However, there is a risk involved here. If a move played for information is legal, the stone will remain in that place, possibly a position that is not very useful.

RQ6 What will be the impact of the scoring method on the behaviour of the MC methods?

Two different ways of scoring a simulated game can be used. In territory scoring, the value of a game is the difference in territory between the player and the opponent. In 0/1 scoring, a game is either won or lost. Because this influences the basic values with which the MC methods compute their evaluations, it will be interesting to see how this impacts playing style.

1.7 Thesis Outline

The structure of the remainder of this thesis is as follows. In Chapter 2 we will introduce our application and the general approach our players should follow. How we deal with referee feedback, relevant to RQ4, will be the subject of Subsection 2.3.4. After that, a succession of players will be given. Section 2.4 will deal with the different forms of MC evaluation used, relevant to research questions RQ1 and RQ2. Section 2.5 will explain what heuristics we use, relevant to RQ3. Section 2.6 will explain the different methods of scoring and their implications, relevant to RQ6.

Chapter 3 will outline our experimental setup, and describe the behaviour and results of our players. These experiments will attempt to answer the research questions RQ1 (Subsections 3.2.1 and 3.2.4), RQ2 (Subsections 3.2.1, 3.2.2, and 3.2.4), RQ3 (Subsection 3.2.5) and RQ6 (Subsection 3.2.3). We will try to answer RQ4 and RQ5 based on some general observations about the games played in our experiments, but no specific experiments have been done to answer these questions.

In Chapter 4 we will draw our conclusions. The research questions will be answered in Section 4.1, and we will discuss the problem statement in Section 4.2. Section 4.3 discusses what questions remain for future research.

Chapter 2

AI for Phantom Go

In this chapter we describe our approach for making a Phantom Go playing program. The structure of our application (Section 2.1) and our communication protocol (Section 2.2) are given. After that, we describe the general approach for building players (Section 2.3) and describe what players were implemented (Sections 2.4 and 2.5). Eight players are introduced in order of increasing complexity. The performance of these players will be described in Chapter 3. Finally we make some observations about determining the value of a simulated game (Section 2.6) and give a list of all players (Section 2.7).

Chapter contents: Structure, Communication Protocol, General Approach, Monte-Carlo Strategies, Heuristics, Scoring

2.1 Structure of the Application

For implementing Phantom Go playing programs, two different modules are needed. As in the real world, the main entities of the application are a referee and two players. The *program* is defined as the entire package we developed, the *player* as the process determining which moves to play, i.e., the algorithm used. The players and the referee are implemented as separate processes and communicate using the Go Text Protocol (Farnebäck, 2002). Thus, it is clear what information the players have at any given moment. An advantage of this approach is that it could be used to play games against other Go programs as well. The game board as it is seen during a game is shown in Figure 2.1.

In order to play against humans and in tournaments, an interface without referee was developed, which connects to only one program. In this case, the operator has to be the referee. See Figure 2.2. In order to analyse the search trees produced by the UCT-based algorithm, the VT tree visualisation tool was used with the implementation.[1] An example situation is depicted in Figure 2.3. For the GTP-module and displaying purposes, we have used classes from the open-source project GoGui.[2] Several utilities from Mark Boon's open-source library GoTools were used to model Go features.[3]

2.2 Communication Protocol

Since GTP is designed for regular Go, we need to add a few extra commands. To avoid confusion, we chose to define our own commands instead of using existing ones in a different way. All our commands start with the abbreviation PG- (Phantom Go) to avoid conflicts with existing commands. The following list describes our extension to the GTP protocol.

- **PG-SuggestMove** The referee asks a player to try playing a move. The player replies, for instance by sending 'F5'. If the move is illegal, the referee will use this command again, this time with the moves the player has already tried (this turn) as arguments:

  ```
  PG-SuggestMove F5 ... \n
  ```

[1]VT, by Joris Borsboom, available at http://www.cs.unimaas.nl/g.chaslot/.
[2]GoGui is available at http://gogui.sourceforge.net/.
[3]GoTools is available at http://www.sente.ch/pub/software/tesuji/.

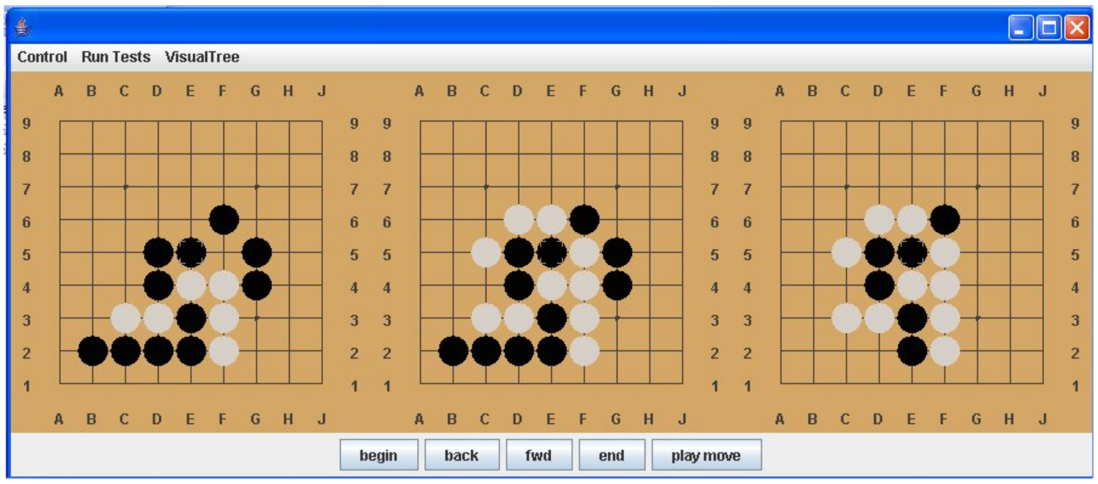
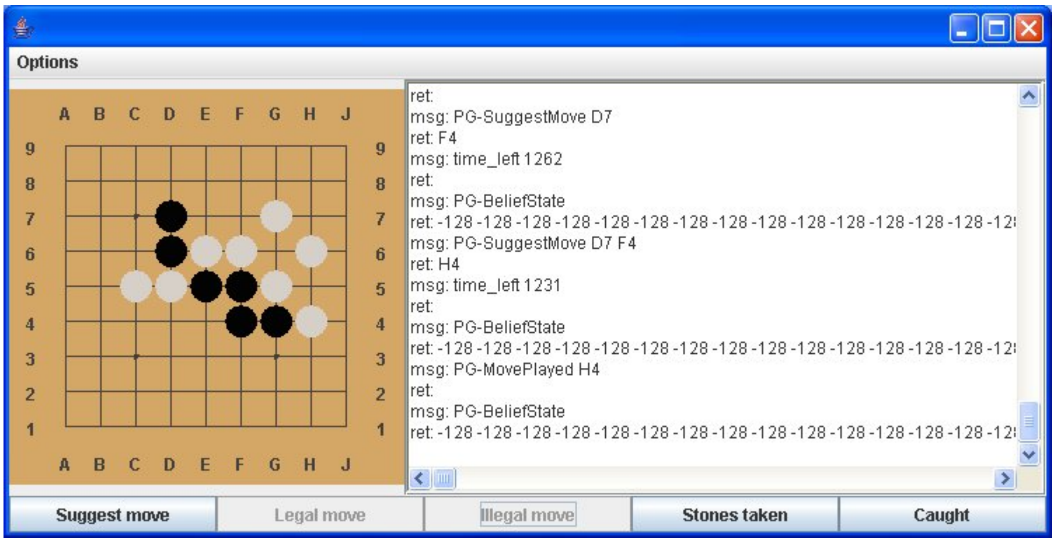


Figure 2.1: The game board. The central board is the referee's board, left and right are the boards as perceived by the players.



```
ret:
msg: PG-SuggestMove D7
ret: F4
msg: time_left 1262
ret:
msg: PG-BeliefState
ret: -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -12
msg: PG-SuggestMove D7 F4
ret: H4
msg: time_left 1231
ret:
msg: PG-BeliefState
ret: -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -12
msg: PG-MovePlayed H4
ret:
msg: PG-BeliefState
ret: -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -128 -12
```

Figure 2.2: The stand-alone board. Referee feedback is given using the buttons below.

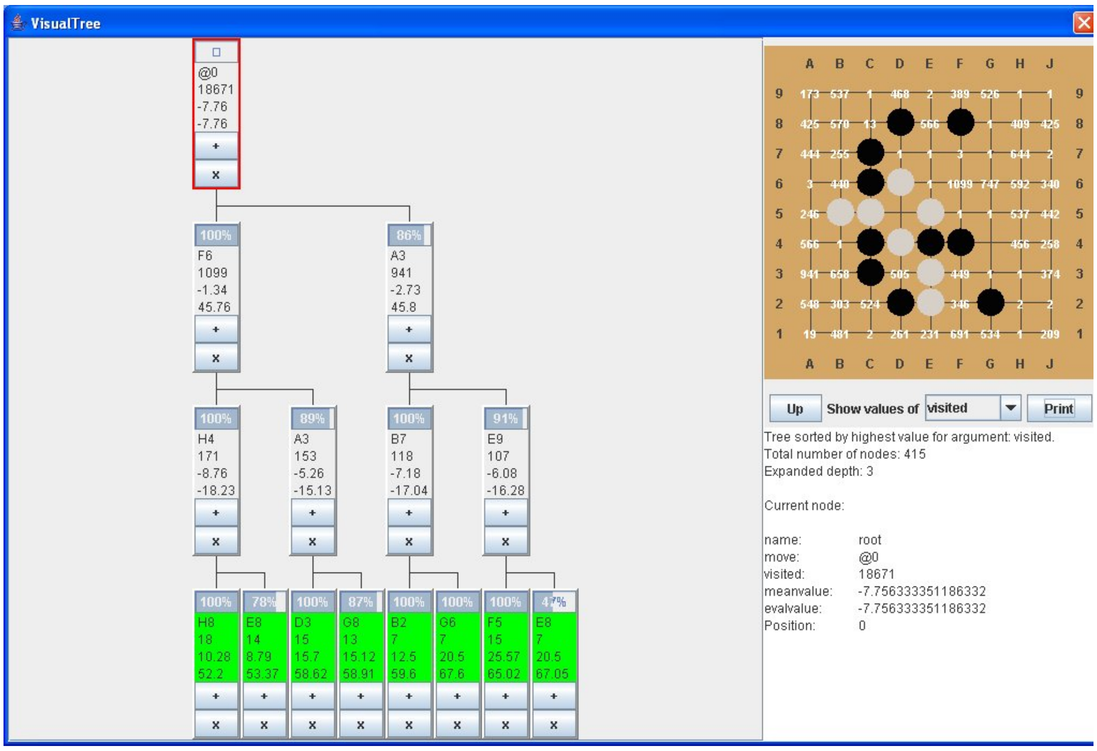
- **PG-TriedMove** Every time a player tries playing a move that turns out to be illegal, this command is sent to the opponent.
- **PG-MovePlayed** If a player tries playing a move that is legal, this is sent, without argument, to the opponent, and, with as argument the played move, to the player trying the move.
- **PG-StonesTaken** If a move captures stones the opponent is told which of his stones are taken. This takes as arguments the stones that are being removed from the board:

  ```
  PG-StonesTaken E4 E3 ... \n
  ```

- **PG-Caught** The player taking the stones will also be told stones are taken, but not which ones. Argument to this is the number of stones he caught:

  ```
  PG-Caught 2\n
  ```

- **PG-BeliefState** Utility command, used to display the state the players believe the game to be in.



```
Tree sorted by highest value for argument: visited.
Total number of nodes: 415
Expanded depth: 3

Current node:

name:          root
move:          @0
visited:       18671
meanvalue:     -7.756333351186332
evalvalue:     -7.756333351186332
Position:      0
```

Figure 2.3: An example of the search tree as depicted by VT.

2.3 General Approach

The computer algorithms we are using for playing Phantom Go should consist of 4 phases for each of which different strategies could be applied. This section outlines all four phases.

1. Preprocessing, in which moves to be evaluated are selected.
2. Filling in the opponent's 'unknown' stones.
3. Evaluating selected moves and playing the 'best' move.
4. Analyzing feedback from played move and updating the player's own board state.

2.3.1 Preprocessing

In the preprocessing phase, a selection of moves to be evaluated is made. Because this selection process is much faster than the evaluation process of phase 3 it will save a significant amount of time if moves that are clearly suboptimal can be filtered. However, if the move that the evaluation process would have suggested is removed here, the results will suffer. It is therefore important to be careful at this stage.

Especially in Phantom Go, where parts of the board state are unknown, it is hard to determine which moves are good or bad at this stage. Since throwing out the wrong moves at this stage could lead to very bad results, we have decided not to use any preprocessing. We simply evaluate all possible moves.

2.3.2 Filling in the opponent's unknown stones

Since the number of moves played is known to all players, the number of opponent stones on the board unknown to the player is easily computed. For practical purposes we ignore the possibility that the opponent has passed. Then we get the following formula.

$$Stones_{unknown} = N - (Stones_{known} + Stones_{taken})$$

N is the number of moves the opponent has made, $Stones_{known}$ the number of opponent stones on the player's board, and $Stones_{taken}$ the number of opponent stones the player has taken. For any useful evaluation of the selected moves, these stones should be taken into account. Of course, there is no certain way of determining the positions of these stones. However, there are more and less likely positions, and this may be exploited through the use of heuristics.

2.3.3 Evaluating moves

After phases 1 and 2 a choice has to be made between the moves selected at phase 1. Because of the high complexity and presence of imperfect information in Phantom Go, Monte-Carlo methods seem the obvious choice for evaluating moves, as we stated in Section 1.4. Most of the remainder of this Chapter is about this evaluation process.

When a move is selected, the move is sent to the referee and the player waits for a reply about the effects of that move.

2.3.4 Feedback and board state

After a move has been played, a response from the referee is given. The different possibilities and corresponding actions are:

1. Legal: The move is updated on the player's own board and played on the reference board.

2. Stones captured (n): The move is updated on the board of the player whose stones have been captured. For the other player the following is done: for all regions enclosed by the playing of this move (typically only one, but more are possible)

   (a) compute the number of empty intersections and opponent stones in each enclosed region,

   (b) find all configurations equaling n,

   (c) remove all stones in these configurations.

   While this never leaves opponent stones on the board that might have been taken, it is also overzealous. All possible combinations are removed, and thus information is destroyed. How big this drawback may prove to be remains to be seen, and perhaps a different way of dealing with situations like this will have to be devised.

3. Illegal: place a stone of the opponent's colour on the tried position, start over at phase 1: selecting moves. This is correct in case a move on an opponent stone has been made. However, if the move was a suicide or ko move, it is incorrect. Still, we do not expect to win a ko fight in the dark, and the position will usually be filled by the time we get around to it. One exception may be the simplest variety of ko, in which case a stone has just been taken. Suicide is a trickier subject as any algorithm will lose opportunities because a suicide position may become a capture position if the group is surrounded.

Play continues until the only moves left are either inside one's own eyes or suicides. Since we are using Chinese rules, this does not affect the score.

2.4 Monte-Carlo Players

Using the structure described in the previous section, a succession of eight Phantom Go playing methods has been developed. These all build upon each other and become gradually more and more complex. They will be introduced over the course of this Section and Section 2.5. For a short list of all players see Section 2.7. The entire package of methods was named InTheDark and presented as a Phantom Go playing program. We will first focus on the different forms of MC evaluation used and after that, in Section 2.5, explore the heuristics that were tried.

