**University of Alberta**

**Library Release Form**

**Name of Author**: Xiaozhen Niu

**Title of Thesis**: Recognizing Safe Territories and Stones in Computer Go

**Degree**: Master of Science

**Year this Degree Granted**: 2004

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Xiaozhen Niu
10742-86 Ave
Edmonton, Alberta
Canada, T6E 2M9

**Date**: ____________________

**University of Alberta**

# RECOGNIZING SAFE TERRITORIES AND STONES IN COMPUTER GO

by

**Xiaozhen Niu**

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2004

**University of Alberta**

**Faculty of Graduate Studies and Research**

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Recognizing Safe Territories and Stones in Computer Go** submitted by Xiaozhen Niu in partial fulfillment of the requirements for the degree of **Master of Science**.

Martin Müller

Robert Hayes

Jonathan Schaeffer

**Date**:

# Abstract

Computer Go is a most challenging research domain in the field of Artificial Intelligence. Go has a very large branching factor, and whole board evaluation in Go is hard. Even though many game-tree search methods have been successfully implemented in other games such as chess and checkers, the AI community has not yet created a strong Go program due to the above two reasons.

Currently most Go-playing programs use a combination of search and heuristics based on an influence function to determine whether territories are safe. However, to assure the correct evaluation of Go positions, the safety of stones and territories must be proved by an exact method.

This thesis describes new, better search-based techniques including region-merging and a new method for efficiently solving weakly dependent regions for solving the safety of stones and territories. The improved safety solver has been tested in several Go endgame test sets. The performance is compared in the Go program *Explorer* and the state of the art Go program *GNU Go*.

# Acknowledgements

First of all, thanks to my supervisor Martin Müller for all his guidance, comments, and revisions throughout this endeavor. Working with someone with so many ideas and so much experience in the field of computer Go has been a wonderful experience. Martin, I thank you for giving me this opportunity to do research with you and to learn from you.

I would like to thank Jonathan Schaeffer for many reasons. In February 2002 Jonathan gave a talk about computer games in the University of Waterloo. I was happened to be there and was totally fascinated. Then I decided to apply for Master's degree in the University of Alberta right after that wonderful seminar. If I had not attended his seminar, I would not have had the opportunity to come to the University of Alberta, its Computing Science Department, and its GAMES research group. As time goes by, I am more and more convinced that I made the right choice. In addition, Jonathan taught a course in September 2002, in which he explained all the basic concepts about heuristic search so well. Even though at that time I was struggling at his assignment "tournaments", I still felt that it was a great experience in my life. Thank you Jonathan!

To my external examiner Dr. Robert Hayes, I thank you for your time and dedication to read this thesis and providing valuable feedback.

Thank you to my family for all of their support. Four and half years ago I was a chemical engineer. I still remember the moment when I told my parents that I decided to quit my job and switch to computer science. Even though my parents were astonished, but they still supported and encouraged me. Dad and mom, thank you for your understanding and encouragement over the years.

Thank you to Akihiro Kishimoto, Ling Zhao, Adi Botea, Yngvi Björnsson, and the other members of the GAMES group for their helpful discussions and valuable feedback during this research. In addition, thanks to Markus Enzenberger for his helps and explanations to the program *Explorer*.

Thank you to Zhipeng Cai, Gang Xiao, Jun Zhou, Yi Xu, Jiyang Chen, Shoudong Zou, Gang Wu, Xiaomeng Wu, and other graduate students and friends for the joy they gave during the pass two years of graduate studies.

Finally, thank you to Xiaoni Liu for everything.

Xiaozhen Niu April 30, 2004

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Computer Games Research

Games such as chess have long been accepted as useful research test-beds in computing science, for many reasons. First, games have well-defined rules and clearly specified goals, which makes it easier for researchers to measure progress and performance. Second, games can be formally specified and provide non-trivial domains to simulate real-world problems. The relative success obtained by game-playing systems can be applied to problems in other non-game areas. In addition, developing a game-playing program requires the application of theoretical concepts and algorithms to practical situations. By using games as testbeds, many valuable lessons can be obtained while studying the thought processes of the human brain. These lessons will help researchers to reach the ultimate goal for AI, constructing computers that exhibit the intellectual capabilities of human beings.

Over the past 40 years, amazing progress has been made in the field of games. Today, computer programs can beat the strongest human players in many areas. As early as in 1979, the Backgammon program BKG by Hans J. Berliner beat the human world champion Luigi Villa [3]. In 1994, a research team lead by Jonathan Schaeffer developed the checkers program Chinook at the University of Alberta, which won the world man-machine championship [23]. The Othello program Logistello by Michael Buro [5], which is based on a well-tuned evaluation function and machine learning techniques, beat the world champion Mr. Murakami with six straight wins 6-0. Perhaps one of the most remarkable achievements is that the chess program Deep Blue defeated the world chess champion Garry Kasparov in 1997. Since then, the effectiveness of brute-force search has been confirmed in many games. In addition, methods developed in game playing systems can also be used in several areas within mathematics, economics, and computer science such as combinatorial optimization, theorem proving, pattern recognition and complexity theory [8].

## 1.2 Why Study Computer Go?

Go is a two-player perfect information game. Two players compete against each other on a board with 19 by 19 lines for a total of 361 points. Each player puts his stones on the board and seeks to occupy territory. Once the stones are put on the board, they cannot move again, but may be removed if they are completely surrounded by the opponent's stones (captured). The elegant and fascinating complexities of Go arise from the struggle to occupy the most territory. After a game, the player who has the most territory wins the game.

Although many AI methods have been successfully applied to other games, they do not enable the AI community to make a strong Go program. There are two major features that make Go different from other games:

1. Go has a very high branching factor. A Go game normally runs over 200 moves. Each turn offers roughly 250 choices of legal moves on average. The search tree is huge and it has been estimated as about $10^{160}$ nodes. Such a high branching factor makes a deep brute-force search method unfeasible for Go.

2. It is very hard to make a good evaluation function for Go. For Chess and

other games, it is comparably easy to evaluate each piece's value. In contrast, deciding whether two stones have similar values in Go can involve a complicated reasoning process. Humans use many powerful reasoning methods and a lot of knowledge, but computers have difficulties to follow the same approach. Currently no Go program can reach a reasonably high degree of accuracy by using a static evaluation function. Dynamic evaluation is also hard since there is no easy way to convert human knowledge and experience to a program. So far, no clear theoretical model for evaluating Go positions has emerged.

Due to the above reasons, the brute-force search techniques used in other games do not work in Computer Go. As early as in 1978, Berliner predicted [2]:

> ... even if a full-width search program were to become World Chess Champion, such an approach cannot possibly work for Go, and this game may have to replace chess as the task par excellence for AI.

Although much encouraging progress has been made in the past few decades, the strength of current Computer Go programs is still relatively weak. Human amateur players of 8-kyu level (beginner) can beat them easily.

In general, there are plenty of research problems and a large variety of possible methods to investigate in Computer Go. To understand how Go knowledge is gained, processed and used by human players may provide fruitful lessons which lead not only to progress in Go programs, but can also have wide applicability to other applications such as pattern recognition, knowledge representation, machine learning and planning. Thus, Computer Go will remain an attractive and challenging domain for AI research for a long time.

## 1.3 Safety of Territory and the Weakly Dependent Region Problem

The objective of this thesis is to develop search-based methods to recognize safe territory in the game of Go. The project builds on Müller's previous work [14]. The effort is concentrated on developing a high performance safety solver for Go endgames.

In practice, although most games of Go last roughly 250 moves, the difference in final score of a game between two strong players usually turns out to be small. Therefore, no matter how well a program performs in the beginning and the middle of the game, a failure to recognize the safety of territories in the endgame can completely change the game result. Such mistakes even happen occasionally in the games of professional players.

Recognizing the safety of territory is similar to solving a Life and Death problem, but there are several differences. First, a Go program needs to recognize Life and Death throughout the whole game. However, recognizing safe territory normally is used in the endgame or close to the endgame of Go. Second, the goal of the Life and Death recognition is to prove whether target stones in a specific area (region) can live or not. However, to prove that a territory is safe, not only the surrounding boundary stones need to be proved safe, but also the surrounded region needs to be proved safe. This means that no opponent stones can live inside. Therefore, proving territory safe needs to deal with a more complicated goal. Figure 1.1 shows an example where the white surrounding stones are safe but the surrounded region is not.

Several methods have been proposed to prove the safety of territory and stones. Benson proposes an algorithm for *unconditionally alive blocks* [1]. It identifies sets of blocks and basic regions that are safe, even if the attacker can play an unlimited number of moves in a row, and the defender always passes. Müller [14] defined

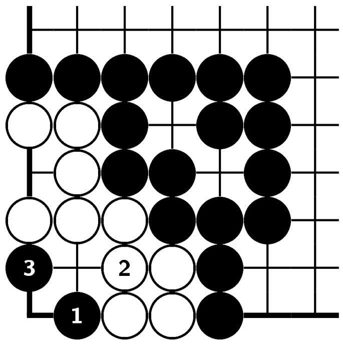


Figure 1.1: Safe white stones, non-safe white region

static rules for detecting safety by *alternating play*, where the defender is allowed to reply to each attacker move. Müller also introduced local search methods for identifying regions that provide one or two *sure liberties* for an adjacent block [14].

The state of the art safety solver in [14] implements Benson's algorithm, static rules and a 6 ply search in the program *Explorer*. However, there are still many remaining problems in recognizing territory safe. One of them is the Weakly Dependent Regions problem. Towards the end of a Go game, the board tends to be divided into many regions. If two regions with the same color share only one boundary block, we call these regions *Weakly Dependent Regions*. Figure 1.2 provides an example. In this figure, the common boundary black block ▲ has only 1 liberty in each of the regions A and B. In local region A, whenever White plays X, the common boundary block ▲ is in atari. So the safety of region B is affected. A similar situation happens in local region B. Therefore, the safety of region A depends on region B and vice-versa. However, simply merging two regions together will make the search space too large, thus it is not feasible in practice.

The previous solver sequentially processes regions one by one and ignores the relationships between them. Therefore, it is unable to solve a problem involving weakly dependent regions.

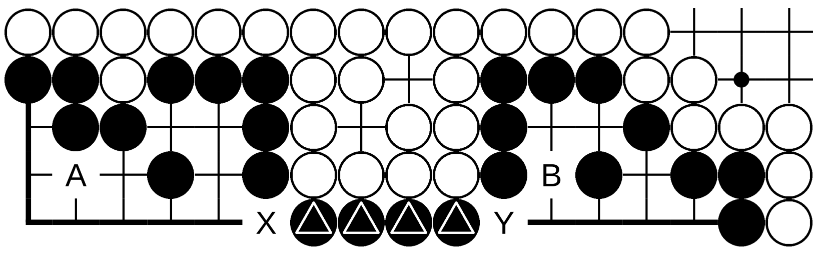


Figure 1.2: An example of weakly dependent regions

## 1.4 Contributions

The research contributions of this thesis include:

- Identifying the major requirements of a high-performance safety solver in Go.
- New region processing techniques. A new, more efficient technique for selectively merging regions is developed.
- A solution to the problem of weakly dependent regions.
- Problem-specific game tree search enhancements such as move ordering and forward pruning.
- The new solver improves the percentage of points proved safe in a standard test set from 26% in [14] to 51%. The speedup observed in our experiments is about 70 times faster than the solver in [14].

## 1.5 Overview of the Thesis

The structure of this thesis is as follows: Chapter 2 introduces basic game-tree algorithms. Chapter 3 surveys relevant work in the field of Computer Go. The basic definitions that are relevant to following chapters are also provided. Chapter 4 describes the techniques used to process regions and to solve weakly dependent regions. Chapter 5 describes the search enhancements. Chapter 6 presents and analyzes experimental results.  Chapter 7 summarizes the research and discusses future work on this project.

# Chapter 2

# Game Tree Search

This chapter provides some background on game tree search and Computer Go. We briefly introduce the concepts of game-tree and minimax search in Section 2.1. In Section 2.2, the standard algorithm of minimax search, Alpha-Beta, is introduced. Section 2.3 discusses common enhancements to Alpha-Beta. Section 2.4 provides a summary of this chapter.

## 2.1 Minimax Search

Go is a two-player zero-sum game, in which the loss of one player is the gain of the other. A player selects a legal move that maximizes the score, while his opponent tries to minimize it. Both players move alternately.

In order to analyze a game, we can construct a graph representation to analyze all possible positions and moves for each player in a game. Figure 2.1 provides an example of such a graph. It is called a *game tree*.

In a typical minimax tree as shown in Figure 2.1, the two players are called *Max player* and *Min player*. By convention, the max player plays first. A *node* in the minimax tree represents a position in a game. The possible moves from a position are represented by unlabelled links in the graph called *branches*. The node at the top which represents the start position is called *root node*. The nodes in which the max player is to play are called *Max nodes*, while nodes in which the min player is to play are called *Min nodes*. By considering all possible moves for both the max and min player, the tree is constructed. If in one node the next player to move has no legal move to continue, then the value of the node is decided by the rules of the game. Such a node is called a *terminal node*. Samuel introduced the term *ply* [20], which represents the distance from the root, i.e. the depth of a game-tree. A *d-ply search* means the program searches $d$ moves ahead from the root node.

Figure 2.1 illustrates a minimax tree. For example, the value of C is 23 because C is a max node, and the max player will choose the maximal value of its children, which is 23. Then the value of 23 is backed up to B by comparing the values of C and J, because B is a min node. After traversing the whole minimax tree, the value 39 is achieved by the path of node A, N, O and R, showing the best play by both players. This path is called a *principal variation (PV)*. The nodes on this path are also called *PV nodes*. In case of ties, there may be several *PV*'s, all with the same value.

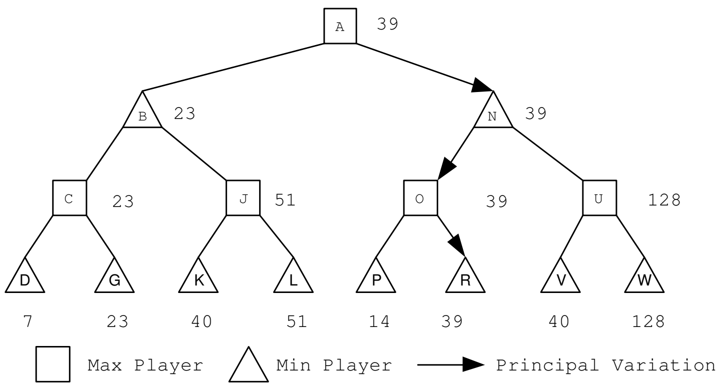


Figure 2.1: Minimax tree

A $d$-ply search of a minimax tree visits all the leaf nodes at the depth of $d$ to determine the minimax value. Let $d$ be the search depth and $b$ the average branching factor at each node, and $N_{minimax}$ be the total number of leaf nodes visited by the minimax algorithm. Then:

$$N_{minimax} = b^d$$

Since the search grows exponentially as a function of the depth $d$, the search depth reached in game-playing programs is limited, especially under tournament conditions. However, the minimax value can be found by visiting fewer leaf nodes. Knuth and Moore showed that the least number is [10]:

$$N_{best} = b^{\lfloor d/2 \rfloor} + b^{\lceil d/2 \rceil} - 1$$

This is a big improvement over minimax. It means that with proper pruning, programs can search up to twice as deep as in full minimax. This is achieved by eliminating nodes from the search that can be shown to be irrelevant to determining the value of the tree. The rest of this chapter discusses enhanced minimax algorithms that try to achieve this best-case result.

