MONTE-CARLO TREE SEARCH WITH HEURISTIC KNOWLEDGE: A NOVEL WAY IN SOLVING CAPTURING AND LIFE AND DEATH PROBLEMS IN GO

by

Peigang Zhang

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Information Technology

Charlotte

2010

Approved by:

Dr. Keh-Hsun Chen

Dr. Jing Xiao

Dr. Zbigniew W. Ras

Dr. Xintao Wu

Dr. Jacek Dmochowski

ABSTRACT

PEIGANG ZHANG. Monte-Carlo tree search with heuristic knowledge: a novel way in solving capturing and life and death problems in Go. (Under the direction of Dr. Keh-Hsun Chen)

Monte-Carlo (MC) tree search is a new research field. Its effectiveness in searching large state spaces, such as the Go game tree, is well recognized in the computer Go community. Go domain-specific heuristics and techniques as well as domain-independent heuristics and techniques are systematically investigated in the context of the MC tree search in this dissertation. The search extensions based on these heuristics and techniques can significantly improve the effectiveness and efficiency of the MC tree search.

Two major areas of investigation are addressed in this dissertation research: I. The identification and use of the effective heuristic knowledge in guiding the MC simulations, II. The extension of the MC tree search algorithm with heuristics. Go, the most challenging board game to the machine, serves as the test bed. The effectiveness of the MC tree search extensions is demonstrated through the performances of Go tactic problem solvers using these techniques.

The main contributions of this dissertation include:

1. A heuristics based Monte-Carlo tactic tree search framework is proposed to extend the standard Monte-Carlo tree search.
2. (Go) Knowledge based heuristics are systematically investigated to improve the Monte-Carlo tactic tree search.
3. Pattern learning is demonstrated as effective in improving the Monte-Carlo tactic tree search.
4. Domain knowledge independent tree search enhancements are shown as effective in improving the Monte-Carlo tactic tree search performances.
5. A strong Go Tactic solver based on proposed algorithms outperforms traditional game tree search algorithms.

The techniques developed in this dissertation research can benefit other game domains and application fields.

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to Prof. Keh-Hsun Chen, my supervisor, for his patient guidance. He supported me through advises, discussions and in many other ways. Without his consistent support I could never finish this dissertation.

I am grateful to Dr. Martin Müller for sharing his Kano book capturing problems library and Dr. Thomas Wolf for sharing his Life&Death problem library.

Also, I would like to thank Dr. Jing Xiao, Dr. Zbigniew W. Ras, Dr. Xintao Wu and Dr. Jacek Dmochowski for their insightful comments to my dissertation.

Mrs. Stella Butterbaugh gave me useful comments to my dissertation format and I thank her.

Finally, I would also like to thank my wife and my daughter for their love and support. I owe them a lot of weekends.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION 1

1.1 Computer Go and Game Tree Search 2

1.2 Monte-Carlo Go 3

1.3 UCT Algorithm 4

1.4 The Extensions of UCT 7

1.5 Research Problems 8

CHAPTER 2: COMPUTER GO TACTIC PROBLEMS 9

2.1 Capturing Problems 9

2.2 Life&Death Problems 10

CHAPTER 3: CAPTURING PROBLEMS AND GAME TREE SEARCH ALGORITHMS 12

3.1 Capturing Problem Formalization 12

3.1.1 Candidate Move Generation 12

3.1.2 Node Evaluation 13

3.1.3 Terminal Test 14

3.2 Game Tree Search Algorithms 14

3.3 $\alpha\beta$ Search 15

3.4 Proof Number Search 16

3.5 Result Analysis 17

3.5.4 Algorithms Comparison 17

3.5.5 Analysis of the Capturing Problems 19

3.5.6 Data Analysis of the $\alpha\beta$ Search Results 19

3.5.7 Data Analysis of the Proof Number Search Results 21

3.6 Remarks 22

CHAPTER 4: MONTE-CARLO TACTIC TREE SEARCH FRAMEWORK 23

4.1 Monte-Carlo Tactic Tree Search Framework 23

4.2 Heuristic Based UCT+ Tree Search 24

4.3 Semi-random Simulation Policy 26

4.4 Local Search Region 27

4.4.1 Local Search Region for Capturing Problems 27

4.4.2 Local Search Region for Life&Death Problems 28

4.4.3 Virtual Boundary 28

4.5 The Determination of the MC Simulation Terminal Status and Terminal Node Tag 29
4.5.4 For Capturing Problems 29
4.5.5 For Life&Death Problems 29
4.5.6 Terminal Node Tag 30
4.6 Exploitation and Exploration Coefficient 30
4.7 Search Result Confidence 30
CHAPTER 5: KNOWLEDGE BASED HEURISTICS 32
5.1 Semi-random Simulation Policy 32
5.1.1 Solid Eye 32
5.1.2 Basic Stone Capturing and Escaping 33
5.1.3 Tactic Initial Probability Weight 34
5.1.4 Creating an Eye instead of Filling an Eye 35
5.1.5 Avoiding Self-Atari 35
5.1.6 Disabling Single Stone Self-Atari 35
5.1.7 Using the Proximity Heuristic 35
5.1.8 Pseudo Ladder 37
5.2 Heuristic Tree Search 37
5.3 Test Results and Analysis 38
5.3.9 No Heuristic Extensions 38
5.3.10 Solid Eye 38
5.3.11 Capture&Escape Probability Weight 39
5.3.12 Tactic Initial Probability Weight 40
5.3.13 Creating an Eye instead of Filling an Eye 41
5.3.14 Avoiding Self-Atari 44
5.3.15 Avoiding Single Stone Self-Atari 44
5.3.16 Ladder 44
5.3.17 Proximity 46
5.3.18 No New Candidate 47
5.3.19 Absence of the Virtual Boundary 49
5.3.20 Heuristic 50

CHAPTER 6: LEARNING PATTERNS 52
6.1 The Pattern Guided MC Simulation Policy 53
6.1.1 How a Pattern Library is Created 53
6.1.2 How Patterns Are Used in the MC Simulations 56
6.2 Using The Trained ANN to Guide the MC Simulation Policy 57
6.2.3 Training the ANN 57
6.2.4 Using the ANN 58
6.3 Test Results and Analysis 59
6.3.5 3*3 Pattern 59
6.3.6 5*5 Pattern 60
6.3.7 ANN 63
CHAPTER 7: DOMAIN INDEPENDENT TREE SEARCH TECHNIQUES 66
7.1 History Heuristic 66
7.2 Sorting Children 67
7.3 Greedy Mode 67
7.4 Heavy Back Up When Terminal Nodes Are Reached 67
7.5 Test Results and Analysis 68
7.5.1 History 68
7.5.2 Greedy Mode 68
CHAPTER 8: RESULTS SUMMARY AND DISCUSSIONS 71
8.1 Results Charts 71
8.2 Results Summary 73
8.3 6-Dan level Life&Death Problem Solved 74
8.4 Contributions and Discussions 74
REFERENCES 76
APPENDIX A: TACTIC SOLVER ARCHITECTURE 80
APPENDIX B: GO TERMS 81

CHAPTER 1: INTRODUCTION

Traditional game tree search algorithms such as $\alpha\beta$ search and Proof Number search explore the game tree to find an optimal path. They use a heuristic evaluation function to approximate the value of the leaf node if the game is not terminated at the leaf node. This technique has been very successful for games such as chess, checker or draught. In the game Go, due to high complexity of its game tree (state space complexity:10^{172}, game tree complexity:10^{360}) and difficult mechanical evaluation of its game positions, these algorithms have not succeeded in producing a strong computer Go player. Through the use of the average value of thousands of random continuations of a game position can form a reasonable evaluation of the position. But this basic Monte-Carlo (MC) simulation on the child nodes of the root node can not converge to the best move. Upper Confidence bounds applied to Trees (UCT) algorithm provides an effective extension to the basic MC simulation technique in game tree search. It can converge to an optimal path given sufficient simulations. The MC tree search algorithm has shown its capacity and efficiency in searching Go game trees to produce much stronger playing programs than the traditional knowledge and search based programs.

The MC tree search algorithm consists of two basic components: the tree search and the MC simulation. Current research results show that adding domain-specific and domain-independent heuristics to both parts can largely increase the game tree search efficiency. In this way, MC tree search becomes UCT+ algorithm while the tree search portion becomes heuristic tree search and the random simulation portion involves heuristic knowledge guided semi-random simulations.

This dissertation tries to answer two important questions:

1. What kind of domain-independent heuristics and techniques can be used to increase the efficiency of the MC tree search?
2. What kind of domain-specific (Go) heuristics can be used to increase the efficiency of the MC tree search?

Go is the only remaining popular board game that has failed to produce a strong computer player, and this is a great challenge to AI researchers. Solving computer Go tactic problems, including capturing and Life&Death, is used as a test bed.

1.1 Computer Go and Game Tree Search

Go is a standard two player zero-sum board game with perfect information. The standard Go board size is 19 x 19. The rule for Go is very simple: Black and White players alternately place stones at an empty intersection; stones that are not connected to free points are removed; repetition or suicide is not allowed. At the end of the game the player who controls the most territory wins.

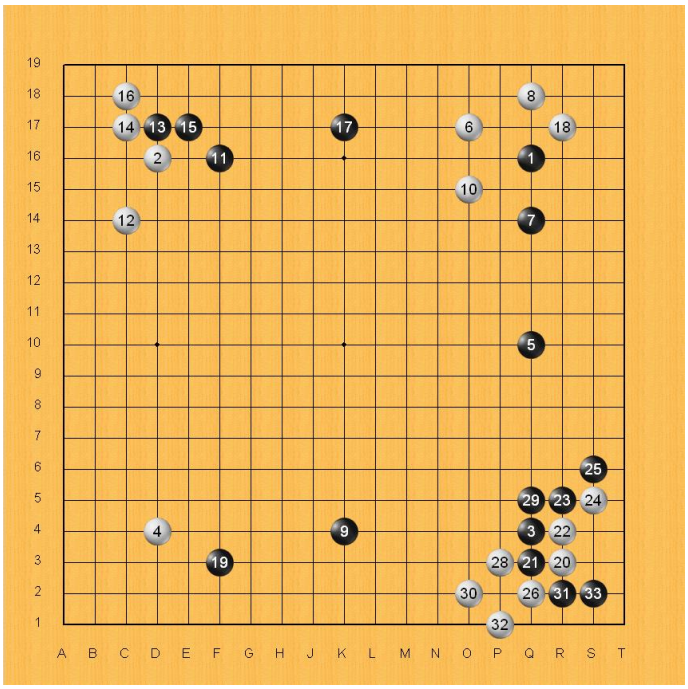


Figure 1.1: The Game of Go

The first Go program can be traced back to the 1960s. In the 1970s Computer Go became a field of research, and from that time on, a lot of Go programs and computer Go competitions emerged. Mller [44] provided an introduction to computer Go. Bouzy [8] had a survey of AI techniques used in computer Go.

Classical game tree search techniques are minimax tree search [39, 13, 1, 37, 45, 48, 52, 54]. They use evaluation functions to approximate the leaf node if a terminal status is not reached. The most famous one is $\alpha\beta$ search [35]. It is very successful in providing strong game programs such as chess (deep blue) [14], checker and Gomoku. But these techniques failed to provide a strong Go program. The failure is mainly due to two reasons: the relatively large search space complexity of Go and the difficulty in providing a good evaluation function. Among the popular board games, Go has the highest degree of search complexity (Go:state space complexity:10^{172}, game tree complexity:10^{360}; chess:state space complexity:10^{50} and game tree complexity:10^{123}). Due to the dynamic nature of Go, the evaluation value usually lacks accuracy and a fair evaluation function would take a relatively

long time to compute (on an average PC: chess programs can evaluate tens of millions of positions while a Go program usually evaluates several thousands positions).

Most top traditional Go programs are knowledge based. They usually combine local tree searches with selectively shallow global tree searches. Intensive Go knowledge from humans is applied to evaluate positions and to select moves [21]. The strength of the best knowledge based program is about 5 kyu, which is below an average experienced amateur player. A knowledge based Go program is also very hard to improve and the performance of the program is largely dependent on the programmer's Go knowledge.

Through the use of classical game tree search methods, some progress has been achieved, such as solving Go on the small board (5 x 5) [31] and solving Life&Death problems in fully enclosed small size regions [37, 55].

To produce a strong Go program, new techniques are needed. Different theories and techniques have been used to create computer Go programs, such as the combinatorial game theory, machine learning [31] (NeuroGo [30]), cognitive modeling and more recently Monte-Carlo Go.

1.2 Monte-Carlo Go

The Monte-Carlo simulation is a stochastic sampling method. In game domains, it is normally used to deal with games of possibility. Go is a deterministic game, but researchers have found that the MC simulation can be used to provide an evaluation function and that the MC evaluation is robust, with global sense and easy to program.

The idea of using the MC simulation to evaluate position is very simple. From a given board position, the black and the white players randomly put stones on the board according to the rules and do not consider putting stones inside a true eye of his side (putting stone inside its true eye is a really bad move). The black and the white players play until there is nowhere to put the stones and then count the territory of each side to determine the winning player. From a given board position, a lot of random self-plays will be conducted and the winning rate of each side will be calculated. The idea is that the better winning rate would reflect a better board status for a certain side.

Brugmann (1993) [5] was the first to develop a Go program (Gobble) using the MC simulation to evaluate positions, but he did not use tree search techniques. Thus, even with sufficient time, his algorithm would not guarantee to converge to the optimal move. Bouzy adopted the MC simulation idea and combined it with min-max tree search [6]; he also used knowledge to guide random simulations which thus became semi-random simulations [9, 11, 12]. But this min-max tree search algorithm was not efficient. Bouzy also used iterative deepening and progressive pruning strategies

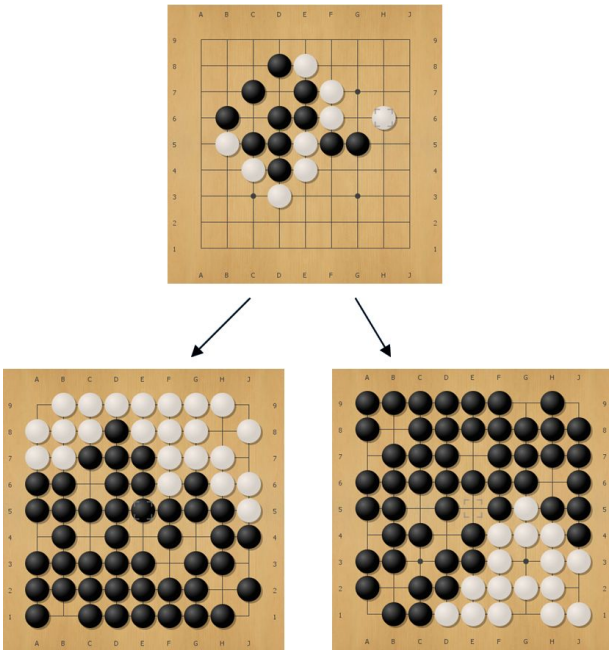


Figure 1.2: Evaluating Go Position Using the MC Simulation

to improve the tree search efficiency [10], but good moves can be pruned due to evaluation inaccuracies. Coulom [26] developed an efficient selectivity and backup algorithm in the MC tree search and the success of his program (CrazyStone) shows the power of the MC tree search. Almost concurrently, Kocsis [40] developed a similar bandit [4] based MC planning algorithm named UCT (Upper Confidence bounds applied to Trees) and this algorithm is more elegant, theoretically founded and easy to implement. MoGo [32] is the first Go program to use the UCT strategy to guide the tree search with great success. In 2007, MoGo surpassed classical knowledge based Go Programs for the first time at a formal international computer Go competition.

The research results of the MC Go has already been used at game domains (such as Amazons and Backgammon) and other application fields [16].

1.3 UCT Algorithm

In this dissertation, the UCT algorithm is adopted as the MC tree search strategy. The UCT algorithm contains two basic parts: the tree search and the MC evaluation. The pseudo code below shows the basic structure of the UCT algorithm.

Pseudo Code 1.1: The Basic Structure of the UCT Algorithm

```
while (nuSimulation > 0){
    nuSimulation--;
    //1. Start from the root node
```

```
UCTNode * pNode = m_pRootNode;
//2. Select the most promising node by UCB function
while(pNode->son != NULL) {
    pNode = SelectByUCB1(pNode);
}
//3. Create children for the currentleaf node
// if it is not the first visit
if (pNode->nuVisit > 0){
    if(CreateChildrenNodes(pNode))
        pNode = pNode->son;
}
//4. Calculate value by playing a semi-random
// game extension (play-out)
// then counting the territory
int value = GetValueByMC(pNode);
pNode->value += value;
pNode->nuVisit++;
//5. Update ancestor nodes' values
UpdateValue(pNode, value);
}
```

SelectByUCB1 function finds the child node with the best UCB1 value.

$$ChildNodeToExplore = argmax(UCB1(parent, node)) = argmax(\mu + \sigma) \tag{1.1}$$

where $\mu = \frac{node.value}{node.nuVisit}$ is the winning rate of this move. $\sigma = \sqrt{\frac{ln(parent.nuVisit)}{ExEValue \times node.nuVisit}}$ is the upper bound (uncertainty) of this move. ExEValue is the coefficient that balances search exploitation and exploration.

Note: Node.value in figure 1.3 is from the max-node point of view. At real calculations it should be modified to negamax-style.

The UCT tree search is a best-first tree search. Compared with traditional minimax tree search, it has its unique best child selection, child expansion, leaf nodes evaluation and value backup strategy. UCT algorithm continuously grows a tree in memory. It selects the best path to a leaf node according to the UCB1 function. If the leaf node is not the first to be visited, its child nodes will be created

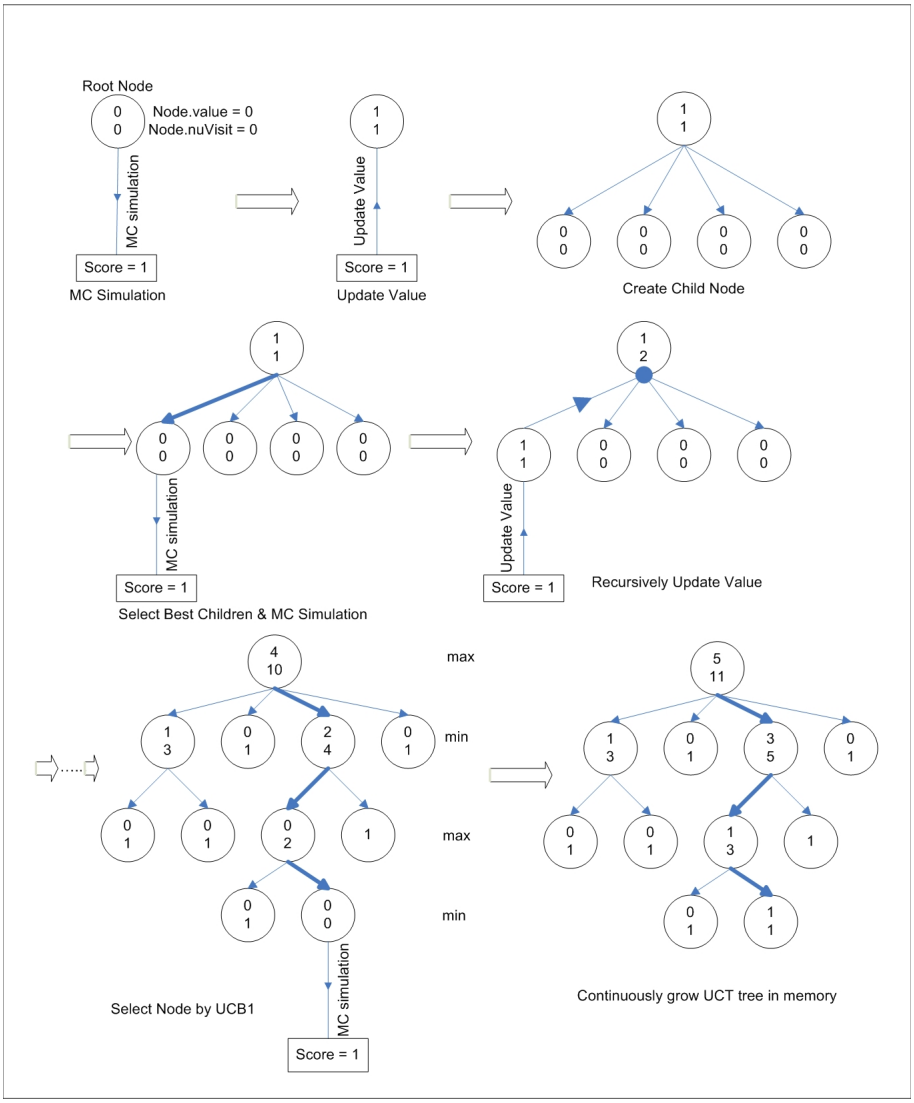


Figure 1.3: The UCT Tree Search

and a child will be selected. From this position, an MC simulation will be performed and the result will be collected and backed up.