2.4.1 Monte-Carlo evaluation

Monte-Carlo evaluation is used in phase 3 to determine which of the moves selected at phase 1 is chosen. Random games are played and the move which obtains the best value according to a statistical evaluation of the random games is played. We introduce three different players which use different ways of determining values for moves.

Before each simulated game, the opponent's unknown stones are redistributed. This is done to ensure that a move is evaluated according to a number of possible boards, instead of just one possible board. Play continues until the only moves left are either inside one's own eyes or suicides. Since we are using Chinese rules, this does not affect the score. After that results are stored, a new board is generated and we start again. This process continues until the time allocated for this process has been fully used.

Random Player Firstly, we define a random player for testing purposes. This player simply plays a random move anywhere on the board that is not occupied by one of his own stones or a perceived opponent stone. Additionally, it avoids playing in his own eyes.

Standard Monte Carlo Secondly, we use standard Monte Carlo evaluation of moves. In standard MC evaluation all selected moves are evaluated in turns. The move is played as the first move of a random game and the value for that game, the score, is stored in the position for that move. Once all time has been used, for all selected moves the mean value of the games[4] is computed and the move with the highest value is chosen as the best move. This is a very basic method and will need relatively many random games to produce good play. For regular Go, this approach has been used by Bouzy and Helmstetter (2004).

All-as-first Thirdly, we implemented all-as-first evaluation of moves. The all-as-first method of evaluation is used by Cazenave (2006) and dates back to Brügmann (1993). Instead of only assigning a value to the first move that was played in a game, the value is assigned to all moves m_i that were played by the player before his opponent. Similarly, the value is also added to all moves m_i which the opponent played first. Once time runs out, the value of playing a move m_i is determined by computing the mean value of the games in which m_i was played first by the player, subtracted by the mean value of the games in which m_i was first played by the opponent. Thus the program tries to play on the position that is most beneficial to itself and bad for the opponent.

The advantage of using all-as-first is that it generates data much faster than standard MC. However, it assumes that the order in which the moves are played does not matter, as long as a move is played before the opponent played it. Since there are situations in Go where the order in which moves are played does matter, it might produce some erroneous results.

2.4.2 UCT

Next to the methods described in the previous subsection we are also using a different form of MC evaluation based on the work of Kocsis *et al.* (2006). Now becoming more generally known as UCT (Upper Confidence bounds applied to Trees), the basic idea of this method is to build a tree-structured search space using random games. Because the results of random games are used to determine which nodes to expand next, the search here is much more focussed on relevant move sequences than in standard MC.

UCT explained

UCT is a form of Monte-Carlo Tree Search (MCTS). In general, MCTS repeatedly applies a best-first-search iteration at the top level. The algorithm runs through four stages iteratively. These stages are described below, and the actions UCT takes are given.

1. **Move selection**: Starting from the root, the child with the highest evaluation value is selected. UCT chooses the child node which maximizes (Black to move) or minimizes (White to move) the following formula:

[4]Section 2.6 describes how to determine the value of a game.

$$\overline{X}_i + C\sqrt{\frac{\ln t(N)}{t(N_i)}}$$

$\overline{X}_i$ is the average value of the child node, $t(N)$ the number of times the parent node was visited and $t(N_i)$ the number of times the child node was visited. C is a constant determining the balance between exploration and exploitation. This procedure is repeated until a leaf node is reached.

2. **Expansion**: When a leaf node is reached, a decision is made whether to expand it. To prevent uncontrolled growth of the tree, and save memory, UCT only expands a leaf node when it is visited for the second time. All possible moves in this position are stored as child nodes. After a node gets expanded, one of its children is chosen for evaluation.
3. **Leaf-node evaluation**: A Monte-Carlo evaluation (*simulated game* or *simulation*) is done for the board position represented by the leaf node. This consists of selecting random moves, playing out an entire game. The game is then scored, giving an evaluation value.
4. **Back propagation**: In this stage the evaluation value is backed up through the tree and the values of the nodes are updated accordingly. We simply add the score to the total score of the node, which is divided by the number of times it was visited to produce $\overline{X}_i$.

This algorithm can be stopped at any time. The node that is played is the child of the root (depth 1) that was explored most.

Move-filling strategies

When constructing the tree, an obvious question is at which point the unknown opponent stones are taken into account. Two different strategies are proposed: late random opponent-move guessing and early probabilistic opponent-move guessing. The combinations of UCT with these strategies become our fourth and fifth players.

Late random guessing The tree is constructed according to the ordinary UCT algorithm. Once a leaf node is reached, the unknown opponent moves are filled in randomly and a simulated game is played. See the left side of Figure 2.4.

While this is clear and simple, there are a few problems with this approach. First, a random distribution of opponent stones might not accurately reflect the game state. Worse, if the unknown moves are filled in after traversing the tree, potentially fatal opponent stones will be missed because moves will probably be made on this position. This leads to careless play and an overestimation of the player's own chances. To deal with these problems a different strategy is suggested.

Early probabilistic guessing This approach is based on the assumption that promising opponent moves might already have been played by the opponent. At depth 1, where the player's own initial move is considered, the exploration is identical to UCT. At depth 2, however, opponent moves are added to the game board according to the number of times they have been selected in previous iterations. A move is selected according to the following probability:

$$C \times \frac{t(N_i)}{t(N)} \times Stones_{unknown}$$

In this formula, $t(N)$ is the number of times the parent node was visited, $t(N_i)$ the number of times the child node was visited, and $Stones_{unknown}$ is the number of unknown opponent stones. C is a constant that can be used to place more or fewer stones at this stage, thus leading to more conservative play or more optimistic play. In our experiments this value has been set to 1, in order not to bias the method.

Moves that are selected are placed on the board. After this, of the child nodes that have not been placed, the node with the highest UCT-score is selected and explored further. While traversing this subtree, it is not possible to select nodes that have already been selected at depth 2. Once a leaf node is reached, any remaining unknown stones will be placed randomly and a simulated game is played. See the right side of Figure 2.4.

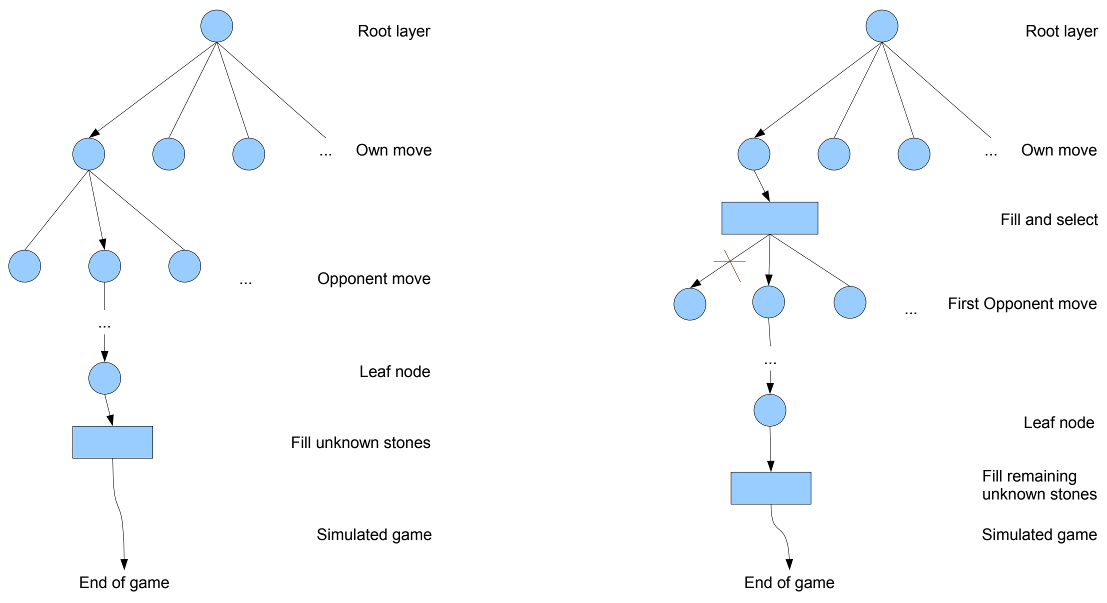


Figure 2.4: The late random move-filling strategy (left) and the early probabilistic move-filling strategy (right).

2.5 Heuristics and Knowledge

Up to now, all proposed solutions have been based on evaluation by means of random games. Almost no knowledge about the game (except for the rules) has been used. We introduce further methods, some based on knowledge, some on educated guesses, that might help improve the behaviour of our program.

2.5.1 Heuristics in MCTS

One of the drawbacks of UCT and other variants of Monte-Carlo Tree Search when compared to all-as-first is that it needs more simulated games to produce good results. Whereas all-as-first uses the result of a game to evaluate all moves in that game, in MCTS the result is used to evaluate only one move. One way of improving the performance of MCTS is therefore using heuristic guidelines to determine which moves should be explored more.

One way to do this is described by Chaslot *et al.* (2007). By adding a progressive bias to the move-selection criterium of UCT, they obtained a new formula:

$$\overline{X}_i + C\sqrt{\frac{\ln t(N)}{t(N_i)}} + f(N_i)$$

$f(N_i)$ is a function which computes a certain heuristic value for the child node N_i. This way, the search will start out by trying promising nodes according to $f(N_i)$. As the search runs longer this factor should decrease, so it does not interfere with the evaluation process too much.

What would be a good value of $f(N_i)$ for Phantom Go? Since a good evaluation function for Go is a notoriously difficult problem, we use some approximation. For this the concept of *influence* is used, which is a measure of potential. It gives an indication how much of the board could be controlled at the end of the game. We compute influence according to the method as proposed by Zobrist (1969), using an implementation from Mark Boon's GoTools. The function returns the influence for each intersection of the board. We then compute the total figure, by adding up the values of all intersections.

For each possible move in a certain board position the gain in influence could be computed by subtracting the current influence score from the influence score after the move has been played. Since each player usually wants to control as much territory as possible, positions which give a bigger increase in influence are more desirable.

Since computing the influence score is a relatively time-consuming procedure, we chose to compute it only once for each intersection, right after the player has been asked for a move. While computing it for each node would give a more accurate figure, the benefits of this will not outweigh its drawback: a loss in the number of simulated games that can be played in the allocated time. A value for White as well as a value for Black has to be computed. While this does not strictly speaking give a correct value, it seems like a good enough approximation of the actual influence value of a certain move.

This leads to the following definition for $f(N_i)$:

$$f(N_i) = \frac{C}{\sqrt{t(N)}} \times (inf_{i.move} - inf_0)$$

$t(N)$ is the number of times the parent node has been visited. $inf_{i.move}$ is the influence score computed for the move associated with the child node, inf_0 is the influence score for the board position at the root of the tree. C is a constant controlling how much weight should be given to the influence score as opposed to the normal UCT-evaluation. The combination of UCT with the influence-based bias in move-selection becomes our sixth player.

2.5.2 Influence-based opponent-move guessing

Apart from using the influence function in the selection of moves, one can also use the influence function as a heuristic to determine which opponent stones might already have been played. This figure can then be used to improve the *late random opponent-move guessing* (see Subsection 2.4.2). By accumulating the gain in influence for each possible opponent move, a grand total of possible gain is produced: inf_{total}. Then the probability that a node will be selected can be set to:

$$\frac{inf_{i.move}}{inf_{total}} \times Stones_{unknown}$$

A new distribution of unknown stones has to be made before each simulated game. The same probabilities can be used for generating this new distribution though, thus keeping the effort put into this procedure rather low. The combination of UCT with the influence-based opponent-move guessing becomes our seventh player.

2.5.3 Hybrid strategies

We have noticed a distinct difference in the way the all-as-first player and the UCT-based players perform in the opening stages of the game. In these stages there is typically very little information about the opponent. Where all-as-first usually develops strongly connected groups near the centre of the board, UCT-based players are much more optimistic and play all over the board, attempting to secure territory everywhere. The behaviour of the all-as-first player is certainly more desirable here.

However, UCT-based players seem to play better moves in more complex situations, where much information is already uncovered. Especially in situations where the order in which the moves are played makes a big difference, all-as-first frequently plays the wrong move first. This leads to the idea of a hybrid player: the first n moves are played by the all-as-first algorithm to acquire a strong starting position. After n moves UCT-based players can play out the game. This hybrid player is our eighth type of player.

At the start of the game all-as-first with less time seems to suggest the same moves as all-as-first with normal time settings. Only when a situation becomes more complex, the number of simulated games seems to make a real difference. Because of this, an additional heuristic is used in the hybrid players. This allocates less time to the all-as-first phase of playing, thus saving more time for the UCT phase.

2.5.4 Other heuristics

Some extra heuristics are used by all players except the random player. We describe three of them. The first one is used to define how much time the selected algorithm receives to determine which move to play. The remaining time is simply divided by a constant. Therefore, much time will be spent analyzing positions at the start of the game and gradually less time will be used as the game progresses. For 9×9 games this constant has been experimentally set to 20.

Secondly, playing in one's own eyes is prevented for the player to move as well as any simulated opponent moves. To play in your own eyes is nearly always a suboptimal move, since it only weakens your own position without threatening your opponent (Chaslot *et al.*, 2006). This happens both in the simulated games and in the construction of search trees. Eyes are implemented as intersections which are completely surrounded by a player's own stones, unless some of these stones are in atari. If some of the surrounding stones are in atari, the position is not qualified as an eye, to allow for connecting stones into groups.

A third heuristic is used to fix a problem that sometimes arises due to the possibly faulty way in which we handle the reply 'illegal'. Because suicide positions are treated as if an opponent stone was occupying that position, those positions will not be played again even if they might have become a killing position. To deal with this problem a third class of stones has been defined: 'safe' opponent stones. If a group of normal ('unsafe') opponent stones has been surrounded completely, all intersections in that group are tried to see if it might be possible to kill the group. If this behaviour is unsuccessful the stones in the group are qualified as 'safe' stones, and this behaviour will not be applied to this group again. Though this results in trying many moves, it takes very little time to execute.

2.6 The Effect of the Scoring Method on MC Evaluation

There are two distinct ways of scoring a simulated game. At first we scored the result of a game as the difference in territory between one player and the other. This usually leads to the move which gives the biggest gain in territory. Thus, the player's behaviour is quite greedy and sometimes loses track of the ultimate goal: controlling more territory than your opponent at the end of the game. We call this a *strategic* disadvantage.

In contrast to this, a game can also be evaluated as either a loss or a win. This is referred to as *0/1 scoring*, 0 points for a loss, 1 point for a win. Because of this different scoring method, the algorithms tend to focus on the moves that have a bigger probability of winning the game. This leads to more adequate risk taking. Because of this, many Go programs[5] now use 0/1 scores in their evaluation. A drawback of this method is that it does not focus on the moves which produce the most gain, thus leading to worse *tactical* play. A second drawback may be that because of the presence of hidden opponent stones, a perceived small win may actually be a small loss.

2.7 List of players

To conclude this Chapter we give a list of the players we introduced in the previous Sections, and the abbreviations we will be using for them in the next Chapter.

1. *Random*: Random play
2. MCE_{std}: Monte-Carlo evaluation with standard sampling
3. MCE_{all}: Monte-Carlo evaluation with all-as-first sampling
4. UCT_{rand}: UCT with late random opponent-move guessing
5. UCT_{prob}: UCT with early probabilistic opponent-move guessing
6. $UCT_{infmove}$: UCT with influence-based move selection
7. $UCT_{inffill}$: UCT with influence-based opponent-move guessing
8. $Hybrid_n$: Hybrid player, (n) being the number of moves played with all-as-first evaluation

[5]For instance, MoGo, ManGo, and CrazyStone use 0/1 scoring.