## 2.2 Alpha-Beta

In a minimax tree, it is not necessary to explore every node to get the correct minimax value. Some branches can be cut off safely. For example, max(5, min(2, X)) will always return 5 no matter what the value of X is. This is the basic idea of Alpha-Beta pruning.

The Alpha-Beta algorithm has been in use by the computer game-playing community since the end of the 1950's [4, 24, 10]. Alpha-Beta uses two parameters $\alpha$ and $\beta$, which form a *search window* ($\alpha$, $\beta$) to test pruning conditions. $\alpha$ represents a lower bound and $\beta$ represents an upper bound. Values outside the search window do not affect the minimax value of the root.

Alpha-Beta starts searching the root node with $\alpha$ = -$\infty$ and $\beta$ = +$\infty$, and it traverses the game tree in a depth-first manner until a leaf node is reached. Then the value of the leaf node is evaluated and backed up to its parent node to become a bound. As more nodes are explored, the bounds become tighter, until finally a minimax value is found inside the search window.

Figure 2.2 shows an example of the Alpha-Beta algorithm's progress, which is modified from [17]. Let us assume that Alpha-Beta searches in a left-to-right order. At the root node A, Alpha-Beta is called with a search window (-$\infty$, +$\infty$) and passes the initial window to search A, B, C, D and E. Node E is a leaf. It returns its minimax value $g$ of 22 to its parent. At node D, the values of $g$ and $\beta$ are updated to 22. Since $g > \alpha$ (because $22 > -\infty$) the search continues to its next child F. This node is searched with a window of (-$\infty$, 22). Parent D returns 7, which is the minimum of 22 and 7. Parent C updates $g$ and $\alpha$ to 7. In node C, its next child G is searched since $7 < +\infty$. The search window for node G becomes (7, +$\infty$). Node G returns the minimum of 19 and 71 to C, and C returns the maximum of 7 and 19 to B. Since node B is already as low as 19 and B is a min node, the value of B will never increase. In node B the search is continued to explore node J. Since node J is a min node and the $g$-value 19 becomes an upper bound, the search window for J is reduced to (-$\infty$, 19), which means that parent B already has an upper bound of 19. Therefore, if in any of the children of B a $lowerbound > 19$ occurs, the search can be stopped. In node J the search is continued to its child K, which returns a value of 53. This causes a cutoff of its siblings in node J because 53 is not less than 19.

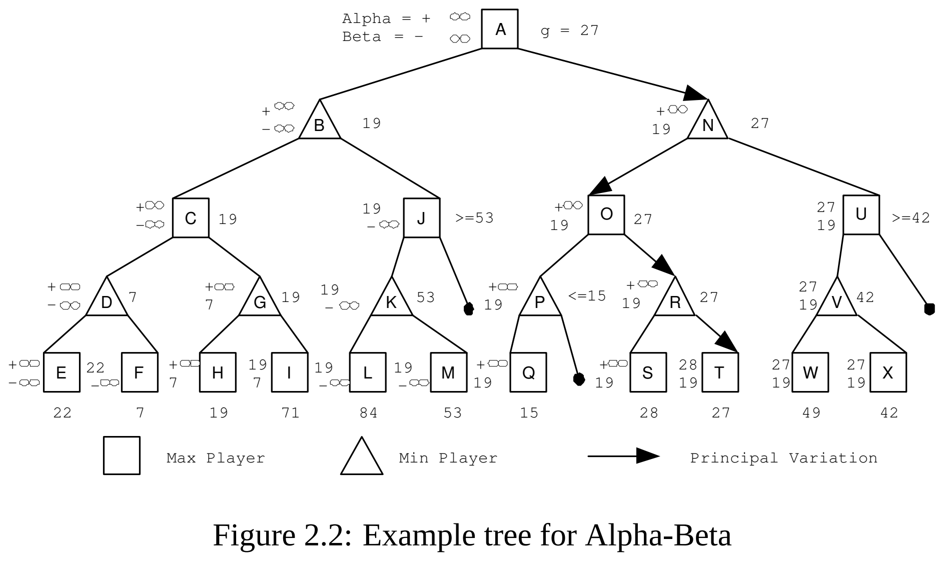


Figure 2.2: Example tree for Alpha-Beta

At the root node A the g-value is updated to the new lower bound of 19. Searching the sub-tree below N can still increase this $g$-value. Nodes N, O, P and Q are all searched with the window (19, $+\infty$). Node Q returns 15, and it causes a cutoff at its parent P since 15 is outside of the search window. Consequently, node P also returns 15. Next nodes R, S, T, U, V, W and X are searched. The sub-tree below V returns 42. This causes a cutoff in its parent U since 42 is not smaller than 27. Node U returns 42 and node N returns the minimum of 27 and 42, and root A returns the maximum of 19 and 27. Finally, the minimax value of the tree has been found, which is 27.

## 2.3 Alpha-beta Enhancements

### 2.3.1 Selective Search

In Alpha-Beta, the backed-up values of leaves are used for pruning. A pruning method like this is sometimes called backward pruning. A drawback of this approach is that it searches all nodes to the same depth. Thus, a bad move gets searched as deeply as a promising good move. To address this problem, many selective search methods have been developed. The main idea of selective search is that some of the “non-promising” branches should be discarded in order to reduce the size of the search tree. In contrast to backward pruning, pruning methods used in selective search are called forward pruning. One example of selective search is N-best search [9]. It only considers the N best moves at each node; all other moves are directly pruned. When the search depth becomes larger, the value of N is decreased accordingly. In addition, a successful example of selective extension is the ProbCut algorithm, presented by Buro [6]. ProbCut uses information from a shallow Alpha-Beta search to decide with a certain probability whether a deep search would yield a value outside the current window. In the game of Othello, ProbCut has been shown to be effective in investigating the relevant variations more deeply.

Selective search is an effective way to reduce the size of the search tree, perhaps to even less than the minimal game tree. However, it has several drawbacks. First, the heuristics used to select "good" or "bad" moves are very application-dependent. An obviously "bad" move at a low level (close to the root) could turn out to be a winning move after a deeper search. Therefore, ignoring such a "bad" move might slow down the search or even miss the win. Second is the performance measurements. In fixed-depth search, improvements mean more cutoffs in the search tree. Therefore, one only needs to compare the sizes of the tree and the search speed while measuring the algorithm performance. However, since selective search artificially cuts off the search tree, the quality of decision becomes more important.

Despite these disadvantages, developing a good forward pruning method is still worth trying, because in the search tree really bad moves should not be considered at all. How to develop a reliable forward pruning strategy combined with sound heuristic knowledge, is still an open problem.

### 2.3.2 Move Ordering

To improve the efficiency of Alpha-Beta pruning, the moves at each node should be ordered so that the most promising ones can be examined first. A minimax tree that is ordered so that the first child of a max node has the highest value, or a value high enough to cause a cutoff. And the first child of a min node has the lowest value or low enough, is called a *best-ordered tree (minimal tree)*. Figure 2.3 shows the minimal tree of the example in Figure 2.2.

The minimal tree has three kinds of nodes, which are defined by Knuth and Moore in [10]. Type 1 nodes form the path from the root to the best leaf (the principal variation). Therefore they are also called *PV nodes*. Type 2 nodes in the minimal tree have only one child; other children have been cut off. They are also called *CUT nodes*. Type 3 nodes have all children, therefore they are also called

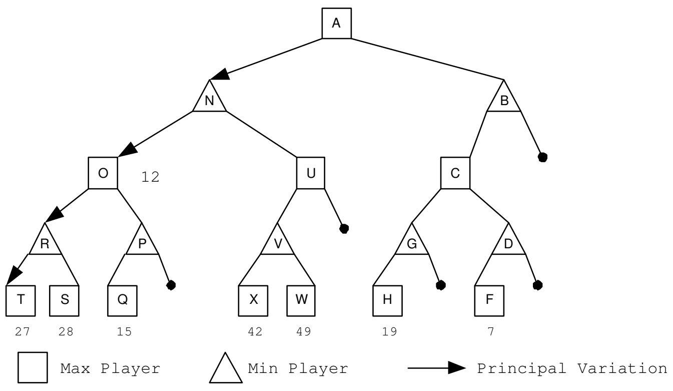


Figure 2.3: Minimal Alpha-Beta tree

*ALL nodes*. For the PV nodes, the minimax value is computed. The value in CUT and ALL nodes can only be worse or equal to the minimax value. Therefore, CUT and ALL nodes are only used to prove that it is unnecessary to search further.

Many approaches have been proposed to improve move ordering. A first approach is to use application-dependent knowledge. For example in chess, a capture normally leads to an advantage in material. Therefore, moves can be ordered by the value of captured pieces. In addition, several other approaches do not rely on application-dependent knowledge. These approaches are proven to be powerful for ordering moves at an interior node. For example, Slate and Atkin developed the killer heuristic [25], which maintains only the two most frequently occurring ”killer” moves at each search depth. Schaeffer presents another powerful technique called history heuristic, which automatically finds moves that are repeatedly good [21, 22]. The history heuristic is a generalization and improvement upon the killer heuristic. It contains a history table for moves. Whenever a move causes a cut-off or turns out to be a good move, the history score of this move increases accordingly. For a node in the search tree, the possible moves are ordered by their scores stored in the history table. In this way, the history heuristic provides an effective way to identify good moves throughout the tree, rather than using information of nodes at the same search depth.

### 2.3.3 Iterative Deepening and Transposition Tables

The basic idea of iterative deepening arose in the early 1970's for the following two reasons. First, for many early game-playing programs, a simple fixed depth search normally can only reach a very shallow depth, especially if it has to be done under tournament conditions. Therefore, it is necessary to find a good time control mechanism. Second, a shallow search in a game-playing system is normally a good approximation of a future deeper search. Slate and Atkin proposed the iterative deepening approach in 1977 [25]. The basic idea is as follows: before doing a d-ply search, perform a 1-ply search, which can be done almost immediately. Then increase the search depth step by step to 2, 3, 4, ..., (d-1) ply searches. Since the search tree grows exponentially, the previous iterations normally take much less time compared to the last iteration. If an iteration takes too long to return the solution, the program can just abort the current iteration and use the result from the previous iteration.

Although at first sight iterative deepening seems very inefficient because interior nodes have been searched over and over again, in experiments iterative deepening is actually more efficient than a direct d-ply search. The efficiency of iterative deepening is based on the transposition table. The best moves from the previous iteration can be stored and reused to improve the move ordering. Therefore, the overhead cost of the d-1 iterations is usually recovered through a better move ordering, which leads to a faster search in iteration d.

In many application domains, the search space is a graph, not a tree. Transposition tables can also be used to prevent re-expansion of searched nodes that have multiple parents [12, 22]. After searching a node, information about this node such as the best score, depth, upper bound, lower bound, and whether the score is exact, is stored in the table. During the search, whenever the same position recurs, the tree search algorithm checks the table before searching it. If the current node is found, then the information from the previous search might be used directly. From this point of view, using a transposition table is an example of exact forward pruning.

In general, transposition tables are implemented as hash tables. By far the most popular method for implementing a transposition table is proposed by Zobrist in 1970 [28]. By using Zobrist's method to generate the hash key, the information stored in the hash table can be retrieved directly and rapidly.

### 2.3.4 Variable Window Search

In the Alpha-Beta algorithm, the bounds $\alpha$ and $\beta$ form the search window. If the value of a node falls outside the search window, a cut-off can occur when value is larger than $\beta$ but not when value is smaller than $\alpha$. Normally using a wider search window means visiting more nodes, and using a smaller search window means visiting fewer nodes. By default, the search window for Alpha-Beta is set to ($-\infty$, $+\infty$). Therefore, reducing the window artificially seems to be a good way to achieve more cut-offs. However, Alpha-Beta already uses all the return values from leaves to reduce the window as much as possible, and guarantees that the minimax value can be found. Reducing the search window artificially runs the risk that the minimax value cannot be found. In this case, re-search in the window with proper bounds is necessary.

In practice, many studies have reported that the cost of re-search is relatively small compared to the benefits of having a well-narrowed search window [12, 7, 16] because of the transposition table. Since variable window search is not used in this thesis, here we only briefly discuss several widely used techniques.

In many games the values of parent nodes and child nodes are related. If we can estimate an initial value for Alpha-Beta to narrow the search window in the beginning of the search, then we can achieve more cut-offs. This window is called an *aspiration window* because we expect the result will fall into the bounds of the window.

Knuth and Moore introduced the following three properties of Alpha-Beta [10]. Let $g$ be the return value of Alpha-Beta and $F(n)$ be the minimax value of node $n$. The postcondition has the following three cases:

1. $\alpha < g < \beta$ (success), $g = F(n)$.

2. $g \leq \alpha$ (fail low), then $g \leq F(n)$.

3. $g \geq \beta$ (fail high), then $g \geq F(n)$.

By using an aspiration window in an Alpha-Beta search, in the first case we have found the exact minimax value cheaply. In the other two cases, we need to perform a re-search. Since the failed search also returns a bound, the re-search can benefit from a window smaller than the initial window (-$\infty$, +$\infty$). In general, aspiration window search is used at the root of the tree. A reasonable estimation can be derived from a relatively cheap shallow search. In practice, this estimation can be derived from iterative deepening.

*Null-window* pushes the narrowed-window-plus-re-search technique to its limit. If a window is set to ($\alpha$, $\alpha$ +1) instead of ($\alpha$, $\beta$), it is called a null window. For example, let alpha be the value of the leftmost child. When performing the null window search for the rest of siblings, if the returned value is smaller than or equal to alpha, we can prune this node safely because it is not better than the leftmost node. In this case, the null window search ensures the maximum cutoffs. If the returned value is bigger than alpha, then this node becomes the new candidate as a PV node. Therefore, it should be re-searched with a wider window to get its exact value. Many studies have proven that the savings outweigh the overhead of re-search [12, 7, 16].

Several widely used Alpha-Beta improvements have been proposed such as Scout [15], NegaScout [19], and Principal Variation Search (PVS) [11] . They all use the idea of null window search.

A further improved Alpha-Beta algorithm is MTD(f) [18], which is simpler and more efficient than previous algorithms. MTD(f) gets its efficiency by using only null window search. Since null window search will only return a bound on the minimax value, MTD(f) has to call Alpha-Beta repeatedly to adjust the search towards the minimax value. In order to work, MTD(f) needs a first estimate of the minimax value. The better the first guess is, the more efficient MTD(f) performs because it will call Alpha-Beta less times. In general, MTD(f) works in an iterative deepening framework. A transposition table is necessary for MTD(f).

## 2.4 Summary

The Alpha-Beta tree-searching algorithm has been in use since the end of the 1950's. Most successful game-playing programs use the Alpha-Beta algorithm with enhancements like move ordering, iterative deepening, transposition tables, narrow search windows. Forty years of research have improved Alpha-Beta's efficiency dramatically. However in Computer Go, there is no direct evidence that deeper search will automatically lead to better performance of a Go program.

# Chapter 3

# Terminology and Previous Work

## 3.1 Terminology and Go Rules

Our terminology is similar to [1, 14], with some additional definitions. Differences are indicated below. A *block* is a connected set of stones on the Go board. Each block has a number of adjacent empty points called *liberties*. A block that loses its last liberty is *captured*, i.e. removed from the board. A block that has only one liberty is said to be *in atari*. Figure 3.1 shows two black blocks and one white block. The small black block ▲ contains two stones, and has five liberties (two marked A and three marked B).