UCT algorithm searches for the best child according to the UCB1 function, and it balances the search between the exploration and exploitation. It considers those nodes with the better value already gained and those nodes with higher uncertainties (fewer samples). The success of the UCT algorithm is mainly due to the combination of the MC simulation and the tree search guided by the UCB1 function. The MC simulation is a stochastic method and evaluation inaccuracy does exist; however, with the number of simulations increases, the mean value becomes more trustworthy.

Theoretical analysis of UCT shows:

1. The probability of selecting the best action converges to 1.

2. The search adapts to the effective size of the search tree.

Notes of the UCT algorithm:

- UCT is not a deterministic algorithm like $\alpha\beta$ search or proof number search. The search process contains stochastic portions.
- UCT is an algorithm that can stop at any time. Given more computing resources, the result would better converge to the optimal path (near optimal path).

1.4 The Extensions of UCT

The original version of UCT algorithm uses little domain knowledge. Researchers find that using Go knowledge to guide the MC simulations can greatly improve the efficiency of UCT algorithm [25]. Through the use of the semi-random simulations, UCT tree search can converge to the optimal path efficiently so as to increase the playing strength.

Extensions of the basic MC tree search are an active research field. No unified architecture has been provided yet, different researchers have different approaches.

Below is a classification of methods of using knowledge to guide the MC simulations:

1. Limited knowledge version of the UCT algorithm uses very limited knowledge, including Random move + Go rules + Not fill true eyes.
2. Prior knowledge. Knowledge comes from prior human experience which includes capturing, extension, patterns, etc. This knowledge is hand coded to Go programs. Examples include CrazyStone2006 and MoGo2006.
3. Prior knowledge + machine learning. Examples include CrazyStone2007 [27] and MoGo2007 [33].
4. Knowledge is automatically extracted from Go data, with or without little prior knowledge.

Some extensions are also used at the tree search part, for example:

- Adjusting the UCB1 function to affect the behavior of exploration and exploitation.
- Iterative widening.
- Progressive unpruning [19].
- Use the grand parent heuristic [53].
- RAVE (Rapid action value estimation) [33].

Figure 1.4 come from Don Dailey's Post on Computer Go Mail List and it shows that the UCT algorithm scales very well when computation time increases. No diminish return appears in the test.

Some basic parallel UCT algorithms are introduced to utilize large scale computing resource [17], but further investigation is needed.

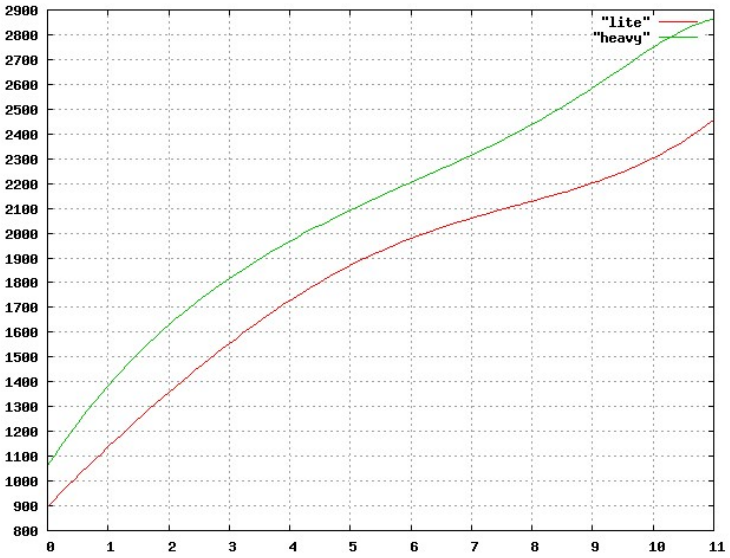


Figure 1.4: Scalability of UCT (X-axis: lg(*time*) Y-axis: Elo Ranking)

1.5 Research Problems

The research problems are listed below:

- How can the quality of the MC simulation be improved?
  - How can an efficient MC simulation be provided, especially after adding knowledge?

    The simulation should be fast in order to be applicable.
  - How does the inductive bias that comes from knowledge affect the effectiveness of the algorithm?

    Research shows that a strong policy does not guarantee a strong game player.
  - What knowledge is important to guide the semi-random simulation? Can this knowledge be extracted automatically from existing data or online search? If so, how can this knowledge be extracted with given prior knowledge?
  - How can prior knowledge and knowledge from machine automatical learning be combined?
- How can the tree search efficiency be improved?
  - How can knowledge be used to improve the efficiency of the tree search?
  - Do traditional game tree search heuristics also apply to the MC tree search? What can be used to speed up the tree search process?

CHAPTER 2: COMPUTER GO TACTIC PROBLEMS

Go tactic problems such as capturing [24, 57], connection [22], and Life&Death [23, 55] problems are a set of important problems in Go.

When facing a Go board status, human players usually would perform a lot of tactical calculations as well as global considerations. It is believed that global search and local tactical search should be combined to achieve a strong computer Go program, especially at the 19 x 19 board.

Tactic problems are also very widely used for teaching and practicing Go. A lot of Go books are dedicated to provide tactic problems for different Go levels (from the low kyu level to the high dan level).

It's generally believed that solving tactic problems is equally hard as the Go game itself. An ordinary tactic problems usually has tens of candidate moves and needs tens of moves to reach the goal, thus the search tree usually is huge (more than 10^{10} nodes).

Solving tactic problems usually requires a deep and narrow search. Through the use of traditional game tree search techniques, some progress has been made to produce strong tactical problem solvers [24, 57]. But traditional game tree search techniques fails in solving some difficult problems (such as some difficult capturing problems, unbounded Life&Death problems [55]).

2.1 Capturing Problems

The goal of the Capturing search is to find out if a block of stones can be captured or not and the best move sequence to accomplish such a goal. Capturing calculation is basic and essential to computer Go programs. Capturing algorithm can be used in post-game capturing problem solving or embedded in actual game engines to conduct real time capturing calculations.

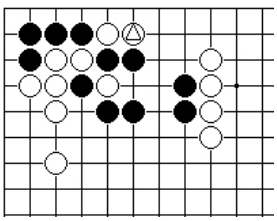


Figure 2.1: One Go Capturing Problem

In real time calculations the time is rather limited and the efficiency of the capturing algorithm

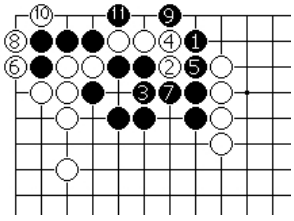


Figure 2.2: The Capturing Sequence

is very important.

Capturing problems provide a very good test bed for game tree search algorithms. Some researchers have proposed general purpose algorithms that uses capturing problems as examples. These algorithms is rather slow for Go capturing problems. To get better and more practical results, further investigations are needed.

In the authors' previous work [57] , highly selective based heuristic search was used in solving capturing problems in Go and desirable results were achieved.

How do human expert players solve capturing problems? When an advanced human Go player faces a capturing problem, he usually can find the weakness of the target block quickly and can perform a narrow and deep reading. Most of the time he can find the weaknesses in the shape of blocks through the use of the knowledge based on his prior experience.

In this dissertation, Kano book series 3 and 4 are used as the capturing test problem sets. Kano book 3 is designed for intermediate level Go players and Kano book 4 is designed for expert level Go players. Kano book 3 contains 61 capturing problems and Kano book 4 contains 51 capturing problems.

2.2 Life&Death Problems

The goal of Life&Death is to make at least two eyes (or produce seki) for defense side, while offence side is to destroy the eye shape to kill blocks inside a given region.

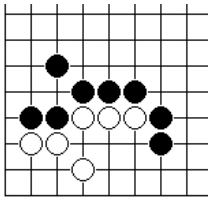


Figure 2.3: One Go Life&Death Problem

Through the use of extended $\alpha\beta$ search, Thomas Wolf's GoTools [55] provides a strong (strong amateur level) Life&Death solver for bounded Life&Death problems. GoTools contains powerful rules for static Life&Death recognition, elaborate move ordering heuristics and a refined tree search-

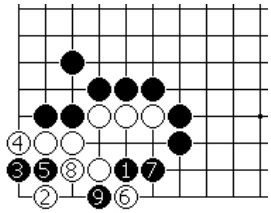


Figure 2.4: The Life&Death Move Sequence

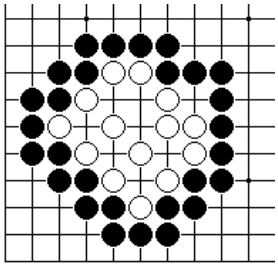


Figure 2.5: A Bounded Life&Death Test Problem Comes from GoTools

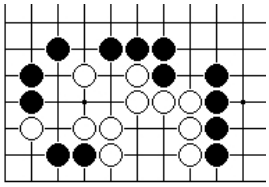


Figure 2.6: An Unbounded Life&Death Problem

ing algorithm. In Akihiro's dissertation [37], he showed that proof number search is also very efficient for bounded Life&Death problems.

For unbounded Life&Death problems, traditional tree search programs fail to provide a strong solver. The performance of normal programs on Life&Death problems is not above their overall level of skill.

This dissertation is primarily focused on unbounded Life&Death problems. Bounded Life&Death problems are also tested for comparison purpose.

In this dissertation, Life&Death Encyclopedia(chinese version) by Japanese Go master Segoe Kensaku (SK) is used as Life&Death problem set(for both training and test purpose). 60 randomly chosen problems are used as test problems. 100 bounded Life&Death problems from GoTools (thanks to Thomas Wolf for providing his Life&Death problem library to the author) are also used as test problems.

CHAPTER 3: CAPTURING PROBLEMS AND GAME TREE SEARCH ALGORITHMS

Game tree search algorithm is a core component in building a strong game-playing program. Alpha-Beta search is the most widely used traditional game tree search algorithm. Proof-number search and its variants have become more and more popular and successful in recent years.

Some issues in designing a game search engine include:

- What are the characteristics of each search algorithm?
- For a specific problem, which search algorithm performs better?

In this chapter, I explore different classical game tree search algorithms in solving Go capturing problems. First I design and implement $\alpha\beta$ search with its extensions and pn-search with its variants in solving capturing problems. Then the results of these algorithms are compared and insights with the features of each algorithm are gained. Finally I analyze some data produced in the searching process to find some patterns that can be used to predict the search outcome.

The performance of the capturing algorithm is favorable as well as practical. The algorithm can be used to solve post game capturing problems or to perform real time capturing calculations in computer Go tournament matches. Our results show that pn-search is better than $\alpha\beta$ search in solving capturing problems. And I can find some patterns in the searching process that can help us to predict the search outcome when there is inadequate time to solve the problem. The work also provides a framework that can be used in solving other Go sub-problems and other games.

3.1 Capturing Problem Formalization

In the following sections, I shall introduce the strategy of candidate move generation and move ordering, terminal status test, and internal and leaf node evaluation. Then I shall use the same methods of move generation, node evaluation, and terminal test with different search algorithms.

3.1.1 Candidate Move Generation

The candidate moves are mostly adjacent to the target block or the last move. I define related points of a block to be the first liberties and second liberties of the block and the first liberties of its adjacent opponent block with no more than 3 liberties. I use a simple evaluation function: putting a stone on the candidate move point and then calculate:

$$PointEvaluation = \#Block'sFirstliberties * 4 + \#Block'sSecondliberties \tag{3.1}$$

Then I choose those points with higher evaluation values from the related points of the block.

I consider 5 types of blocks:

- The target block
- The blocks in the target block's crucial chain
- The block of the last move
- The adjacent opponent's blocks
- The adjacent blocks of the last move

Each type of blocks can produce some related points with a minimum and a maximum number threshold. And basically I select the candidate moves from those related points of the 5 types of blocks.

I use two types of goals to help select candidate moves: capturing goal and escape goal. If the color of the current move is the same as the target block I set the goal as escape, otherwise I set the goal as capturing. With different goals the strategies of selecting candidate moves are different. For example, if the goal is escape, it can not make a move so that the target block can be ladder captured.

I also use some other basic conditions to prune the candidate moves. The move should not fill a solid eye of a block with the same color. The number of total candidate moves is limited to 15. This number is big enough for most situations.

Figures 3.1 and 3.2 show the target block and the candidate moves of a Capturing problem. The white stone marked with the triangle is the target, and it can be captured if black is the next to move. Those stones marked with the number are the candidate moves to capture the marked stone, and location 2 is the right move to capture the target.

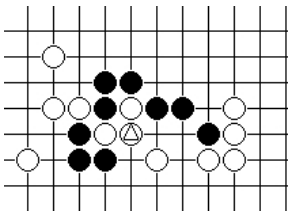


Figure 3.1: Target Block

3.1.2 Node Evaluation

I use the evaluation of the crucial chain of the target block as the internal and terminal node evaluation:

$$NodeEvaluation = \#Crucialchain'sFirstliberties * 4 + \#Crucialchain'sSecondliberties \quad (3.2)$$

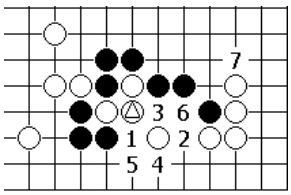


Figure 3.2: Candidate Moves

3.1.3 Terminal Test

For the terminal status test, I set an upper bound for the node evaluation. If the node evaluation exceeds this upper bound, the target block will be considered as capturing failed. If the target block has already been captured or can be ladder captured it will be considered as captured.

3.2 Game Tree Search Algorithms

For two players, zero sum, perfect information board games, such as Go, Chess, and Checker, Game tree search algorithm is essential to a strong game playing program. The goal of a game tree search is to find the best value and the best move sequence of a minimax game tree or an AND/OR game tree. Usually the size of the game tree is too big to be solved by currently available computers. The strategies for reducing the size of the game tree include reducing branching factors by reducing the candidate moves, using evaluation when search reaches at some certain depths and using search route selection strategies. Candidate move selection and the use of evaluation largely depend on the domain knowledge. Search route selection strategy is sometimes independent of the domain knowledge. Depth first search and best first search are the most popular search route selection strategies.

$\alpha\beta$ search is a depth first search algorithm and it is the most widely used game tree search algorithm for modern game engines [35]. Move ordering of candidate moves is essential to the efficiency of $\alpha\beta$ search. There are many extensions to $\alpha\beta$ search including iterative deepening, transposition table, history heuristic and others, which mostly contribute to getting a better move order. Through the use of $\alpha\beta$ iterative deepening search, one can get the search result of a certain depth and use it as an approximate solution when the search time limit is reached. The disadvantage of $\alpha\beta$ iterative deepening search is that it is difficult to reach deep depths.

Pn-search [1] is a best first search algorithm. It is very powerful in solving the non-uniform game tree. Pn-search algorithm needs to keep the whole search tree that it expands, thus causes heavy memory usage. In recent years, several depth first variants of pn-search, such as pn* [48], df-pn and df-pn+ [45] have emerged. Through the use of some thresholds of the proof number and the

disproof number, these algorithms would adopt a depth first manner to search the tree but the tree they expanded is similar or identical to the same as pn-search. These depth first variants can save memory yet with similar performances as pn-search.

Until now $\alpha\beta$ search with its extensions is still the most widely used game tree search algorithms. Pn-search and its variants become more and more popular [37].

3.3 $\alpha\beta$ Search

The idea of $\alpha\beta$ search is using the alpha value - the worst possible score for the max node, and the beta value - the worst possible score for the min node. Through the use of these values a lot of branches can be pruned [39].

With the best ordering, the number of nodes that $\alpha\beta$ search needs visit to solve a problem is only approximate $w^{\frac{d}{2}}$, where w is the average number of candidate moves and d is the search depth. In the worst case, it would search the whole game tree to find the solution.

I use $\alpha\beta$ search with iterative deepening and the transposition table. I increase the depth by 2 at each iteration. I use Zobrist [59] hash and liner probing for the transposition table.

Several criteria can be used to make empirical comparisons among different algorithms - CPU time, terminal nodes visited and the total nodes visited [13]. I use the total nodes expanded and the CPU time as the metrics of the performances of the algorithms.

I use the capturing problems of Kano's book 3 to test our algorithms. Search time is limited to 200 seconds for each problem. Table 3.1 contains the test results of $\alpha\beta$ search including those with and without the transposition table.

With the transposition table the performance is improved with less time consumed and fewer nodes expanded. Storing the value of tree nodes on the table can avoid recalculating the same position. And in iterative deepening the transposition table would also help improving move ordering.

The quality of the move order is essential to the efficiency of $\alpha\beta$ search. It is impossible to find a unified evaluation function to give the perfect move order for every problem. It can be helpful to use pattern database in the future to provide a better move ordering.

Investigating problems related to why $\alpha\beta$ search failed, I find three reasons.

- It is very difficult for $\alpha\beta$ search to reach very deep depths under the time constraint. For those problems that requires deep searches, $\alpha\beta$ search normally ran out of time and failed.
- The move generation did not provide all possible moves - the winning key move was missing. Or the best move was ordered with rather low priority and the search was made inefficient.
- Solving the problem requires other kinds of knowledge such as connection, Life&Death ect.

3.4 Proof Number Search

Pn-search is a best first search algorithm and it is very powerful in solving non-uniform game trees. The idea of pn-search is that it always expands the node that is possible to prove or disprove the goal of the AND/OR game tree with the least efforts [1]. Pn-search divides nodes into two types: AND nodes and OR nodes. At every AND node it tries to disprove the node and at every OR node it tries to prove the node. It uses two numbers associated with every node to achieve that goal: the proof number - the minimum number of leaf nodes to be expanded to prove the node and the disproof number - the minimum number of leaf nodes to be expanded to disprove the node. For a search tree I can view it as an AND/OR tree or a minimax tree, the AND node is the same as the MIN node and the OR node is the same as the MAX node.

Pn-search algorithm needs to keep the whole game tree that it expends in memory thus it uses a lot of memory. In recent years, there have emerged several depth first variants of pn-search such as PN*, df-pn and df-pn+. They use a hash table to store intermediate results. Memory requirement can be reduced to the size of the hash table.

Pn-search and df-pn set the proof number and the disproof number to 1 if the node is a non-terminal node. Df-pn+ uses evaluation function h and cost to calculate the proof number and the disproof number. Similarly I use evaluation value with pn-search to calculate the proof number and the disproof number at frontier nodes and I call the algorithm pn+.

For pn+,

$$pn = E^2, dn = (\frac{1}{E} \times 25) \tag{3.3}$$

For df-pn+,

$$cost = 0, h.pn = E^2, h.dn = (\frac{1}{E} \times 25) \tag{3.4}$$

Where E is the node evaluation, pn is the proof number and dn is the disproof number.

I use pn, pn+, df-pn, and df-pn+ to solve capturing problems. The candidate move generation and terminal status test are the same as in $\alpha\beta$ search. For these algorithms I use total nodes of the search tree and search time as the search control. For df-pn and df-pn+, some candidate moves are generated but not expanded, thus the number of nodes counted may appear fewer than pn-search and pn+.