Chapter 3

Experiments and Results

In this chapter we describe our experiments. Five different sets of self-play competitions are held, in an attempt to answer our research questions. The setup of these experiments is described in Section 3.1. Results are given and explained in Section 3.2. In Section 3.3 our participation in the Computer Olympiad 2007 is described, and an example game is given.

3.1 Experimental Setup

Five different sets of self-play experiments were conducted. Experiments were done on two machines, both running Linux. Because we have had to use different machines, different time settings were used. However, all games within one experiment were always conducted on the same machine and with the same time settings, so this should not mean our results are any less valid. A version of our program without a graphical user interface and the possibility to play multiple games was used for these experiments. All games were played using Chinese rules and a komi of 6.5.

In the first set-up, a tournament was held evaluating the different forms of Monte-Carlo evaluation without domain-specific knowledge. Five players (*Random*, MCE_{std}, MCE_{all}, UCT_{rand}, and UCT_{prob}) played an all-against-all style tournament. This was done in order to explore research questions RQ1 and RQ2.

In the second set-up, the performance of the hybrid player ($Hybrid_n$) was tested, with different settings for the number of games played with all-as-first (n). It played against the strongest contenders of the first tournament. This experiment was conducted to test an assumption we made about RQ2, namely that all-as-first opens stronger, while tree search produces better results in more complex situations where more information is already uncovered.

In the first two experiments, all algorithms use territory scoring. In a third set-up, the effect of scoring was evaluated by playing two different versions of the same algorithm against each other, one with territory scoring and one with 0/1 scoring. This was done in order to answer research question RQ6.

In a fourth set-up, the performances of Monte-Carlo methods were assessed again by playing them against each other, this time with 0/1 scoring. This was done to ensure that the analysis we made of the Monte-Carlo methods in the first experiments still holds when using 0/1 scoring.

Finally, in a fifth set-up, the impact of the influence heuristics was assessed, by playing UCT_{rand} against UCT enhanced with heuristics. These experiments were conducted with research question RQ3 in mind, although we only test methods based on the concept of influence.

The results of these five sets of experiments are given in Subsections 3.2.1 to 3.2.5.

3.2 Results and Discussion

The results of the different series of experiments are given and discussed in this section.

3.2.1 Monte-Carlo strategies

The results of the tournament comparing Monte-Carlo methods are given in Table 3.1. These experiments were done by playing 50 games of Phantom Go for each combination of players, alternating Black and White. The time setting for each match is 10 minutes per game per player. Experiments were carried out on a computing node with four AMD Opteron 3.4 processors and 32 GB RAM.

	Random	MCE_{std}	MCE_{all}	UCT_{rand}	UCT_{prob}	total wins	wins without *Random*
Random	×	0	0	0	0	0	×
MCE_{std}	50	×	14	4	14	82	32
MCE_{all}	50	36	×	28	44	158	108
UCT_{rand}	50	46	22	×	42	160	110
UCT_{prob}	50	36	6	8	×	100	50

Table 3.1: Results of the pairwise comparison between Monte-Carlo methods.

First, we notice that all methods used beat the random player (*Random*) every time. However, this is not a very important result, since the level of the random player is very low. More interesting are the comparisons between the different forms of MC evaluation. In the remainder of this Subsection we do not take the games against *Random* into account.

Moreover we see that MCE_{all} (36 victories) outperforms MCE_{std} (14 victories). Of the two UCT-based players, the one with late random opponent-move guessing (UCT_{rand}, 42 victories) achieves better results than the variant with early probabilistic opponent-move guessing (UCT_{prob}, 8 victories). Although UCT_{rand} scores the highest number of total victories, MCE_{all} beats it by a small margin in the direct confrontation. Both these players show a comparable number of total victories. MCE_{std}, being the simplest method, performs worst.

All-as-first The main reason why MCE_{all} beats MCE_{std} seems to be that it uses the results of each simulated game more exhaustively, thus faster increasing the evaluation values of the moves which are crucial to the outcome of the game. Especially in a situation where much information is unknown, these are the most important moves. MCE_{std} performs quite weakly because it needs many simulated games to produce good evaluation results for all moves.

UCT with late random guessing An advantage of the two UCT-based methods is that they can reason about move sequences in complex situations. Paradoxically, this is also their weakness as they tend to overestimate their chances. UCT_{rand} can play very carelessly because by building the game tree it assumes that it can always reply to an opponent move. The algorithm fails to take into account the possibility that the opponent move may already have been made or that its reply to that move may not be possible. This leads to an overestimation of the value of moves. A simplified example is given in Figure 3.1. On the left side a part of the search tree constructed by UCT_{rand} is shown. UCT_{rand} is White. At the root (depth 0) the player is unaware of the opponent stone at the bottom left. When considering which move to play (depth 1) UCT_{rand} assumes it can always respond to the opponent attacking its position (depth 2 and 3). However, as the right side of Figure 3.1 shows, the actual play works out very differently because of the black stone on the bottom left already in place.

UCT with early probabilistic guessing The weak results of UCT_{prob} are somewhat unexpected. Because unknown opponent stones are dealt with at depth 2 instead of at the leaf node, we expected it would not suffer from the same drawback as UCT_{rand}, or at least to a lesser extent. While UCT_{prob} can still overestimate moves in this way, it does not happen as often. A qualitative analysis shows that there are two other problems with the early probabilistic opponent-move guessing heuristic.

Firstly, UCT_{prob} can rank weak moves higher than strong moves if the most explored answering moves of that node are also weak moves. This happens when, in the first few traversions of the node, a number of weak opponent moves at depth 2 accidentally are valued as strong moves. In other UCT-algorithms this happens as well. However, because of the early probabilistic opponent-move guessing, two effects hinder the algorithm from re-evaluating these bad moves and continuing the search by exploring more promising branches.

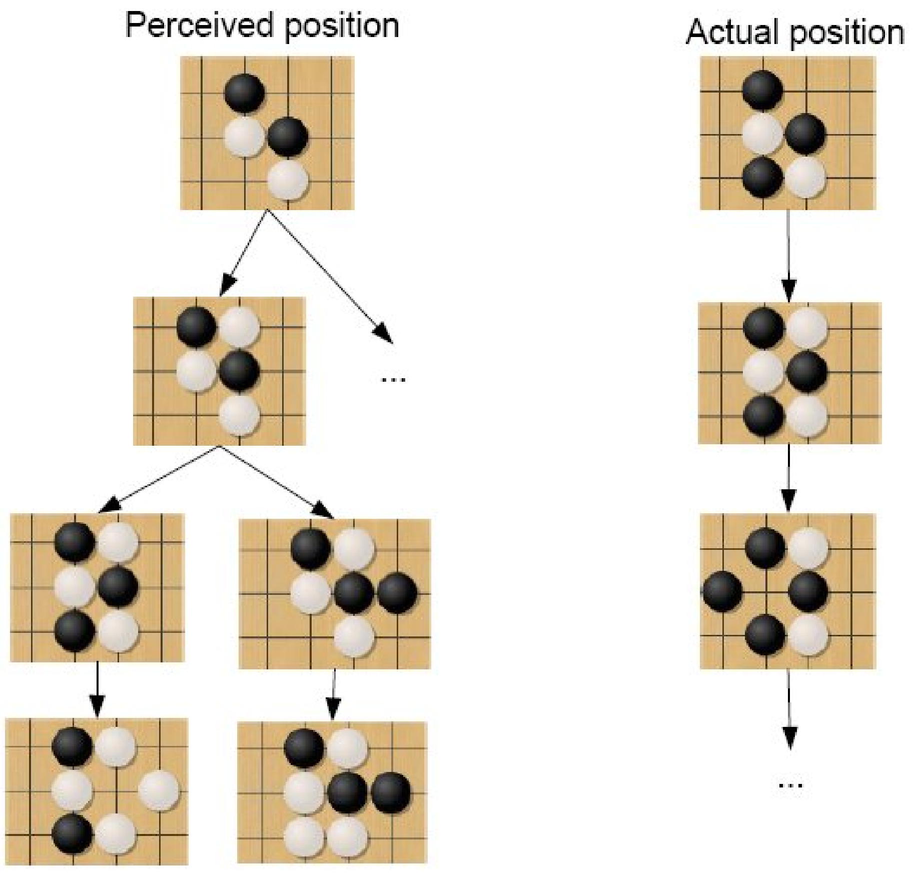


Figure 3.1: A simplified example of the overestimation of a move by UCT_{rand}.

1. Opponent moves which have been traversed many times at depth 2, have a high probability of being picked to fill the board as unknown stones. Thus, there is a smaller chance of them being chosen for further exploration. This in turn leads to a situation in which faulty evaluations of weak moves at depth 2 can endure for a longer time because they are not available for reevaluation.

2. Because in subsequent traversions unknown opponent stones are distributed according to the evaluation moves have at depth 2, new evaluations of the move at depth 1 will be carried out with unknown stones filled in at weaker positions. This leads to a higher evaluation value for the move at depth 1, and the move chosen for exploration at depth 2.

Especially if a number of weak opponent moves have been valued too high, these effects can strengthen each other. If one weak move is chosen as an unknown stone, and the other for exploration, the move chosen for exploration has a bigger chance of getting updated with a higher value because it plays against a board with weaker opponent stones. Many more simulated games are necessary for the algorithm to realise it was overevaluating a node compared to, for instance, UCT_{rand}.

In some cases, this behaviour can last until the end of the time allocated for selecting a move. Even though the rest of the tree is developed as one would expect, ranking the best moves at the top, the move which was overestimated is still the one that has been traversed most. This results in the return of a weak move, which is only chosen because of the weak answering moves the algorithm explored at depth 2.

Secondly, UCT_{prob} sometimes decides not to attack a weak group of opponent stones because it assumes the moves necessary to save it have already been made. Because these moves are typically ranked as good moves at depth 2, it assumes these moves have already been played. This leads to an underestimation of its own chances, and in turn to a reluctance to take risks.

These two problems, persistence of weak moves and an aversion of taking risks in some situations, produce erratic play, in which good tactical play is interrupted by weak moves and the player sometimes decides not to pursue what seem like good chances.

UCT vs All-as-first Finally, the best UCT-based player (UCT_{rand}) does not outperform the best player without tree search (MCE_{all}). Both play roughly at the same level, and suffer from their own drawbacks. Besides the overestimation of moves and their sometimes careless play, UCT-based players typically play disjoint stones in the opening phase of the game. Because of the tree-based nature of their search, they assume that these can be connected at a later stage of the game. MCE_{all} typically plays strongly connected stones near the centre of the board, which is a much better tactic in the early stages of a game of Phantom Go. Many human players employ this tactic as well, because it leads to bigger groups which are harder to kill, and hopefully divides the opponent stones into two or more weaker groups which cannot be connected.

Once the game progresses, more information about the game is known, and the positions become more complex, different qualities become more important. Because MCE_{all} lacks a search structure, it cannot reason about sequences of moves. UCT_{rand} can determine sequences, which leads to better tactical play. Even though it can play carelessly, this is still a big enough advantage to even the score against MCE_{all}, which usually has a stronger position after the opening stages of the game.

3.2.2 The performance of the hybrid player

To test the hybrid player it played games in different configurations against the best 'pure' methods: all-as-first evaluation and UCT with late random opponent-move sampling. The hybrid player is a combination of these two. The results are given in Table 3.2. The $Hybrid_5$, $Hybrid_{15}$, and $Hybrid_{25}$ refer to a hybrid player which shifts from all-as-first evaluation to UCT evaluation after 5, 15, and 25 moves, respectively. These experiments were done under the same conditions as in Subsection 3.2.1.

	$Hybrid_5$	$Hybrid_{15}$	$Hybrid_{25}$	total score
MCE_{all}	23 - 27	25 - 25	18 - 32	66 - 84
UCT_{rand}	14 - 36	20 - 30	18 - 32	52 - 98
total score	37 - 63	45 - 55	36 - 64	118 - 182

Table 3.2: Results of the two best MC methods against hybrid players.

The hybrid player was introduced to make up for some of the disadvantages of the pure algorithms, mainly the weak opening game of UCT_{rand} and the weak tactical play in complex situations of MCE_{all}. The hybrid player generally outperforms the pure methods, with a total score over all games of 118 - 182. This indicates that our approach is valid, and does overcome both problems at least somewhat. However, its victory over MCE_{all} (66 - 84) is less convincing than its victory over UCT_{rand} (52 - 98). This indicates that of the respective disadvantages, the weak opening of UCT_{rand} is the bigger drawback.

It is hard to draw a conclusion about the ideal point of switching methods from these experiments. Whereas $Hybrid_5$ and $Hybrid_{25}$ both win with a similar margin (37 - 63 and 36 - 64), $Hybrid_{15}$ wins with a smaller margin (45 - 55). This is somewhat unexpected, because 15 moves are usually enough to acquire a strong starting position, whereas 5 seems like a small number, and 25 too long. Then again, complex situations can arise at the start of the game, and disjoint playing can still be a problem in later stages. Perhaps it would be better to analyse the situation a game is in before deciding whether to switch, instead of switching after a preset number of moves.

3.2.3 The effect of scoring

To determine the effect of using 0/1 scoring to evaluate the simulated games, all 'standard' methods as used in Subsection 3.2.1 played against themselves, one player with territory scoring, one player with 0/1 scoring. These experiments were done on a computing node with two Pentium III processors. Because less simulated games can be played in the allocated time, compared to our earlier experiments, two batches of experiments were done.

In the first round of experiments 50 games were played, switching Black and White. Each player was allocated 10 minutes for each game. The results of these games are given in Table 3.3. In the second round of experiments, only 30 games were played, but each player was allocated 30 minutes for each game. The results of these games are given in Table 3.4.

	Territory scoring	0/1 scoring
MCE_{std}	23	27
MCE_{all}	11	39
UCT_{rand}	22	28
UCT_{prob}	16	34

Table 3.3: Results of the comparison between MC-methods with territory scoring and 0/1 scoring. This experiment has been conducted with 10 minutes time for each player.

	Territory scoring	0/1 scoring
MCE_{std}	12	18
MCE_{all}	9	21
UCT_{rand}	7	23
UCT_{prob}	7	23

Table 3.4: Results of the comparison between MC-methods with territory scoring and 0/1 scoring. This experiment has been conducted with 30 minutes time for each player.

The first major conclusion that can be drawn from these experiments is that 0/1 scoring outperforms territory scoring. Apparently the strategic advantages of 0/1 scoring outweigh its tactical disadvantages. When observing the games this becomes quite apparent: while territory-based algorithms try to claim as much territory as they believe is feasible, 0/1-based algorithms tend to play for a minimal win.

When we look at the first experiment in more detail, especially the result of all-as-first seems spectacular, showing the biggest improvement over its territory-based twin. However, this is mostly an effect of all-as-first's ability to accomplish good results with small amounts of information. The results of the second experiment show that the two UCT-based methods actually show a slightly bigger improvement over their respective territory-based siblings.

When analyzing the games played between an algorithm with territory scoring and an algorithm with 0/1 scoring, there is one very distinct difference. While the 0/1-based player does seem to miss some opportunities to claim more territory, it always tries to consolidate a position in which it can win, and tries to attack if it is in a position which seems to be lost. This is a major improvement over the algorithms with territory scoring which sometimes forget to defend a won position while trying to claim more territory. The inverse can also be a problem: trying to defend territory which will not be enough to win the game and forgetting to attack.