Given a color $c \in \{Black, White\}$, let $A_{\neg c}$ be the set of all points on the Go board which are *not* of color $c$. Then a *basic region* of color $c$ (called a region in [1, 14]) is a maximal connected subset of $A_{\neg c}$. Each basic region is surrounded by blocks of color $c$. In this thesis, we also use the concept of a *merged region*, which

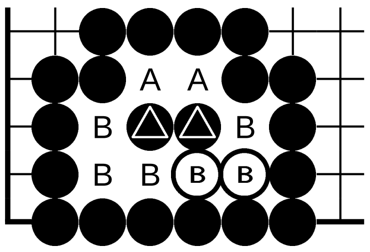


Figure 3.1: Blocks, basic regions and merged regions

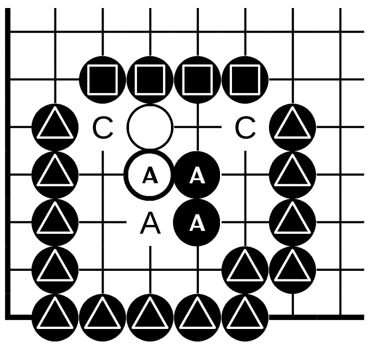


Figure 3.2: The interior and cutting points of a black region

is the union of two or more basic regions of the same color. We will use the term region to refer to either a basic or a merged region. In Figure 3.1 $A$ and $B$ are basic regions and $A \cup B$ is a merged region.

We call a block $b$ *adjacent* to a region $r$ if at least one point of $b$ is adjacent to one point in $r$. A block $b$ is called *interior block* of a region $r$ if it is adjacent to $r$ but no other region. Otherwise, if $b$ is adjacent to $r$ and at least one more region it is called a *boundary block* of $r$. We denote the set of all boundary blocks of a region $r$ by $Bd(r)$. In Figure 3.1, the black block ▲ is a boundary block of the basic region A but an interior block of the merged region $A \cup B$. The *defender* is the player playing the color of boundary blocks of a region. The other player is called the *attacker*.

Given a region, the *interior* is the subset of points not adjacent to the region's boundary blocks. There may be both attacker and defender stones in the interior. A *cutting point* is a point that is adjacent to two or more boundary blocks. In Figure 3.2, the black region has two boundary blocks marked by triangles and squares separately. The interior consists of four points marked $A$, and this region contains two cutting points marked $C$.

The *accessible liberties* of a region is the set of liberties of all boundary blocks in the region. A point $p$ in a region is called a *potential attacker eye point* if the attacker could make an eye there, provided the defender passes locally. Figure 3.3

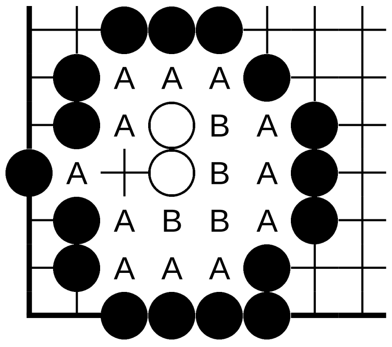


Figure 3.3: Accessible liberties (A) and potential attacker eye points (B) of a black region

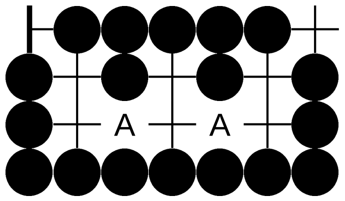


Figure 3.4: Intersection points (A) of a black region

shows some examples.

An intersection point of a region is an empty point $p$ such that $region - \{p\}$ is not connected and $p$ is adjacent to all boundary blocks. In Figure 3.4, the black region has two intersection points, which are marked by letter A.

If two basic regions have one or more common boundary blocks, we call these two regions *related*. By further analyzing the relationship between related regions, we distinguish between *strongly dependent* regions, which share more than one common boundary block, and *weakly dependent* regions with exactly one common boundary block. In Figure 3.5 on the left, two basic black regions A and B are related. Further, they are strongly dependent because they have two common boundary blocks (marked by triangles). In Figure 3.5 on the right, the two basic black regions C and D are weakly dependent because they have only one common boundary block (marked by a square).

A $Nakade$ shape is a region that will end up as only one eye [27]. Therefore it

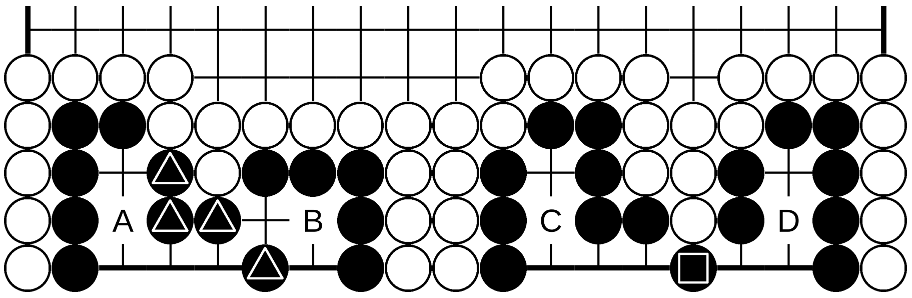


Figure 3.5: Strongly and weakly dependent regions

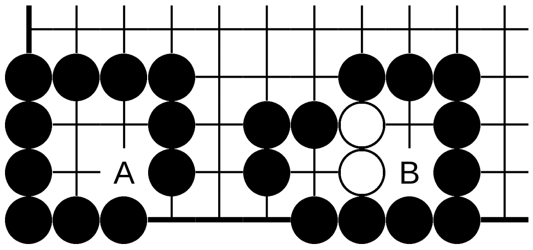


Figure 3.6: Two black nakade shapes

is not sufficient to live. In Figure 3.6 left and right, both black regions A and B are nakade shapes.

Our results are mostly independent of the specific Go rule set used. As in previous work [1, 14], suicide is forbidden. Our algorithm is incomplete in the sense that it can only find stones that are safe by two sure liberties [14]. Because ko requires a global board analysis and the problem can turn out to be very complicated, we exclude cases such as conditional safety that depends on winning a ko, and also less frequent cases of safety due to double ko or snapback. Figure 3.7 provides an example of double ko. In this figure, neither black nor white can win both ko fights in A and B in one move. Therefore, the black block ▲ and white block ◻ are safe even though they only have one sure eye.

Figure 3.8 provides an example of snapback. In this figure, the white block ◻ has only 1 liberty. However, if black captures this block by playing at A, white can immediately recapture the black block ▲ and remains safe.

In addition, the safety solver does not yet handle coexistence in *seki*. Figure 3.9

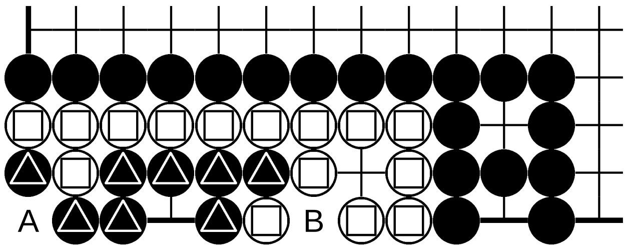


Figure 3.7: An example of double ko

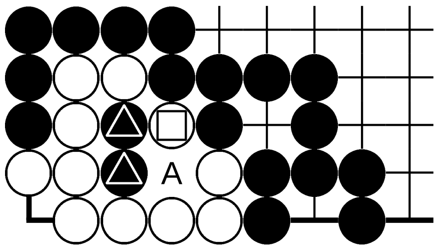


Figure 3.8: An example of snapback.

provides two examples of seki. On the left, black block ■ and white block □ share two common liberties marked A and B. On the right, black block ▲ and white block △ both have one sure eye, and share one common liberty marked C.

## 3.2 Previous Work

Benson's algorithm for *unconditionally alive blocks* [1] identifies sets of blocks and basic regions that are safe, even if the attacker can play an unlimited number of moves in a row, and the defender passes on every turn. Benson's algorithm is a start-

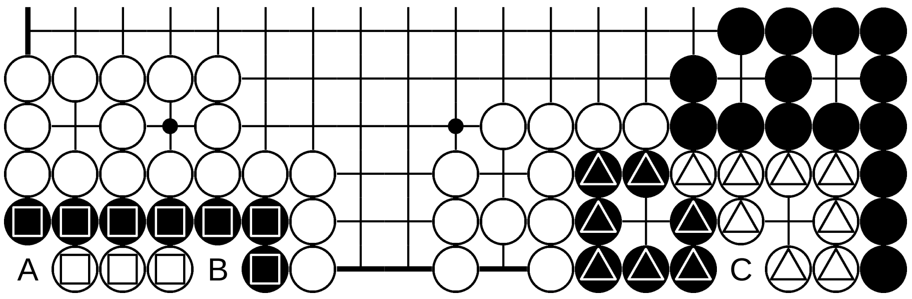


Figure 3.9: Two examples of seki

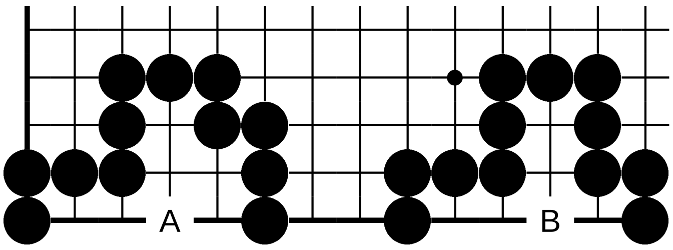


Figure 3.10: Two black regions are alive

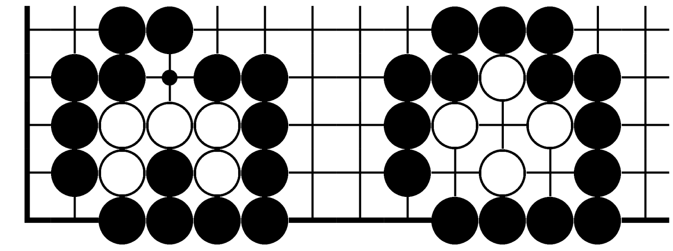


Figure 3.11: Two black regions are not alive

ing point for recognizing safe territories and stones, and it is also the first theorem in the theory of Go. However, it has limited applications in practice. Müller [14] defined static rules for detecting safety by *alternating play*, where the defender is allowed to reply to each attacker move. Müller also introduced local search methods for identifying regions that provide one or two *sure liberties* for an adjacent block. Experimental results for a preliminary implementation in the program *Explorer* were presented for Benson's algorithm, static rules and a 6 ply search.

Van der Werf implemented an extended version of Müller's static rules to provide input for his program that learns to score Go positions [26]. Vilà and Cazenave developed static classification rules for many classes of regions up to a size of 7 points [27].

The following figures provide several examples that are modified from [27]. They all can be identified by using the static eye classification. In Figure 3.10, both black regions A and B are alive no matter who plays first and no matter what the surrounding conditions are. In Figure 3.11, both black regions are not unconditionally

ally alive. In the left, if black loses all the external liberties, then it will be in atari. In the right, the black region is not alive due to a ko fight inside. If black wins the ko, then the region is alive. If white wins the ko, then the region turns out to be a size 6 nakade shape.

## 3.3 Definitions

The following definitions, adapted from [14], are the basis for our work. They are used to characterize blocks and territories that can be made safe under alternating play, by creating two sure liberties for blocks, and at the same time preventing the opponent from living inside the territories. During play, the liberty count of blocks may decrease to 1 (they can be in atari), but they are never captured and ultimately achieve two sure liberties.

Regions can be used to provide either one or two liberties for a boundary block. We call this number the *Liberty Target* $LT(b,r)$ of a block $b$ in a region $r$. A search is used to decide whether all blocks can reach their liberty target in a region, under the condition of alternating play, with the attacker moving first and winning all ko fights.

**Definition:** Let $r$ be a region, and let $Bd(r) = \{b_1, \ldots, b_n\}$ be the set of non-safe boundary blocks of $r$. Let $k_i = LT(b_i, r)$ , $k_i \in \{1, 2\}$, be the liberty target of $b_i$ in $r$. A defender strategy $S$ is said to *achieve all liberty targets* in $r$ if each $b_i$ has at least $k_i$ liberties in $r$ initially, as well as after each defender move.

Each attacker move in $r$ can reduce the liberties of a boundary block by at most one. The definition implies that the defender can always regain $k_i$ liberties for each $b_i$ with his next move in $r$. The following definition of life under alternating play is analogous to Benson's:

**Definition:** Let $EL(b)$ be the external safe liberties of a block $b$. A set of blocks $B$ is *alive under alternating play* in a set of regions $R$ if there exist liberty targets

$LT(b, r)$ and a defender strategy $S$ that achieves all these liberty targets in each $r \in R$ and

$$\forall b \in B \;\; EL(b) + \sum_{r \in R} LT(b, r) \geq 2$$

Note that this construction ensures that blocks in $B$ will never be captured. Initially each block has two or more liberties. Each attacker move in a region $r$ reduces only liberties of blocks adjacent to $r$, and by at most 1 liberty. By the invariant, the defender has a move in $r$ that restores the previous liberty count. Each block in $B$ has at least one liberty overall after any attacker move and two liberties after the defender's local reply. In addition, if a block has one sure external liberty ($EL(b)$ = 1), then the sum of liberty targets for such a block can be reduced to 1. If $EL(b)$ = 2, then the block is already safe ad need not be considered here.

**Definition:** We call a region $r$ *1-vital* for a block $b$ if $b$ can achieve a liberty target of one in $r$, and *2-vital* if $b$ can achieve a liberty target of two.

## 3.4 Recognition of Safe Regions

The attacker *cannot live inside* a region surrounded by safe blocks if there are no two nonadjacent potential attacker eye points, or if the attacker eye area forms a nakade shape (as introduced in Section 3.1). The current solver uses a simple static test for this condition as described in [14].

The state of the art safety solver in [14] implements Benson's algorithm, static rules and a 6 ply search in the program *Explorer*. However, there are still many remaining problems in recognizing territory safe. One of them is the Weakly Dependent Regions problem. The solver sequentially processes regions one by one and ignores the relationships between them. Therefore, it is unable to solve a problem involving weakly dependent regions.

# Chapter 4

# Safety Solver

## 4.1 Search Engine

The search engine in the program *Explorer* [13] is an Alpha-Beta search framework with enhancements including iterative deepening and transposition table as described in Chapter 2). Other enhancements to this Alpha-Beta framework such as move ordering and heuristic evaluation functions will be described in Chapter 5.

The safety solver uses this search engine and includes the following sub-solvers:

**Benson solver** Implements Benson's classic algorithm [1] to recognize unconditional life.

**Static solver** Uses static rules to recognize safe blocks and regions under alternating play, as described in [14]. No search is used.

**1-vital solver** Uses search to find regions that are 1-vital for one or more boundary blocks. As in [14] there is also a combined search for 1-vitality and connections in the same region, that is used to build chains of safely connected blocks.

**Generalized 2-vital solver** Uses searches to prove that each boundary block of a given region can reach a predefined liberty target. For *safe blocks*, the target is 0, since their safety has already been established by using other regions.

Blocks that have one sure external liberty (eye) outside of this region are defined as *external eye blocks*. For these blocks the liberty target is 1. For all other non-safe boundary blocks the target is 2 liberties in this region. All the search enhancements described in the next section were developed for this solver.

The 2-vital solver in [14] could not handle external eye blocks. It tried to prove 2-vitality for all non-safe boundary blocks.

**Expand-vital solver** Uses search to prove the safety of partially surrounded areas, as in [14]. This sub-solver can also be used to prove that non-safe stones can connect to safe stones in a region.

## 4.2 High-level Outline of Safety Solver

Figure 4.1 shows the processing steps on a final position of a 19$\times$19 game from test set 1 in Section 6.1. In this typical example, much of the board has been partitioned into relatively small basic regions that are completely surrounded by stones of one player.

The basic algorithm of the safety solver for this example is as follows:

1. The static solver is called first. It is very fast and resolves the simple cases. The result is shown in Figure 4.2. In this position, the static solver can solve a total of 9 basic regions A, B, C, D, E, F, G, H and I. The stones that have been proved safe or dead for attacker stones inside are marked by triangles.