I test the algorithms on the capturing problems of Kano's book 3. Search time is limited to 200

seconds for each problem. Table 3.1 contains the test results.

Pn-search and its variants solves more problems with less time; fewer nodes are expanded; and it goes a little deeper than $\alpha\beta$ search. The problems that can not be solved by pn-search are a subset of the problems that can not be solved by $\alpha\beta$ search.

Another advantage of pn-search and its variants is that they do not require move orders of those candidate moves. This would save some time.

Investigation of those unsolved problems I find that the failure was mainly caused by the following factors:

- The move generation does not provide all possible moves.
- Solving the problem requires other kinds of knowledge such as connection, Life&Death.

3.5 Result Analysis

3.5.4 Algorithms Comparison

Table 3.1: Results of Different Algorithms in Solving Capturing Problems in Kano Book 3

	Solved	Unsolved	Time(ms)	Nodes Explored	Search Depth
$\alpha\beta$ without transposition table	52	9	5502	28746	5.6
$\alpha\beta$ with transposition table	53	8	2785	13187	5.8
Proof number search(pn)	55	6	2148	18607	7.2
pn+ (with heuristic)	55	6	1303	10931	7.0
df-pn	55	6	2259	5792	7.4
df-pn+	55	6	1648	2175	6.7

In table 3.1, the time represents the average time used for solved problems in milliseconds; the nodes are the average number of nodes expanded for solved problems; and the depth is the average search depth for solved problems.

For most problems, pn-search and its variants performed better than $\alpha\beta$ search algorithms. Most problems can be solved within 1 second. The unsolved problems of pn-search, pn+, df-pn and df-pn+ are the same 6 problems and they are a subset of the problems that are unsolved by $\alpha\beta$ search. Among solved problems by all algorithms, $\alpha\beta$ search performed better than pn-search on 13 problems; pn-search performed better than $\alpha\beta$ search on 29 problems. Their performances were equal on the rest 13 problems. Table 3.1 shows that in terms of time cost, pn+ is slightly better than pn-search and df-pn+ is slightly better than df-pn. When I combine the heuristic evaluation with the proof number and the disproof number, the algorithms expanded fewer nodes to prove or disprove the root.

Figures 3.3, 3.4 and 3.5 show the performance comparison of the algorithms with time restriction to each problem. It can be find out that if time is restricted to 100 milliseconds, pn+ has the best

performance. Figures 3.3 and 3.4 show that the performance of df-pn is slightly better than df-pn+ when the time is restricted within 1 second.

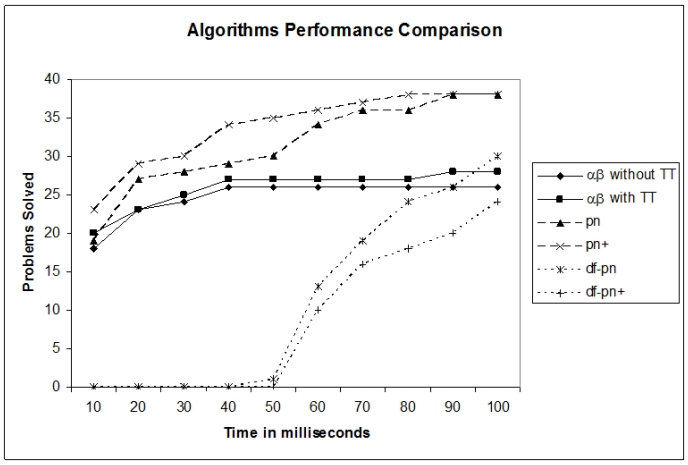


Figure 3.3: Problems Solved within 100 Milliseconds

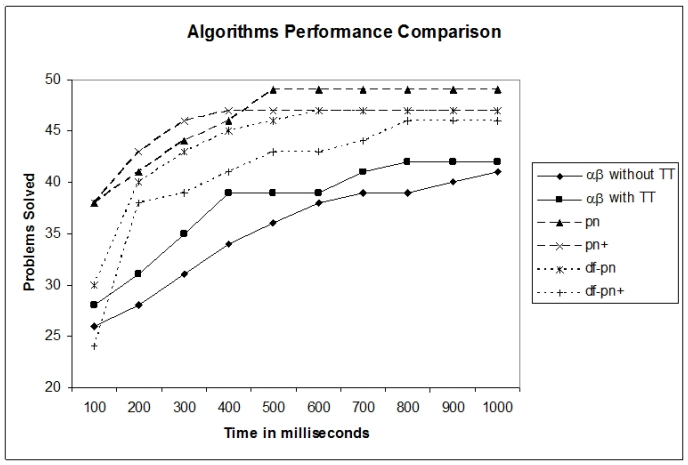


Figure 3.4: Problems Solved within 1 Second

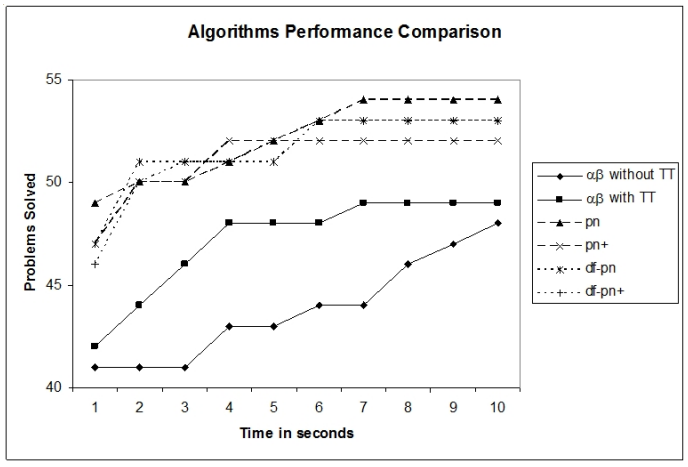


Figure 3.5: Problems Solved within 10 Seconds

From figures 3.4 and 3.5, it can be find out that pn-search and its variants have similar perfor-

mances and they are much better than $\alpha\beta$ search.

3.5.5 Analysis of the Capturing Problems

The search results of the 61 capturing problems in Kano book 3 are: 49 problems are successfully solved as captured, 6 problems are successfully solved as escaped and 6 problems are unsolved.

For 13 problems, $\alpha\beta$ search uses less time than pn-search. Figure 3.6 is an example that $\alpha\beta$ search uses 640 ms and expands 3139 nodes and pn-search uses 3384 ms and expands 16791 nodes.

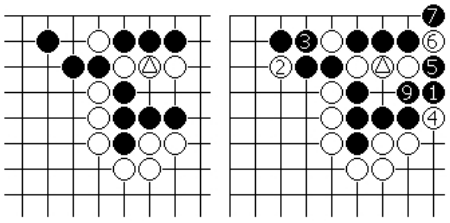


Figure 3.6: A Problem $\alpha\beta$ Search Solved Faster than pn-search

For 29 problems pn-search uses less time than $\alpha\beta$ search. Figure 3.7 is an example that $\alpha\beta$ search used 33s and expanded 133384 nodes and pn-search used 260 ms and expanded 2094 nodes. The searching process of pn-search just like the way of human thinking: perform a narrow and deep search.

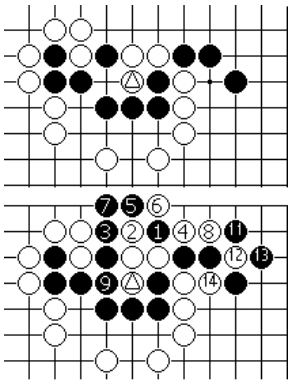


Figure 3.7: A Problem pn-search Solved Faster than $\alpha\beta$ Search

3.5.6 Data Analysis of the $\alpha\beta$ Search Results

I record the best value $\alpha\beta$ search gets at each depth when it performs the iterative deepening. I find that there exist some similar patterns of the variation of the best value with different types of problems. Figures 3.6, 3.7 and 3.8 are examples of the variation of the best value with depth deepening.

The best value is the evaluation of the target crucial chain's first liberty and second liberty after several plies. For the capture side the smaller the better while for the escape side the bigger the better. When the target is captured the value becomes 0.

For those successfully captured problems, the best value can become a little bit higher before falling down to 0. For those successfully solved as escaped problems, the best value usually becomes higher and higher. For those unsolved problems, the best value stays in a confined range.

Figure 3.8 is the best value of $\alpha\beta$ search with search deepening of an example of captured problems. Figure 3.9 is the best value of $\alpha\beta$ search with search deepening of an example of un-captured problems. Figure 3.10 is the best value of $\alpha\beta$ search with search deepening of an example of unsolved problems.

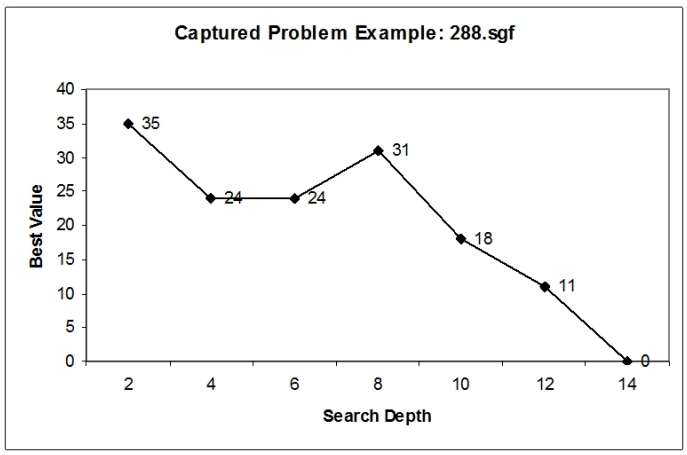


Figure 3.8: The Best Value Example 1

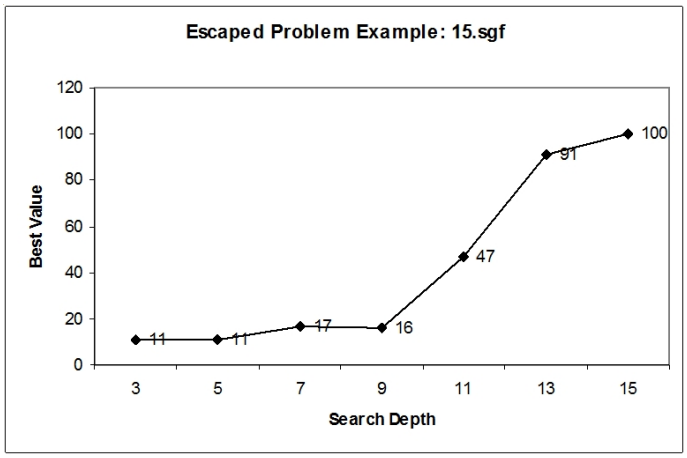


Figure 3.9: The Best Value Example 2

Further investigation is needed to find out that if this pattern can help to predict the search result when the result is unknown. For real game engines the time spent for a specific search usually is very limited and inadequate for a thorough search. It will be very helpful if this kind of prediction can become trustworthy.

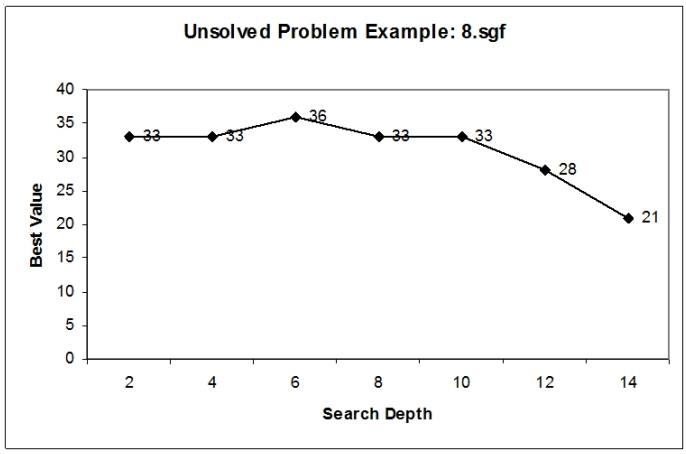


Figure 3.10: The Best Value Example 3

3.5.7 Data Analysis of the Proof Number Search Results

I record the proof number and the disproof number of the root node while pn-search is processing search. I also find that there exist some similar patterns of the variation of the proof number and the disproof number of the root node after more nodes expanded. Figures 3.11 and 3.12 are examples of the variation of the proof number and the disproof number of the root node with search proceeding and more nodes being expanded.

Proof number is the minimum number of leaf nodes to be expanded to prove the node and the disproof number is the minimum number of the leaf nodes to be expanded to disprove the node. If the proof number becomes 0 the target is captured. If the disproof number becomes 0 the target escapes. For those target successfully captured problems, the proof number usually increases until it reaches to the peak and then decreases until the problem is solved; the disproof number usually continues increases. For those successfully solved problems with the target escapes, the proof number usually continues to increase and the disproof number usually increases to a certain peak and then decreases until the problem is solved. For problems that is unsolved due to lack of time, the proof number and the disproof number usually both continue to increase.

Figure 3.11 is the variation of the proof number and the disproof number during pn-search of an example of captured problems. Figure 3.12 is the variation of the proof number and the disproof number during pn-search of an example of un-captured problems.

Further investigation is needed to find out that if this pattern can help to predict the search result when the result is still unknown.

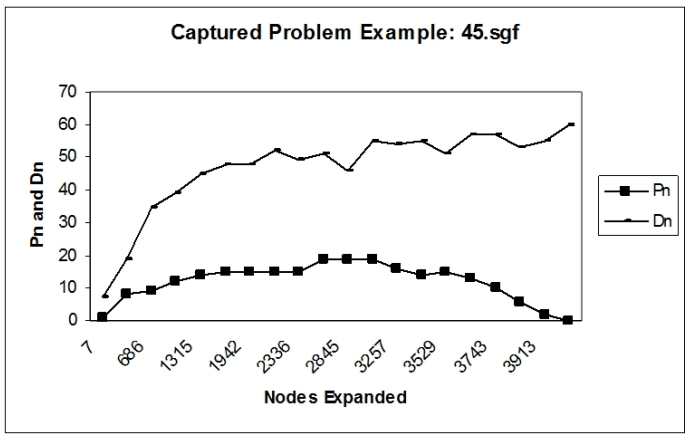


Figure 3.11: The Variation of the Proof Number and the Disproof Number Example 1

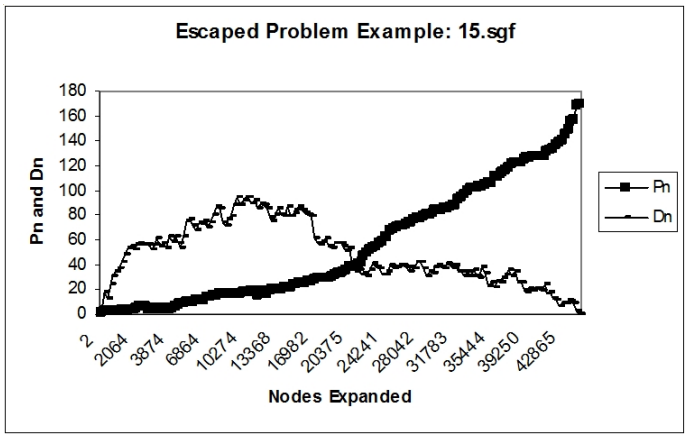


Figure 3.12: The Variation of the Proof Number and the Disproof Number Example 2

3.6 Remarks

The performances of these capturing algorithms are compared favorably against other published results. Pn-search and its variants have better performances in solving capturing problems. They can solve most problems within 1 second and can be used by Go programs in real time matches.

The reasons for the unsolved problems fall into two basic categories: the quality of the domain evaluation and the need for other types of knowledge such as connection and Life&Death etc. To improve the performance, I need to improve domain evaluation and incorporate knowledge from solvers of other Go tactic problems.

CHAPTER 4: MONTE-CARLO TACTIC TREE SEARCH FRAMEWORK

4.1 Monte-Carlo Tactic Tree Search Framework

In this section, the framework of heuristic based Monte-Carlo tactic search will be addressed.

Standard Monte-Carlo tree search contains two parts: the tree search part and the semi-random simulation part. Heuristic based Monte-Carlo tree search extends both parts:

- A heuristic based UCT+ tree search is proposed to extend standard UCT tree search.
- A heuristic based semi-random simulation policy is extended by knowledge.

Comparing to global Monte-Carlo tree search, Monte-Carlo tactic tree search has two additional extensions: local search region and tactic specific terminal status determination.

For global Monte-Carlo tree search, usually all legal moves (except those that would fill their own eyes) are added as candidate moves at tree search part and thus become possible moves at the semi-random simulation part. Different from global Monte-Carlo tree search, heuristic based Monte-Carlo tactic search usually use local search region at both tree search part and semi-random simulation part. The scale of the local search region depends on the specifics of the problems which usually is the problem affected board region. For global Monte-Carlo tree search, semi-random simulation result (score) is collected when no legal moves (except those that would fill their own eyes) are available. For the Monte-Carlo tactic search, the semi-random simulation terminal status determination would also depend on the specifics of the problems.

Pseudo code of Monte-Carlo tactic tree search.

Pseudo Code 4.2: Monte-Carlo Tactic Tree Search

```
while (nuSimulation > 0)
{
    nuSimulation--;
    //1. Start from the root node
    UCTPlusNode * pNode = m_pRootNode;

    //2. Select the most promising node by UCT+ function
```

```
    while(pNode->son != NULL)
    {
        pNode = SelectByUCB1Plus(pNode);
    }
    //3. Create children for the currentleaf node if it is not the first visit
    //using local search region
    if (pNode->nuVisit > 0)
    {
        if(CreateChildrenNodesLocal(pNode))
            pNode = pNode->son;
    }
    //4. Calculate value by MC using extended semi-random simulation
    // Using tactic specific terminal status determination
    int value = GetValueByMCTactic(pNode);
    pNode->value += value;
    pNode->nuVisit++;
    //5. Update ancestor nodes' value
    UpdateValueHeavy(pNode, value);
}
```

Table 4.1: Comparison of Standard Global MCTS and Proposed Tactic MCTS Framework

	Standard global MCTS	Tactic MCTS
Semi-random simulation policy	Standard semi-random simulation	Standard + tactic specific heuristic based semi-random simulation
Tree search	UCT	A heuristic based UCT+ tree search
Candidate move	All legal moves (except those that would fill their own eyes)	Local search region
Simulation terminal status determination	Play until no legal (except those that would fill their own eyes) moves are available	Tactic problems specific terminal status determination

4.2 Heuristic Based UCT+ Tree Search

The standard UCB1 function is extended to considering the heuristic (domain knowledge dependent) and the history (domain knowledge independent) value part. The heuristic value part is used to enable the heuristic value to affect the decision of the next child node to explore, it represents

adding domain knowledge dependent bias to the UCB1 function. The history value part is used to enable the history value to affect the decision of the next child node to explore, it represents adding domain independent bias to the UCB1 function.

$$ChildNodeToExplore = argmax(UCB1Plus) = argmax(\mu + f_{UCB1} + f_{his} + f_{heu}) \tag{4.1}$$

$$\mu = \frac{n_{nuWin}}{n_{nuVisit}} \tag{4.2}$$

$$f_{UCB1} = \sqrt{\frac{ln(n_{parentnuVisit})}{ExEValue \times n_{nuVisit}}} \tag{4.3}$$

$$f_{his} = S(n_{nuVisit} - C_{his}) \times V_{his} \tag{4.4}$$

Check chapter 7 for more details about f_{his}.

$$f_{heu} = S(n_{nuVisit}) \times V_{heu} \tag{4.5}$$

$$S(x) = 2 \times (1 - \frac{1}{1 + e^{\frac{-\sqrt{x}}{C_S}}}) \tag{4.6}$$

S(x) is a transformed sigmoid function, its domain is $[0, +\infty)$, its range is (0, 1]. It is used to reduce the effect of the history part as well as the heuristic part as the number of simulations increasing, because the bias introduced by the heuristic value and the history value should become 0 when the number of simulations increases to ∞, just as f_{UCB1}.