3.2.4 Monte-Carlo strategies with 0/1 scoring

Since the results of 0/1 scoring are so much better than the results of territory scoring, it was deemed useful to rerun the tournament of Subsection 3.2.1, this time with 0/1 scoring. This experiment was conducted using the same settings as the experiments in Subsection 3.2.3. First a tournament amongst MCE_{std}, MCE_{all}, UCT_{rand} and UCT_{prob} was held with 10 minutes time per player per game. 50 games were played between all players. The results are given in Table 3.5.

	MCE_{std}	MCE_{all}	UCT_{rand}	UCT_{prob}	total wins
MCE_{std}	×	3	7	7	17
MCE_{all}	47	×	43	40	130
UCT_{rand}	43	7	×	25	75
UCT_{prob}	43	10	25	×	78

Table 3.5: Results of the pairwise comparison between Monte-Carlo methods using 0/1 scoring. Tournament with 10 minutes time per game per player.

These results reinforce the results of Table 3.3. MCE_{std} does not improve a great deal, and performs worse than in the territory-based tournament (see Table 3.1) with only 17 wins now against 32 wins before.

Notable is also the improvement of UCT_{prob}, which wins 78 games now against 50 before. Apparently it suffers less from the drawback described in Subsection 3.2.1 when using 0/1 scoring, because the impact of an erroneous good result is not as big a factor as when using territory scoring. MCE_{all}, which also showed the biggest improvement in Table 3.3, is the clear winner here. It also showed the biggest improvement against its territory-based counterpart. However, that effect was less distinct in our experiment with 30 minutes time, and was mostly due to its ability to produce good play with very few simulated games. Therefore another tournament was played between MCE_{all}, UCT_{rand} and UCT_{prob} with 30 minutes time per game per player. 32 games were played per match. The results are given in Table 3.6.

	MCE_{all}	UCT_{rand}	UCT_{prob}	total wins
MCE_{all}	$\times$	22	25	47
UCT_{rand}	10	$\times$	15	25
UCT_{prob}	7	17	$\times$	24

Table 3.6: Results of the pairwise comparison between Monte-Carlo methods using 0/1 scoring. Tournament with 30 minutes time per game per player.

Both UCT_{prob} and UCT_{rand} show a better result with more time to decide on a move. However, under these settings they still do not beat MCE_{all}. Apparently UCT needs more time to improve its performance against all as first when using 0/1 scoring as opposed to territory scoring.

To test this assumption, and to have one final try at beating MCE_{all}, we ran an experiment on the same computing node as used in Subsection 3.2.1, this time with 20 minutes time per player. In this MCE_{all} competed against UCT_{rand}, UCT_{prob}, and $Hybrid_{15}$, all using 0/1 scoring. Each match consisted of 50 games. The results are given in Table 3.7

	MCE_{all}
UCT_{rand}	14 - 36
UCT_{prob}	22 - 28
$Hybrid_{15}$	28 - 22

Table 3.7: Results of the match between tree-based methods and all-as-first, using 0/1 scoring. Played with 20 minutes time per game per player.

UCT_{rand} performs similar to the tournament on the slower computing node. Apparently, just more computing time will not be enough to overcome the disadvantages of overestimation by UCT with late random opponent-move guessing. Surprisingly, UCT_{prob} performs much better against MCE_{all} than before. It seems that the disadvantages of the early probabilistic opponent-move guessing, in particular the persistence of bad moves with good evaluations, can at least be partly overcome by playing more simulated games.

The hybrid player still beats MCE_{all}, thus reinforcing our earlier conclusion that UCT performs better at later stages in the game, also when using 0/1 scoring.

3.2.5 The impact of the influence heuristic

To assess the value of the influence heuristic, two different algorithms using influence were implemented. Both were based on UCT, but enhanced with an influence-based feature. In the first algorithm the total gain in influence was used to guess the location of the unknown opponent stones ($UCT_{inffill}$). In the second the total gain in influence was used to determine which moves should be explored first while constructing the search tree ($UCT_{infmove}$). They were tested against UCT with late random opponent-move guessing (UCT_{rand}). Experiments were carried out with both territory scoring and 0/1 scoring.

The results of the first algorithm against UCT_{rand} are given in Tables 3.8 and 3.9. The experiments were carried out on the same computing node and with the same setting as described in Subsection 3.2.3. Especially in the experiments with 10 minutes time per player, the results of UCT with influence-based move guessing are worse than those of the random move guessing. The results with 30 minutes per player are better, but the influence-based algorithm still does not beat UCT_{rand}, although it does produce a tie in the experiment where 0/1 scoring was used.

	UCT_{rand}	$UCT_{inffill}$
Territory scoring	34	16
0/1 scoring	28	22

Table 3.8: Results of the comparison between UCT with late random opponent-move guessing and UCT with influence-based opponent-move guessing for territory scoring and 0/1 scoring. This experiment has been conducted with 10 minutes of time per game per player.

	UCT_{rand}	$UCT_{inffill}$
Territory scoring	17	13
0/1 scoring	15	15

Table 3.9: Results of the comparison between UCT with late random opponent-move guessing and UCT with influence-based opponent-move guessing for territory scoring and 0/1 scoring. This experiment has been conducted with 30 minutes of time per game per player.

These results seem to indicate that our assumption that the positions which give the biggest increase in total influence are the positions which are most likely to already have been taken, is incorrect. In experiments with more time, and the experiments with 0/1 scoring, the results have been somewhat better because the longer running time and the focus on strategic play diminish the effect of opponent-stone guessing.

The influence-based move selection heuristic ($UCT_{infmove}$) was tested with C set very low, 0.1 for territory scoring and 0.001 for 0/1 scoring. C is the constant controlling how much weight the influence score should be given in relation to the normal UCT evaluation of a node. By setting it very low it essentially functions as a tiebreaker. This does not mean it has very little impact. When exploring the children of a node for the first time, they all have the same, high score. The influence heuristic decides in which order they get explored. When using 0/1 scoring it also decides between nodes for which the same amount of games have been played and which have won the same amount of these games. Since these situations occur very often, the influence heuristic will still have a big effect on the overall behaviour. This also holds for the evaluation of nodes at the topmost level, which usually get explored very often, as the heuristic decides which moves to make further down the tree, and thus affects the evaluation value which gets backed up the tree. The results of this second algorithm against UCT_{rand} are given in Tables 3.10 and 3.11.

	UCT_{rand}	$UCT_{infmove}$
Territory scoring	20	30
0/1 scoring	30	20

Table 3.10: Results of the comparison between UCT with late random opponent-move guessing and UCT with influence-based move selection and late random opponent-move guessing for territory scoring and 0/1 scoring. This experiment has been conducted with 10 minutes of time per game per player.

The results in Table 3.10 may seem a little weird. How can the algorithm with influence-based move guessing perform better with territory scoring and worse with 0/1 scoring? This is caused by the different ways in which ties can arise, as explained above. In the territory-scoring experiment the heuristic only decides between nodes which have not been explored yet. This makes the search more directed at the leaf nodes of the tree. However, as soon as all siblings have been visited at least once, the search is guided by the scores of the simulated games. Therefore the heuristic does not impact the decisions between inner nodes of the tree. In the experiment with 0/1 scores, siblings that have been explored the same number of times and won the same amount of games have the same score. The influence heuristic decides between these nodes in these cases, thus giving it a longer lasting impact on which nodes are chosen, throughout the tree. This way the tree is explored much more according to the evaluation of the heuristic. As total gain in influence is not always a good measure of how good a move is, this leads to worse results than random tie-breaking.

In the experiment with a longer running time (see Table 3.11) the tree-building algorithm can run for quite a long time. This means that the initial impact of the influence heuristic gets canceled out

	UCT_{rand}	$UCT_{infmove}$
Territory scoring	17	13
0/1 scoring	14	16

Table 3.11: Results of the comparison between UCT with late random opponent-move guessing and UCT with influence-based move selection and late random opponent-move guessing for territory scoring and 0/1 scoring. This experiment has been conducted with 30 minutes of time per game per player.

through the big number of simulated games that get played. A heuristic producing moves that have a bigger probability of producing a simulated game with a good score would tend to focus the search more. However, this does not hold for the influence heuristic which ends up exhibiting the same behaviour as random move selection.

The influence heuristic performs worse than random move-guessing and similar to the move-selection of standard UCT because it is very one-sided. It usually gives useful values at the start of the game, when there are still many empty intersections. However, it completely disregards tactical play. The life-and-death status of a group is never a consideration, so a dead group that is not taken off the board yet can be evaluated as very useful. When defending territory, the moves with the highest gain in influence are not the best moves to prevent opponent attacks. Therefore the results in complex situations are often worse than random, which at least gives equal chances to all moves.

3.3 Playing Strength

Apart from testing the different algorithms against each other, a question we would like to have answered is what the general level of our program is. How good is it really? We competed in the 2007 Computer Olympiad, which we describe below. Also, an example game is given to give an idea of the playing style of the program.

3.3.1 Computer Olympiad

A preliminary version of InTheDark has competed in the 2007 Computer Olympiad in Amsterdam. InTheDark played against Tristan Cazenave's GoLois. This is the follow-up of the program described by Cazenave (2006). The match was played under slightly different rules, as referee feedback now also included which opponent stones were caught, instead of only the number. The time setting for the matches was 30 minutes each. InTheDark lost the match 6 - 0.

The version of InTheDark that competed was a hybrid of the all-as-first algorithm, which played the first 15 moves, and UCT with early probabilistic opponent-move guessing. It used territory scoring to value the simulated games. Both early probabilistic move guessing and territory scoring have since been evaluated as suboptimal (see Section 3.2). However, at the time our research was still focussed on producing good *tactical* play with a limited number of simulated games. In difficult situations, these settings produce relatively good moves, but overall *strategical* play is weak.

The version of GoLois that we played against had changed from the program described by Cazenave (2006) in a few ways. Firstly, it no longer employed all-as-first, but standard Monte-Carlo evaluation of moves. Secondly, it did not use territory scoring but 0/1 scoring. And lastly, it ran on 24 parallel CPU's, thus analyzing on average 300,000 simulated games per move. InTheDark played on one desktop machine, analyzing 80,000 simulated games for the first move. This number decreased as the game progressed.

As a positive remark, in many games InTheDark has been in pretty good positions, and had the possibility to win. It lost the games because of overconfidence, and a tendency to make the wrong strategic choices, most notably trying to claim more territory instead of defending a position which would have won the game. After the match, we adjusted the direction of our research and are hoping to have a rematch in the future. A few examples of the games played at the Computer Olympiad can be found in Appendix B.

3.3.2 An example game

In order to have a better insight into the level of the program we analyze an example game from our experiments. It is between $Hybrid_{15}$ (Black) and MC_{all} (White), both using 0/1 scoring. It was played on an AMD Opteron 3.4 processor with 20 minutes time for each player.

There are more example games in Appendix B.

```
b: E5                   w: D5
b: (D5) E6              w: (E5 E6) E4
b: (E4) F4              w: (F4) D6
b: (D6) D4              w: (D4) E3
b: (E3) E7              w: (E7) D3
b: (D3) C4              w: (C4) C3
b: (C3) B4              w: (B4) D7
b: (D7) B3              w: D8
b: F5                   w: (B3) B2
b: F3                   w: C2
b: (D8 B2 C2) E8        w: (E8 F3) E2
```

The current position of the board and the belief states are shown in Figure 3.2. Both players are playing strongly connected groups, trying to divide the opponent's stones and possibly kill some. In this game the players have almost full information about each others position, which is rare. Because the players are still both using the all-as-first algorithm at this phase of the game and have had pretty much the same information throughout the game, they will often value the same positions highly. In this way they will keep playing on the same positions as their opponent, and thus still have much information.

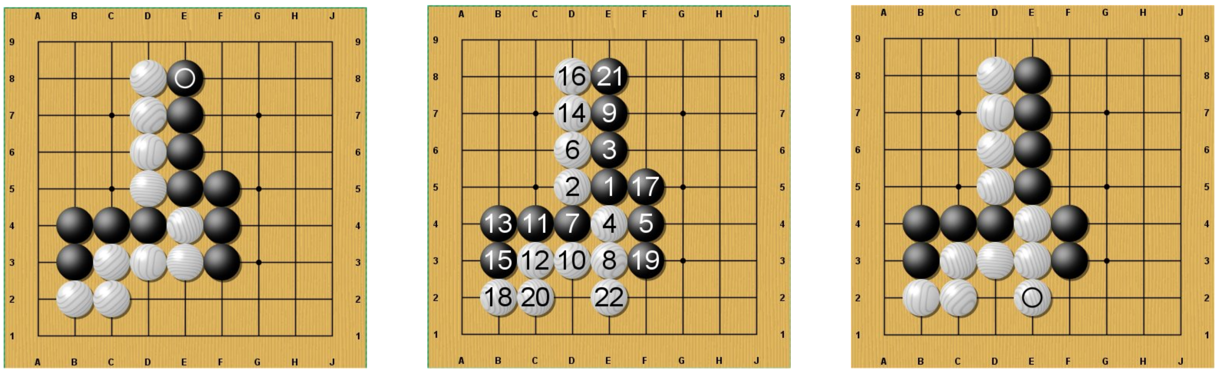


Figure 3.2: The reference board (middle), belief state of Black (left) and White (right) after move 22.

```
b: (E2) B6              w: (B6 F5) B7
b: (B7) C8              w: C7
b: (C7) B8              w: A7
b: D9                   w: A2
b: (A7) C9              w: (D9 C9) E9
b: (E9) A8              w: B9
```

The current position of the board and the belief states are shown in Figure 3.3. The black player, by now using the UCT algorithm with late random opponent-move guessing, realises that just holding on to the right side of the board is not going to be enough to win. He tries to claim more territory in the upper left corner, also threatening the group there. White tries to guard that corner with moves at E9 and B9, but does not realise how big the group there already is.

```
b: F9                   w: F2
b: (F2) G2              w: (A8 B8 C8 G2) G3
```

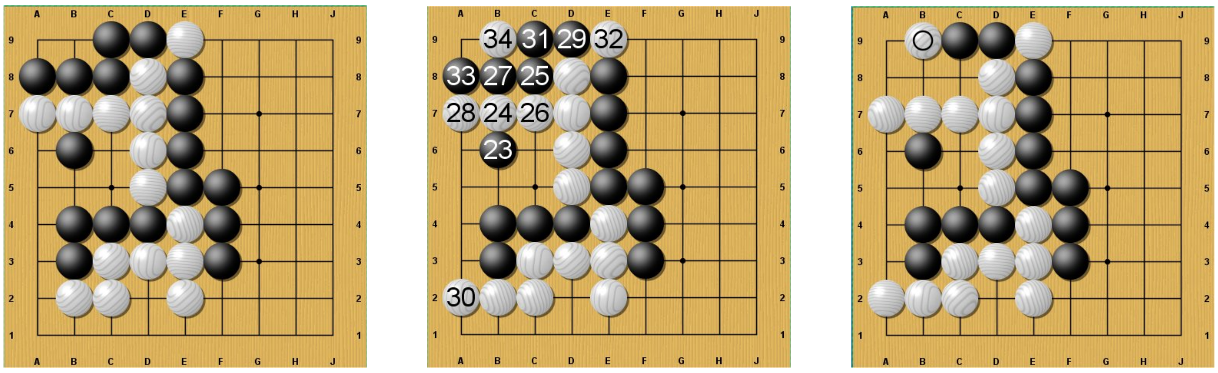


Figure 3.3: The reference board (middle), belief state of Black (left) and White (right) after move 34.

```
b: (G3) H2              w: (H2) H3
b: (H3) A6              w: J3
b: G4                   w: (A6) C5
b: H4                   w: B5
b: J2                   w: A4
```

The current position of the board and the belief states are shown in Figure 3.4. White realises it has lost the upper left corner and now tries to claim the bottom right. Black is ignoring its chances to kill the upper left white group to prevent this. However, after White plays J3 it assumes the group is safe for at least a few moves, and goes back to saving his upper left group. Black concentrates on the bottom right corner and takes the group.