2. The 2-vital solver is called for each region. As a simple heuristic to avoid computations that most likely will not succeed, searches are performed only for regions up to size 30. Many small regions remaining in this position can not be solved because they are related regions. In this step, since the 2-vital solver treats regions separately, it only solves 2 more regions J and K. The

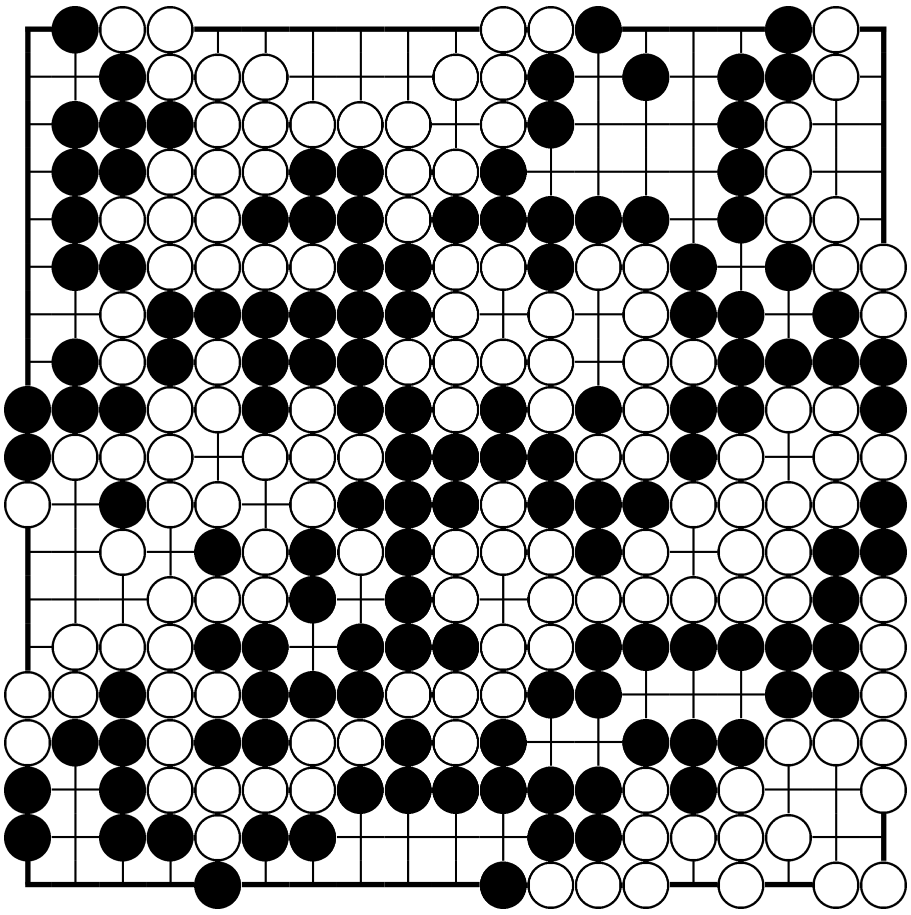


Figure 4.1: A whole board example (before step 1)

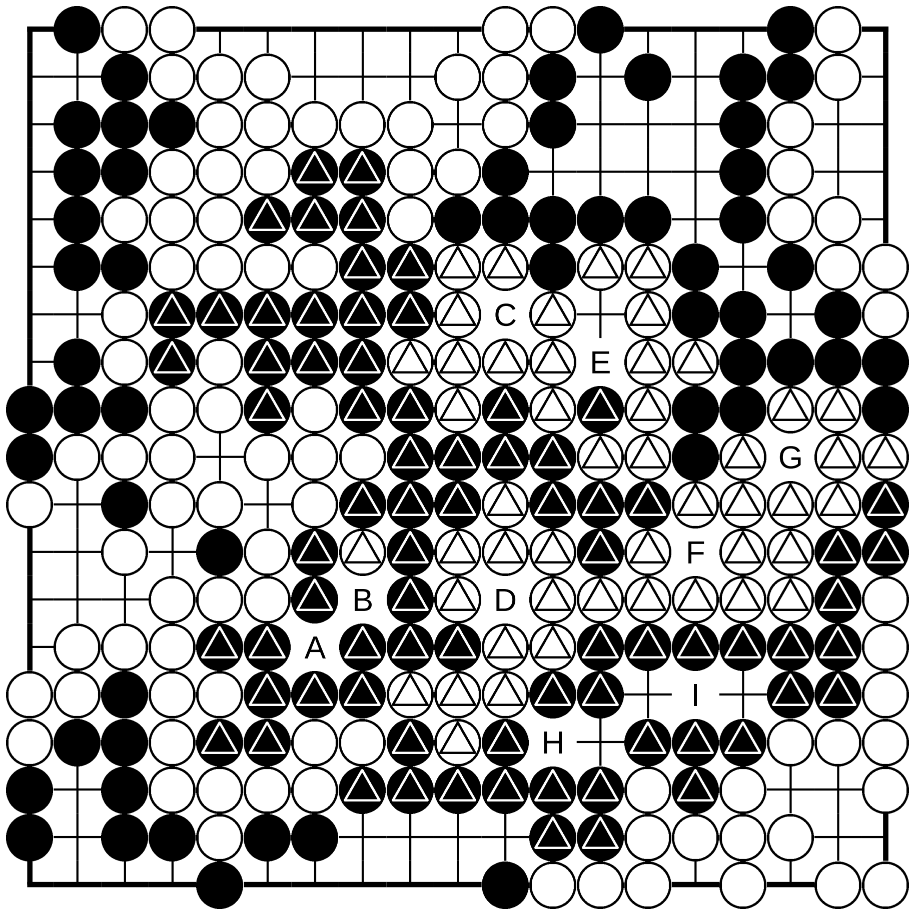


Figure 4.2: The result of step 1

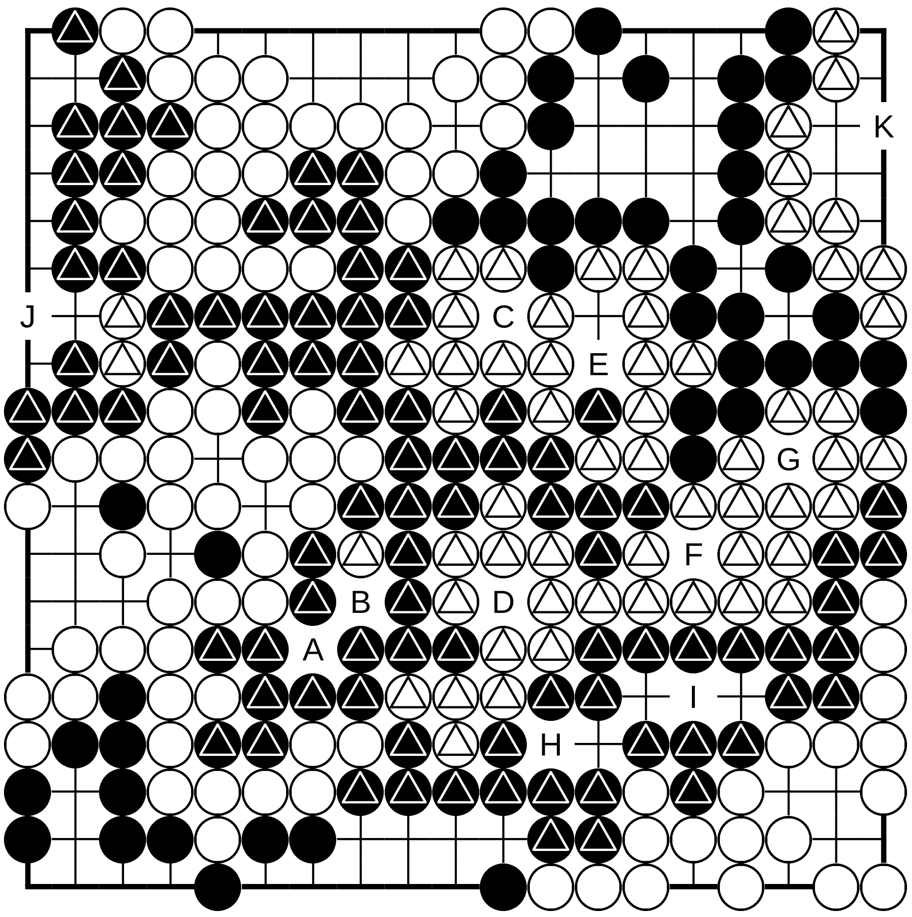


Figure 4.3: The result of step 2

sizes of these two regions are small, 12 and 7 points respectively. Figure 4.3 shows the result.

3. The expand-vital solver is called for regions that have some safe boundary blocks. The safety of those blocks has been established by using other regions. In this example, the expand-vital solver does not solve any region at this step.

   Müller’s previous solver [14] only used the steps so far. The result is shown in Figure 4.3.

4. (New) Region merging. After the previous steps, all the easy-to-prove safe basic regions have been found. In this step the remaining unproven related

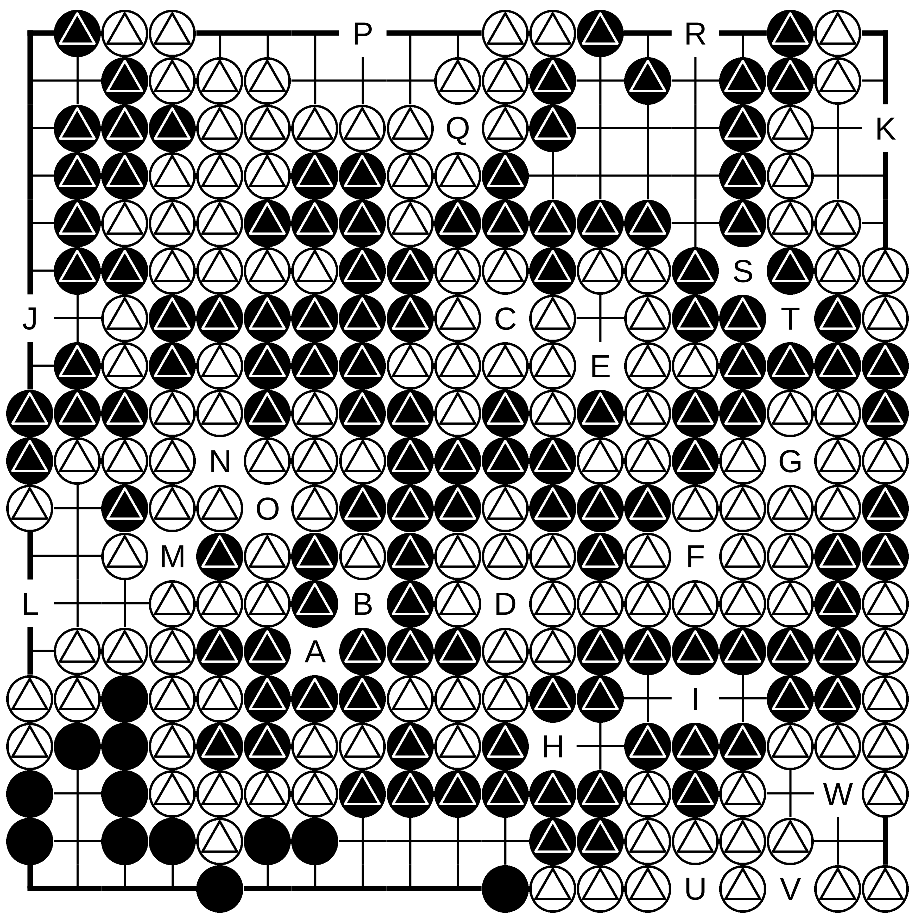


Figure 4.4: The result of step 4

regions are merged. For each small-enough merged region (up to size 14 in the current implementation) the generalized 2-vital solver is called. The mechanism is described in detail in Section 4.3. Figure 4.4 shows the result of this step. The solved merged regions are $P \cup Q$, $L \cup M \cup N \cup O$, $W \cup V$ with an external eye U for white and $R \cup S \cup T$ for black. Most of the remaining related regions have been solved except for two weakly dependent black regions at the bottom.

5. (New) Weakly dependent regions. A new algorithm deals with weakly dependent regions. In this step both the 1-vital solver and the 2-vital solver are used to prove whether a region is 2-vital safe or not. A detailed descrip-

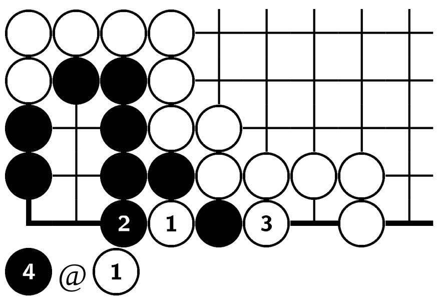


Figure 4.5: The black region is a 2-vital region

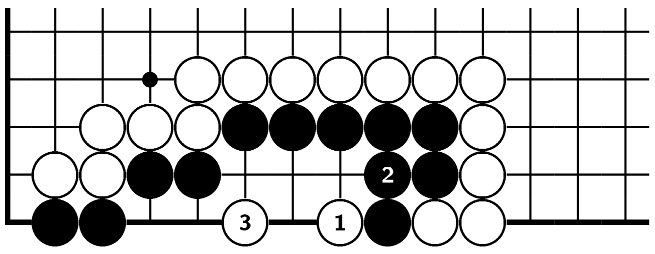


Figure 4.6: The black region is not a 2-vital region

tion about processing weakly dependent regions is given in Section 4.4. One of the remaining weakly dependent black regions X is solved. Figure 4.5 demonstrates the proving process.

However, the other remaining black region Y cannot be proved as a 2-vital region in this step, even though it has a safe boundary block. Figure 4.6 provides a modified example of region Y to demonstrate the reason. In this figure, when white plays move 1 black has to connect because it is atari. After white plays move 3, the black region turns out to be a nakade shape. Therefore it is not 2-vital safe.

Figure 4.7 shows the result of this step.