$n_{nuVisit}$ is the number of simulations that go through this node. n_{nuWin} is the number of winning simulations for this node among all above visited simulations. $n_{parentnuVist}$ is the number of simulations that go through the parent node. V_{his} is the history value of the node; it will be discussed in chapter 7 in details. V_{heu} is the heuristic value of the node; it will be discussed in chapter 5 in details. C_{his} is the coefficient for the history value, it represents when to start to use History Heuristic (the bigger, the latter to start to use the history value). C_S is the coefficient for the transformed sigmoid function S, it represent the decaying rate of this transformed sigmoid function (the bigger, the slower the change from 1 to 0).

Pseudo code for node selection:

Pseudo Code 4.3: Node Selection

```
UCTPlusNode * SelectByUCB1Plus (UCTPlusNode * pNode)
{
        return argmax(UCB1Plus(pNode));
}
```

4.3 Semi-random Simulation Policy

Probability weight is used to guide the semi random simulation policy. The possibility of being chosen is dependent on probability weight.

For a given board status:

l_1, l_2, ... l_n represent the available legal positions (legal move) inside a local search region.

w_1, w_2, ... w_n represent the probability weight of those legal positions.

$$w_{total} = \sum w_i \tag{4.7}$$

$$p_i = \frac{w_i}{w_{total}} \tag{4.8}$$

p_i is the possibility of l_i being chosen as the next move. $\sum p_i = 1$.

Pseudo code of semi-random simulation.

Pseudo Code 4.4: Semi-Random Simulation

```
int GetValueByMCTactic (UCTPlusNode * pNode)
{
        if(IsLeafNode(pNode))
                return pNode->leafValue;
        while (NotInTerminalStatus())
        {
                //get a random bp
                int bp = GenRandomMoveByWeight();
                ExecuteMove(bp);
    }
        Return EvaluateBoard();
}
```

In chapter 5, semi-random simulation policy extended by Go knowledge is systematically investigated.

- Solid Eye
- Default Probability Weight
- Creating an Eye instead of Filling an Eye
- Avoiding Self-Atari
- Avoiding Single Stone Self-Atari
- Ladder
- Proximity

In chapter 6, the semi-random simulation is extended by learning patterns. 3*3 and 5*5 exact pattern match and ANN are used as learning pattern methods.

4.4 Local Search Region

Global MC-UCT search usually considers all legal moves as candidate moves except those that fill their own solid eyes. For tactic problems, considering all legal moves obviously is not efficient. Candidate moves come from legal moves inside the local search region except those that fill their own solid eyes. The search region may dynamically grow as explained below.

4.4.1 Local Search Region for Capturing Problems

- Initial candidate move region
  - The target block
    - First liberties, second liberties and third liberties
  - Adjacent opponent of the target block
    - First liberties and second liberties
- Floating Search Region
  - After a move (for both UCT tree and Monte Carlo simulation) this move's adjacent points are added to the candidate move list

Figure 4.1 is a local search region for a Capturing problem. The block marked with the triangle is the target to capture and the positions marked with the X are the initial candidate moves inside local search region.

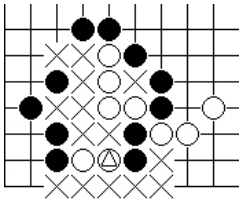


Figure 4.1: Local Search Region for a Capturing Problem

4.4.2 Local Search Region for Life&Death Problems

- Initial candidate move region
  - The target block
    - First liberties, second liberties and third liberties
  - Adjacent opponent blocks of the target block
    - First liberties and second liberties, if adjacent opponent blocks are strong (measure by heuristic), then no opponent blocks' second liberty or even first liberty will be added
- Floating Search Region
  - After a move (for both UCT tree and Monte Carlo simulation) this move's adjacent points are added to the candidate move list

Figure 4.2 is a local search region for a Life&Death problem. Blocks marked with the triangle are the target to kill, and the positions marked with the X are the initial candidate moves inside the local search region.

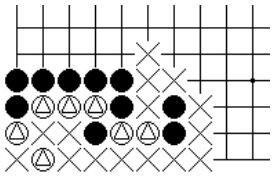


Figure 4.2: Local Search Region for a Life&Death Problem

See section 7.5 for test results of the floating search region dynamically create new candidate moves.

4.4.3 Virtual Boundary

Virtual boundary is defined as an enlarged bounding box of blocks of Life&Death problems. A bounding box is the smallest rectangle region that contains the problem blocks; virtual boundary is

a rectangle region that enlarges this bounding box by two extra lines at each direction. The floating search region is constraint inside the virtual boundary. Figure 4.3 shows an example of virtual boundary. The positions marked with X are the virtual boundary and the possible candidate moves are inside (boundary not included) the virtual boundary.

For Life&Death problems, the goal for defense side is usually to make two true eyes locally instead of escape to outside regions. Thus the virtual boundary is introduced to reduce candidate moves (both for the tree search part and the MC simulation part.

Different from the bounded region of the bounded Life&Death problems, the virtual boundary is still a very big region.

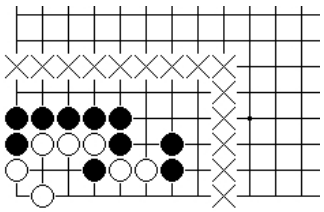


Figure 4.3: Virtual Boundary for Life&Death Problems

See section 7.5 for test results of the virtual boundary.

4.5 The Determination of the MC Simulation Terminal Status and Terminal Node Tag

When to stop a semi-random simulation and to collect the simulation result depends on the type of problems.

4.5.4 For Capturing Problems

For MC capturing simulations, two stop criteria are used.

1. If the target block is removed from the board, it will return with capturing succeeded.
2. If the number of liberty of the target block exceeds a given threshold or the number of moves from the beginning to the current board exceeds a given threshold, it will return with escaping succeeded.

4.5.5 For Life&Death Problems

For the MC Life&Death simulations, two stop criteria are used.

1. If all target blocks are removed from the board, it will return with target dead.
2. If the number of moves from the beginning to the current board exceeds a given threshold, it will return with target alive.

4.5.6 Terminal Node Tag

The global MC-UCT search usually considers a UCT tree node to be a terminal node if no more legal moves exist except those that fill their own eyes. For the tactic UCT tree, it is important to put a terminal node tag for those nodes with an already known definite value and to store the value in the node for later to use. There is no need to do an MC evaluation for those nodes. To decide whether a node is a terminal node or not, some heuristics such as ladder status and the number of liberties can be used. In our implementation, at the time the node is created, some heuristics are used to determine whether this node is a terminal node or not. If it is a terminal node the value will be stored and a terminal node tag will be set. There will be no more expansion from this node. When the UCT tree search visits this node at a future time, the value will be returned directly.

4.6 Exploitation and Exploration Coefficient

When computing f_{UCB1} in equation 4.3, an ExEValue coefficient (exploitation and exploration value) is needed. A series of tests to determine the best ExEValue constant were performed (For combined Kano book 3 and Kano book 4 Capturing problems, the search extension is the ladder setting with search early termination(see Section 4.7) used; the maximum simulation 500k). It shows that $ExEValue = 10$ is the best choice for both correctness and efficiency.

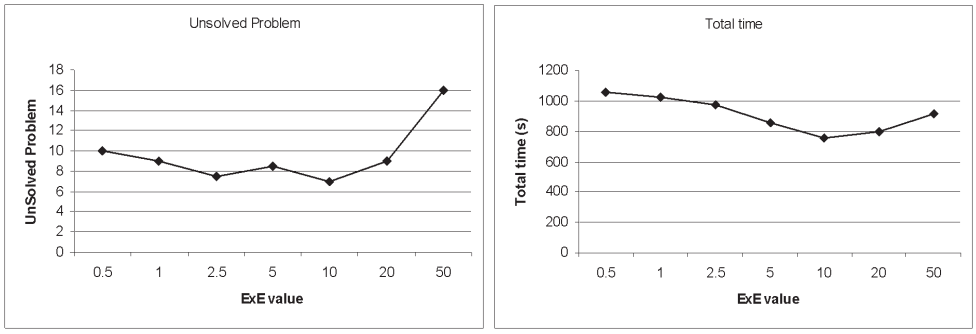


Figure 4.4: Correctness and Efficiency with Different ExEValue

4.7 Search Result Confidence

Monte-Carlo tree search is a statistical based algorithm. Unlike traditional game tree search algorithms such as $\alpha\beta$ search and PNS search that can solve the problem by outputting the optimal path of the search tree, Monte-Carlo tree search would output the children node with statistical values that including the root node winning rate and the child nodes winning rate. The confidence of the search result relies on these statistical values. Three statistical values are important to the confidence: the winning rate of the root node ($root_{winrate}$), the winning rate of the best child node ($bestchild_{winrate}$) and the winning rate difference between the best child node and the second to the

best child node ($bestchild_{winrate} - secondtobestchild_{winrate}$). Practically I can set a threshold for these statistical values, if all the values exceed the threshold the search will stop earlier.

The following thresholds are used at 4.6. $ConfidenceThreshold_{rootwinrate} = 0.5$, 50% root winning rate means that the problem is favorable to the root node. $ConfidenceThreshold_{bestchildwinrate} = 0.75$, 75% winning rate means the best child performs pretty well. $ConfidenceThreshold_{besttosecond} = 0.05$, 5% difference is enough to show that the best child is far better than the second to the best child. This threshold should be disabled in circumstances that multiple nodes are possible equally good.

To avoid adding more noise to the test results, the search result confidence is not used for the tests in chapter 5, 6 and 7.

CHAPTER 5: KNOWLEDGE BASED HEURISTICS

In this chapter, Go specific knowledge will be used to extend the MC semi-random simulation policy and Heuristics from Go knowledge will be added to improve the MC tree search.

5.1 Semi-random Simulation Policy

Probability weight is used to guide the semi random simulation policy. The possibility of being chosen is dependent on the probability weight.

For a given board status:

$l_1, l_2, \ldots\ l_n$ represents the available legal positions (legal moves) inside a local search region.

$w_1, w_2, \ldots\ w_n$ represents the probability weight of those legal positions.

$$w_{total} = \sum w_i \tag{5.1}$$

$$p_i = \frac{w_i}{w_{total}} \tag{5.2}$$

p_i is the possibility of l_i being chosen as the next move. $\sum p_i = 1$.

A default probability weight w_i for a legal move l_i is 10.

Probability weight w_i for an illegal move l_i is 0.

5.1.1 Solid Eye

A solid eye or true eye is a liberty enclosed by the same color stones. A solid eye can not be occupied by the other side unless the whole block is dead.

Filling your own solid eye usually means suicide, thus it's meaningless to fill your own solid eyes.

Figure 5.1 shows an solid eye example. The points marked with the X are solid eyes, and they are also called true eyes.

Contrary to a solid eye is a false eye which can not serve as a true eye.

Figure 5.2 shows an false eye example. The point marked with the X is a false eye.

The solid eye rule is $w_i = 0$, if l_i is a solid eye position.

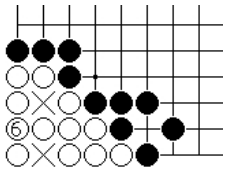


Figure 5.1: Solid Eye

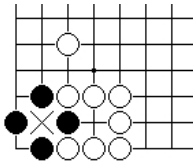


Figure 5.2: False Eye

5.1.2 Basic Stone Capturing and Escaping

Stone capturing and stone escaping from the atari (only one liberty left) status are the rudimental Go knowledge. Stone capturing means the elimination of a block of opponent stones and removal from the board. Stones escape from atari means the prevention of stones from being immediately captured by the opponent. Capturing and escaping are the direct responses when the atari status occurs.

Figure 5.3 is an example of capturing and escape. Black can capture white at the position marked with the X while white can escape at the position marked with the X.

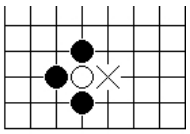


Figure 5.3: Capturing and Escape

One situation is the capture of stones to avoid atari.

Figure 5.4 is an example of capturing stones to avoid atari. Black stone marked with the triangle is in atari, and this atari will resolve by capturing the white stone marked with the square at X.

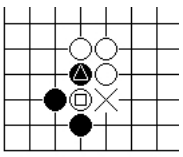


Figure 5.4: Capturing Stones to Avoid Atari

Capturing, escape and capturing to avoid atari are given higher probability weight.

Probability Weight for Capturing Problems

$w_i = 1000 \times nu_{Stone}$, if move at l_i would free stones from atari. nu_{Stone} is the number of stones that have escaped.

$w_i = 10000 \times nu_{Stone}$, if move at l_i would capture stones. nu_{Stone} is the number of stones that have been captured.

$w_i = 50000 \times nu_{Stone}$, if move at l_i would capture stones so as to avoid atari. nu_{Stone} is the number of stones that have been captured.

Probability Weight for Life&Death Problems

$w_i = 1000 \times nu_{Stone}$, if move at l_i would free stones from atari. nu_{Stone} is the number of stones that have escaped.

$w_i = 10000 \times nu_{Stone}$, if move at l_i would capture stones. nu_{Stone} is the number of stones that have been captured.

$w_i = 50000 \times nu_{Stone}$, if move at l_i would capture stones so as to avoid atari. nu_{Stone} is the number of stones that have been captured.

The setting of these weights (together with those inside below section) is largely decided by limited tests. It is very unlikely to perform a thorough test to choose the optimum values due to the computation feasibility. (perform one Go Tactic test is very time consuming already)

5.1.3 Tactic Initial Probability Weight

Different initial probability weights are assigned to positions inside local search region. Different types of problems have different initial probability weights.

Initial Probability Weight for Capturing Problems

$w_i = 400$, if l_i is a first liberty of the target blocks.

$w_i = 100$, if l_i is a second liberty of the target blocks.

$w_i = 25$, if l_i is a third liberty of the target blocks.

$w_i = 50$, if l_i is a first liberty of the adjacent opponent blocks of the target blocks.

$w_i = 12$, if l_i is a second liberty of the adjacent opponent blocks of the target blocks.

$w_i = 11$, if l_i is a new added points to this local search region.

$w_i = 25$, if l_i is a first liberty of an adjacent opponent block of a target block's adjacent opponents.

Initial Probability Weight for Life&Death Problems

$w_i = 400$, if l_i is a first liberty of the target blocks.

$w_i = 100$, if l_i is a second liberty of the target blocks.

$w_i = 25$, if l_i is a third liberty of the target blocks.

$w_i = 50$, if l_i is a first liberty of the adjacent opponent blocks of the target blocks.

$w_i = 12$, if l_i is a second liberty of the adjacent opponent blocks of the target blocks.

$w_i = 11$, if l_i is a new added points to this local search region.

$w_i = 25$, if l_i is a first liberty of an adjacent opponent block of a target block's adjacent opponents.

5.1.4 Creating an Eye instead of Filling an Eye

In figure 5.5 white would fill a potential eye by playing at the position marked with the square. In this situation adjacent legal position (the position marked with the X) will be chosen to replace the move that can fill potential eye.

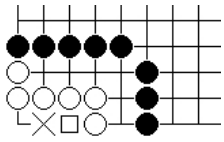


Figure 5.5: Creating an Eye instead of Filling an Eye

This rule would not affect probability weight, it would adjust a generated random move to an adjacent position.

5.1.5 Avoiding Self-Atari

In figure 5.6 white would put himself in self-atari situation by playing at the position marked X.

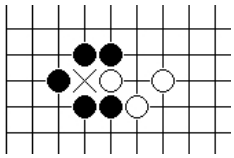


Figure 5.6: Avoiding Self-Atari

In situation that a self-atari random move is generated, the probability weight will be reduced once to $w_i = 5$ and a new random move will be regenerated. If the probability weight of the position has already been reduced, the position will be used.

5.1.6 Disabling Single Stone Self-Atari

Sometimes it is necessary to disable Avoiding Single Stone Self-Atari to accomplish some tesuji.

In figure 5.7, black will destroy white's eye by playing at the position marked with the X, and this would make one stone in self-atari situation happen.

This rule would disable above Avoiding Self-Atari rule if only one stone is in atari.

5.1.7 Using the Proximity Heuristic

In human Go games, usually consecutive moves are made at very close region, in another word, a move is largely affected by the last move. The positions near the last move should be given higher

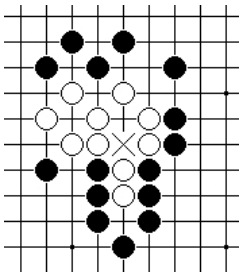


Figure 5.7: Single Stone Self-Atari

priority in the selection process, this is called proximity heuristic [29].

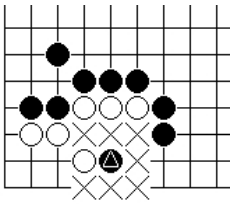


Figure 5.8: Proximity Value around the Last Move

A proximity table is used to enable proximity heuristics.

Below is the pseudo code for using the proximity table.

Pseudo Code 5.5: Using the Proximity Table

```
ExecuteRandomMove (last move)
{
    ... ...
    //calculate new proximity table
    if (UseProximity)
    {
        //fading the proximity map by half
        For (all positions)
        {
                proximityTable[black] /= 2;
                proximityTable[white] /= 2;
        }
        //update the last move's adjacent and diagonal positions' proximity
        //table by restore to maximum proximity value
        UpdateProximityTable(last move);
    ... ...
```

```
        }
}
```

The random move used to generate the next move is $w_i + Wproximity_i$. $Wproximity_i = 0$ at the beginning of the MC simulation. $Wproximity_i = w_i$ if l_i is the adjacent or the diagonal position of the last move.

5.1.8 Pseudo Ladder

Ladder knowledge is a kind of basic tesuji to capture two-liberty blocks. It's a natural extension from the capturing and escaping heuristics.

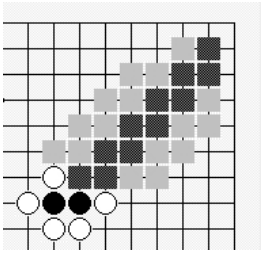


Figure 5.9: Ladder Capturing Sequence

Through the use of the pseudo ladder extension, higher weights are put on those liberties of the two-liberty blocks if those liberties are responsible for the ladder capture of those blocks. The pseudo ladder capture is used to calculate the ladder status. It can handle over 99% ladder problems and consume very little time compared with a regular ladder solver. This extension costs 30%-40% extra time to perform an MC simulation. Experimental results show that it significantly improves the performance of the Go capturing MC-UCT algorithm, thus the extra time is well spent.

5.2 Heuristic Tree Search

This section would demonstrate how f_{heu} is calculated and affect the tree search process.

$$ChildNodeToExplore = argmax(UCB1Plus) = argmax(\mu + f_{UCB1} + f_{his} + f_{heu}) \tag{5.3}$$

$$f_{heu} = S(n_{nuVisit}) \times V_{heu} \tag{5.4}$$

V_{heu} is simply defined by ladder (including capture and escpae) knowledge.

$V_{heu} = 1$, if this child node can ladder capture the opponent blocks.

$V_{heu} = 0$, if this child node can be ladder captured by the opponent blocks.

5.3 Test Results and Analysis

Kano book series 3 and 4 are used as the capturing test problem sets. Kano book 3 is designed for the intermediate level Go players, while Kano book 4 is designed for expert level Go players. Kano book 3 contains 61 capturing problems and Kano book 4 contains 51 capturing problems.

Life&Death Encyclopedia(chinese version) by Japanese Go master Segoe Kensaku (SK) is used as a Life&Death problem set. 60 randomly chosen problems are used as test problems. 100 bounded Life&Death problems from GoTools are also used as test problems.

Capturing and Life&Death tactic search are tested under different MC tactic search extension settings. For each MC tactic search extension setting, tests are performed with 3 different number of simulations (10k, 50k, 250k). For each test, The best move from the MC tactic search is collected and compared with the problem answer; if they are the same, the test will be marked with pass; if they are different, the test will be marked with fail. CPU time used to perform the test is listed also. As CPU time consumption is largely dependent on the extension implementation details such as the data structure and the performance tuning, thus it is listed for reference purpose. The number of passed problems is mainly used to analyze the extensions.