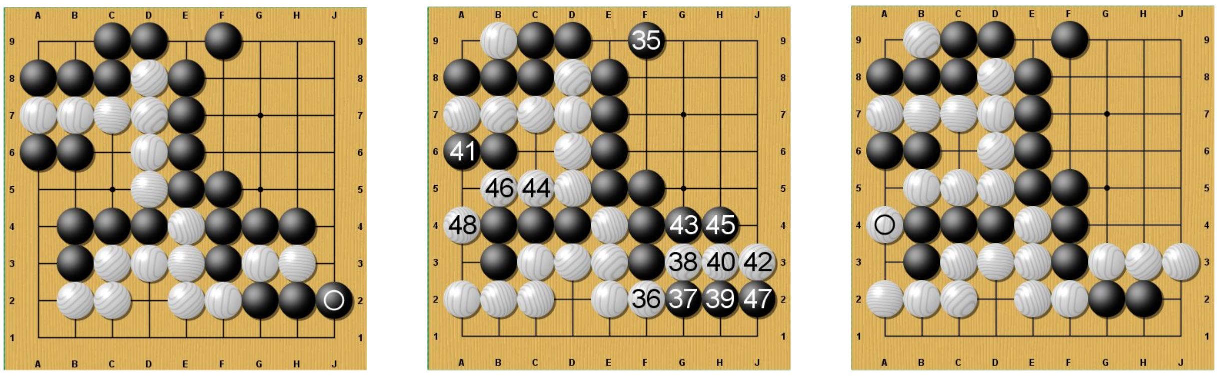


Figure 3.4: The reference board (middle), belief state of Black (left) and White (right) after move 48.

```
b: (J3) J4              w: J1
b: (C5 B5 J1) H1        w: G8
b: (G8) F8              w: H7
b: G7                   w: J8
b: (H7) G9              w: H9
b: H8                   w: (G8) J7
b: (J8 H9) G8
```

The current position of the board and the belief states are shown in Figure 3.5. After the bottom right corner has fallen to Black, the only way White can win is by claiming territory inside Black's upper right corner. It tries to do so, but Black defends. After move 61, the game is won for Black. The rest of the game is given below, although it essentially consists of random moves. Black knows it has won, White knows it has lost.

The remainder of the game went as follows. The reference boards after moves 79, 88, and 105 are given in Figure 3.6.

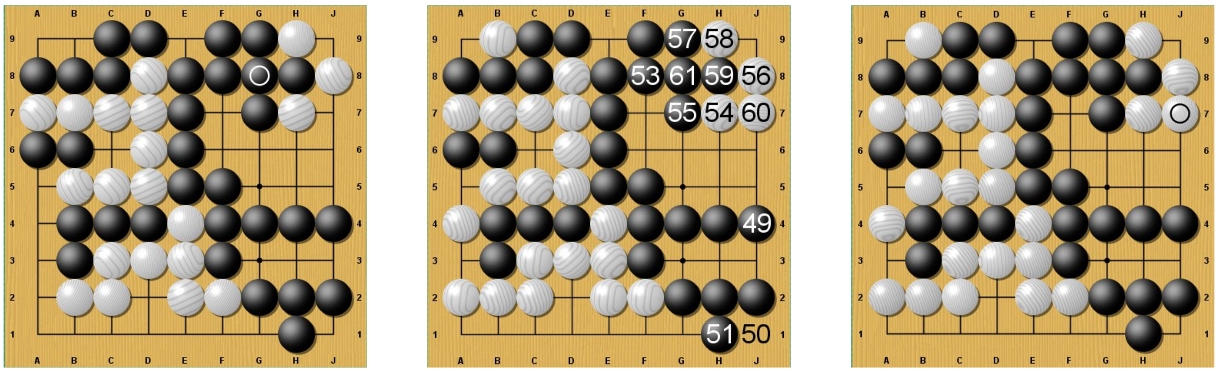


Figure 3.5: The reference board (middle), belief state of Black (left) and White (right) after move 61.

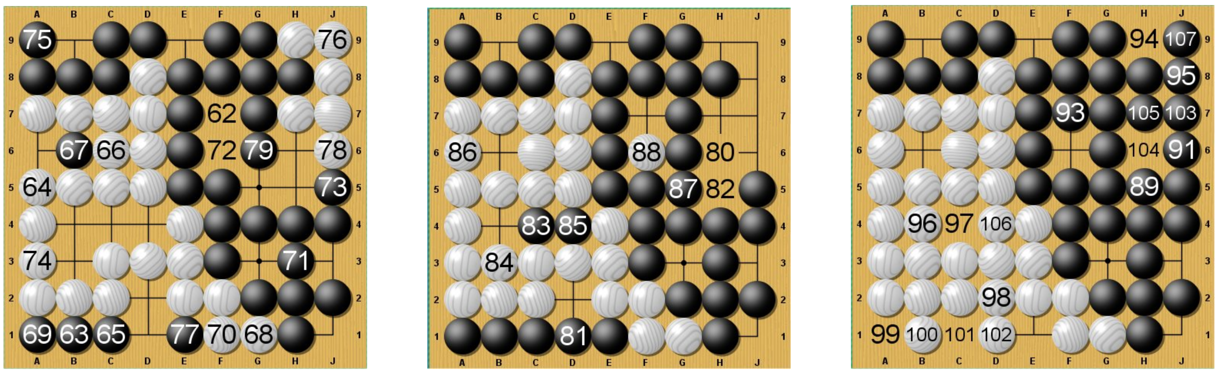


Figure 3.6: The reference boards after moves 79, 88, and 105.

```
                      w: F7
b: B1                 w: A5
b: C1                 w: C6
b: B6                 w: G1
b: (A3) A1            w: F1
b: H3                 w: F6
b: J5                 w: (J3) A3
b: A9                 w: J9
b: E1                 w: (E1 J5) J6
b: G6                 w: H6
b: D1                 w: H5
b: (A2) C4            w: B3
b: (D2 B3 J6 G1) D4   w: (B1 D1 C4 C1 A1 D4 H3) A6
b: (F1) G5            w: F6
b: (F6) H5            w: (H5) H6
b: (H6) J6            w: H7
b: F7                 w: H9
b: J8                 w: (J9 J8 J6) B4
b: C4                 w: (C4) D2
b: A1                 w: (A1) B1
b: C1                 w: (C1) D1
b: (H7) J7            w: H6
b: (H6 H9) H7         w: D4
b: J9                 w: PASS
b: PASS
score game : 2.5
```

Chapter 4

Conclusions and Future Research

In this chapter we return to our research questions, and draw conclusions (Section 4.1). Then we give an answer to the initial problem statement (Section 4.2). Finally, we suggest possibilities for future research (Section 4.3).

Chapter contents: Conclusions, Research Questions, Problem Statement, Future Research

4.1 Answers to the Research Questions

In drawing our conclusions we will address our research questions as stated in Section 1.6. After this, our problem statement will be addressed.

4.1.1 Monte-Carlo methods

Our first research question was as follows.

RQ1: Will other Monte-Carlo methods perform better than all-as-first?

We compared four Monte-Carlo methods for Phantom Go: (1) Monte-Carlo evaluation with standard sampling, (2) Monte-Carlo evaluation with all-as-first sampling, (3) UCT with late random opponent-move guessing, and (4) UCT with early probabilistic opponent-move guessing. The empirical finding by Cazenave (2006) that Monte-Carlo evaluation with all-as-first sampling yields a reasonable Phantom-Go player was reproduced.

The experimental results described in Subsection 3.2.1 show that Monte-Carlo evaluation with all-as-first sampling outperforms both Monte-Carlo evaluation with standard sampling and UCT with early probabilistic opponent-move guessing. While it does not outperform UCT with late random opponent-move guessing, it does play at a comparable level.

Therefore, the answer to our first research question is negative: of the three other Monte-Carlo methods investigated, none performs better than all-as-first. Of course, this does not imply that other MC-methods might not yield better results (see Section 4.3).

4.1.2 Tree search

The second research question was as follows.

RQ2: How much improvement will tree search produce?

Surprisingly, our experiments showed that players based on UCT could not outperform the best player based on Monte-Carlo evaluation although a UCT-based player can reach a comparable level of play. The reason for this is that UCT is considerably weaker in the early stages of a game. However, UCT-based methods certainly did improve play at later stages in the game when situations become more complex. As our experiments with hybrid players showed, MCTS will produce better moves when used at that stage of the game.

There are still some problems when applying tree search to Phantom Go which hold for all complex games with uncertainty about the opponent's position. These originate from the overestimation of the value of a move because of unknown opponent stones. Two ways of dealing with these unknown stones in MCTS were suggested: (1) late random opponent-move guessing and (2) early probabilistic opponent-move guessing. However, these have not been able to make an improvement significant enough to outperform Monte-Carlo evaluation with all-as-first sampling.

4.1.3 Integrating domain-specific knowledge

Our third research question was as follows.

> RQ3: Can we improve on standard Monte-Carlo algorithms by integrating domain-specific knowledge?

There are two ways in which we tried to improve on the basic Monte-Carlo algorithms. We attempted to increase the accuracy of the unknown opponent stones and we tried to select better moves while traversing the search tree. The heuristic we used for both approaches was based on the concept of influence. However, the total gain in influence did not turn out to be a very good heuristic to predict the most important moves.

Our results indicate that the approaches of using domain knowledge to improve the moves selected in the tree and the unknown opponent stones is valid. Moves with a higher evaluation by our heuristic got played, or selected as unknown stone, more often. It is necessary to be careful when choosing a heuristic though, because as our experiments have shown a bad heuristic will produce results worse than, or at best equal to, those of random selection.

4.1.4 The evaluation of referee feedback

Our fourth research question was as follows.

> RQ4: How big will the drawbacks of our (possibly faulty) evaluation of referee feedback be?

Since our approach to deal with referee feedback can give an incorrect board position, we had to determine whether this leads to a decrease in playing strength. A partial solution to this problem has been implemented in our program. The player tries to kill completely surrounded groups. While this does lead to some successful captures, the behaviour is often invoked too late. Especially in the case in which part of what appears to be a solid group could have been caught, this behaviour is often not triggered until after the opponent has solved the problem.

However, in by far the most cases, our approach does generate a useful board position. While occasional mistakes are inconvenient, they do not prevent the program from playing a reasonably strong game. In some cases misinterpretations may lead to losing a game. Mistakes that could prove disastrous are a mistaken belief an opponent group could be killed (see for instance Appendix B.2) or an adversity to fill one's eyes until it is proven that stones are in atari (see for instance Appendix B.3). These mistakes only turn up in a small percentage of the games though, so in conclusion we may state that the drawbacks of our evalation of referee feedback are not very big. However, a better way of determining which intersections might not actually be occupied could still bring improvement, as less games will be lost due to mistakes. See Section 4.3 for further suggestions.

4.1.5 Uncovering information

The fifth research question was as follows.

> RQ5: How much information about the opponent's position is typically uncovered during the game, and can this information be maximised?

In uncovering information during the game a certain amount of luck is involved. Especially in the early stages trying to play on an occupied intersection or playing just beside it is less a matter of good play than plain luck. One effect is definitely clear though: the more information is uncovered, the bigger the chances are of more illegal moves. This is so because once some opponent stones are known, the intersections near those stones will become good moves to play. They are also more likely to be occupied by an opponent stone.

Even though this indicates the importance of acquiring information, we have not tried to include a separate heuristic trying to to find out more about the opponent's position. There are two reasons for this:

1. The intersections that are of the biggest interest to the player are usually the moves that get the best Monte-Carlo evaluation. Therefore the player often tries to play on these intersections anyway. A separate behaviour would be a waste of time.

2. Trying to play on an intersection with the intention of acquiring information, i.e., hoping for the referee to say 'illegal', might conflict with the more important goal of playing good moves. If a move turns out to be legal after all, the player might end up with a useless stone.

The question of uncovering information is second to the question of dealing with unknown stones during Monte-Carlo evaluation. One exception to this could be a behaviour which 'guards your back', trying to make sure the opponent does not try to build groups in territory the player regards as unthreatened.

4.1.6 Scoring methods

The last research question was as follows.

> RQ6: What will be the impact of the scoring method on the behaviour of the MC methods?

In answer to this last question we can be clear. The Monte-Carlo methods all accomplish better results when using 0/1 scoring as opposed to territory scoring. This is mainly due to the fact that they focus on winning the game, instead of focussing on claiming territory, which is merely the means towards winning the game. Strategically, 0/1 scoring is to be preferred.

Having said that, 0/1 scoring does seem to need much more simulated games than territory scoring to converge to the optimal tactical moves. Especially in UCT-based methods, the search will not focus on moves which are important to the outcome of the tree search as fast as when using territory scoring. Perhaps we should not abolish territory scoring altogether just yet.

4.2 Problem Statement Revisited

In Chapter 1, we stated our problem as:

> How can we apply the existing knowledge about Monte-Carlo methods to computer Phantom Go in order to produce a strong playing program?

After having answered the research questions in the previous section, we believe to have achieved more insight into the ways MC methods can be applied to Phantom Go and, more generally, to problems with uncertainty. However, our research objective was not met. At the Computer Olympiad, we lost to GoLois and our methods do not significantly improve on the methods used by Cazenave (2006). Obviously, more research is needed to reliably apply Monte-Carlo approaches to domains with uncertainty.

4.3 Future Research

A number of directions for future research in Phantom Go come to mind. We describe eight of them.

1. One idea is to see whether applying grouping nodes for UCT in Phantom Go as was tested for regular Go by Saito *et al.* (2007) is feasible. This approach could counterbalance the overestimation-effect explained in Subsection 3.2.1.

2. While our experiments with the total gain in influence as a heuristic were unsuccessful, other heuristics can be thought of. Better heuristics can, and should, be searched for.

3. Much research has been done on patterns in regular Go. Especially the work of Gelly *et al.* (2006) on the use of patterns in UCT-based algorithms seems a viable approach to build better search trees. This approach might work for Phantom Go too. Apart from this, these patterns might also be used to guess the unknown opponent stones, based on the stones that are already known.

4. Small tactical searches which do account for all configurations of opponent stones could be incorporated into the Monte-Carlo approaches. While Parker *et al.* (2006) showed that in complex games with partial information overconfidence outperforms paranoia, overconfidence about the strength of one's own groups is still a big problem in automated players. A mechanism to evaluate the life-or-death status of a group could improve our performance. This could be implemented like the Metapositions explored by Sakuta and Iida (2000). While this approach seems infeasible for a complete game of Phantom Go, it would offer a way of determining whether groups can be caught or not.

5. In out experiments with early probabilistic opponent-move guessing we tried to exploit some of the information already produced by the UCT-algorithm. The way in which we decided to use it influenced the search itself, leading to worse results. However, this does not negate the fact that there is much useful information in the search tree. Especially in situations dealing with uncertainty it could give a major improvement if we could exploit this information. More research will be needed to see whether a way of using information already available in the search tree can be devised that does not interfere with the search itself. One idea here would be to use the average territory score of a node as a tiebreaker, while using 0/1 scoring as the main valuating method.