6. (New) As in step 3, the Expand-vital solver is called for those regions for which one or more new safe boundary blocks have been found. In this step, the expand-vital solver can easily solve the last weakly dependent black region Y. Figure 4.8 shows the result of this step. In this example, the solver

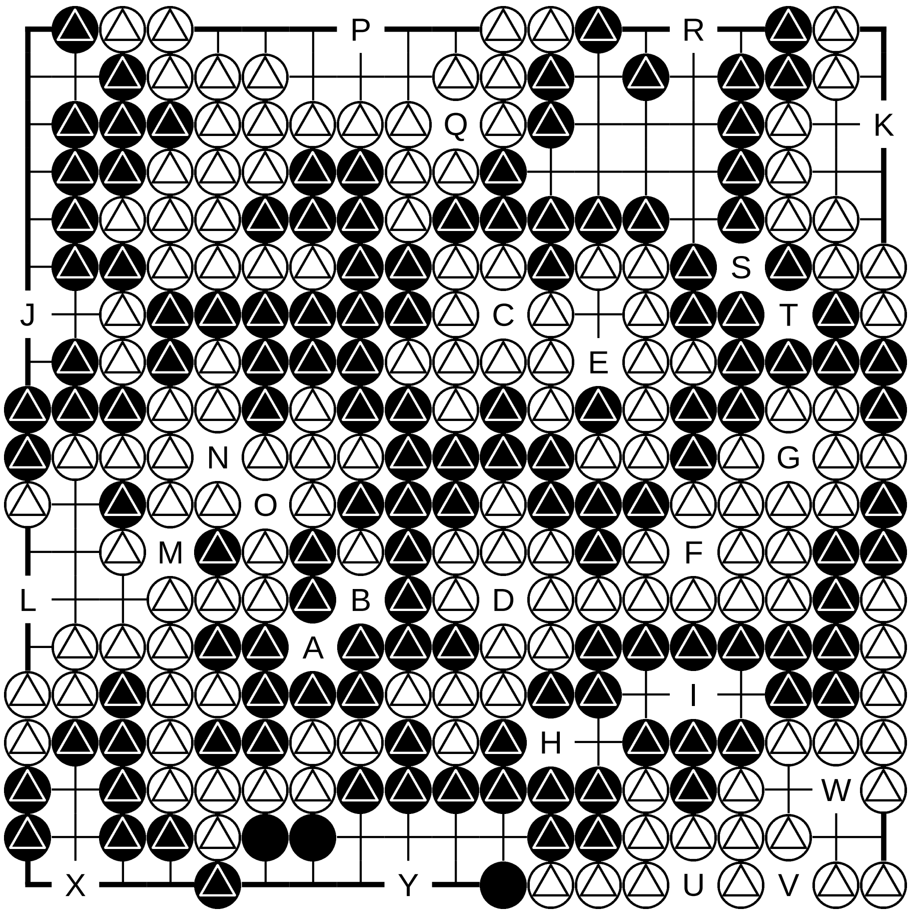


Figure 4.7: The result of step 5

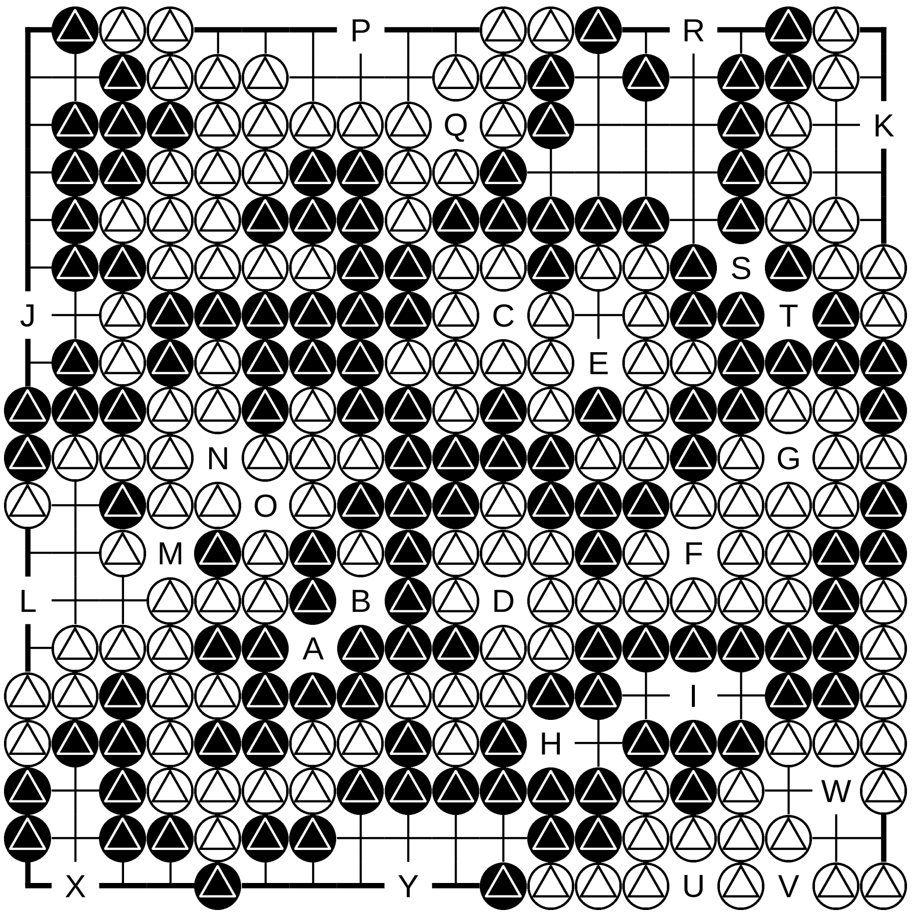


Figure 4.8: The result of step 6

succeeds in proving the safety of every point on the board.

## 4.3 Region Merging

One of the major drawbacks of Müller's previous solver [14] is that it processes basic regions one by one and ignores the possible relationship between them. Figure 4.9 shows an example of two strongly dependent regions. The previous solver treats regions A and B separately, and neither region can be solved. However the merged region A $\cup$ B can be solved easily.

The first algorithm step of region merging scans all regions and merges all related regions. They are either strongly or weakly dependent. After the merging

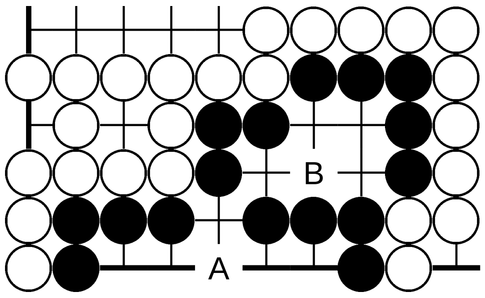


Figure 4.9: Two related regions

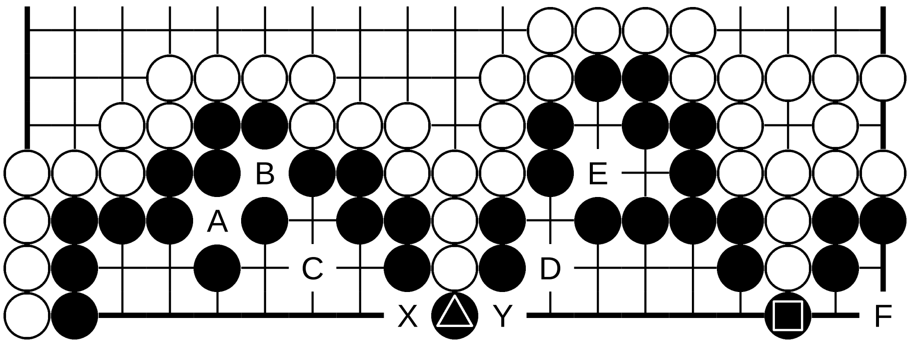


Figure 4.10: Strongly and weakly dependent regions

step, the 2-Vital solver is used to recognize safe merged regions.

This method can solve simple cases such as the one in Figure 4.9. However, since merging all related regions often creates a very large merged region, the search space can become too large.

Our current solver uses a two-step merging process. In the first step, strongly dependent basic regions are merged. In the second step groups of *weakly dependent regions* are formed. A group can contain both basic regions and merged regions computed in the first step. Figure 4.10 shows an example.

In this figure, there are a total of 6 related black regions A, B, C, D, E, and F. Since the huge outside region contains surrounding white stones that are already safe, we do not need to consider it here.

A simple merge yields a combined new region with size 32, which is too large to be fully searched in a reasonable time. Two-step merging creates the following result: The first step identifies connected components of strongly dependent regions and merges them. A, B and C are strongly dependent and are merged into a new region $R_1 = A \cup B \cup C$. Next D and E are merged into $R_2 = D \cup E$. Region F is not strongly dependent on any other region and is not merged. The second step identifies weak dependencies between $R_1$, $R_2$ and F and builds the group. $R_1$ and $R_2$ are weakly dependent through block ▲, and $R_2$ and F are weakly dependent through block ■. The result is a group of weakly dependent regions $\{R_1, R_2, F\}$ with region sizes of 15, 14 and 3 respectively. The regions within a group are not merged but searched separately, as explained in the next section.

The common boundary block between two weakly dependent regions has both *internal* and *external* liberties relative to each region. For example, for block ▲ and $R_2 = D \cup E$, the liberty Y is internal and the liberty X is external.

## 4.4 Weakly Dependent Regions

We distinguish between two types of weak dependencies. In type 1, the common boundary block has more than one liberty in both weakly dependent regions. For example, in Figure 4.11 the shared boundary block of regions A and B has more than 1 liberty in each region. In type 1 dependencies, our search in one region does not consider the external liberties of the common block.

In type 2 weak dependencies, the common boundary block has only one liberty in at least one of the weakly dependent regions. Figure 4.12 provides a typical example of type 2 weakly dependent regions. The black block ▲ has only 1 liberty in each of the regions X and Y. We need to consider the external liberties for the common block because moves in region X will affect the result of the region Y. However, we do not want to merge these two regions because of the resulting increase in problem size.

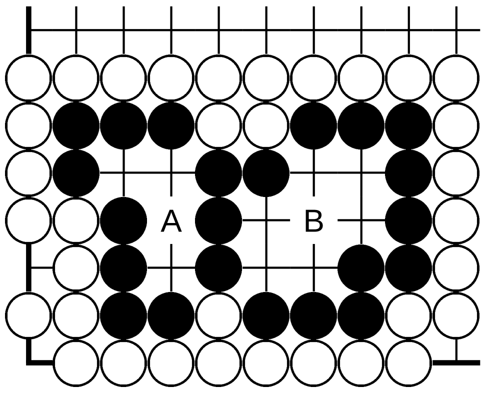


Figure 4.11: First type of weakly dependent regions

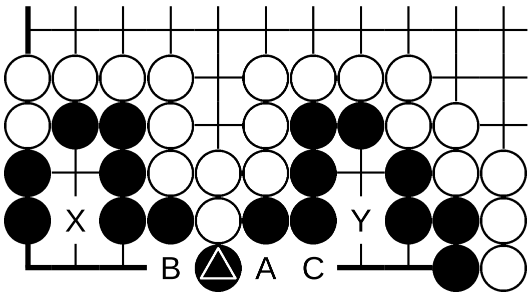


Figure 4.12: Second type of weakly dependent regions

Figure 4.13 demonstrates the separate 2-vital search processing for regions X and Y. In local region X, considering an external attacker move at A is necessary. Whenever White plays A, the common boundary block ▲ loses its external liberties. Therefore, it is in atari. In this case, Black will connect at B in response. A similar situation happens in region Y. If White plays at A, since common boundary block ▲ is in atari, Black is forced to answer at C to capture this white block. In addition, considering one external move in B is also necessary for proving region Y safe. Therefore, if we consider region X and Y locally, both regions can be proved safe.

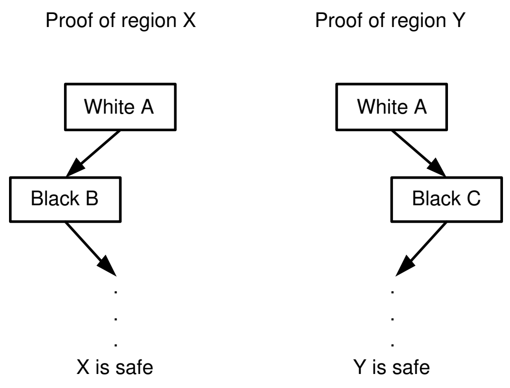


Figure 4.13: Separate searches in regions X and Y

However, from the global point of view we need to handle the relationship between regions. In real game if White plays at A, for region X Black should connect at B while for region Y Black should play a move at C to capture the white stone. In this situation, which move should Black play? Figure 4.14 demonstrates the 2-vital search processing for both weakly dependent regions X and Y.

If White plays in A, since A is located at region Y, we look at that region first. If the white block at A only has one liberty, we always play the capture move in region Y. After removing the white block in A, from X's point of view the common boundary block ▲ will gain one external liberty again. Therefore, in this case after White A and Black C in region Y the result will not affect the local search in

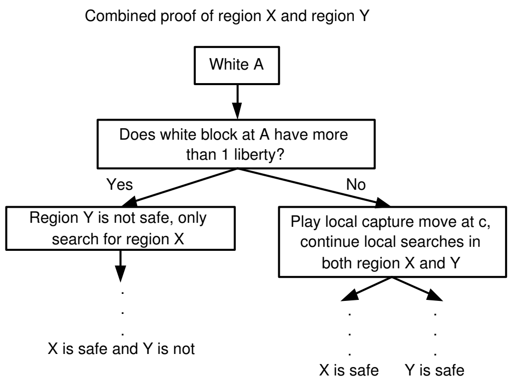

Combined proof of region X and region Y
White A
Does white block at A have more than 1 liberty?
Yes
No
Region Y is not safe, only search for region X
Play local capture move at c, continue local searches in both region X and Y
X is safe and Y is not
X is safe
Y is safe

Figure 4.14: Search considering both region X and Y

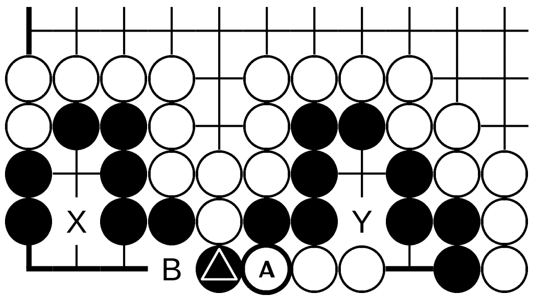


Figure 4.15: White block in A has more than 1 liberty

region X.

If the white block at A has more than one liberty such as in the example shown in Figure 4.15, then the situation is different. In region Y, since the black block ▲ already lost all its internal liberties and we can not guarantee that it can achieve more external liberties outside of region Y, the safety search for region Y will fail immediately. For region X, even if the region Y is not safe, since white A is an external move for its local search, black will answer at B locally. Therefore, in this example, region X can be proved safe and region Y can not.

The pseudo code in Figure 4.16 describes the method for processing groups of weakly dependent regions.

```
for each weakly dependent group $G$
    if ( total size of all regions in $G < 14$) // 14 is a constant determined empirically
        $r_G$ = merge all regions in $G$;
        call 2-vital solver for $r_G$;
    else
        for each region $r \in G$
            for each shared boundary block $b$ between $r$ and another region $r_2 \in G$
                do a 1-vital search for $b$ in $r_2$;
            if (all 1-vital searches succeed)
                reduce liberty target for all tested boundary blocks to 1;
                call 2-vital solver for $r$;
            else
                reduce liberty target for all successfully tested boundary blocks to 1;
                take unproved (1-vital search not successful) blocks as special blocks;
                generate external moves for special blocks (for both attacker/defender);
                call 2-vital solver for $r$;
```

Figure 4.16: Search for weakly dependent groups

## 4.5 Other Improvements

The following further enhancements were made to the solver beyond the version described in [14].

**External eyes of blocks** If a boundary block of a region $r$ has one sure liberty elsewhere, this information is stored and used in the search for $r$ by lowering the liberty target for that block. In Figure 4.17, after Black plays the first move, the previous solver recognizes that both two white boundary blocks (marked by squares and triangles) could be in atari, and returns the result that the region is not safe. However, since the white boundary block marked by triangles has one external eye, the liberty target for this block can be reduced to 1. By using this additional information it becomes possible to prove the region safe.

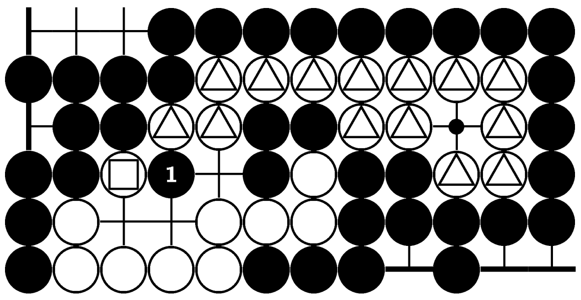


Figure 4.17: Block with an external eye

# Chapter 5

# Search Enhancements

## 5.1 Move Generation and Move Ordering

In this work, we focus on proving that a region and its boundary blocks are safe. Therefore we have concentrated our efforts on generating and ordering the defender's moves. For the attacker, all legal moves in the region plus a pass move are generated. When processing weakly dependent regions as described in Section 4.4, extra moves outside of the region might be generated for either attacker or defender. The attacker is allowed to immediately recapture a ko. Therefore, the attacker will always win a ko fight inside a region.