All tests were executed on a 3.0GHz 4-CPU PC. Most tests were executed twice.

5.3.9 No Heuristic Extensions

In the Capturing search, more than half of the problems passed the test and scales very well (passed problems, Kano book 3, 10k: 39 pass, 50k: 47 pass, 250k: 53 pass; Kano book 4, 10k: 28 pass, 50k: 35 pass, 250k: 38 pass).

In the Life&Death search, almost all problems failed the test.

The reason that the Capturing search performs much better than the Life&Death search is that the Solid Eye rule is very important for the Life&Death search. Without the Solid Eye rule, it is almost impossible to create 2 eyes.

5.3.10 Solid Eye

The Solid Eye rule is essential to both the Capturing and the Life&Death search.

In the Capturing search, more problems passed the test (Kano book 3, 10k: +7 pass, 50k: +9 pass, 250k: +5 pass; Kano book 4, 10k: +7 pass, 50k: +2 pass, 250k: +1 pass).

In the Life&Death search, many problems get solved (Average passed problems: GoTools problems, 10k: 90 pass, 50k: 93 pass, 250k: 98 pass; SK problems, 10k: 32 pass, 50k: 36 pass, 250k: 40 pass) while almost no problems get solved before the Solid Eye rule is applied.

Table 5.1: Extensions Setting: No Heuristic Extensions

Solid Eye	No
Capture&Escape Probability Weight	No
Tactic Initial Probability Weight	No
Creating an Eye instead of Filling an Eye	No
Avoiding Self-Atari	No
Avoiding Single Stone Self-Atari	No
Ladder	No
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

Table 5.2: Test Results: No Heuristic Extensions

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	22 (36%)	39 (64%)	200.6
50k	61	14 (23%)	47 (77%)	924.5
250k	61	8 (13%)	53 (87%)	3895
Kano Book 4				
10k	51	23 (45%)	28 (55%)	238.5
50k	51	16 (31%)	35 (69%)	950.9
250k	51	13 (25%)	38 (75%)	4247.9
GoTools Problems				
10k	100	91 (91%)	9 (9%)	52.9
	100	91 (91%)	9 (9%)	52.3
50k	100	88 (88%)	12 (12%)	263.1
	100	88 (88%)	12 (12%)	266.4
250k	100	90 (90%)	10 (10%)	1338.2
	100	90 (90%)	10 (10%)	1327.9
SK Problems				
10k	60	58 (97%)	2 (3%)	72.1
	60	58 (97%)	2 (3%)	71.7
50k	60	58 (97%)	2 (3%)	359
	60	58 (97%)	2 (3%)	358.8
250k	60	58 (97%)	2 (3%)	1791.3
	60	58 (97%)	2 (3%)	1800.6

5.3.11 Capture&Escape Probability Weight

Capture&Escape Probability Weight improves search performance for both the Capturing search and the Life&Death search (Kano book 3, 10k: +6 pass, 50k: +3 pass, 250k: +2 pass; Kano book 4, 10k: +0 pass, 50k: +0 pass, 250k: +3 pass; GoTools problems, 10k: +8 pass, 50k: +7 pass, 250k:

Table 5.3: Extensions Setting: Solid Eye

Solid Eye	Yes
Capture&Escape Probability Weight	No
Tactic Initial Probability Weight	No
Creating an Eye instead of Filling an Eye	No
Avoiding Self-Atari	No
Avoiding Single Stone Self-Atari	No
Ladder	No
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

Table 5.4: Test Results: Solid Eye

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	15 (25%)	46 (75%)	195.4
50k	61	5 (8%)	56 (92%)	770.8
250k	61	3 (5%)	58 (95%)	3474.5
Kano Book 4				
10k	51	16 (31%)	35 (69%)	358.7
50k	51	14 (27%)	37 (73%)	1504.5
250k	51	12 (24%)	39 (76%)	6282.1
GoTools Problems				
10k	100	10 (10%)	90 (90%)	71.3
50k	100	7 (7%)	93 (93%)	349.3
250k	100	2 (2%)	98 (98%)	1684.9
SK Problems				
10k	60	28 (47%)	32 (53%)	181.1
50k	60	24 (40%)	36 (60%)	896.4
250k	60	20 (33%)	40 (67%)	4217.5

+1.5 pass; SK problems, 10k: +5.5 pass, 50k: +9.5 pass, 250k: +9 pass).

5.3.12 Tactic Initial Probability Weight

The Tactic initial Probability Weight improves the search performance when there is a small number of simulations (such as 10k simulations), but the performance is lowered when there is a huge number of simulations (such as 250k). (Kano book 3, 10k: +0 pass, 50k: -2 pass, 250k: -1 pass; Kano book 4, 10k: +2 pass, 50k: +2 pass, 250k: -1 pass; GoTools problems, 10k: +0 pass, 50k: -1.5 pass, 250k: +0 pass; SK problems, 10k: +3.5 pass, 50k: -3.5 pass, 250k: -3 pass).

Table 5.5: Extensions Setting: Capture&Escape Probability Weight

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	No
Creating an Eye instead of Filling an Eye	No
Avoiding Self-Atari	No
Avoiding Single Stone Self-Atari	No
Ladder	No
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

Table 5.6: Test Results: Capture&Escape Probability Weight

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	9 (15%)	52 (85%)	309.3
50k	61	2 (3%)	59 (97%)	1443.5
250k	61	1 (2%)	60 (98%)	6392.3
Kano Book 4				
10k	51	16 (31%)	35 (69%)	553.3
50k	51	14 (27%)	37 (73%)	2211.9
250k	51	9 (18%)	42 (82%)	9759.6
GoTools Problems				
10k	100	2 (2%)	98 (98%)	118.2
	100	2 (2%)	98 (98%)	118.7
50k	100	0 (0%)	100 (100%)	571.5
	100	0 (0%)	100 (100%)	571.1
250k	100	1 (1%)	99 (99%)	2780.6
	100	0 (0%)	100 (100%)	2788.8
SK Problems				
10k	60	22 (37%)	38 (63%)	291.2
	60	23 (38%)	37 (62%)	291.9
50k	60	14 (23%)	46 (77%)	1473.4
	60	15 (25%)	45 (75%)	1446.1
250k	60	11 (18%)	49 (82%)	7341.8
	60	11 (18%)	49 (82%)	7252

A better tuned initial probability weight parameters should be able to improve the performance.

5.3.13 Creating an Eye instead of Filling an Eye

Creating an Eye instead of Filling an Eye rule reduces the Capturing search performance (Kano book 3, 10k: +0 pass, 50k: +0 pass, 250k: -2 pass; Kano book 4, 10k: -6 pass, 50k: -2 pass, 250k:

Table 5.7: Extensions Setting: Tactic Initial Probability Weight

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	No
Avoiding Self-Atari	No
Avoiding Single Stone Self-Atari	No
Ladder	No
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

Table 5.8: Test Results: Tactic Initial Probability Weight

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	9 (15%)	52 (85%)	128.8
50k	61	4 (7%)	57 (93%)	577.8
250k	61	2 (3%)	59 (97%)	2571.4
Kano Book 4				
10k	51	14 (27%)	37 (73%)	206.2
50k	51	12 (24%)	39 (76%)	984
250k	51	10 (20%)	41 (80%)	4414.9
GoTools Problems				
10k	100	2 (2%)	98 (98%)	51.7
	100	2 (2%)	98 (98%)	51.7
50k	100	1 (1%)	99 (99%)	253.9
	100	2 (2%)	98 (98%)	253.6
250k	100	1 (1%)	99 (99%)	1245.2
	100	0 (0%)	100 (100%)	1245
SK Problems				
10k	60	20 (33%)	40 (67%)	121
	60	18 (30%)	42 (70%)	123.1
50k	60	17 (28%)	43 (72%)	627.6
	60	19 (32%)	41 (68%)	640
250k	60	15 (25%)	45 (75%)	3156.4
	60	13 (22%)	47 (78%)	3116.5
1250k	60	11 (18%)	49 (82%)	16415.8
	60	10 (17%)	50 (83%)	16232.8

-4 pass) but improves the Life&Death search performance (GoTools problems, 10k: +0 pass, 50k: -0.5 pass, 250k: -0.5 pass; SK problems, 10k: -3 pass, 50k: +3.5 pass, 250k: +3 pass)

Eye shape knowledge is very important in solving Life&Death problems because the goal of

Life&Death problems is to create two eyes.

Table 5.9: Extensions Setting: Creating an Eye instead of Filling an Eye

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	Yes
Avoiding Self-Atari	No
Avoiding Single Stone Self-Atari	No
Ladder	No
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

Table 5.10: Test Results: Creating an Eye instead of Filling an Eye

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	9 (15%)	52 (85%)	130.5
50k	61	4 (7%)	57 (93%)	576.6
250k	61	4 (7%)	57 (93%)	2663.6
Kano Book 4				
10k	51	20 (39%)	31 (61%)	203.4
50k	51	14 (27%)	37 (73%)	911.8
250k	51	14 (27%)	37 (73%)	4280.9
GoTools Problems				
10k	100	2 (2%)	98 (98%)	51
	100	2 (2%)	98 (98%)	51.2
50k	100	2 (2%)	98 (98%)	248.2
	100	2 (2%)	98 (98%)	245
250k	100	2 (2%)	98 (98%)	1223.5
	100	0 (0%)	100 (100%)	1230.1
SK Problems				
10k	60	21 (35%)	39 (65%)	121.7
	60	23 (38%)	37 (62%)	122.7
50k	60	12 (20%)	48 (80%)	618.8
	60	17 (28%)	43 (72%)	621.8
250k	60	12 (20%)	48 (80%)	3129.3
	60	10 (17%)	50 (83%)	3158.2

5.3.14 Avoiding Self-Atari

Avoiding Self-Atari rule improves the Capturing search performance (Kano book 3, 10k: +2 pass, 50k: +0 pass, 250k: +1 pass; Kano book 4, 10k: +2 pass, 50k: +0 pass, 250k: +2 pass) and slightly improves the Life&Death search performance (GoTools problems, 10k: +0 pass, 50k: +1 pass, 250k: +1 pass; SK problems, 10k: +4.5 pass, 50k: +0 pass, 250k: +0 pass).

Table 5.11: Extensions Setting: Avoiding Self-Atari

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	Yes
Avoiding Self-Atari	Yes
Avoiding Single Stone Self-Atari	No
Ladder	No
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

5.3.15 Avoiding Single Stone Self-Atari

Avoiding Single Stone Self-Atari rule slightly reduces the Capturing search performance (Kano book 3, 10k: -1 pass, 50k: -1 pass, 250k: -1 pass; Kano book 4, 10k: +0 pass, 50k: +0 pass, 250k: +0 pass) and does not affect the Life&Death search performance (GoTools problems, 10k: -0.5 pass, 50k: +0.5 pass, 250k: -0.5 pass; SK problems, 10k: +1.5 pass, 50k: -1.5 pass, 250k: +0.5 pass).

Allow single stone self-atari is good for the Capturing search.

5.3.16 Ladder

Ladder knowledge significantly improves the Capturing search performance (Kano book 3, 10k: +4 pass, 50k: +3 pass, 250k: +3 pass; Kano book 4, 10k: +6 pass, 50k: +6 pass, 250k: +7 pass) but lowers the Life&Death search performance (GoTools problems, 10k: -1 pass, 50k: -0.5 pass, 250k: +0 pass; SK problems, 10k: -9 pass, 50k: +0 pass, 250k: -3.5 pass).

It's very interesting to see that while ladder knowledge is very useful to the Capturing search, it is not useful to the Life&Death search. Just as the eye shape knowledge is useful to the Life&Death search while not useful to the Capturing search, different tactic searches favor different knowledge.

Table 5.12: Test Results: Avoiding Self-Atari

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	7 (11%)	54 (89%)	137.1
50k	61	4 (7%)	57 (93%)	573.2
250k	61	3 (5%)	58 (95%)	2599.8
Kano Book 4				
10k	51	18 (35%)	33 (65%)	241.8
50k	51	14 (27%)	37 (73%)	1121
250k	51	12 (24%)	39 (76%)	4637.5
GoTools Problems				
10k	100	1 (1%)	99 (99%)	55.3
	100	2 (2%)	98 (98%)	56
50k	100	1 (1%)	99 (99%)	270.6
	100	1 (1%)	99 (99%)	271.3
250k	100	0 (0%)	100 (100%)	1332.3
	100	0 (0%)	100 (100%)	1332.5
SK Problems				
10k	60	18 (30%)	42 (70%)	127.1
	60	17 (28%)	43 (72%)	129.5
50k	60	15 (25%)	45 (75%)	660.8
	60	14 (23%)	46 (77%)	653.5
250k	60	10 (17%)	50 (83%)	3286.5
	60	12 (20%)	48 (80%)	3198.4

Table 5.13: Extensions Setting: Avoiding Single Stone Self-Atari

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	Yes
Avoiding Self-Atari	Yes
Avoiding Single Stone Self-Atari	Yes
Ladder	No
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

The right knowledge would efficiently reduces the semi-random simulation policy space and thus improves the search performance, while unrelated knowledge would add bias to the semi-random simulation policy space and thus reduces the search performance.

Table 5.14: Test Results: Avoiding Single Stone Self-Atari

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	8 (13%)	53 (87%)	141.8
50k	61	5 (8%)	56 (92%)	566.7
250k	61	4 (7%)	57 (93%)	2667.9
Kano Book 4				
10k	51	18 (35%)	33 (65%)	262.8
50k	51	14 (27%)	37 (73%)	1071.7
250k	51	12 (24%)	39 (76%)	4882.3
GoTools Problems				
10k	100	2 (2%)	98 (98%)	57.8
	100	2 (2%)	98 (98%)	58
50k	100	0 (0%)	100 (100%)	284.1
	100	1 (1%)	99 (99%)	283
250k	100	1 (1%)	99 (99%)	1382.7
	100	0 (0%)	100 (100%)	1394.3
SK Problems				
10k	60	16 (27%)	44 (73%)	136.1
	60	16 (27%)	44 (73%)	135.2
50k	60	18 (30%)	42 (70%)	681.1
	60	14 (23%)	46 (77%)	692.6
250k	60	10 (17%)	50 (83%)	3430.2
	60	11 (18%)	49 (82%)	3439.1

Table 5.15: Extensions Setting: Ladder

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	Yes
Avoiding Self-Atari	Yes
Avoiding Single Stone Self-Atari	Yes
Ladder	Yes
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

5.3.17 Proximity

It's very clear that the Proximity extension significantly reduces both the Capturing search performance (Kano book 3, 10k: -12 pass, 50k: -6 pass, 250k: -6 pass; Kano book 4, 10k: -13 pass, 50k: -12 pass, 250k: -7 pass) and the Life&Death search performance (GoTools problems, 10k: -20 pass,

Table 5.16: Test Results: Ladder

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	4 (7%)	57 (93%)	255.8
50k	61	2 (3%)	59 (97%)	1086.6
250k	61	1 (2%)	60 (98%)	5273.6
Kano Book 4				
10k	51	11 (22%)	40 (78%)	506.5
50k	51	8 (16%)	43 (84%)	2265.1
250k	51	5 (10%)	46 (90%)	9524.8
GoTools Problems				
10k	100	3 (3%)	97 (97%)	123
	100	3 (3%)	97 (97%)	123.4
50k	100	1 (1%)	99 (99%)	585
	100	1 (1%)	99 (99%)	589.5
250k	100	1 (1%)	99 (99%)	2874.9
	100	0 (0%)	100 (100%)	2862.2
SK Problems				
10k	60	25 (42%)	35 (58%)	295.5
	60	25 (42%)	35 (58%)	287.3
50k	60	18 (30%)	42 (70%)	1418.2
	60	14 (23%)	46 (77%)	1431.4
250k	60	16 (27%)	44 (73%)	6931.4
	60	12 (20%)	48 (80%)	7006.8

50k: -8.5 pass, 250k: -8.5 pass; SK problems, 10k: -10 pass, 50k: -15.5 pass, 250k: -11 pass).

Table 5.17: Extensions Setting: Proximity

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	Yes
Avoiding Self-Atari	Yes
Avoiding Single Stone Self-Atari	Yes
Ladder	No
Proximity	Yes
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

5.3.18 No New Candidate

Using the floating search region (dynamically creating candidate moves) significantly improves both the Capturing search performance and the Life&Death search performance. Kano book 3, 10k:

Table 5.18: Test Results: Proximity

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	16 (26%)	45 (74%)	81.2
50k	61	8 (13%)	53 (87%)	357.6
250k	61	5 (8%)	56 (92%)	1659.5
Kano Book 4				
10k	51	24 (47%)	27 (53%)	135.6
50k	51	20 (39%)	31 (61%)	613.1
250k	51	12 (24%)	39 (76%)	2768.4
GoTools Problems				
10k	100	21 (21%)	79 (79%)	64.6
	100	25 (25%)	75 (75%)	64.1
50k	100	7 (7%)	93 (93%)	340.2
	100	12 (12%)	88 (88%)	342.6
250k	100	8 (8%)	92 (92%)	1812.4
	100	10 (10%)	90 (90%)	1816.3
SK Problems				
10k	60	35 (58%)	25 (42%)	114.8
	60	35 (58%)	25 (42%)	112.1
50k	60	34 (57%)	26 (43%)	567
	60	29 (48%)	31 (52%)	569.5
250k	60	27 (45%)	33 (55%)	2854.7
	60	23 (38%)	37 (62%)	3000.3

Table 5.19: Extensions Setting: No New Candidate

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	No
Avoiding Self-Atari	No
Avoiding Single Stone Self-Atari	No
Ladder	No
Proximity	No
Create New Candidate	No
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

+1 pass, 50k: +3 pass, 250k: +5 pass; Kano book 4, 10k: +4 pass, 50k: +7 pass, 250k: +4 pass. GoTools problems, 10k: +0 pass, 50k: -0.5 pass, 250k: +0 pass; SK problems, 10k: +4.5 pass, 50k: +5.5 pass, 250k: +4.5 pass.

Table 5.20: Test Results: No New Candidate

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	10 (16%)	51 (84%)	20.2
50k	61	7 (11%)	54 (89%)	98.9
250k	61	7 (11%)	54 (89%)	459
Kano Book 4				
10k	51	18 (35%)	33 (65%)	241.8
50k	51	19 (37%)	32 (63%)	108.8
250k	51	14 (27%)	37 (73%)	497.8
GoTools Problems				
10k	100	2 (2%)	98 (98%)	51.3
	100	2 (2%)	98 (98%)	50.7
50k	100	1 (1%)	99 (99%)	248.2
	100	1 (1%)	99 (99%)	251.6
250k	100	0 (0%)	100 (100%)	1227.6
	100	1 (1%)	99 (99%)	1235
SK Problems				
10k	60	23 (38%)	37 (62%)	40.7
	60	24 (40%)	36 (60%)	41
50k	60	24 (40%)	36 (60%)	204.8
	60	23 (38%)	37 (62%)	202.4
250k	60	18 (30%)	42 (70%)	1033.9
	60	19 (32%)	41 (68%)	1027.3

Table 5.21: Extensions Setting: Absence of the Virtual Boundary

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	No
Avoiding Self-Atari	No
Avoiding Single Stone Self-Atari	No
Ladder	No
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	No
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

5.3.19 Absence of the Virtual Boundary

Virtual boundary knowledge is only used for the Life&Death search. It improves the Life&Death search performance especially when there are fewer simulations (GoTools problems, 10k: -0.5 pass, 50k: -0.5 pass, 250k: +0 pass; SK problems, 10k: +8 pass, 50k: -1 pass, 250k: +0 pass) and reduces

Table 5.22: Test Results: Absence of the Virtual Boundary

	Tests	Fail	Pass	CpuTime (s)
GoTools Problems				
10k	100	0 (0%)	100 (100%)	51.7
	100	3 (3%)	97 (97%)	52.2
50k	100	1 (1%)	99 (99%)	255.7
	100	1 (1%)	99 (99%)	257
250k	100	0 (0%)	100 (100%)	1238.2
	100	1 (1%)	99 (99%)	1247
SK Problems				
10k	60	25 (42%)	35 (58%)	413.7
	60	29 (48%)	31 (52%)	417.2
50k	60	18 (30%)	42 (70%)	2108.3
	60	16 (27%)	44 (73%)	2110.8
250k	60	15 (25%)	45 (75%)	10865.8
	60	13 (22%)	47 (78%)	10734.7

70% of the search time.