6. Recently, a new approach to combine all-as-first evaluation and UCT has been proposed by Gelly and Silver (2007). The results of this approach in regular Go are promising. Gelly and Silver use a gradual shift from all-as-first to UCT, using the values determined by all-as-first in constructing their search tree. This seems like a much more graceful solution than our hybrid methods, and we would like to see what results this approach brings in Phantom Go.

7. As shown in Chapter 3, the result of tree-based search methods improves when more time for simulating games is available. Research into parallelization like the work by Cazenave and Jouandeau (2007) and the use of faster machines will probably also improve the performance of MCTS for Phantom Go.

8. Finally, it would be interesting to see if a different way of dealing with perceived opponent stones can be devised. Especially in the case of former suicide positions, our current approach sometimes misses opportunities. An interesting idea in this is to assign a certain *belief value* to intersections not occupied by the own stones, instead of the all-or-nothing approach of placing an opponent stone or not. Finding out how to work with these belief values in Monte-Carlo evaluation is a promising research objective.

In conclusion: many items can and should still be investigated to improve the level of Phantom Go programs.

References

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002a). Finite Time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, Vol. 47, No. 2/3, pp. 235–256. [4]

Auer, P., Cesa-Bianchi, N., Freund, Y., and Schapire, R. E. (2002b). The Nonstochastic Multiarmed Bandit Problem. *SIAM Journal on Computing*, Vol. 32, No. 1, pp. 48–77. [4]

Bolognesi, A. and Ciancarini, P. (2006). Searching over Metapositions in Kriegspiel. *Computers and Games, 4th International Conference, CG 2004, Ramat-Gan, Israel, July 5-7, 2004, Revised Papers* (eds. H. Jaap van den Herik, Yngvi Björnsson, and Nathan S. Netanyahu), Vol. 3846 of *Lecture Notes in Computer Science*, pp. 246–261, Springer. [3]

Bolognesi, A. and Ciancarini, P. (2007). Moving in the Dark: Progress through Uncertainty in Kriegspiel. *Proceedings of the Computer Games Workshop 2007 (CGW2007)* (eds. H. J. van den Herik, Jos Uiterwijk, Mark Winands, and Maarten Schadd), number 07-06 in MICC Technical Report Series, pp. 27–37. [3]

Bouzy, B. (2002). A Small Go Board Study of Metric and Dimensional Evaluation Functions. *3rd Computer and Games Conference* (eds. J. Schaeffer, M. Müller, and Yngvi Björnsson), Vol. 2883 of *Lecture Notes in Computer Science*, pp. 376–392, Springer. [3, 5]

Bouzy, B. (2003). Mathematical Morphology Applied to Computer Go. *IJPRAI*, Vol. 17, No. 2, pp. 257–268. [5]

Bouzy, B. (2005). Associating Domain-dependent Knowledge and Monte Carlo Approaches within a Go Program. *Information Sciences, Heuristic Search and Computer Game Playing IV*, Vol. 175, No. 4, pp. 247–257. [5]

Bouzy, B. (2006a). Move Pruning Techniques for Monte-Carlo Go. *Advances in Computer Games: 11th International Conference, Acg 2005, Taipei, Taiwan, September 6-8, 2005, Revised Papers* (eds. H.J. van den Herik, S.-C. Hsu, T.-S. Hsu, and H.H.L.M. Donkers), Vol. 4250 of *Lecture Notes in Computer Science*, pp. 104–119, Springer. [4]

Bouzy, B. (2006b). History and Territory Heuristics for Monte-Carlo Go. *New Mathematics and Natural Computation*, Vol. 2, No. 2, pp. 1–8. [5]

Bouzy, B. and Cazenave, T. (2001). Computer Go: An AI-Oriented Survey. *Artificial Intelligence*, Vol. 132, No. 1, pp. 39–103. [4]

Bouzy, B. and Helmstetter, B. (2004). Monte Carlo Go Developments. *Advances in Computer Games, Many Games, Many Challenges, 10th International Conference, ACG 2003, Graz, Austria, November 24-27, 2003, Revised Papers* (eds. H. Jaap van den Herik, Hiroyuki Iida, and Ernst A. Heinz), Vol. 263 of *IFIP*, pp. 159–174, Kluwer. [4, 5, 13]

Brügmann, B. (1993). Monte Carlo Go. White paper. [4, 13]

Cazenave, T. (1998). Strategic Evaluation in Complex Domains. *FLAIRS 98, Sanibel.* [5]

Cazenave, T. (2003). Metarules to Improve Tactical Go Knowledge. *Information Sciences*, Vol. 154, Nos. 3–4, pp. 173–188. [5]

Cazenave, T. (2006). A Phantom Go Program. *Advances in Computer Games, 11th International Conference, ACG 2005, Taipei, Taiwan, September 6-9, 2005. Revised Papers* (eds. H. J. van den Herik, S.-C. Hsu, T.-S. Hsu, and H. H. L. M. Donkers), Vol. 4250 of *Lecture Notes in Computer Science*, pp. 120–125, Springer. [2, 3, 4, 6, 13, 26, 31, 33]

Cazenave, T. and Helmstetter, B. (2005). Combining tactical search and Monte-Carlo in the game of Go. *IEEE 2005 Symposium on Computational Intelligence and Games* (eds. G. Kendall and S. Lucas), pp. 171–175. [5]

Cazenave, T. and Jouandeau, N. (2007). On the Parallelization of UCT. *Proceedings of the Computer Games Workshop 2007 (CGW2007)* (eds. H. J. van den Herik, Jos Uiterwijk, Mark Winands, and Maarten Schadd), number 07-06 in MICC Technical Report Series, pp. 93–101. [34]

Chaslot, G., Saito, J.-T., Uiterwijk, J. W. H. M., Bouzy, B., and Herik, H. J. van den (2006). Monte-Carlo Strategies for Computer Go. *BNAIC06: Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence* (eds. P.Y. Schobbens, W. Vanhoof, and G. Schwanen), pp. 83–90, University of Namur, Namur, Belgium. [17]

Chaslot, G. M. J. B., Winands, M. H. M., Uiterwijk, J. W. H. M., and Herik, H. J. van den (2007). Progressive Strategies for Monte-Carlo Tree Search. *JCIS workshop.* [15]

Ciancarini, P. and Favini, G.P. (2007). A Program to Play Kriegspiel. *ICGA Journal*, Vol. 30, No. 1, pp. 3–24. [3]

Davies, J. (1992). The Rules of Go. *The Go Player's Almanac* (ed. Richard Bozulich). Ishi Press. http://www.cs.cmu.edu/ wjh/go/rules/Chinese.html. [2]

Farnebäck, G. (2002). Specification of the Go Text Protocol, version 2, draft 2. http://www.lysator.liu.se/ gunnar/gtp/gtp2-spec-draft2/gtp2-spec.html. [9]

Gelly, S. and Silver, D. (2007). Combining online and offline knowledge in UCT. *ICML '07: Proceedings of the 24th international conference on Machine learning* (ed. Z. Ghahramani), pp. 273–280, ACM Press, New York, NY, USA. ISBN 978–1–59593–793–3. [34]

Gelly, S., Wang, Y., Munos, R., and Teytaud, O. (2006). Modification of UCT with Patterns in Monte-Carlo Go. Technical report, INRIA. [5, 34]

Herik, H. J. van den, Uiterwijk, J.W.H.M, and Rijswijck, J. van (2002). Games solved: Now and in the Future. *Artificial Intelligence*, Vol. 134, pp. 277–311. [1]

Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo Planning. *17th European Conference on Machine Learning (ECML)* (eds. J. Fürnkranz, T. Scheffer, and M. Spiliopoulou), Vol. 4212 of *LNAI*, pp. 282–293, Springer. [4]

Kocsis, L., Szepesvári, C., and Willemson, J. (2006). Improved Monte-Carlo Search. Working paper. [6, 13]

Parker, A., Nau, D., and Subrahmanian, V.S. (2005). Game-Tree Search with Combinatorially Large Belief States. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 254–259. [3]

Parker, A., Subrahmanian, V.S., and Nau, D. (2006). Overconfidence or Paranoia? Search in Imperfect-Information Games. *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 1045–1050. [3, 34]

Rijswijck, J. van and Müller, M. (2001). Report on the Second International Conference on Computers and Games. *ICGA Journal*, Vol. 24, No. 1, pp. 49–54. [3]

Saito, J.-T., Winands, M. H. M., Uiterwijk, J. W. H. M., and Herik, H. J. van den (2007). Grouping Nodes for Monte-Carlo Tree Search. *Proceedings of the Computer Games Workshop 2007 (CGW2007)* (eds. H. J. van den Herik, Jos Uiterwijk, Mark Winands, and Maarten Schadd), number 07-06 in MICC Technical Report Series, pp. 125–132. [33]

Sakuta, M. and Iida, H. (2000). Solving Kriegspiel-Like Problems: Examining Efficient Search Methods. *Computers and Games 2000*, Vol. 2063 of *Lecture Notes in Computer Science*, pp. 56–75, Springer-Verlag. [3, 34]

Schaeffer, J. (1989). The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, No. 11, pp. 1203–1212. ISSN 0162–8828. [5]

Schaeffer, J. and Plaat, A. (1997). Kasparov versus Deep Blue: The Rematch. *ICCA Journal*, Vol. 20, No. 2, pp. 95–101. [1]

Werf, E.C.D van der (2005). *AI Techniques for the Game of Go.* Ph.D. thesis, Universiteit Maastricht, The Netherlands. [5]

Zobrist, A. (1969). A Model of Visual Organization for the Game of Go. *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 103–111, AFIPS Press, Montvale, NJ. [5, 15]

Appendix A

Algorithms

Our main algorithms are given in (Java-style) pseudocode.

A.1 Standard Monte-Carlo Evaluation

```
function suggestmove
{
 double[][] results;
 while(currenttime - starttime < timeLeft / timeFactor)
 {
  for(i = 0; i < boardSize; i++)
  {
   for(k = 0; k < boardSize; k++)
   {
    if(board(i+1, k+1) == EMPTY &&
     !eye(i+1, k+1))
    {
     clone = clonedadmin();
     clone.play(i+1, k+1)
     clone.fillUnknownOpponent();
     score = simGame(clone);
     results[i][k] += score;
    }
   }
  }
 }
 if(color == BLACK)
 {
  return index of argmax(results);
 }
 else if(color == WHITE)
 {
  return index of argmin(results);
 }
}

function simgame(clone)
{
 possibles = get empty positions(clone);
 while(!bothpassed)
 {
  move = randomNoEye(possibles);
  if(move < 0)
  {
   if(onePassed)
   {
    bothPassed = true;
   }
   else
   {
    onePassed = true;
    changeColor();
   }
   else
   {
    onePassed = false;
    taken = clone.playmove(move);
    possibles.add(taken);
    changeColor();
   }
  }
  if(use01)
  {
   return Utils.score01(clone);
  }
  return Utils.score(clone);
 }
}
```

A.2 All-as-First Monte-Carlo Evaluation

```
function suggestmove
{
 double[][] results;
 int[][] games;
 double[][] oppResults;
 int[][] oppGames;

 while(currenttime - starttime < timeLeft / timeFactor)
 {
  clone = clonedadmin();
  clone.fillUnknownOpponent();
  score = simGame(clone);
  moves = getMoveStack();
  colors = getMoveColors();
  playedpositions;
  for(i = 0; i < moves; i++)
  {
   if(moves[i] ! member of playedpositions)
   {
    playedpositions.add(moves[i]);
    if(colors[i] == color)
    {
     results[getX(moves[i])][getY(moves[i])] += score;
     games[getX(moves[i])][getY(moves[i])]++;
    }
    else
    {
     oppResults[getX(moves[i])][getY(moves[i])] += score;
     oppGames[getX(moves[i])][getY(moves[i])]++;
    }
   }
  }
 }
 if(color == BLACK)
 {
  return index of argmax(results/games - oppresults/oppgames);
 }
 else if(color == WHITE)
 {
  return index of argmin(results/games - oppresults/oppgames);
 }
}
```

A.3 UCT with Late Random Opponent-Move Guessing

```
function suggestmove
{
 tree = initiateTree();
 while(currenttime - starttime < timeLeft / timeFactor)
 {
  visitedNodes;
  clone = clonedadmin();
  currentNode = tree.root;
  visitedNodes.add(currentNode);
  while(currentNode.hasChild)
  {
   if(blacktomove)
   {
    currentNode = argmax(currentnode.children.uctscore)
   }
   else
   {
    currentNode = argmin(currentnode.children.uctscore)
   }
   clone.play(currentNode.move);
   vistitedNodes.add(currentNode);
  }
  if(currentNode.visited > 0)
  {
   expandNode(currentNode);
   if(blacktomove)
   {
    currentNode = argmax(currentnode.children.uctscore)
   }
   else
   {
    currentNode = argmin(currentnode.children.uctscore)
   }
   clone.play(currentNode.move);
   vistitedNodes.add(currentNode);
  }
  clone.fillUnknownOpponent();
  score = simGame(clone);
  for(i = 0; i < visitedNodes; i++)
  {
   visitedNodes[i].value += score;
   visitedNodes[i].visited ++;
  }
 }
 return index of argmax(tree.root.children.visited);
}
```

A.4 UCT with Early Probabilistic Opponent-Move Guessing

```
function suggestmove
{
 tree = initiateTree();
 while(currenttime - starttime < timeLeft / timeFactor)
 {
  visitedNodes;
  clone = clonedadmin();
  currentNode = tree.root;
  visitedNodes.add(currentNode);
  while(currentNode.hasChild)
  {
   if(blacktomove)
   {
    if(depth == 2)
    {
     fillWithProb(clone);
    }
    currentNode = argmax(currentnode.children.uctscore) && currentNode.move == EMPTY
   }
   else
   {
    if(depth == 2)
    {
     fillWithProb(clone);
    }
    currentNode = argmin(currentnode.children.uctscore) && currentNode.move == EMPTY
   }
   clone.play(currentNode.move);
   vistitedNodes.add(currentNode);
  }
  if(currentNode.visited > 0)
  {
   expandNode(currentNode);
   if(blacktomove)
   {
    currentNode = argmax(currentnode.children.uctscore) && currentNode.move == EMPTY
   }
   else
   {
    currentNode = argmin(currentnode.children.uctscore) && currentNode.move == EMPTY
   }
```

```
    clone.play(currentNode.move);
    vistitedNodes.add(currentNode);
   }
   clone.fillUnknownOpponent();
   score = simGame(clone);
   for(int i = 0; i < visitedNodes; i++)
   {
    visitedNodes[i].value += score;
    visitedNodes[i].visited ++;
   }
  }
   return index of argmax(tree.root.children.visited);
 }

 function fillWithProb(clone)
 {
  for(i = 0; i < currentNode.children; i++)
  {
   random = randomNumber[0, 1];
   if((C * (children[i].visited * currentNode.visited) * unknownstones) > random)
   {
    clone.play(children[i].move, opposingColor);
   }
  }
```

A.5 The Influence Heuristic

The influence heuristic is used in two ways. Subsection A.5.1 describes the heuristic itself. Subsection A.5.2 describes how it is used to guess opponent stones. Subsection A.5.3 describes how it is used to determine which node to explore during tree construction.