Currently there is no move ordering for the attacker. For the defender, the following safe forward pruning technique is used: When a boundary block of a region is in atari, only moves that can possibly avert the capturing threat, such as extending the block's liberties or capturing the attacker's adjacent stones, are generated. If no such forced moves are found, all legal moves for the defender are generated.

For ordering the defender's moves, both a high priority move detector and a normal scoring system are used. The detector analyzes the purpose of the attacker's previous move, and classifies the situation as one of three priorities:

1. Attacker's move close to one of the empty cutting points.

2. Attacker's move extending one or more cutting blocks.

3. Other attacker move.

For priority 1 and 2 positions, a set of high priority moves according to the attacker's motivation is generated first. For priority 1, most likely the attacker is trying to cut, so the cutting points close to this move, as well as the cutting points' 8 neighbor points, have high priority. For priority 2, most likely the attacker is trying to expand its own cutting block. Capturing this block is an urgent goal for the defender. Therefore, all liberties of this block are given high priority. The number of adjacent empty points is used to order liberties.

All moves in priority 3 positions and all remaining moves in priority 1 and 2 positions are sorted according to a score that is computed as a weighted sum:

$$\textit{Move score} = f_1 * \textit{LIB} + f_2 * \textit{NDB} + f_3 * \textit{NAB} + f_4 * \textit{CB} + f_5 * \textit{AP}.$$

The formula uses the following features:

1. Liberties of this defender's block (LIB)
2. Number of neighboring defender's blocks (NDB)
3. Number of neighboring attacker's blocks (NAB)
4. Capture bonus (CB): 1 if an opponent block is captured, 0 otherwise
5. Self-atari penalty (AP): -1 if move is self-atari, 0 otherwise

The following set of weights worked well in our experiments: $f_1 = 10, f_2 = 30, f_3 = 20, f_4 = 50, f_5 = 100.$

## 5.2 Evaluation Functions

### 5.2.1 Heuristic Evaluation Function

The evaluation function in [14] used only three values: *proven-safe*, *proven-unsafe* and *unknown*. Since most of the nodes during the search evaluate to *unknown*, we can improve the search by using a heuristic evaluation to differentiate nodes in this category. The heuristics are based on two observations:

1. An area that is divided into more subregions is usually easier to evaluate as *proven-safe* for our static evaluation function.

2. If the attacker has *active blocks* with more than 1 liberty, it usually means that the attack still has more chances to succeed.

Let NSR be the number of subregions and NAB be the number of the attacker's active blocks. Then the heuristic evaluation of a position is calculated by the following formula:

$$eval = f_1 * \mathit{NSR} + f_2 * \mathit{NAB}, \quad f_1 = 100, f_2 = -50$$

### 5.2.2 Exact Evaluation Function

The exact evaluation function recognizes positions that are *proven-safe* or *proven-unsafe*. A powerful function is crucial to achieve good performance. However, there is a tradeoff between evaluation speed and power. In our evaluation function there are two types of exact static evaluations, *HasSureLiberties()* and *StaticSafe()*.

*HasSureLiberties()* is a quick static test to check whether all boundary blocks of a region have two sure liberties and the opponent cannot live inside the region. It uses the following two conditions for checking:

1. All empty points inside the regions are liberties of some boundary blocks.

2. The region has two or more intersection points as described in Section 3.1, or it has two separate eyes.

Condition 1 implies that there is no eye space for the attacker. Condition 2 utilizes the *miai* strategy. If there are two equal-value points, a miai strategy means

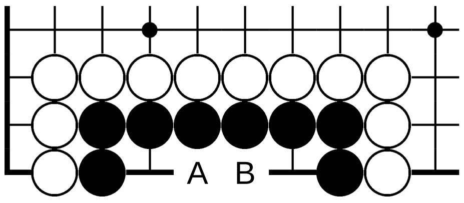


Figure 5.1: An example of miai

that no matter which point one player chooses, the other player can always get the other point. Figure 5.1 shows an example of a miai strategy. In the black region, White cannot occupy both points A and B in one move. Therefore, the black region is alive under alternating play. If both conditions of *HasSureLiberties()* are satisfied, then the region is safe.

*StaticSafe()* is a simplified static safety solver which takes the subregions created by the search into account. It takes the set of all points of the region as input and processes the following steps:

1. Generate all the subregions and blocks inside the input region.

2. As in Benson's algorithm [1], find all the healthy subregions for blocks. A region is *healthy* for a block if the block is adjacent to all empty points of the region.

3. Implement Benson's algorithm [1] to find all the blocks that have two or more healthy subregions. Mark them as safe.

4. Call *HasSureLiberties()* for each subregion. If the subregion is proven as static 2-vital, then mark this subregion as safe. Otherwise, if all the boundary blocks of this subregion are already marked as safe, and there is no space inside this subregion for the attacker to make two eyes, then also mark this subregion as safe.

5. If all the points in the input point set are marked as safe, then *StaticSafe()*

returns safe. Otherwise, returns non-safe.

Each time *StaticSafe()* is called, it has to compute subregions and boundary blocks that are generated during the search. Furthermore, *HasSureLiberties()* is used for testing each subregion. Therefore, *StaticSafe()* is much slower than *HasSureLiberties()*. The relative speed of the two methods varies widely, but 5–10 times slower is typical. For efficiency, we use the following compromise rule: If the previous move changes the size of a region by more than 2 points, then *StaticSafe()* is used. Otherwise, the quicker *HasSureLiberties()* is used. In contrast, [14] used only a weaker form of *HasSureLiberties()*.

# Chapter 6

# Experiments

The safety solver described here has been developed as part of the Go program *Explorer* [13]. To compare the performance of our current solver with the previous solver [14], our test set 1 is the same, the problem set *IGS_31_counted* from the Computer Go Test Collection [13]. The set contains 31 problems. Each of them is the final position of a 19 $\times$ 19 game played by human amateur players.

Since test set 1 was used to develop and debug the solver, we created an independent test set 2 and test set 3. Test set 2 contains 27 final positions of games by the Chinese professional 9 dan player ZuDe Chen. Test set 3 contains 35 final positions of games by Korean professional Go players. All three sets are available at `http://www.cs.ualberta.ca/˜mmueller/cgo/general.html`.

All experiments were performed on a Pentium 4 with 1.6 Ghz and a 64MB transposition table. The following abbreviations for the solvers and enhancements are used in the tables:

**Benson** Benson's algorithm, as in [14].

**Static-1997** Static solver from [14].

**Search-1997** Search-based solver, 6 ply depth limit, from [14].

**Static-2004** Current version of static solver.

| Version | Safe points | Safe blocks | Safe regions |
|---|---|---|---|
| Benson | 1,886 (16.9%) | 103 (9.2%) | 204 (25.4%) |
| Static-1997 | 2,481 (22.2%) | 168 (15.0%) | N/A |
| Search-1997 | 2,954 (26.4%) | 198 (17.6%) | N/A |
| Static-2004 | 2,898 (25.9%) | 212 (18.9%) | 321 (40.0%) |
| M1 | 4,017 (35.9%) | 326 (29.0%) | 404 (50.4%) |
| M2 | 4,073 (36.4%) | 330 (29.4%) | 406 (50.6%) |
| M3 | 5,029 (44.9%) | 444 (39.5%) | 495 (61.7%) |
| M4 | 5,070 (45.3%) | 451 (40.2%) | 498 (62.1%) |
| M5 | 5,396 (48.2%) | 484 (43.1%) | 519 (64.7%) |
| M6 (Full) | 5,740 (51.3%) | 523 (46.6%) | 548 (68.3%) |
| Perfect | 11,191 (100%) | 1,123 (100%) | 802 (100%) |

Table 6.1: Search improvements in test set 1

**M1** A basic 2-liberties search, similar to the one in [14].

**M2** M1 + consider external eyes of blocks as in Section 4.5.

**M3** M2 + region merging method as in Section 4.3.

**M4** M3 + move ordering and pruning as in Section 5.1.

**M5** M4 + improved heuristic and exact evaluation functions as in Section 5.2.

**M6** Full solver, M5 + weakly dependent regions as in Section 4.4.

## 6.1 Experiment 1: Overall Comparison of Solvers

Table 6.1 shows the results for all methods listed above for test set 1. The set contains 31 full-board positions with a total of $31 \times (19 \times 19) = 11{,}191$ points, 1,123 blocks and 802 regions. For methods M1–M6, a time limit of 200 seconds per region was used. For results with shorter time limits, see Experiment 2.

Table 6.2 shows the results for all methods listed above for test set 2. This test set contains a total of $27 \times (19 \times 19) = 9{,}747$ points, 1,052 blocks and 742 regions.

Table 6.3 shows the results for all methods listed above for test set 3. This test set contains a total of $35 \times (19 \times 19) = 12{,}635$ points, 1,362 blocks and 869 regions.

| Version | Safe points | Safe blocks | Safe regions |
|---|---|---|---|
| Benson | 1,329 (13.6%) | 106 (10.1%) | 160 (21.6%) |
| Static-2004 | 2,287 (23.5%) | 188 (17.9%) | 251 (33.8%) |
| M1 | 3,244 (33.3%) | 273 (25.9%) | 320 (43.1%) |
| M2 | 3,305 (33.9%) | 278 (26.0%) | 325 (43.8%) |
| M3 | 4,079 (41.9%) | 380 (36.1%) | 409 (55.1%) |
| M4 | 4,220 (43.3%) | 394 (37.5%) | 420 (56.7%) |
| M5 | 4,594 (47.1%) | 440 (42.0%) | 455 (61.4%) |
| M6 (Full) | 4,822 (49.5%) | 483 (45.9%) | 481 (64.9%) |
| Perfect | 9,747 (100%) | 1,052 (100%) | 742 (100%) |

Table 6.2: Search improvements in test set 2

| Version | Safe points | Safe blocks | Safe regions |
|---|---|---|---|
| Benson | 1,319 (10.4%) | 86 (6.3%) | 140 (16.1%) |
| Static-2004 | 2,643 (20.9%) | 214 (15.7%) | 282 (32.5%) |
| M1 | 3,906 (30.9%) | 322 (23.6%) | 364 (41.9%) |
| M2 | 4,109 (32.5%) | 353 (25.9%) | 381 (43.8%) |
| M3 | 4,792 (37.9%) | 435 (31.9%) | 449 (51.7%) |
| M4 | 4,887 (38.7%) | 448 (32.9%) | 455 (52.4%) |
| M5 | 5,130 (40.6%) | 472 (34.7%) | 474 (54.5%) |
| M6 (Full) | 5,291 (41.9%) | 499 (36.6%) | 493 (56.7%) |
| Perfect | 12,635 (100%) | 1,362 (100%) | 869 (100%) |

Table 6.3: Search improvements in test set 3

In the results of test set 1, the current static solver performs similarly to the best 1997 solver. Adding search and adding region merging yield the biggest single improvements in performance, about 10% each. The heuristic evaluation function and weakly dependent regions add about 3% each. Other methods provide smaller gains with these long time limits, but they are essential for more realistic shorter times, as in the next experiment.

Results for test set 2 and set 3 are a little bit worse than for test set 1, but that is true even for the baseline Benson algorithm. There does not seem to be a bias of tuning our solver especially for the problems in test set 1.

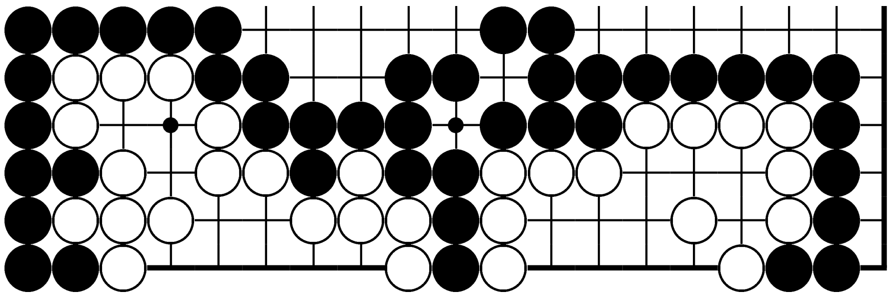
## 6.2 Experiment 2: Detailed Comparison of Solvers

This experiment compares the six search-based methods M1–M6 in more detail on test set 1. The static solver can prove 321 out of 802 regions safe. The best solver M6 can prove 548 regions with a time limit of 200s per region. The remaining 254 regions have not been solved by any method.

A total of (548–321) = 227 regions can be proven safe by search. To further analyze the search improvements, we divide these regions into four groups of increasing difficulty, as estimated by the CPU time used.

Group 1, very easy (regions 322–346): This group contains 25 regions. Most regions in this group have small size, less than 10. All methods M1–M6 solve all 25 regions quickly within 0.1s (0.2s for M1).

Group 2, easy (regions 347–408): This group contains 62 regions. Figure 6.1 shows two examples. Table 6.4 shows the number of regions solved by each method with different time limits. The number in braces is the difference between two methods. The performance of M1 and M2 is not convincing. By using region merging, M3 solves all 62 regions within 0.5s. The more optimized methods M4–M6 solve all within 0.1s. Region merging dramatically improves the performance



Left: A merged white region (Size: 10). Right: A basic white region (Size: 11)

Figure 6.1: Two examples of easy problems in group 2

| Version | M1 | | M2 | | M3 | | M4 | M5 | M6 |
|---|---|---|---|---|---|---|---|---|---|
| T=0.1s | 0 | | 23 | | 38 | | **62** | **62** | **62** |
| T=0.5s | 29 | (+29) | 31 | (+8) | **62** | (+24) | | | |
| T=1.0s | 39 | (+10) | 40 | (+9) | | | | | |
| T=5.0s | 43 | (+4) | 42 | (+2) | | | | | |
| T=10s | 43 | (+0) | 44 | (+2) | | | | | |
| T=50s | 43 | (+0) | 49 | (+5) | | | | | |
| T=200s | 43 | (+0) | 49 | (+0) | | | | | |
| Solved | 43 | | 49 | | **62** | | **62** | **62** | **62** |

Table 6.4: Search results for Group 2, easy (62 regions)

of solving these easy regions.

Group 3, moderate (regions 409–495): This group contains 87 regions. Figure 6.2 shows two examples. The left example in this figure contains two white regions. The smaller white region (size 3) can be treated as an external eye of a white boundary block (as described in Section 4.5). However, since it is not a simple eye, the current solver will merge two white regions together.