5.3.20 Heuristic

Heuristic knowledge applied to the tree search part slightly improves both the Capturing search performance (Kano book 3, 10k: -2 pass, 50k: +1 pass, 250k: +1 pass; Kano book 4, 10k: +2 pass, 50k: +1 pass, 250k: -2 pass) and the Life&Death search performance (GoTools problems, 10k: +0 pass, 50k: -1 pass, 250k: +0.5 pass; SK problems, 10k: +5.5 pass, 50k: -0.5 pass, 250k: +0 pass) especially when there is a small number of simulations. This shows that heuristic knowledge can jump start tree search at early stage.

Table 5.23: Extensions Setting: Heuristic

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	Yes
Avoiding Self-Atari	Yes
Avoiding Single Stone Self-Atari	Yes
Ladder	Yes
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	Yes
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

Table 5.24: Test Results: Heu

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	6 (10%)	55 (90%)	255.3
50k	61	1 (2%)	60 (98%)	1105.8
250k	61	0 (0%)	61 (100%)	5319.5
Kano Book 4				
10k	51	9 (18%)	42 (82%)	526.9
50k	51	7 (14%)	44 (86%)	2194.6
250k	51	7 (14%)	44 (86%)	9674.7
GoTools Problems				
10k	100	2 (2%)	98 (98%)	127.2
	100	4 (4%)	96 (96%)	127.6
50k	100	2 (2%)	98 (98%)	607
	100	2 (2%)	98 (98%)	612.2
250k	100	0 (0%)	100 (100%)	2950.8
	100	0 (0%)	100 (100%)	2956.2
SK Problems				
10k	60	19 (32%)	41 (68%)	287.1
	60	20 (33%)	40 (67%)	287.6
50k	60	19 (32%)	41 (68%)	1416.1
	60	14 (23%)	46 (77%)	1446.3
250k	60	13 (22%)	47 (78%)	7232.1
	60	15 (25%)	45 (75%)	7111.9

CHAPTER 6: LEARNING PATTERNS

In this chapter, pattern learning techniques will be introduced to improve the quality of the MC simulation policy.

In the game of Go, shape is a very important kind of knowledge. More than half of the contents of an ordinary Go tutorial book are focused on the shape knowledge. An experienced player can easily identify the good shape and the bad shape. Good shapes should be formed and bad shapes should be avoided.

In figure 6.1 black plays at the position marked with the triangle to form good shape.

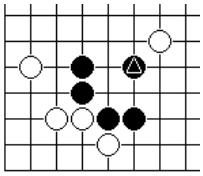


Figure 6.1: Good Shape

In figure 6.2 black plays at the position marked with the triangle to form bad shape.

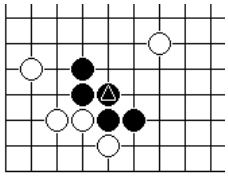


Figure 6.2: Bad Shape

Due to the fact that there is an abundant amount of shape knowledge, complete implementation is not feasible. Besides, cramming an excessive amount of hard-coded shape knowledge into an MC simulation policy would dramatically slow down the simulation speed and make it unusable.

MoGo uses basic hard-coded 3*3 patterns and significantly improves the Go engine strength.

Through the use of a fast indexing method to build and use 3*3 pattern library, GoIntellect increased winning rates against GNUGO by 7.5%.

In this chapter, two methods are used to learn shapes.

1. Learning shapes through the 3*3 and 5*5 pattern libraries.

2. ANN

6.1 The Pattern Guided MC Simulation Policy

In this section, a fast pattern match method that uses 3*3 and 5*5 pattern libraries is used to extend the MC simulation policy.

Shapes are pretty local and that is why a good or bad shape usually can be identified within 3*3, 4*4, or 5*5 region.

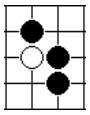


Figure 6.3: A Good Shape for Black

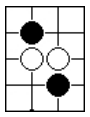


Figure 6.4: A Bad Shape for Black

Local shapes can be learned from game records.

3*3 and 5*5 pattern libraries are built through a given learning dataset set. An extra pattern weight value is given to positions to form matched patterns.

6.1.1 How a Pattern Library is Created

Pseudo code for building a pattern library.

Pseudo Code 6.6: Building Pattern Library

```
{
  foreach (board configuration in solution path)
  {
    foreach (legal move in current board configuration)
    {
        Pattern p = GetCurrentpattern();
        PatternFamily pf = GetPatternFamily(p);
    foreach (Pattern pattern in pf)
    {
    CalculateZobristHash(pattern);

    pattern.nu_appearance ++;
    if( pattern center position == next move)
    pattern.nu_chosen ++;
    SavePatternHashToLibrary(pattern);
    }
    }
  }
}
```

A pattern is defined as an empty (no stone yet) centered board configuration within a given region. In this section, a region is a 3*3 or a 5*5 square.

Learning data set contains 740 Life&Death problems from the Life&Death book, these problems are not used for tests. The board configuration from the solution path (the path to kill the targets or keep the targets alive) of each problem is used to extract patterns.

For each board configuration, every empty centered 3*3 or 5*5 region is saved to the pattern library as a pattern. For each pattern, two parameters are saved together with the pattern: the number of appearance ($nu_{appearance}$ represents how many times this pattern appears on the board as candidate) and the number of times for it to be chosen (nu_{chosen} represents how many times this pattern is chosen for the next move). For a new pattern that is inserted into the pattern library, $nu_{appearance}$ is 1 and nu_{chosen} is 0. If the pattern is already inside the pattern library $nu_{appearance}$ would increment by 1. For a pattern in which the center is used for the next move, nu_{chosen} would

increment by 1.

Considering the board configuration symmetry, rotation and color shift, 16 patterns are in one pattern family. A pattern family is defined as a cluster of patterns that can transform into each other by symmetry, rotation and color shift transformation. If one pattern is inserted into the pattern library, all other 15 patterns of the same pattern family will be also inserted into pattern library.

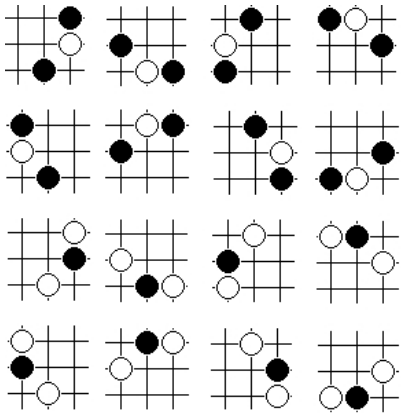


Figure 6.5: Pattern Family

To enable fast access to a pattern library, Zobrist hashing is used to transform a pattern to a Zobrist hash value. A Zobrist hash value is saved to a pattern library instead of the whole pattern configuration. A Pattern library can be pretty big, containing millions of patterns. Using Zobrist hash would guarantee that only constant time is utilized in the search for a certain pattern. Inside a pattern, each position can have four states: empty, black, white and border.

After the pattern library is built, each pattern will be assigned a pattern ranking value providing the number of appearances and the number of times being chosen. The pattern ranking is used as an additional MC simulation policy weight by pattern. One simple way to calculate the ranking is $nu_{chosen}/nu_{appearance}$, but apparently a pattern with 9 appearances and 8 times of being chosen is far better than a pattern with 1 time appearance and 1 time of being chosen. Just like the Upper confidence bound, a lower confidence bound strategy is used to rank the pattern.

if $nu_{chosen} > 0$

$$nu_{rankValue} = \frac{nu_{chosen}}{nu_{appearance}} - \frac{C}{\sqrt{nu_{appearance}}}, C = 1.0 \tag{6.1}$$

if $nu_{chosen} = 0$

$$nu_{rankValue} = 0 \tag{6.2}$$

In figure 6.6 and figure 6.7, the position marked with stone X is the center position of the pattern; the position marked with empty X is out side of the board which we consider that is enlarged border positions and part of the pattern. Most 5by5 patterns contain border positions which may occupy

Table 6.1: Top 10 5by5 Patterns, Ranking by $nu_{rankValue}$

Nu_{chosen}	$Nu_{appearance}$	$Nu_{chosen/nu_appearance}$	$Nu_{rankValue}$
12	12	1	0.711324865
16	18	0.888889	0.653186628
6	6	1	0.59175171
4	4	1	0.5
10	13	0.769231	0.491880671
6	7	0.857143	0.479178384
3	3	1	0.422649731
17	30	0.566667	0.384092481
10	16	0.625	0.375
4	5	0.8	0.352786405

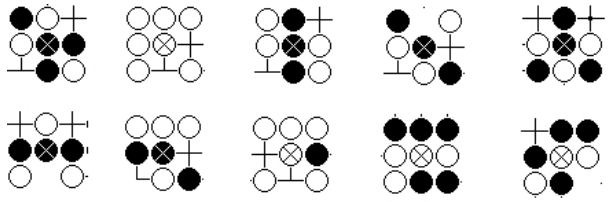


Figure 6.6: Top 10 3by3 Patterns By Ranking

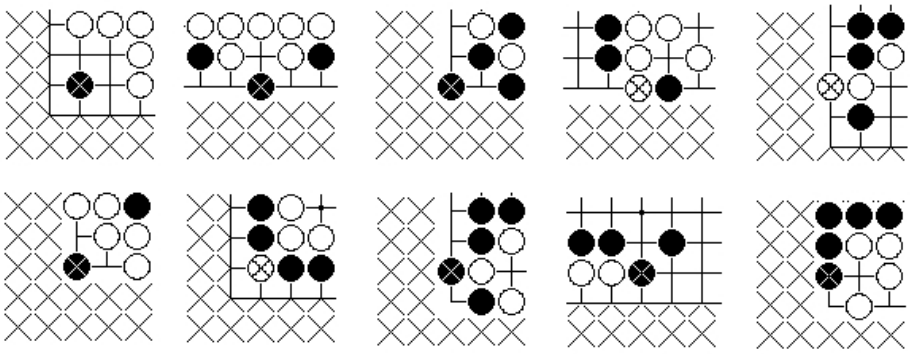


Figure 6.7: Top 10 5by5 Patterns By Ranking

multiple lines of grids.

6.1.2 How Patterns Are Used in the MC Simulations

If the pattern match extension is turned on, matched patterns will be given an additional pattern probability weight.

For matched patterns:

$$w_i\_pattern = nu_{rankValue} \times C_{patternweight}, C_{patternweight} = 50 \tag{6.3}$$

At the beginning of the MC simulation, all patterns surrounding a legal move will be scanned, the Matched pattern legal move will be given an additional pattern probability weight. During the MC simulation, after a stone is put on the board, only patterns whose positions have changed will

be analyzed and their pattern weight updated.

On average, pattern match takes 40% of the MC simulation time.

6.2 Using The Trained ANN to Guide the MC Simulation Policy

One problem associated with the above pattern match method is that it required an exact match. This would require relatively large pattern library and thus a relatively large learning data sets which is not usually feasible. And, for a big pattern, for instance a 5*5 pattern, this would pose a even more serious problem.

In this chapter, artificial neural network (ANN) will be used as a fussy pattern match method to guide the MC simulation policy.

6.2.3 Training the ANN

Using ANN to solve Computer Go problems is a interesting and complex topic. Sophisticated network architecture and Go knowledge based features are usually used to improve prediction quality [30]. Since the effort here is to use ANN to generate semi-random moves which requires fastness, simple network architecture is utilized.

When creating a network, it is necessary to define how many layers, neurons and connections it should have. The ANN will have difficulties learning if the network becomes too large, and it will also tend to over-fit with poor generalization after learning. If the network is too small, it will not be able to represent the the problem.

Standard three-layer network is utilized: one input layer, one hidden layer and one output layer.

The input Layer contains all points inside the bounding box of the problem region (restricted inside 9*9 region, border are included, 81 input neuron altogether to represent the problem region) and the color of the next to play (represented by 1 input neuron), with a total 82 input neurons used. +1 represents black; 0 represents empty; -1 represents white. Values are scaled by 10 for the last move position. +100 represents the border; +2 represents that black is the next to play while -2 represents that white is the next to play (the neuron that represents the next to play is set as the first input neuron).

The output Layer: the output region is the same as the input region, with a total of 81 output neurons are used. +1 represents the position for the next move and for all other positions the value is 0.

Several preliminary evaluation of hidden Layer configuration (80 neurons, 40 neurons, 20 neurons, 10 neurons) are performed and decided by the author's Go knowledge (which considers both prediction generalization and fastness). 20 neurons are chosen as hidden layer configuration.

The training data set used is the same as in the above chapter. It contains 740 Life&Death problems from the Life&Death book. These training problems are not used for tests. The board configuration (here is a restricted problem region) along the solution path (the path to kill the targets or to keep the targets alive) for each problem is used to train the ANN.

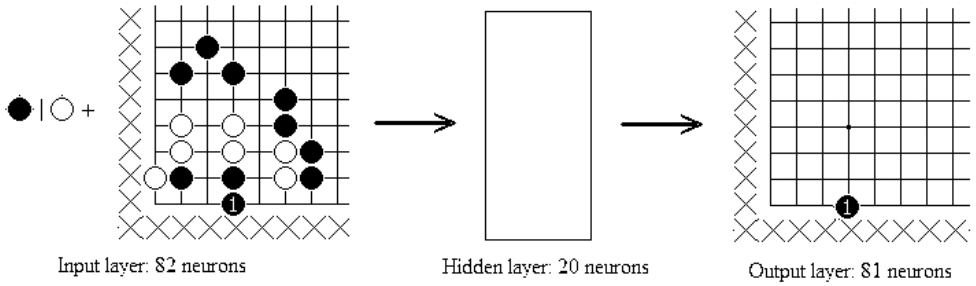


Figure 6.8: ANN Network Architecture

Open source ANN library Fast ANN (FANN, http://leenissen.dk/fann) is used to train the ANN. It uses back propagation training.

Maximum training epoch and desired error are usually served as the training stop criteria. Through the training, the desired error reduces dramatically in the first 100 epoches and becomes stable after 100 epoches. To balance prediction accuracy, generalization and some certain level of randomness for MC simulation purpose, the maximum epoch is set to 100 and the desired error is set to 0.01.

6.2.4 Using the ANN

A trained ANN is used to guide the MC simulation policy. In an MC simulation, the board configuration is used as an input for the trained ANN, while an output board is evaluated. The grid value represents the evaluation value for each board position.

Ei represents the evaluation value for each grid. If Position i is not a legal move, then $E_i = 0$.

$$E_{total} = \sum E_i, w_i\_ANN = \frac{E_i}{E_{total}} \times C_{ANN}, C_{ANN} = 1280 \tag{6.4}$$

w_i_ANN is the additional probability weight from the ANN extension.

Here is a board position evaluation value evaluated by a trained ANN. The blue color represents better evaluation values.

Using ANN to evaluate candidate moves is very computation expensive. This extension uses about 90% of the CPU time of the whole MC simulation.

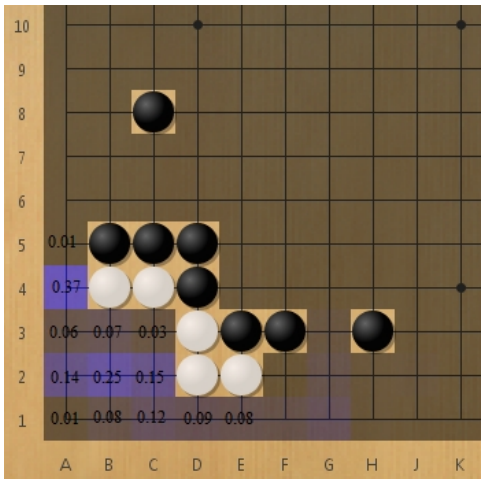


Figure 6.9: ANN Go Board Evaluation

6.3 Test Results and Analysis

Due to the unavailability of a large Capturing problem training set, the pattern library learned from the Life&Death training set is used for both the Capturing and the Life&Death searches.

6.3.5 3*3 Pattern

3*3 pattern is not useful for the Capturing search (Kano book 3, 10k: +1 pass, 50k: +1 pass, 250k: +0 pass; Kano book 4, 10k: -3 pass, 50k: -2 pass, 250k: -1 pass) but improves the Life&Death search performance when the number of the simulation is high (sucha as 250k) (GoTools problems, 10k: +0.5 pass, 50k: +1.5 pass, 250k: +0 pass; SK problems, 10k: -3.5 pass, 50k: -2.5 pass, 250k: +4.5 pass, 1250k: +3.5 pass).

The Life&Death pattern library is useful for the Life&Death search but it is not capable of improving the Capturing search performance. The test result shows that different tactic searches require different pattern libraries.

Enabling 3*3 pattern adds 60% more CPU time, but this time consumption can be improved by better pattern match implementations.

The following tests show that the 3*3 pattern does not affect the search performance while other search extensions are enabled (Kano book 3, 10k: +1 pass, 50k: -1 pass, 250k: +0 pass; Kano book 4, 10k: 0 pass, 50k: 0 pass, 250k: -3 pass; GoTools problems, 10k: +2 pass, 50k: +0 pass, 250k: +0.5 pass; SK problems, 10k: -1.5 pass, 50k: -0.5 pass, 250k: -0.5 pass).

Table 6.2: Extensions Setting: 3*3 Pattern

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	No
Avoiding Self-Atari	No
Avoiding Single Stone Self-Atari	No
Ladder	No
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	Yes
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

Table 6.3: Test Results: 3*3 Pattern

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	8 (13%)	53 (87%)	245.1
50k	61	3 (5%)	58 (95%)	1073.5
250k	61	2 (3%)	59 (97%)	4788.2
Kano Book 4				
10k	51	17 (33%)	34 (67%)	430.9
50k	51	14 (27%)	37 (73%)	1798.9
250k	51	11 (22%)	40 (78%)	8272
GoTools Problems				
10k	100	1 (1%)	99 (99%)	78.8
	100	2 (2%)	98 (98%)	79.5
50k	100	0 (0%)	100 (100%)	389.9
	100	0 (0%)	100 (100%)	388.1
250k	100	0 (0%)	100 (100%)	1933.3
	100	1 (1%)	99 (99%)	1938
SK Problems				
10k	60	23 (38%)	37 (62%)	201.8
	60	22 (37%)	38 (63%)	210.2
50k	60	18 (30%)	42 (70%)	1033.5
	60	23 (38%)	37 (62%)	1040.8
250k	60	9 (15%)	51 (85%)	5186.3
	60	10 (17%)	50 (83%)	5155.5
1250k	60	6 (10%)	54 (90%)	26326.5
	60	8 (13%)	52 (87%)	26201.8

6.3.6 5*5 Pattern

Compared with the 3*3 pattern, the 5*5 pattern does not significantly change the performance of either the Capturing search (Kano book 3, 10k: +3 pass, 50k: -1 pass, 250k: +0 pass; Kano book

Table 6.4: Extensions Setting: Enabling most Extensions

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	Yes
Avoiding Self-Atari	Yes
Avoiding Single Stone Self-Atari	Yes
Ladder	Yes
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	Yes
Tree Search History (f_{his}) Part	Yes
Greedy Mode	Yes
Sorting Children Node	Yes

Table 6.5: Test Results: Enabling most Extensions

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	6 (10%)	55 (90%)	269.7
50k	61	2 (3%)	59 (97%)	1103.8
250k	61	1 (2%)	60 (98%)	5156.3
Kano Book 4				
10k	51	9 (18%)	42 (82%)	507.4
50k	51	7 (14%)	44 (86%)	2156
250k	51	2 (4%)	49 (96%)	9095.6
GoTools Problems				
10k	100	5 (5%)	95 (95%)	124.2
	100	4 (4%)	96 (96%)	123.7
50k	100	2 (2%)	98 (98%)	589.4
	100	2 (2%)	98 (98%)	590
250k	100	1 (1%)	99 (99%)	2847.6
	100	1 (1%)	99 (99%)	2855.5
SK Problems				
10k	60	19 (32%)	41 (68%)	287.9
	60	16 (27%)	44 (73%)	288.6
50k	60	13 (22%)	47 (78%)	1432.6
	60	12 (20%)	48 (80%)	1444.7
250k	60	9 (15%)	51 (85%)	7027.8
	60	8 (13%)	52 (87%)	7092

4, 10k: -3 pass, 50k: -3 pass, 250k: +2 pass) or the Life&Death search (GoTools problems, 10k: -1 pass, 50k: -0.5 pass, 250k: +0.5 pass; SK problems, 10k: +0 pass, 50k: +2 pass, 250k: -3 pass). This is because the 5*5 pattern has a much lower match rate compared with the 3*3 pattern.