A.5.1 Computing the total gain in influence

```
 function computeInfluences
 {
  totalInf = getTotalInfluence(board)
  int[][] ownInf;
  int[][] opponentInf;
  for(i = 0; i < possibles; i++)
  {
   board.change(i.move, ownColor);
   ownInf[getX(i.move)][getY(i.move)] = getTotalInfluence(board);
   board.change(i.move, oppColor);
   opponentInf[getX(i.move)][getY(i.move)] = getTotalInfluence(board);
   board.change(i.move, EMPTY);
  }
 }

 function getTotalInfluence(board)
 {
  infboard = computeInfluences(board)
  totalInf = 0;
  for{i = 1; i < boardsize; i++)
  {
   for{k = 1; k < boardsize; k++)
   {
```

```
   totalInf += infBoard[i][k];
  }
 }
 return totalInf;
}
```

A.5.2 Influence-based opponent-move guessing

```
function fillWithInf(clone)
{
 totalDif = 0;
 for(i = 1; i <= boardSize; i++)
 {
  for(k = 1; k <= boardSize; k++)
  {
   if(clone[i][k] == EMPTY)
   {
    if(!eye && !suicide && !ko)
    {
     totalDif += opponentInf[i][k];
    }
   }
  }
 }
 loop1: for(loopno = 0; loopno < stonesMiss; loopno++)
 {
  double pick = random[0, 1];
  tot = 0;
  loop2: for(i = 0; i < opponentInf; i++)
  {
   for(k = 0; k < opponentInf; k++)
   {
    if(cloned.getBoardPoint((i+1), (k+1)) == EMPTY)
    {
     if(!eye && !suicide && !ko)
     {
      tot += opponentInf[i][k];
      if((tot / totalDif) > pick)
      {
       clone.playMove(i+1, k+1, opponentColor);
       totalDif -= opponentInf[i][k];
       break loop2;
      }
     }
    }
   }
  }
 }
 return clone;
}
```

A.5.3 Influence-based move selection

This function is called from the UCT algorithm instead of the function uctScore.

```
function infUctScore(node)
{
 uctS = uctScore(node);
 if(colorToMove == ownColor)
 {
  dif = ownInf[getX(node.move)][getY(node.move)] - totalInf;
 }
 if(colorToMove == oppColor)
 {
  dif = opponentInf[getX(node.move)][getY(node.move)] - totalInf;
 }
 if(parent.visited == 0)
 {
  return uctS + (infFactor * dif);
 }
 double infS = (infFactor / sqrt(parent.visited)) * infDif;
 return uctS + infS;
}
```

Appendix B

Example games

B.1 InTheDark - GoLois, Game 2

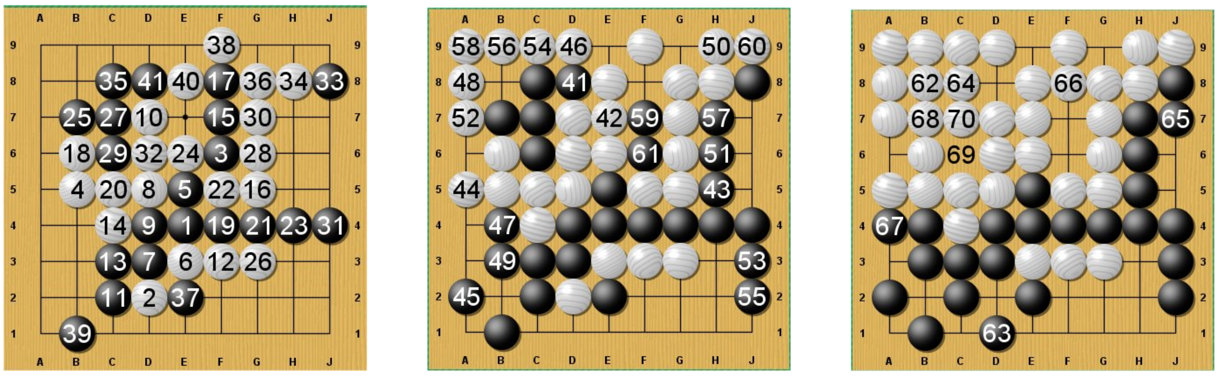


Figure B.1: The reference boards during the game.

Game 2 of the match was a pretty straightforward game, good for explaining the basics. The reference boards after moves 41, 61, and 70 are given in Figure B.1.

```
InTheDark (black)         GoLois (white)
b: E4                     w: D2
b: F6                     w: B5
b: E5                     w: E3
b: (E3) D3                w: D5
b: D4                     w: D7
b: (D2) C2                w: F3
b: C3                     w: (E4) C4
b: F7                     w: G5
b: F8                     w: B6
```

InTheDark, using MC_{all} at this stage, is playing strongly connected stones in the opening stages of this game. GoLois is playing much more unconnected stones.

```
b: (F3 G5) F4             w: (C3) C5
b: G4                     w: (E5 C2) F4
b: H4                     w: E6
```

InTheDark Is trying to divide the board into three separate areas, thus making it hard for GoLois to make living groups. However, it does not realise that F6, F7 and F8 are already cut off from the rest of its stones. It is playing too confidently because the UCT_{prob} algorithm assumes, wrongly, that it can always react to a move at F5 or E6 by connecting at the other position.

```
b: B7                        w: G3
b: C7                        w: (D3 F6) G6
b: (D7) C6                   w: (F7 F4 G4) G7
b: J4                        w: (F8) D6
b: J8                        w: H8
b: (C4) C8                   w: G8
b: E2                        w: F8
b: B1                        w: (C7) E8
b: (D5) D8
```

Now we arrive at the position as shown in the left board of Figure B.1. The belief states of InTheDark and GoLois are shown in Figure B.2. As you can see, GoLois has much information about opponent stones, where InTheDark has discovered very few opponent stones. This is a big disadvantage here, because it has never realised that the stones at F6, F7 and F8 are threatened.

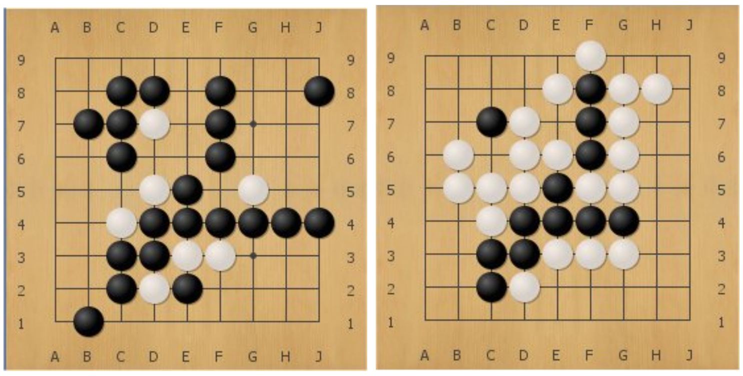


Figure B.2: The board as perceived by InTheDark (left) and GoLois (right) after move 41.

```
                             w: E7
b: (D6 C5) H5                w: (B7) A5
b: A2                        w: D9
b: B4                        w: A8
b: (B5) B3                   w: (J8) H8
b: (A8) H6                   w: A7
b: J3                        w: C9
b: J2                        w: B9
b: H7                        w: A9
b: F7                        w: J9
b: F6
```

Now we arrive at the position as shown in the middle board of Figure B.1. The belief states of InTheDark and GoLois are shown in Figure B.3. After the capture at move 42, InTheDark still could have won the game by making a living group in the top left corner and defending its territory at the bottom of the board. However, since it did not fully realise these positions were threatened it chose to do something else. Because UCT_{prob} is using territory scoring here, instead of consolidating a potentially won position, it tries to claim more territory by venturing into the upper right corner. GoLois plays this much better, because it realizes that it can win the game by holding on to the territory it already controls. By making sure there can never be a living group in the top left corner, even though it does not know how many stones are already there, it effectively wins the game.

```
                             w: B8
b: D2                        w: C8
b: J7                        w: F8
b: A4                        w: B7
b: (J9 H9 A5) C6             w: C7
```

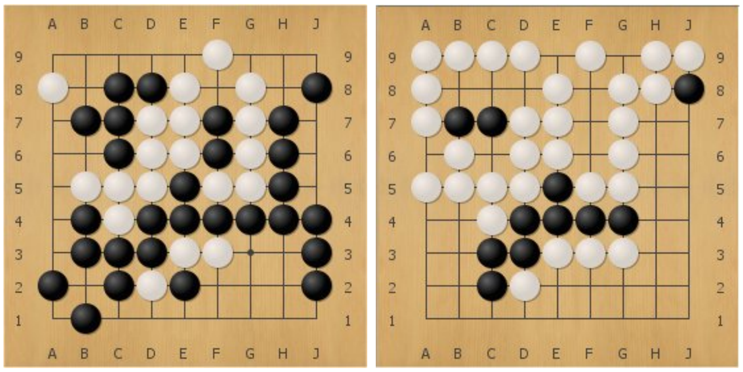


Figure B.3: The board as perceived by InTheDark (left) and GoLois (right) after move 61.

Game over, White wins by 2.5.

B.2 InTheDark - GoLois, Game 6

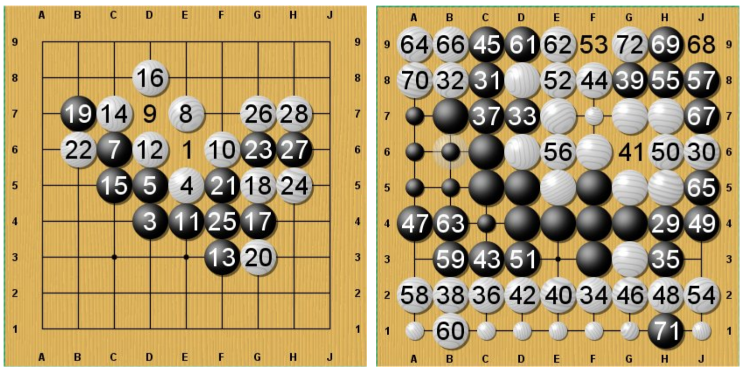


Figure B.4: The reference boards during the game.

Game 6 was something like a comedy of errors and misinterpretations. The reference boards after moves 28 and 72 are given in Figure B.4.

```
InTheDark (black)        GoLois (white)
b: E6                    w: F4
b: D4                    w: E5
b: (E5) D5               w: D6
b: (D6) C6               w: E7
b: D7                    w: F6
b: (F6) E4               w: D6
b: (F4 E6) F3
```

An opening in which both players are fighting over control of the middle of the board. A bug causes InTheDark to play on a position it knows is forbidden because of ko (E6). After this it fills the position with an opponent stone. While this may seem like an innocent mistake, the consequences of this will become apparent in the rest of the game.

```
                         w: (D4 E4 F3) C7
b: C5                    w: D8
b: G4                    w: G5
b: B7                    w: G3
```

```
b: (G3) F5                  w: (F4) B6
```

Apparently GoLois is suffering from a similar bug. F4 is also a move forbidden because of ko.

```
b: G6                       w: (C5) H5
b: (G5) F4                  w: (G6) G7
b: (G7) H6                  w: (H6) H7
```

InTheDark can capture at E6, but does not realise this because of the mistake it made earlier. It also does not realise its stones at G6 and H6 are in atari, so it continues to try and capture the white stones at G5 and H5. See Figure B.5.

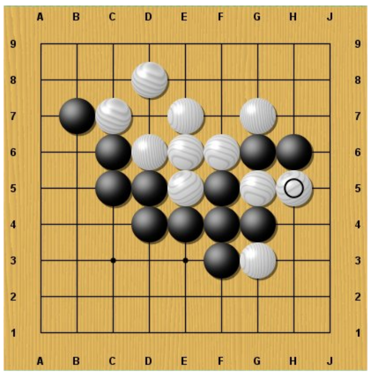


Figure B.5: The board as perceived by InTheDark after move 28.

```
b: (H5) H4                  w: J6
b: C8                       w: (B7) B8
b: D7                       w: (C7 H4) F2
b: H3                       w: (H3) C2
b: C7                       w: B2
b: G8                       w: E2
b: (E2 F2) G6               w: D2
b: C3                       w: F8
b: C9                       w: G2
b: A4                       w: H2
b: (F7) J4                  w: H6
b: D3                       w: E8
b: F9                       w: J2
b: H8                       w: E6
b: J8                       w: A2
b: (D2 E8 C2 G2 F8) B3      w: B1
b: D9                       w: (D9) E9
```

Because of a misrepresentation of enemy stones, InTheDark believes it can kill the white group in the top right corner. However, this group is unconditionally alive. While putting its efforts into this futile attempt, it does not realise GoLois is building a group inside territory it believes it controls. See Figure B.6.

```
b: (E9 A2 B2 H2) B4         w: A9
b: J5                       w: B9
b: J7                       w: (C9 F9) J9
b: (G6 ....) H9             w: A8
b: (B8) H1                  w: (G8) G9
```

At the move (G6 ....) InTheDark tried to find a killing position for the top right white group by playing every possible position. None existed. This meant it had lost the game. White wins by 17.5 points.

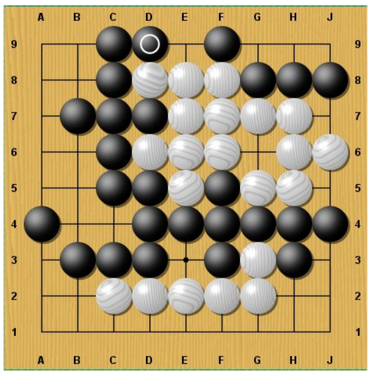


Figure B.6: The board as perceived by InTheDark after move 61.

B.3 UCT_{rand} - UCT_{prob}

A game between two MCTS methods, UCT with late random opponent-move guessing and UCT with early probabilistic opponent-move guessing. This game was played on a Pentium III processor with 30 minutes time per player.

```
b: E5                    w: B2
b: G3                    w: (E5) H6
b: G4                    w: D6
b: C6                    w: E6
b: D4                    w: D5
b: (H6 E6 D5 D6) C5      w: (D4) F5
b: C7                    w: B4
b: F6                    w: E4
b: G5                    w: C4
b: E3                    w: (G4) F3
b: (C4) D3               w: (D3 G5) G6
b: C3                    w: (F6) F7
b: G2                    w: H3
b: (B4) F4               w: C8
```

The reference board and belief states for this position are shown in Figure B.7. Remarkable in this position is the discrepancy in the amount of information the players have about the opponent stones. While Black knows nearly everything, White has very little information. This shows that taking opponent stones also has a drawback: your opponent gets more information.

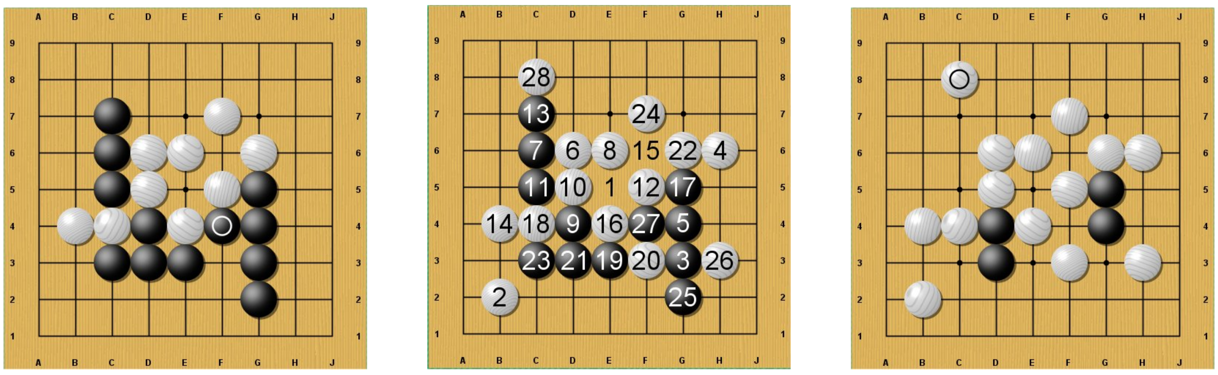


Figure B.7: The reference board (centre) and the belief-states of Black (left) and White (right) after move 28.

```
b: (C8) B5                 w: D2
b: B3                      w: (E3 G3 C3 B3) B7
b: A4                      w: (F4 G2) J6
b: B8                      w: (C6) D7
b: H5                      w: E5
b: F1                      w: (C7 H5) J5
b: (B7) D8                 w: (B8) B9
b: (D7) A8                 w: H2
b: (J5) C9                 w: F2
b: (F3 F2) E2
```

The reference board and belief states for this position are shown in Figure B.8. After the capturing moves at E4, C9, and E2 White has much information about the opponent as well. It urgently needs to secure some territory though.