Table 6.5 contains the test results. In this group, the search enhancements dramatically improve the solver. M1 and M2 solve few problems. M3 can solve 79 regions, but more than half of them need more than 10 seconds. The evaluation function dramatically speeds up the solver. M5 solves all regions within 10 seconds. M6, using weakly dependent regions, solves 23 regions within 0.1s, as opposed to 0 for M5. All 87 regions are solved within 5s. In this category M6 outperforms all

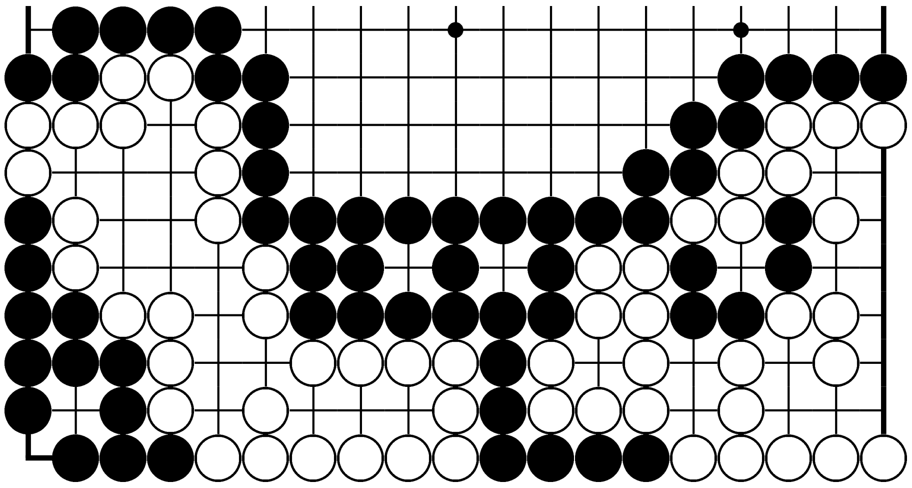

Left: A merged white region (Size: 16). Right: A basic white region (Size: 19)

Figure 6.2: Two examples of moderate problems in group 3.

| Version | M1 | | M2 | | M3 | | M4 | | M5 | | M6 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T=0.1s | 0 | | 0 | | 0 | | 0 | | 0 | | 23 | |
| T=0.5s | 0 | | 0 | | 14 | (+14) | 14 | (+14) | 10 | (+10) | 37 | (+14) |
| T=1.0s | 0 | | 6 | (+6) | 33 | (+19) | 33 | (+19) | 38 | (+28) | 59 | (+22) |
| T=5.0s | 0 | | 6 | (+0) | 38 | (+5) | 38 | (+5) | 68 | (+30) | **87** | (+28) |
| T=10s | 0 | | 8 | (+2) | 38 | (+0) | 40 | (+2) | **87** | (+19) | | |
| T=50s | 0 | | 10 | (+2) | 73 | (+35) | 79 | (+39) | | | | |
| T=200s | 13 | (+13) | 17 | (+7) | 79 | (+6) | 82 | (+3) | | | | |
| Solved | 13 | | 17 | | 79 | | 82 | | **87** | | **87** | |

Table 6.5: Search results for Group 3, moderate (87 regions)

other methods.

Group 4, hard (regions 496–548): This group contains the 53 regions that are solved in 5s–200s by M6. Figure 6.3 shows three examples. In Figure 6.3 (c) there are three white regions (size: 13, 14 and 2). However, the white region in the right corner (size 2) can also be treated as an external eye of a white boundary block (as described in Section 4.5). Therefore, it is possible to further improve the current solver to handle external eyes. Table 6.6 contains the test results. This group includes 20 weakly dependent regions that cannot be solved by M1–M5. Many of these problems take more than a minute even with M6. They represent the limits of

| Version | M1 | | M2 | | M3 | | M4 | | M5 | | M6 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T=0.1s | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| T=0.5s | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| T=1.0s | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| T=5.0s | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| T=10s | 0 | | 0 | | 0 | | 0 | | 11 | (+11) | 11 | (+11) |
| T=100s | 0 | | 0 | | 15 | (+15) | 17 | (+17) | 21 | (+10) | 28 | (+17) |
| T=200s | 5 | (+5) | 5 | (+5) | 17 | (+2) | 20 | (+3) | 33 | (+12) | **53** | (+25) |
| Solved | 5 | | 5 | | 17 | | 20 | | 33 | | **53** | |

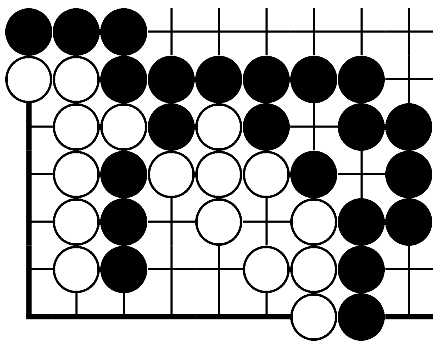
Table 6.6: Search results for Group 4, hard (53 regions)

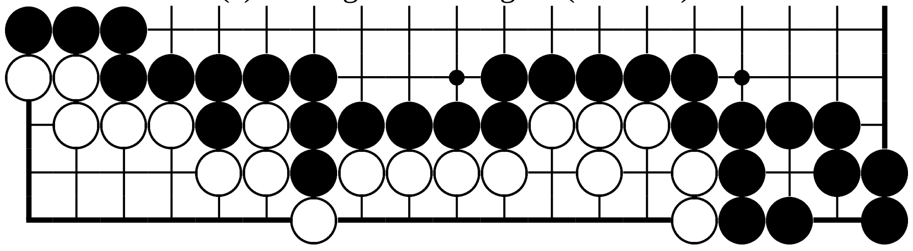
the current solver.

## 6.3 Experiment 3: Comparison with GNU Go

*GNU Go* program is one of the strongest Go programs nowadays. The version we choose is the latest 3.5.6 (available at `http://www.gnu.org/software/gnugo/devel.html`. In *GNU Go*, there is a safety solver that checks the unconditional status of stones on the Go board and returns one of the following five results: *Black territory*, *White territory*, *Live*, *Dead* and *Unknown*. The first four results are exact.

We compare our Benson solver and static solver with the *GNU Go* safety solver. Table 6.7 shows the results for all three test sets. The table shows the number of stones that are proved to be safe. Even though *GNU Go* is a strong Go program, its safety solver is relatively weak. It is a little bit better than our Benson solver, but much worse than our Static solver. Therefore, it is unnecessary to compare *GNU Go*'s safety solver with our other improved solvers.


(a) A merged white region (Size: 17)


(b) Two weakly dependent white regions (Size: 11 and 9)

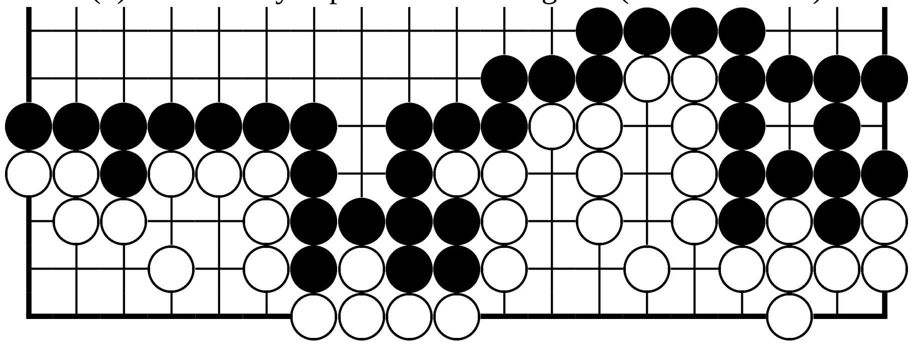

(c) Three weakly dependent white regions (Size: 13, 14 and 2)

Figure 6.3: Three examples of hard problems in group 4

| Version | Set 1 | Set 2 | Set 3 |
|---|---|---|---|
| Benson | 1,886 (16.9%) | 1,329 (13.6%) | 1,319 (10.4%) |
| Static-2004 | 2,898 (25.9%) | 2,287 (23.5%) | 2,643 (20.9%) |
| GNU Go safety solver | 1,926 (17.2%) | 1,335 (13.7%) | 1,330 (10.5%) |
| Perfect | 11,191 (100%) | 9,747 (100%) | 12,635 (100%) |

Table 6.7: Comparison with GNU Go

# Chapter 7

# Conclusions and Future Work

The results of our work on proving territories safe are very encouraging. Using a combination of both new region-processing methods and search enhancements, the current safety solver is significantly faster and more powerful than the previous solver described in [14] and the GNU Go solver. However, most large areas with more than 18 empty points still remain unsolvable due to the size of the search space. Figure 7.1 shows an example. Although this region has only 18 empty points, our current solver could not solve it within 200 seconds and a 14 ply search. In order to handle larger areas, the current solver can be improved in the following areas:

**Move generation** More Go knowledge could be used for safe forward pruning. Instead of generating all legal moves, in many cases the program could analyze the attacker's motivations and generate refutation moves. Move ordering and

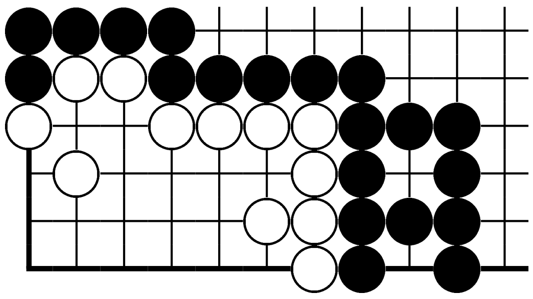


Figure 7.1: Example of an unsolved region (Size: 18)

pruning for the attacker should also be investigated.

**Evaluation function** The current exact evaluation function is all-or-nothing, and tries to decide the safety of the whole input area. If the area becomes partially safe during the search, this information is ignored. However, it would be very useful in order to simplify the further search. Also, more research on fine-tuning the evaluation function is needed.

More future work ideas include:

- Handle special cases such as seki, snapback, double ko.
- Use the solver in Explorer to prove regions unsafe and find successful invasions, or defend against them.
- Develop a heuristic version that can find possible weaknesses in large areas.
- Develop a safety solver using a depth first proof number (df-pn) search engine.

# Bibliography

[1] D.B. Benson. Life in the Game of Go. *Information Sciences*, 10:17–29, 1976. Reprinted in Computer Games, Levy, D.N.L. (Editor), Vol. II, pp. 203-213, Springer Verlag, New York 1988.

[2] H.J. Berliner. A Chronology of Computer Chess and its Literature. *Artificial Intelligence*, 10:201–214, 1978.

[3] H.J. Berliner. Backgammon Computer Program Beats World Champion. *Artificial Intelligence,* 14:205–220, 1980.

[4] A.L. Brudno. Bounds and Valuations for Shortening the Search of Estimates. *Problems of Cybernetics*, 10:225–241, 1963. Appeared originally in Russian in Problemy Kibernetiki, vol.10, pp.141-150, 1963.

[5] M. Buro. *Methods for the Evaluation of Game Positions Using Examples*. PhD thesis, University of Paderborn, Germany, 1994.

[6] M. Buro. ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm. *ICCA Journal*, 18(2):71–76, 1995.

[7] M.S. Campbell and T.A. Marsland. A Comparison of Minimax Tree Search Algorithms. *Artificial Intelligence*, 20:347–367, 1983.

[8] Aviezri S. Fraenkel. Selected Bibliography on Combinatorial Games and Some Related Material. *Proceedings of Symposia in Applied Mathematics,* 43:191–226, 1991.

[9] R.D. Greenblatt, D.E. Eastlake, and S.D. Crocker. The Greenblatt Chess Program. *Fall Joint Computing Conf. Procs.*, 31:801–810, 1967.

[10] D.E Knuth and R.W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6:293–326, 1975.

[11] T.A. Marsland. Relative Performance of Alpha-Beta Implementations. In *Intenational Joint Conferences on Artificial Intelligence (IJCAI'83),* pages 763–766, 1983.

[12] T.A. Marsland and M.S. Campbell. Parallel Search of Strongly Ordered Game Trees. *Computing Surveys*, 14(4):533–551, 1982.

[13] M. Müller. *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory*. PhD thesis, ETH Zürich, 1995. Diss. ETH Nr. 11.006.

[14] M. Müller. Playing it Safe: Recognizing Secure Territories in Computer Go by Using Static Rules and Search. In H. Matsubara, editor, *Game Programming Workshop in Japan '97*, pages 80–86, Computer Shogi Association, Tokyo, Japan, 1997.

[15] J Pearl. Asymptotic Properties of Minimax Trees and Game-Searching Procedures. *Artificial Intelligence*, 14(2):113–138, 1980.

[16] J Pearl. *Heuristics – Intelligent Search Strategies for Computer Problem Solving*. Addison–Wesley, 1984.

[17] A. Plaat. *Research Re: Search and Re-search*. PhD thesis, Tinbergen Institute and Department of Computer Science, Erasmus University, 1996.

[18] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-first Fixed-Depth Minimax Algorithms. *Artificial Intelligence*, 87:55–293, 1996.

[19] A. Reinefeld. An Improvement of the Scout Tree Search Algorithm. *International Computer Chess Association Journal*, 6(4):4–14, 1983.

[20] A.L. Samuel. Some Studies in Marchine Learning. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[21] J Schaeffer. *Experiments in Search and Knowledge*. PhD thesis, University of Waterloo, Canada, 1986. Available as University of Alberta technical report TR86-12.

[22] J Schaeffer. The History Heuristic and Alpha-beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(1):1203–1212, 1989.

[23] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag, 1997.

[24] J.H. Slagle and J.K. Dixon. Experiments with Some Programs that Search Game Trees. *Journal of the ACM*, 16(2):189–207, 1969.

[25] D.J. Slate and L.R Atkin. *Chess 4.5 – The Northwestern University Chess Program*. Springer-Verlag, 1977.

[26] E. van der Werf, J. van den Herik, and J. Uiterwijk. Learning to Score Final Positions in the Game of Go. In J. van den Herik, H. Iida, and E. Heinz, editors, *Advances in Computer Games 10*, pages 143 – 158. Kluwer, 2004.

[27] R. Vilà and T. Cazenave. When One Eye is Sufficient: a Static Classification. In J. van den Herik, H. Iida, and E. Heinz, editors, *Advances in Computer Games 10*, pages 109 – 124. Kluwer, 2004.

[28] A.L. Zobrist. A New Hashing Method with Applications for Game-playing. *Technical report, Department of Computer Science, University of Wisconsin*, 1970. Reprinted in International Computer Chess Association Journal, 13(2):169–173, 1990.

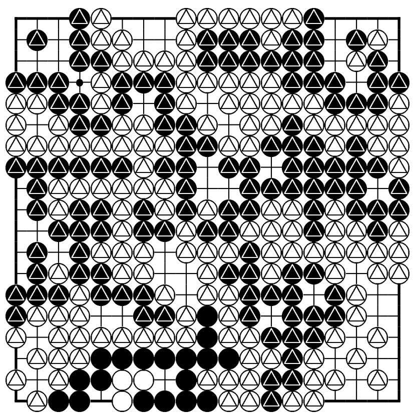
# Appendix A

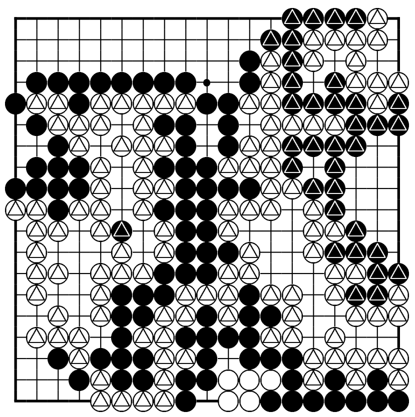
# Test Data

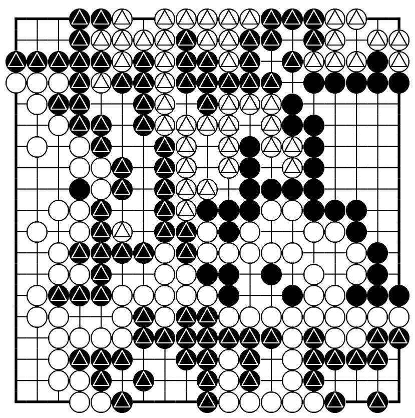
This appendix gives the test positions of test set 1. Only safe and dead stones are marked by triangles. The other two test sets are available at `http://www.cs.ualberta.ca/~games/go/safety/`.