The following tests show that the 5*5 pattern does not improve the search performance while

Table 6.6: Extensions Setting: Other Extensions + 3*3 Pattern

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	Yes
Avoiding Self-Atari	Yes
Avoiding Single Stone Self-Atari	Yes
Ladder	Yes
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	Yes
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	Yes
Tree Search History (f_{his}) Part	Yes
Greedy Mode	Yes
Sorting Children Node	Yes

Table 6.7: Test Results: Other Extensions + 3*3 Pattern

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	5 (8%)	56 (92%)	378.3
50k	61	3 (5%)	58 (95%)	1679.1
250k	61	1 (2%)	60 (98%)	7411.6
Kano Book 4				
10k	51	9 (18%)	42 (82%)	763.1
50k	51	7 (14%)	44 (86%)	3126.6
250k	51	5 (10%)	46 (90%)	14016.7
GoTools Problems				
10k	100	3 (3%)	97 (97%)	155.6
	100	2 (2%)	98 (98%)	155.8
50k	100	2 (2%)	98 (98%)	762.4
	100	2 (2%)	98 (98%)	757.9
250k	100	0 (0%)	100 (100%)	3655.6
	100	1 (1%)	99 (99%)	3672.5
SK Problems				
10k	60	18 (30%)	42 (70%)	373.1
	60	19 (32%)	41 (68%)	374
50k	60	13 (22%)	47 (78%)	1908
	60	13 (22%)	47 (78%)	1890.7
250k	60	9 (15%)	51 (85%)	9406.4
	60	9 (15%)	51 (85%)	9396.5

other search extensions are enabled (Kano book 3, 10k: +2 pass, 50k: +0 pass, 250k: +1 pass; Kano book 4, 10k: -1 pass, 50k: -1 pass, 250k: -4 pass; GoTools problems, 10k: +2.5 pass, 50k: +1 pass, 250k: +0.5 pass; SK problems, 10k: -6 pass, 50k: -2.5 pass, 250k: -1 pass).

Table 6.8: Extensions Setting: 5*5 Pattern

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	No
Avoiding Self-Atari	No
Avoiding Single Stone Self-Atari	No
Ladder	No
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	Yes
ANN	No
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

Table 6.9: Test Results: 5*5 Pattern

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	6 (10%)	55 (90%)	263.3
50k	61	5 (8%)	56 (92%)	1161.5
250k	61	2 (3%)	59 (97%)	5364.2
Kano Book 4				
10k	51	17 (33%)	34 (67%)	486.8
50k	51	15 (29%)	36 (71%)	2110.8
250k	51	8 (16%)	43 (84%)	8739.5
GoTools Problems				
10k	100	3 (3%)	97 (97%)	82.3
	100	3 (3%)	97 (97%)	82
50k	100	1 (1%)	99 (99%)	401.5
	100	1 (1%)	99 (99%)	403.8
250k	100	1 (1%)	99 (99%)	1995.8
	100	1 (1%)	99 (99%)	2004.9
SK Problems				
10k	60	18 (30%)	42 (70%)	222.7
	60	20 (33%)	40 (67%)	221
50k	60	18 (30%)	42 (70%)	1115.8
	60	14 (23%)	46 (77%)	1107
250k	60	18 (30%)	42 (70%)	5551.5
	60	16 (27%)	44 (73%)	5574.2

6.3.7 ANN

Due to computation resource restrictions (It is very time consuming when ANN runs with 250k MC simulations), only 10k MC simulations and 50k MC simulations Life&Death tests are performed. ANN does not help to improve the Life&Death search performance (GoTools problems, 10k: -3

Table 6.10: Extensions Setting: Other Extensions + 5*5 Pattern

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	Yes
Avoiding Self-Atari	Yes
Avoiding Single Stone Self-Atari	Yes
Ladder	Yes
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	Yes
ANN	No
Tree Search Heuristic (f_{heu}) Part	Yes
Tree Search History (f_{his}) Part	Yes
Greedy Mode	Yes
Sorting Children Node	Yes

Table 6.11: Test Results: Other Extensions + 5*5 Pattern

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	4 (7%)	57 (93%)	415.1
50k	61	2 (3%)	59 (97%)	1733.2
250k	61	0 (0%)	61 (100%)	7794
Kano Book 4				
10k	51	10 (20%)	41 (80%)	867
50k	51	8 (16%)	43 (84%)	3309.5
250k	51	6 (12%)	45 (88%)	14691.7
GoTools Problems				
10k	100	2 (2%)	98 (98%)	161.1
	100	3 (3%)	97 (97%)	160.4
50k	100	1 (1%)	99 (99%)	770.5
	100	1 (1%)	99 (99%)	770.1
250k	100	0 (0%)	100 (100%)	3764.4
	100	1 (1%)	99 (99%)	3742
SK Problems				
10k	60	23 (38%)	37 (62%)	397.5
	60	24 (40%)	36 (60%)	403.5
50k	60	13 (22%)	47 (78%)	2005.8
	60	16 (27%)	44 (73%)	1992.8
250k	60	10 (17%)	50 (83%)	9936.8
	60	9 (15%)	51 (85%)	9714.4

pass, 50k: +0 pass; SK problems, 10k: -2.5 pass, 50k: -1 pass).

ANN does not help to improve the Life&Death search performance while other extensions (same as other extensions in pattern section) are enabled (GoTools problems, 10k: +1.5 pass, 50k: -1 pass; SK problems, 10k: -2.5 pass, 50k: +2 pass).

Table 6.12: Extensions Setting: ANN

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	No
Avoiding Self-Atari	No
Avoiding Single Stone Self-Atari	No
Ladder	No
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	Yes
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	No
Greedy Mode	No
Sorting Children Node	No

Table 6.13: Test Results: ANN

	Tests	Fail	Pass	CpuTime (s)
GoTools Problems				
10k	100	4 (4%)	96 (96%)	1117.2
	100	6 (6%)	94 (94%)	1123.7
50k	100	1 (1%)	99 (99%)	4854.8
	100	2 (2%)	98 (98%)	4815.8
SK Problems				
10k	60	23 (38%)	37 (62%)	694.9
	60	20 (33%)	40 (67%)	688.9
50k	60	20 (33%)	40 (67%)	3329.4
	60	18 (30%)	42 (70%)	3482

Table 6.14: Test Results: Other Extensions + ANN

	Tests	Fail	Pass	CpuTime (s)
GoTools Problems				
10k	100	3 (3%)	97 (97%)	1400
	100	3 (3%)	97 (97%)	1420.5
50k	100	4 (4%)	96 (96%)	6136.7
	100	2 (2%)	98 (98%)	6140.8
SK Problems				
10k	60	21 (35%)	39 (65%)	1076.5
	60	19 (32%)	41 (68%)	1083.5
50k	60	11 (18%)	49 (82%)	5234.5
	60	10 (17%)	50 (83%)	5187.2

CHAPTER 7: DOMAIN INDEPENDENT TREE SEARCH TECHNIQUES

In this chapter, several techniques are used to enhance the Monte Carlo tree search part. These techniques are Go (knowledge) independent, thus all these techniques can be used as general Monte Carlo tree search extensions.

7.1 History Heuristic

History heuristic is a very useful extension for alpha-beta search [47]. The idea of the history heuristic is to use the completed tree search to guide later tree searches. Usually a history heuristic table is used to save the search result for each point.

The history Heuristic idea is borrowed to accelerate the tree search. The idea of the history heuristic is to accelerate searching the unexplored part of the tree by using the pre-explored part of the tree.

If a board position has a higher frequency in passing through the Monte Carlo tree search, that would normally be an indication of the importance of that board position.

f_{his} in 5.3 is defined as follows:

if $nu_{rootvisit} \geq C_{his}$,

$$f_{his} = S(n_{nuVisit} - C_{his}) \times V_{his} \tag{7.1}$$

S(x) is the transformed sigmoid function (see section 5.2 for definition).

if $n_{nuRootVisit} < C_{his}$, $f_{his} = 0$

$C_{his} = 1000$;

A history value for a board position i is defined as the ratio between the total number of the simulations passing through that board position within the tree and total number of the simulations.

s represents a simulation.

$nu_{visit}(s, i) = 1$, if s contains position l_i in the tree part

$nu_{visit}(s, i) = 0$, if s does not contains position l_i at the tree part

$V_{his}(i) = \frac{\sum nu_{visit}(s,i)}{n_{nuRootVisit}}$

$n_{nuRootVisit}$ is the total number of simulations.

A history value table is used to save the V_{his} the value for every board position.

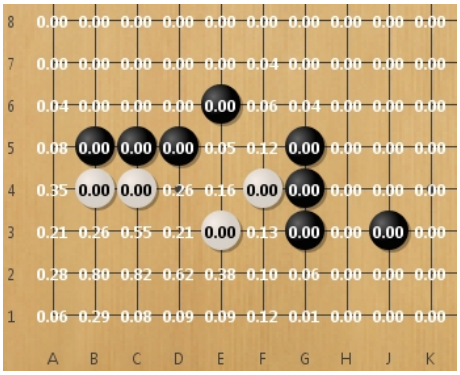


Figure 7.1: History Values of Board Positions after 10k Simulations

The history heuristic is used to jump start the tree search.

7.2 Sorting Children

Sorting children nodes by heuristics is essential to $\alpha\beta$ search, the performance of which is largely dependent on how well the children nodes is ordered.

In this section, the technique of children sorting is discussed. The MC simulation probability weight for each position is simply used as the rank criteria to sort the children nodes. When children nodes are created, they are sorted by the MC simulation probability weight and inserted into the tree according to this order. At later tree search stage, better ranked children have higher priority to be chosen if they have never been chosen before.

7.3 Greedy Mode

Greedy mode is a simplified version of iterative widening technique, which is easy to implement.

If a child with $\mu > C_{greedy}$ is found, stop searching other unexpanded nodes.

A tested setting of the greedy threshold is:

$C_{greedy} = 95\%$

In the greedy mode, part of the tree (some nodes) are skipped in the exploration.

7.4 Heavy Back Up When Terminal Nodes Are Reached

A terminal node (with a terminal node tag) stores a definite value while the MC evaluation returns a value with uncertainty. It may be possible to accelerate the search process through heavy back up when the search process reaches a terminal node. A series of tests was performed. At the time the search process reaches a terminal node I treat it as a k-fold backup (backup like k simulations, the number of visit increments by k and the value increment by 0 or k). The test result shows that an 8-fold backup is the best.

Table 7.1: Back up Heavily While Reaching Terminal Nodes

k	Unsolved b3	Time b3 (s)	Unsolved b4	Time b4 (s)	Total unsolved	Total time (s)
1	2	251	7	438	9	689
2	0	496	8	1001	8	1497
4	1	436	7	367	8	803
8	0	235	6	464	6	699
16	2	546	8	324	10	870
32	2	281	6	368	8	649
64	1	528	10	396	11	924
128	4	235	12	265	16	500
256	3	308	9	267	12	575
512	7	339	13	352	20	691
1024	5	322	17	214	22	536

7.5 Test Results and Analysis

7.5.1 History

The history extension applied to the tree search improves both the Capturing search performance (Kano book 3, 10k: +0 pass, 50k: +0 pass, 250k: +0 pass; Kano book 4, 10k: +1 pass, 50k: +1 pass, 250k: +0 pass) and the Life&Death search performance (GoTools problems, 10k: +0 pass, 50k: +0.5 pass, 250k: +0.5 pass; SK problems, 10k: +5 pass, 50k: -1 pass, 250k: +4 pass).

Table 7.2: Extensions Setting: History

Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	Yes
Avoiding Self-Atari	Yes
Avoiding Single Stone Self-Atari	Yes
Ladder	Yes
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	No
Tree Search History (f_{his}) Part	Yes
Greedy Mode	No
Sorting Children Node	No

7.5.2 Greedy Mode

Greedy mode is slightly improves search performance (Kano book 3, 10k: -1 pass, 50k: +1 pass, 250k: -1 pass; Kano book 4, 10k: +0 pass, 50k: +2 pass, 250k: +2 pass) and the Life&Death search

Table 7.3: Test Results: History

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	4 (7%)	57 (93%)	263.8
50k	61	2 (3%)	59 (97%)	1166.5
250k	61	1 (2%)	60 (98%)	5238.4
Kano Book 4				
10k	51	10 (20%)	41 (80%)	518.6
50k	51	7 (14%)	44 (86%)	2149.8
250k	51	5 (10%)	46 (90%)	9159
GoTools Problems				
10k	100	3 (3%)	97 (97%)	123.9
	100	3 (3%)	97 (97%)	124.1
50k	100	0 (0%)	100 (100%)	601
	100	1 (1%)	99 (99%)	600.7
250k	100	0 (0%)	100 (100%)	2919.6
	100	0 (0%)	100 (100%)	2918.5
SK Problems				
10k	60	19 (32%)	41 (68%)	288.1
	60	21 (35%)	39 (65%)	286.7
50k	60	18 (30%)	42 (70%)	1489.1
	60	16 (27%)	44 (73%)	1457.6
250k	60	11 (18%)	49 (82%)	7053.9
	60	9 (15%)	51 (85%)	6993.9

performance (GoTools problems, 10k: -2 pass, 50k: -0.5 pass, 250k: -0.5 pass; SK problems, 10k: +3.5 pass, 50k: +2 pass, 250k: -0.5 pass).

Table 7.4: Extensions Setting: Extensions without Greedy

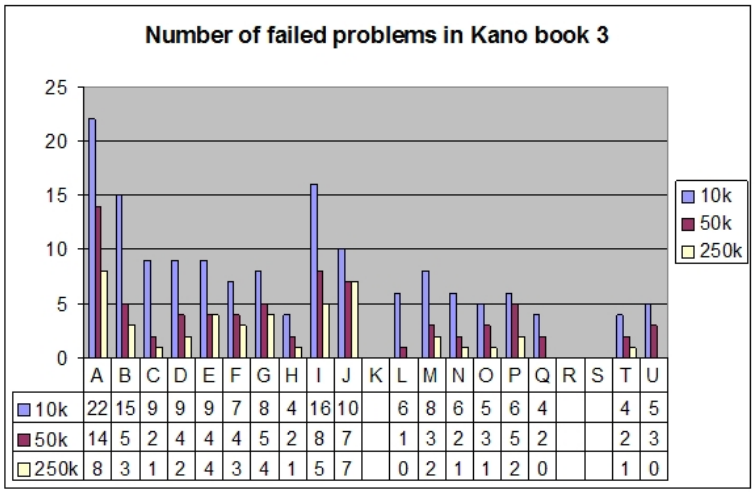
Solid Eye	Yes
Capture&Escape Probability Weight	Yes
Tactic Initial Probability Weight	Yes
Creating an Eye instead of Filling an Eye	Yes
Avoiding Self-Atari	Yes
Avoiding Single Stone Self-Atari	Yes
Ladder	Yes
Proximity	No
Create New Candidate	Yes
Using Virtual Boundary	Yes
3*3 Pattern	No
5*5 Pattern	No
ANN	No
Tree Search Heuristic (f_{heu}) Part	Yes
Tree Search History (f_{his}) Part	Yes
Greedy Mode	No
Sorting Children Node	Yes

Table 7.5: Test Results: Extensions without Greedy

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	5 (8%)	56 (92%)	289.2
50k	61	3 (5%)	58 (95%)	1224.3
250k	61	0 (0%)	61 (100%)	5352.3
Kano Book 4				
10k	51	9 (18%)	42 (82%)	559.7
50k	51	9 (18%)	42 (82%)	2367.8
250k	51	4 (8%)	47 (92%)	10880.3
GoTools Problems				
10k	100	2 (2%)	98 (98%)	127.9
	100	3 (3%)	97 (97%)	128
50k	100	1 (1%)	99 (99%)	617.6
	100	2 (2%)	98 (98%)	618.9
250k	100	0 (0%)	100 (100%)	2963.7
	100	1 (1%)	99 (99%)	2936.8
SK Problems				
10k	60	20 (33%)	40 (67%)	292.4
	60	22 (37%)	38 (63%)	294.1
50k	60	13 (22%)	47 (78%)	1473
	60	16 (27%)	44 (73%)	1450.4
250k	60	7 (12%)	53 (88%)	7129.6
	60	9 (15%)	51 (85%)	7165.2

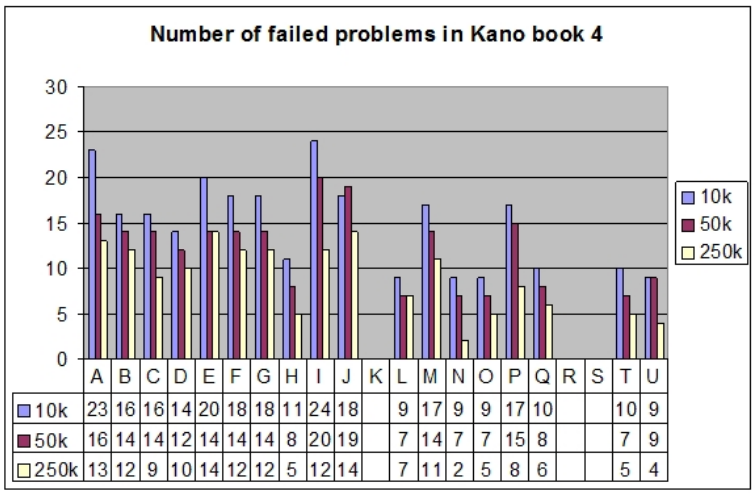
CHAPTER 8: RESULTS SUMMARY AND DISCUSSIONS

8.1 Results Charts



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
10k	22	15	9	9	9	7	8	4	16	10		6	8	6	5	6	4			4	5
50k	14	5	2	4	4	4	5	2	8	7		1	3	2	3	5	2			2	3
250k	8	3	1	2	4	3	4	1	5	7		0	2	1	1	2	0			1	0

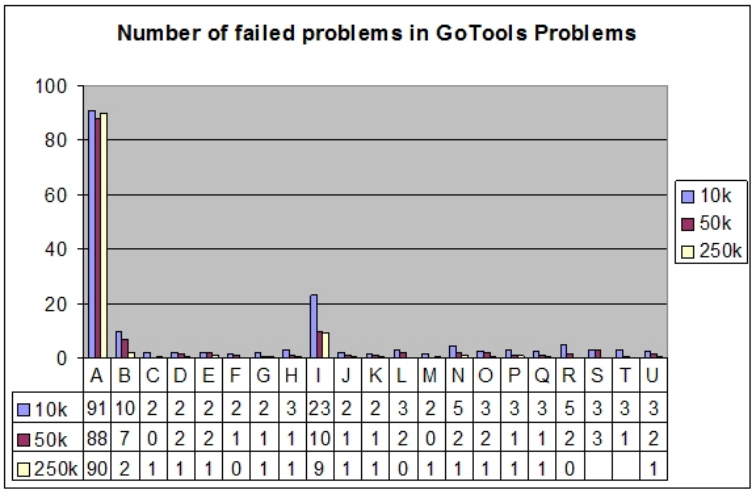
Figure 8.1: Number of Failed Kano Book 3 Problems