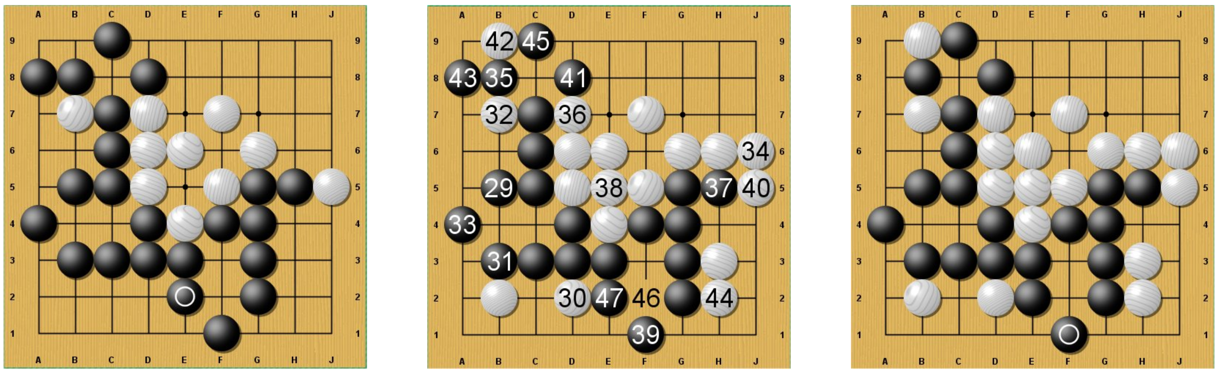


Figure B.8: The reference board (centre) and the belief-states of Black (left) and White (right) after move 47.

```
                           w: E8
b: F2                      w: B1
b: A6                      w: (F2) H4
b: D9                      w: C2
b: (B1) B4                 w: A2
b: A9                      w: D1
b: (D2 D1 A2 B2 C2) J4     w: (J4) J3
b: H1                      w: F8
b: (J6 E8 H2 E5) B6        w: (H1) F9
```

The reference board and belief states for this position are shown in Figure B.9. Black thought it was ahead, but White has managed to form a small living group in the left bottom corner before Black found out about it. White now has a good chance of winning the game. Black attacked at J4, but that stone was taken.

```
b: (F8) G7                 w: (A8 C4 G7) H7
b: J2                      w: (D9) A3
b: G8                      w: H8
b: (H8) J7                 w: G9
b: E9                      w: (A5) G7
b: (H3) J4                 w: J3
b: H3
```

The reference board and belief states for this position are shown in Figure B.10. White should have connected the stones in the left bottom corner with the rest of its stones. However, the heuristic which prevents playing in its own eyes prevented that. It would not allow that move until the group is in atari. White never realised it was in atari. By losing the group, it lost the game. Black defends well by playing H3. The last few moves were played as follows.

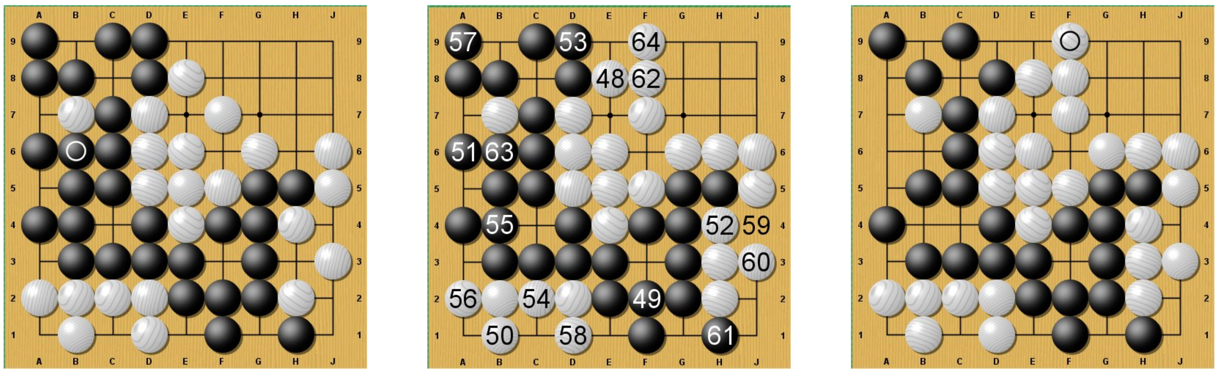


Figure B.9: The reference board (centre) and the belief-states of Black (left) and White (right) after move 64.

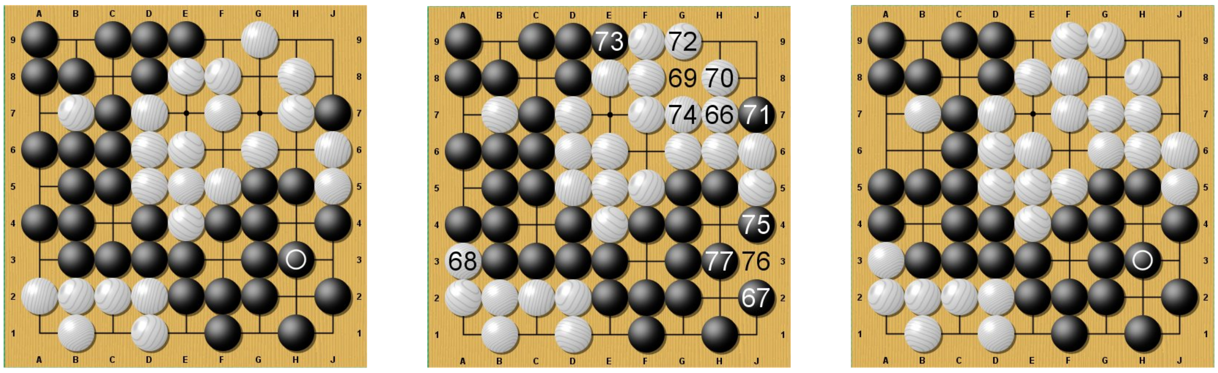


Figure B.10: The reference board (centre) and the belief-states of Black (left) and White (right) after move 77.

```
                            w: (A7 A6) E1
b: (E1 A3) A7               w: (J7) J8
b: (G7 F9) J9               w: (E9) H9
b: PASS                     w: PASS
score: 2.5
```

Black wins by 2.5 points.

B.4 UCT_{prob} - MC_{all}

This is a game played during our experiments. It is between UCT_{prob} (Black) and MC_{all} (White), both using 0/1 scoring. It was played on an AMD Opteron 3.4 processor with 20 minutes time for each player.

```
b: E7                                   w: E5
b: E6                                   w: E4
b: (E5) D5                              w: (E6) F6
b: (E4) D3                              w: F5
b: E3                                   w: F7
b: F3                                   w: (E3 D3 F3) D4
b: D6                                   w: C4
b: (D4 C4 F6 F5) E8                     w: C3
b: (C3) C2                              w: (C2) B3
b: B5                                   w: F8
b: C5                                   w: G4
b: B2                                   w: G3
```

```
b: B4                                   w: G2
b: (G4 B3 G3) A3                        w: G5
b: (F7 G2) F2
```

The reference board and belief states for this position are shown in Figure B.11. Black is defending here, but also has much more information. The all-as-first algorithm (White) tries to divide the board into smaller pieces. However, it does not close the last intersections on the edge. This is a drawback of the all-as-first algorithm. Since there are always multiple ways in which you can connect the string to the edge of the board, the importance of doing this is overlooked by the algorithm. Thus these moves are not seen as very urgent. This allows Black to play at A3 and connect its two groups.

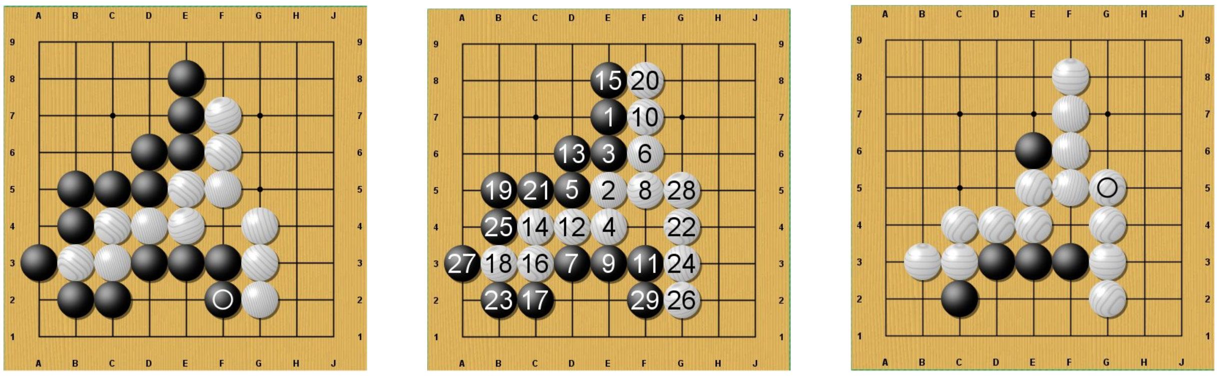


Figure B.11: The reference board (centre) and the belief-states of Black (left) and White (right) after move 29.

```
                                        w: (B2 F2) F9
b: (F8) G1                              w: (G1) H2
b: E9                                   w: (B5) H6
b: A2                                   w: H1
b: F1                                   w: H7
b: (F9 H1 H2) D2                        w: G7
b: F4
```

The reference board and belief states for this position are shown in Figure B.12. The Black move at F4 wins the game. Black has now 0.5 point more than White. While the White moves at G7, H7, and H6 were good moves for defending the territory, they were not the most important moves. White did not realise this because it has very little information about the opponent's position, and thinks it can easily claim some more territory. At the next move White will realise it has a problem.

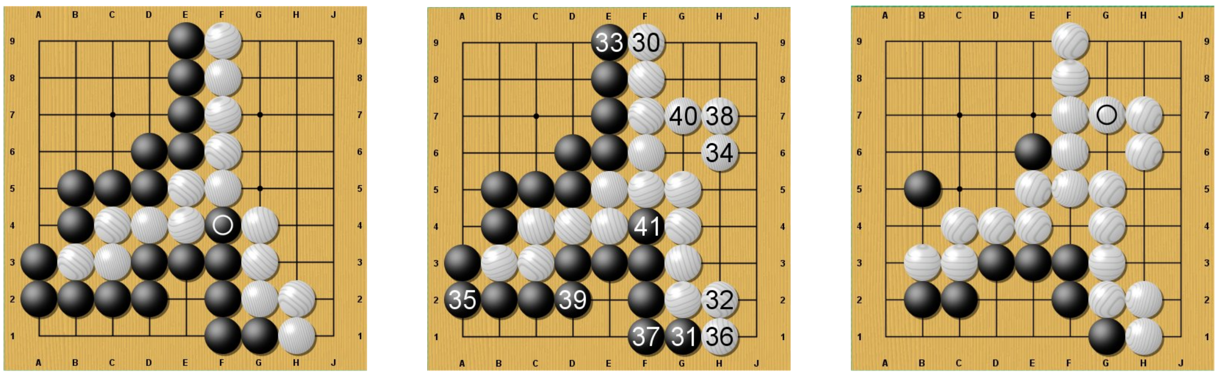


Figure B.12: The reference board (centre) and the belief-states of Black (left) and White (right) after move 41.

```
                                          w: (C5 E7 E8 E9 A3 D5 F4 B4 F1
                                              D2 D6 A2) A4
b: C8                                     w: (C8) C7
b: (C7) D8                                w: (D8) B8
b: (A4 B8) A5                             w: B6
b: C9                                     w: A6
b: (A6 B6) B9                             w: (B9 C9) A8
```

The reference board and belief states for this position are shown in Figure B.13. White realised it had to try something, and attempted to form a group in the upper left corner. After the Black move at B9 this is no longer possible and the game is over. The moves played until the board was filled are given below.

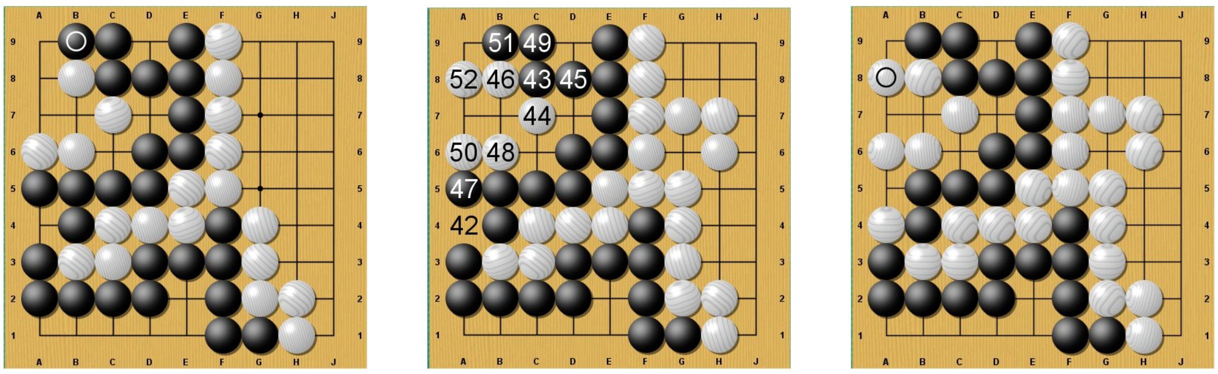


Figure B.13: The reference board (centre) and the belief-states of Black (left) and White (right) after move 52.

```
b: J1                                     w: A1
b: A7                                     w: C6
b: H3                                     w: B1
b: J3                                     w: C1
b: J5                                     w: D1
b: G8                                     w: E1
b: J8                                     w: B7
b: E2                                     w: E1
b: B1                                     w: D1
b: C1                                     w: E1
b: H4                                     w: (B1 J8) H8
b: D1                                     w: D7
b: J2                                     w: A9
b: (G6 C6 G5) H5                          w: G9
b: J6                                     w: (J1) J9
b: (H6 D7) H9                             w: (J6 J9 J5 J3 H3 H5) J7
b: (J7 H7 G8 J7 F9 G9 H8 G8 F8 F7 G7
    H7 H6 G6 F6 B3 C3 C4 D4 E4 E5 F5
    G5 G4 G3 G2 H2 H1 A9) A7              w: (H9) J9
b: A8                                     w: H9
b: B6                                     w: J8
b: B7                                     w: (A6) D7
b: C6                                     w: (A9 A8 B8 B7 B6 C6 H4 J2) J4
b: J1                                     w: H3
b: C7                                     w: H4
b: J5                                     w: H5
b: J2                                     w: (J1 J2 J5) J3
b: J2                                     w: J1
b: (H3 H5) PASS                           w: J6
```

```
b: PASS                                 w: PASS
score : 0.5
```

Black wins by 0.5 point.