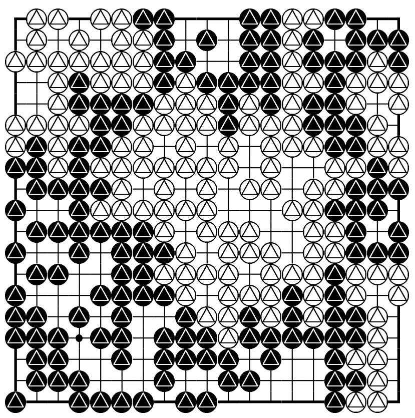
## A.1 Test Positions

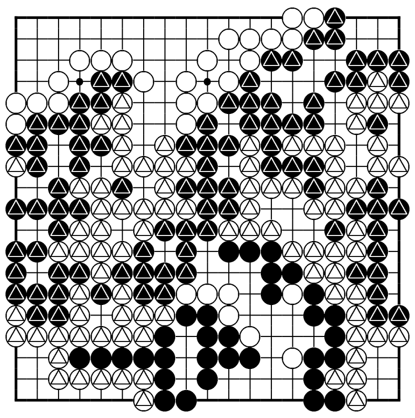
Test Set 1 (31 board positions).

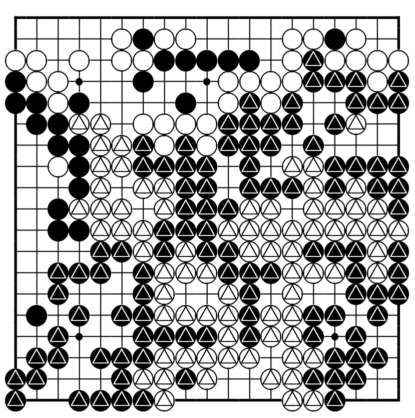

Position 1

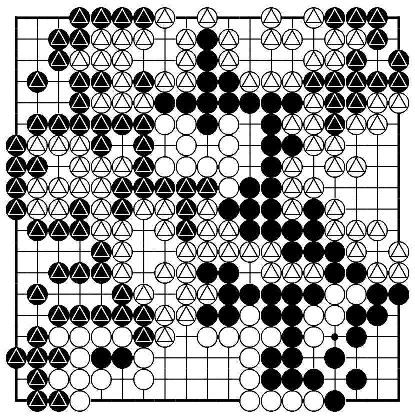

Position 2

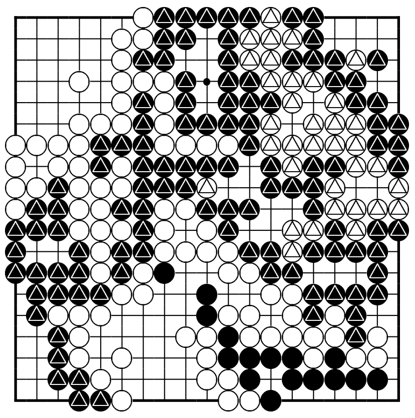

Position 3

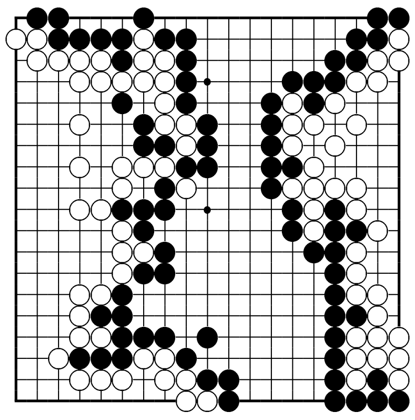

Position 4

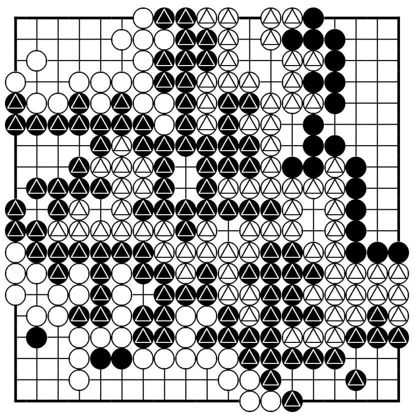

Position 5

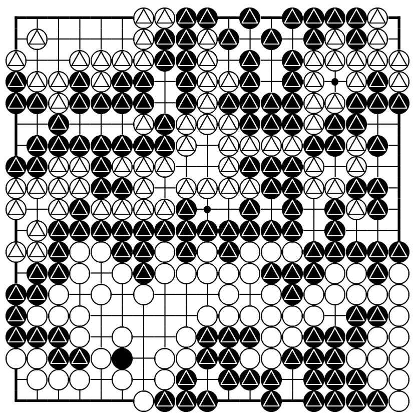

Position 6

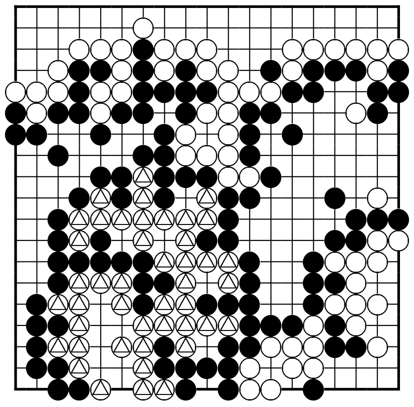

Position 7

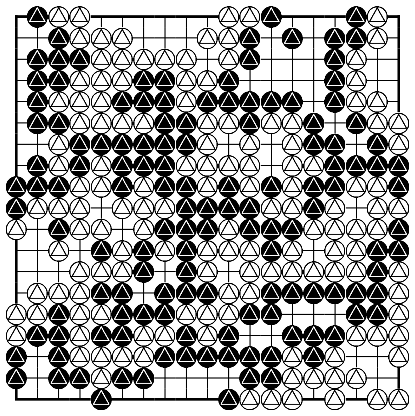

Position 8

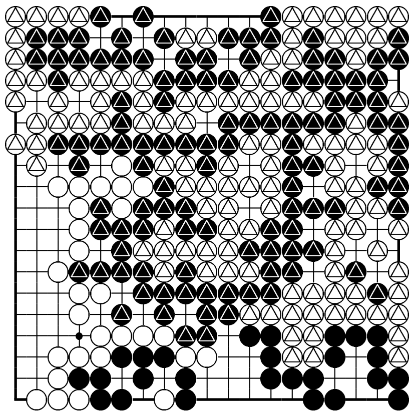

Position 9

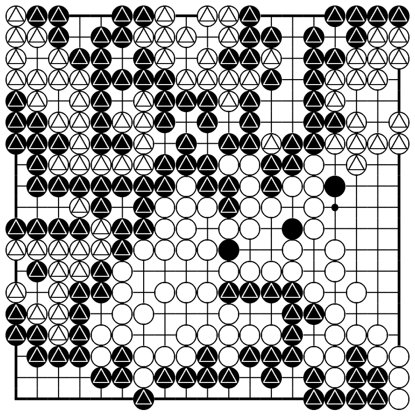

Position 10

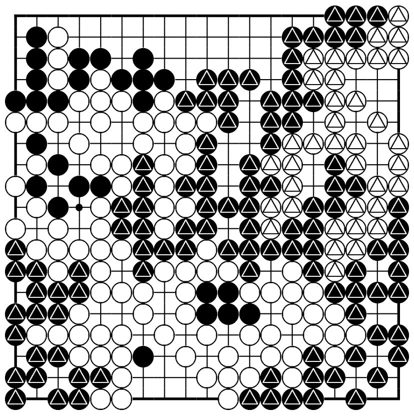

Position 11

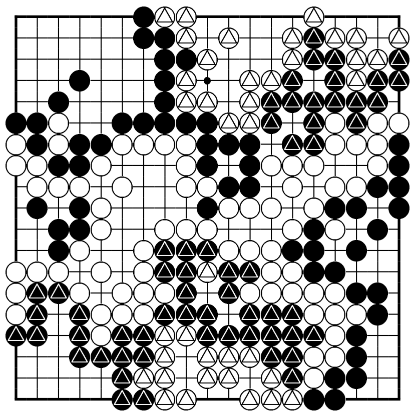

Position 12

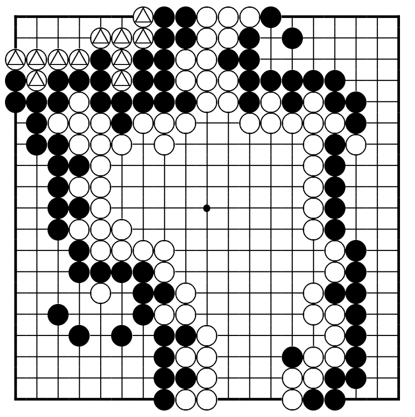

Position 13

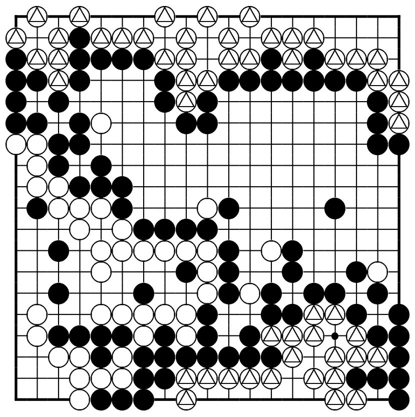

Position 14

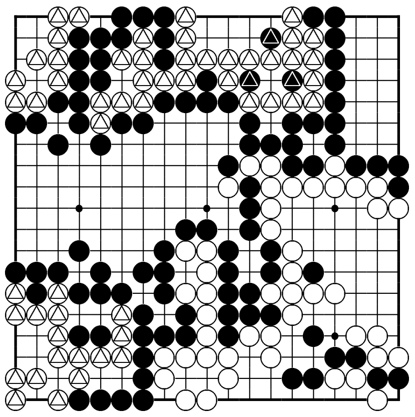

Position 15

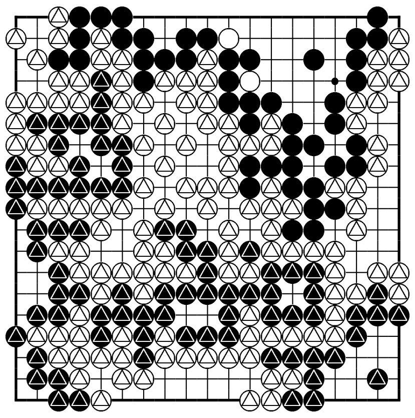

Position 16

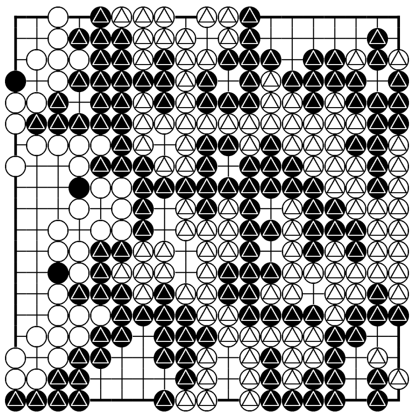

Position 17

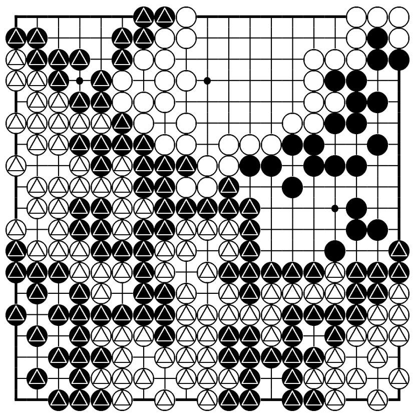

Position 18

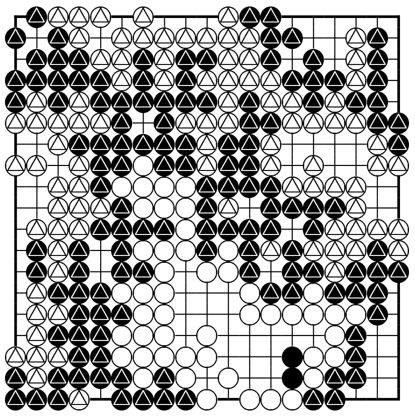

Position 19

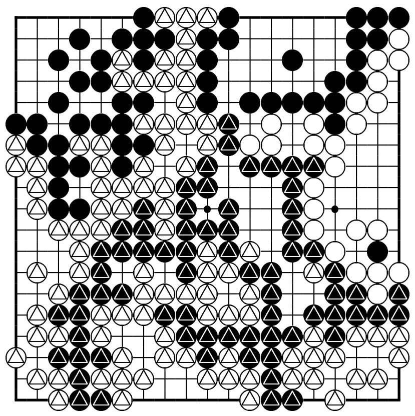

Position 20

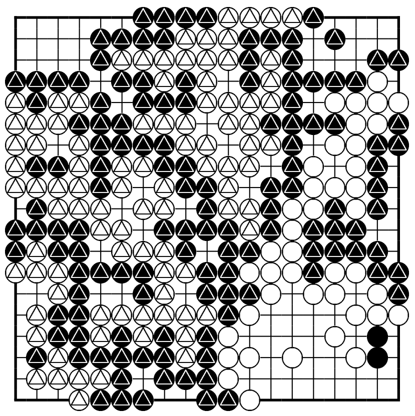

Position 21

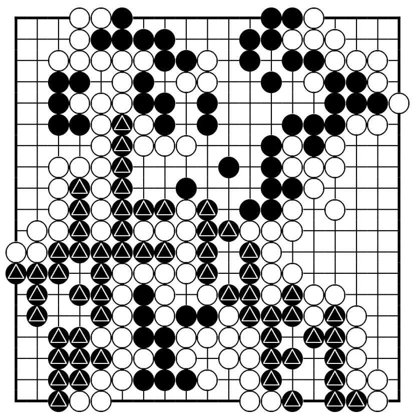

Position 22

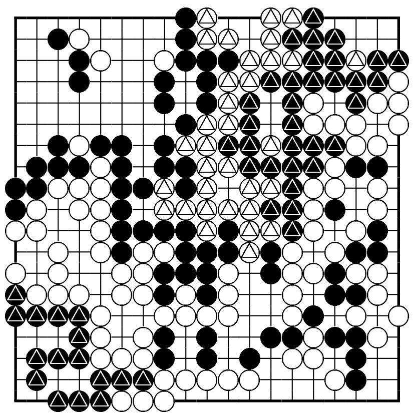

Position 23

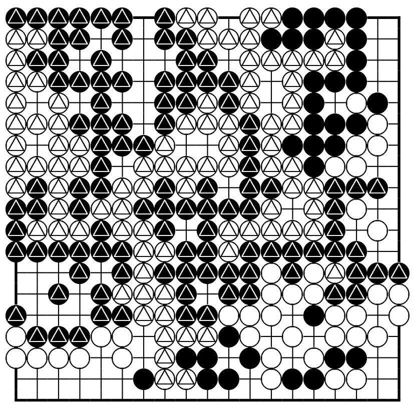

Position 24

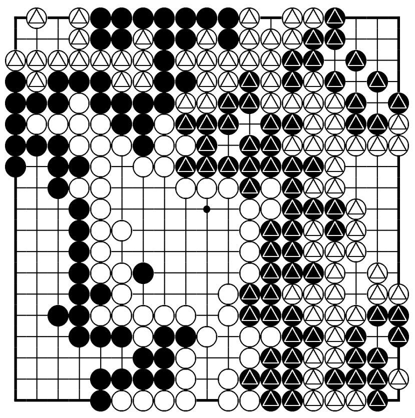

Position 25

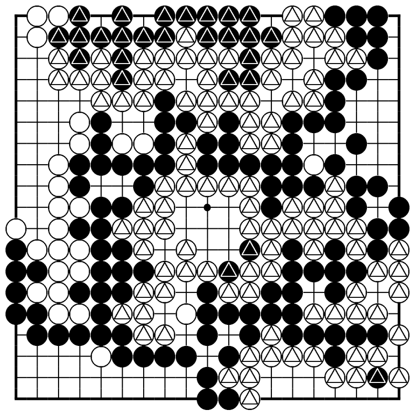

Position 26


Position 27


Position 28


Position 29


Position 30


Position 31