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
10k	23	16	16	14	20	18	18	11	24	18		9	17	9	9	17	10			10	9
50k	16	14	14	12	14	14	14	8	20	19		7	14	7	7	15	8			7	9
250k	13	12	9	10	14	12	12	5	12	14		7	11	2	5	8	6			5	4

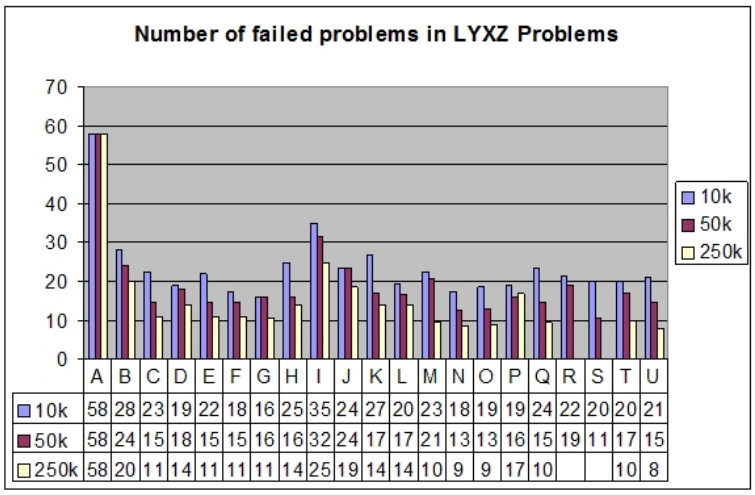
Figure 8.2: Number of Failed Kano Book 4 Problems

- A: No Heuristic Extensions
- B: Solid Eye
- C: Capture&Escape Probability Weight
- D: Tactic Initial Probability Weight
- E: Creating an Eye instead of Filling an Eye



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
10k	91	10	2	2	2	2	2	3	23	2	2	3	2	5	3	3	3	5	3	3	3
50k	88	7	0	2	2	1	1	1	10	1	1	2	0	2	2	1	1	2	3	1	2
250k	90	2	1	1	1	0	1	1	9	1	1	0	1	1	1	1	1	0			1

Figure 8.3: Number of Failed GoTools Problems



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
10k	58	28	23	19	22	18	16	25	35	24	27	20	23	18	19	19	24	22	20	20	21
50k	58	24	15	18	15	15	16	16	32	24	17	17	21	13	13	16	15	19	11	17	15
250k	58	20	11	14	11	11	11	14	25	19	14	14	10	9	9	17	10			10	8

Figure 8.4: Number of Failed SK Problems

- F: Avoiding Self-Atari
- G: Avoiding Single Stone Self-Atari
- H: Ladder
- I: Proximity
- J: No New Candidate
- K: Absence of the Virtual Boundary
- L: Heuristic
- M: 3*3 Pattern
- N: Enabling most Extensions
- O: Other Extensions + 3*3 Pattern
- P: 5*5 Pattern

- Q: Other Extensions + 5*5 Pattern
- R: ANN
- S: Other Extensions + ANN
- T: History
- U: Extensions without Greedy

8.2 Results Summary

Table 8.1: Search Extensions Summary

Search Extensions	Capturing Search	Life&Death Search	Note
Solid Eye	✓	✓	
Capture&Escape Probability Weight	✓	✓	
Tactic Initial Probability Weight	✓x	✓x	(1)
Creating an Eye instead of Filling an Eye	x	✓	
Avoiding Self-Atari	✓	✓	
Avoiding Single Stone Self-Atari	x	≈	
Ladder	✓	x	
Proximity	x	x	
Create New Candidate	✓	✓	
Using Virtual Boundary	N/A	✓	
3*3 Pattern	?	✓	(2)
5*5 Pattern	≈	≈	
ANN	x	x	
Tree Search Heuristic (f_{heu}) Part	✓	✓	
Tree Search History (f_{his}) Part	✓	✓	
Greedy Mode	✓x	✓x	(1)
Sorting Children Node	✓	✓	

(1): It Depends on the number of simulations. (2): It improves search performance when the number of simulations is high.

Here are the test results for the searches configured with the best extension settings.

- Ladder is very useful for the Capturing search, but it is not useful for the Life&Death search.
- Eye shape is useful for Life&Death search, but it is not useful for the Capturing search.
- 3*3 pattern is useful and 5*5 pattern is not useful.
- The Life&Death patterns worsen the Capturing search.
- Greedy mode is not always desirable, especially when the number of simulations becomes high.

- History heuristic is useful, especially when the number of the simulations at a moderate level (neither very high nor low).
- Capturing problems : for kano book 4, 250k simulation can solve all problems except two. (compared with the previous 6 unsolved problems with 500k simulations using the traditional game tree search algorithms).

Table 8.2: Test Results: Best Extensions

	Tests	Fail	Pass	CpuTime (s)
Kano Book 3				
10k	61	5 (8%)	56 (92%)	268.2
50k	61	3 (5%)	58 (95%)	1145.1
250k	61	0 (0%)	61 (100%)	5123.6
Kano Book 4				
10k	51	10 (20%)	41 (80%)	561.1
50k	51	6 (12%)	45 (88%)	2286.6
250k	51	2 (4%)	49 (96%)	8886.9
GoTools Problems				
10k	100	1 (1%)	99 (99%)	155.6
50k	100	0 (0%)	100 (100%)	747.7
250k	100	0 (0%)	100 (100%)	3667.5
SK Problems				
10k	60	17 (28%)	43 (72%)	365.7
50k	60	12 (20%)	48 (80%)	1905
250k	60	7 (12%)	53 (88%)	9410.3

- can solve almost all the problems from Thomas Wolf.

8.3 6-Dan level Life&Death Problem Solved

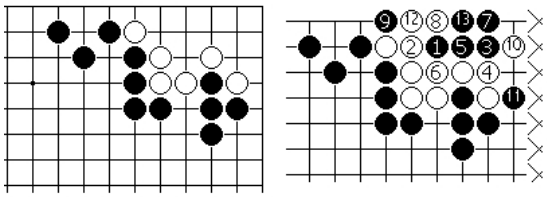


Figure 8.5: 6D Life&Death Problem Solved within 250k Simulations

Here is an example of hard Life&Death Problem Solving. A 6D Life&Death Problem(come from Dr. Anders Kierulf's SmartGo Life&Death problem library) Solved within 250k Simulations At the early stage of the search, the right move is the second to the best child. The right move stably stays as the best child after 80k simulations ($bestchild_{winrate} - secondtobestchild_{winrate} > 0.05$). After 250k simulations, the $root_{winrate} = 0.49$, the $bestchild_{winrate} = 0.56$, the $bestchild_{winrate} - secondtobestchild_{winrate} = 0.18$

($root_{winrate}$), the winning rate of the best child node ($bestchild_{winrate}$), the winning rate difference between the best child node and second to the best child node ($bestchild_{winrate} - secondtobestchild_{winrate}$)

8.4 Contributions and Discussions

The main contributions of this dissertation are:

1. A heuristics based Monte-Carlo tactic tree search framework is proposed to extend the standard Monte-Carlo tree search.

2. (Go) Knowledge based heuristics are systematically investigated to improve the Monte-Carlo tactic tree search.
3. Pattern learning proved effective in improving the Monte-Carlo tactic tree search.
4. Domain knowledge independent tree search enhancements improve the Monte-Carlo tactic tree search performance.
5. A strong Go Tactic solver based on proposed algorithms outperforms the traditional game tree search algorithms.

Future work may includes solving multi-tactic problems; solving other tactic problems such as connection; combining tactic search with global search to provide a strong 19x19 computer Go player;

The techniques developed from this dissertation research may benefit other game domains and application fields.

REFERENCES

[1] Allis, L.V., Meulen, M. van der, and Herik, H.J. van den. Proof-number search, Artificial Intelligence 66 (1994) (1), pp. 91-124.

[2] Allis, L.V. (1994). Searching for solutions in games and artificial intelligence, Ph.D. Thesis, University of Limburg, Masstricht, The Netherlands.

[3] Nobuo Araki, Kazuhiro Yoshida, Yoshimasa Tsuruoka, and Jun'ichi Tsujii. Move prediction in Go with the maximum entropy method. In 2007 IEEE Symposium on Computational Intelligence and Games, 2007.

[4] Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multi-armed bandit problem. Machine Learning, 47, 235-256.

[5] Bruegmann, B. (1993). Monte-Carlo Go. http://www.cgl.ucsf.edu/go/Programs/Gobble.html.

[6] Bouzy, B., Associating shallow and selective global tree search with Monte Carlo for 9x9 Go. In van den Herik, H. J., Bjornsson, Y., and Netanyahu, N. S. Editors, Fourth International Conference on Computers and Games, Ramat-Gan, Israel, 2004.

[7] Bouzy, B., Helmstetter, B., Monte-Carlo Go developments, ACG. Volume 263 of IFIP., Kluwer (2003) 159-174.

[8] Bouzy, B., Cazenave, T., Computer Go: An AI oriented survey, Artificial Intelligence, Volume 132, Issue 1, October 2001, Pages 39-103.

[9] Bouzy, B. and Chaslot, G., Monte-Carlo Go reinforcement learning experiments. In Computational Intelligence in Games, 2006.

[10] Bouzy, B., Move pruning techniques for Monte-Carlo Go. In Advances in Computer Games 11, Taipei, Taiwan, 2005.

[11] Bouzy, B., Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. Joint Conference on Information Sciences, Cary (2003) 505-508

[12] Bouzy, B., History and territory heuristics for Monte-Carlo Go. New Mathematics and Natural Computation, 2(2):1-8, 2006.

[13] Campbell, M., Marsland, T., A Comparison of Minimax Tree-Search Algorithms, Artificial Intelligence 20 (4): 347-367 1983.

[14] Campbell, M., Hoane, A., Jr. and Hsu, F., Deep Blue, Artificial Intelligence, Volume 134, Issues 1-2, January 2002, Pages 57-83.

[15] Cazenave, T., Abstract Proof Search, Computers and Games 2000, LNCS 2063, pp. 39-54, Hamamatsu (2000).

[16] Cazenave, T., Combining Tactical Search and Monte-Carlo in the Game of Go. In IEEE CIG 2005, 2005.

[17] Cazenave, T., and Jouandeau, N., On the parallelization of UCT. In CGW 2007, pages 93-101, June 2007

[18] Chaslot, G., Chatriot, L., Gelly, S., Hoock, J., Perez, J., Rimmel, A., Teytaud, O., Combining expert, offline, transient and online knowledge in Monte-Carlo exploration (2008), http://www.lri.fr/ teytaud/eg.pdf

[19] Chaslot G.M.J.B., Winands M.H.M. Winands, Uiterwijk J.W.H.M., van den Herik H.J., and Bouzy B. Progressive strategies for Monte-Carlo tree search. JCIS workshop 2007, 2007

[20] Chaslot, G., De Jong, S., Saito, J.-T., and Uiterwijk, J. W. H. M. 2006. Monte-Carlo Tree Search in Production Management Problems. In Pierre-Yves Schobbens, Wim Vanhoof, and Gabriel Schwanen, editors, Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium, pages 91-98.

[21] Chen, K., Computer Go: Knowledge, Search, and Move Decision, ICGA Journal, Vol. 24, No. 4, 203-215 (2001).

[22] Chen, K., Soft and Hard Connectivity in Go, Proceedings of the 8th International Conference on Computer Science and Informatics, 282-286, 2005.

[23] Chen, K. and Chen, Z., Static Analysis of Life&Death in the game of Go, Information Sciences, Vol. 121, Nos. 1-2, pp. 113 134 (1999).

[24] Chen, K. and Zhang, P., A Heuristic Search Algorithm for Capturing Problems in Go, ICGA JOURNAL 29 (4): 183-190 December 2006.

[25] Chen, K. and Zhang, P., Monte-Carlo Go with Knowledge Guided Simulations, ICGA Computer Games Workshop 2007, 2007

[26] Coulom, R., Efficient selectivity and backup operators in monte-carlo tree search. In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy (2006).

[27] Coulom, R., Computing Elo ratings of move patterns in the game of Go. ICGA Computer Games Workshop 2007, 2007

[28] Drake, P., and Uurtamo, S. 2007. Heuristics in Monte Carlo Go. In Proceedings of the 2007 International Conference on Artificial Intelligence, CSREA Press.

[29] Drake, P., Pouliot, A., Schreiner, N., Vanberg, B., The Proximity Heuristic and an Opening Book in Monte Carlo Go, 2006, Computer Go Bibliography.

[30] Enzenberger, M. (2003). Evaluation in Go by a neural network using soft segmentation. 10th Advances in Computer Games Conference (pp. 97-108).

[31] Eric C. D. van derWerf. AI techniques for the game of Go. PhD thesis, Universiteit Maastricht, 2004.

[32] Gelly, S., Y. Wang, R. Munos and O. Teytaud, Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, France (2006).

[33] Gelly, S., Silver, D., Combining Online and Offline Knowledge in UCT. International Conference of Machine Learning, ICML 2007, Corvallis Oregon USA, p. 273-280

[34] H. Jaap van den Herik, Jos W. H. M. Uiterwijk, Jack van Rijswijck, Games solved: now and in the future, Artificial Intelligence Volume 134 , Issue 1-2 (January 2002) p. 277 - 311

[35] Junghanns, A. Are There Practical Alternatives to Alpha-Beta?, ICCA JOURNAL 21 (1): 14-32 MAR 1998.

[36] Kano, Y., Graded Go Problems For Beginners, Volume Three, Intermediate Problems, Kiseido Publishing Company, ISBN 4-906574-48-3 (1987).

[37] Kishimoto, A., Correct and Efficient Search Algorithms in the Presence of Repetitions, PhD Dissertation, University of Alberta, (2005).

[38] Julien Kloetzer and Hiroyuki Iida, The Monte-Carlo Approach in Amazons, ICGA Computer Games Workshop 2007, 2007

[39] Knuth, D., Moore, R., Analysis of Alpha-Beta Pruning, Artificial Intelligence 6 (4): 293-326 1975.

[40] Kocsis, L. and Szepesvari, C., Bandit based monte-carlo planning, European Conference on Machine Learning, pp. 282-293 (2006).

[41] Franois Van Lishout, Guillaume Chaslot, and Jos Uiterwijk, Monte-Carlo Tree Search in Backgammon,ICGA Computer Games Workshop 2007, 2007

[42] Michalewicz, Z., Genetic algorithms + data structures = evolution programs. SpringerVerlag, 1992.

[43] Mitchell, T. M., 1997. Machine Learning. The McGraw-Hill Companies, Inc.

[44] Müller, M., Computer Go, Artificial Intelligence, Volume 134, Issues 1-2, January 2002, Pages 145-179.

[45] Nagai, A., Df-pn Algorithm for Searching AND/OR Trees and its Applications, Ph.D. Thesis, Department of Information Science, University of Tokyo, 2002.

[46] Russell, S., and Norvig, P. 2003. Artificial Intelligence: A Modern Approach, second edition. Upper Saddle River, NJ: Prentice Hall.

[47] Schaeffer, J., The History Heuristic and Alpha-Beta Search Enhancements in Practice, IEEE Transactions on Pattern Analysis and Machine Intelligence 11 (11): 1203-1212 NOV 1989.

[48] Seo, M., Iida, H., and Uiterwijk, J.W.H.M.. The PN*-Search Algorithm: Application to Tsume-Shogi. Artificial Intelligence, Vol. 129, Nos. 1-2, pp. 253-277. (2001)

[49] Silver, D., Sutton, R., and Mller, M. (2007). Reinforcement learning of local shape in the game of Go. 20th International Joint Conference on Artificial Intelli- gence (pp. 1053-1058).

[50] Sutton, R., and Barto, A. (1998). Reinforcement learning: An introduction. Cambridge, MA: MIT Press.

[51] Tesauro, G., Temporal Difference Learning and TD-Gammon, Communications of the ACM, 38, (1995), pp. 58-68.

[52] Thomsen, T., Lambda-Search in Game Trees - with Application to Go, ICGA Journal, Vol. 23, No. 4, pp. 203-217 (2000)

[53] Wang, Y., Gelly, S. (2007). Modifications of UCT and sequence-like simulations for Monte-Carlo Go. IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii (pp. 175-182).

[54] Winands, M.H.M., Informed Search in Complex Games, Ph.D. thesis, Universiteit Maastricht, Maastricht, The Netherlands, ISBN 90-5278-429-9, (2004).

[55] Wolf, T., Forward pruning and other heuristic search techniques in tsume go, Information Sciences 122 (no. 1) (2000) pp. 59-76.

[56] Haruhiro Yoshimoto, Kazuki Yoshizoe, Tomoyuki Kaneko, Akihiro Kishimoto, Kenjiro Taura, Monte Carlo Go Has a Way to Go, AAAI 2006

[57] Zhang, P. and Chen, K., Using Different Search Algorithms to Solve Computer Go Capturing Problems, Proceedings of 2006 Chinese Computer Games Conference, pp. 55-61 (in Chinese) (2006).

[58] Zhang, P. and Chen, K., Monte-Carlo Go Tactic Search, 10th International Conference on Computer Science & Informatics (CSI), 2007

[59] Zobrist, A.L., A New Hashing Method with Application for Game Playing, Techn. Rep. #88, Univ. of Wisconsin, Madison, WI 53706 (1970).

APPENDIX A: TACTIC SOLVER ARCHITECTURE

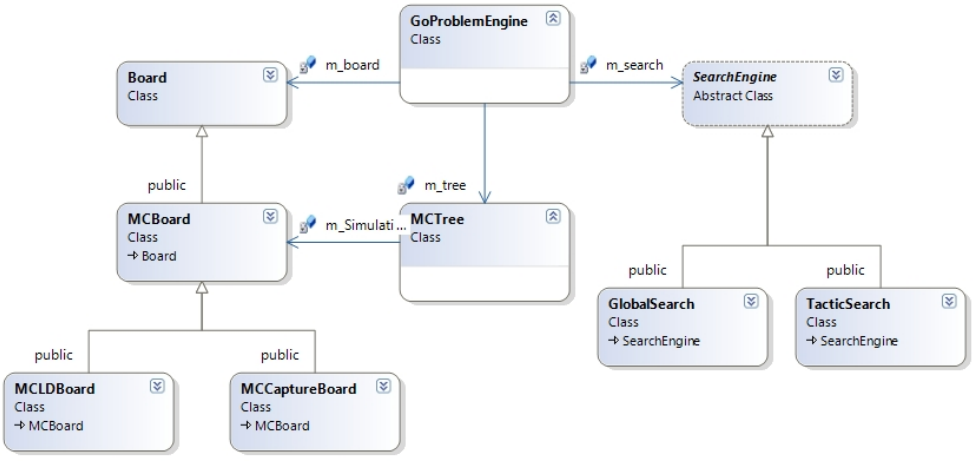


Figure A.1: Class Diagram of Tactic Solver

APPENDIX B: GO TERMS

adjacent On the Go board, two intersections are adjacent if they have a line but no intersection between them.

alive Stones that cannot be captured are alive. Alive stones normally have two eyes or are in seki.

atari Stones are said to be in atari if they can be captured on the opponent's next move, i.e., their block has only one liberty.

block Connected stones of the same color.

dan Master level. Higher numbers are better.

eye A surrounded area providing one safe liberty. Two eyes are usually necessary to make stones safe.

false eye An intersection surrounded by stones of one color which does not provide a sure liberty.

ko A single stone capture that can lead to repetition.

ladder A simple capturing sequence which can take many moves.

liberty An empty point adjacent to a stone, or a block of stones.

seki Coexistence of Black and White blocks that don't have two eyes.

suicide A move that does not capture an opponent block and leaves its own block without a liberty.

tesuji A skillful tactical move.