ABSTRACT

Title of Thesis:	MONTE CARLO TREE SEARCH AND MINIMAX COMBINATION – APPLICATION OF SOLVING PROBLEMS IN THE GAME OF GO
	Jonathan Fun Lin, Master of Science, 2017
Thesis Directed By:	Dr. Michael Fu, Smith School of Business and the Institute for Systems Research, University of Maryland at College Park

Monte Carlo Tree Search (MCTS) has been successfully applied to a variety of games. Its best-first algorithm enables implementations without evaluation functions. Combined with Upper Confidence bounds applied to Trees (UCT), MCTS has an advantage over traditional depth-limited minimax search with alpha-beta pruning in games with high branching factors such as Go. However, minimax search with alpha-beta pruning still surpasses MCTS in domains like Chess. Studies show that MCTS does not detect shallow traps, where opponents can win within a few moves, as well as minimax search. Thus, minimax search performs better than MCTS in games like Chess, which can end instantly (king is captured). A combination of MCTS and minimax algorithm is proposed in this thesis to see the effectiveness of detecting shallow traps in Go problems.

MONTE CARLO TREE SEARCH AND MINIMAX COMBINATION – APPLICATION OF SOLVING PROBLEMS IN THE GAME OF GO

by

Jonathan Fun Lin

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2017

Advisory Committee:
Dr. Michael Fu, Chair
Dr. Steven Marcus, Committee Member
Dr. Jeffrey Herrmann, Committee Member

Dedication

To my family.

Acknowledgements

I would like to acknowledge the following people, all of whom contributed to this thesis.

I would like to thank all members from UMD Go Club for helping me design the model of Go problems. I would like to thank Justin Teng, the former president of UMD Go Club, for operating the Go Club.

I would like to thank Dr. Steven Marcus and Dr. Jeffrey Herrmann for serving on my thesis committee.

Finally, I would like to thank my advisor, Dr. Michael Fu, for his guidance and support.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

MCTS	Monte Carlo Tree Search
UCT	Upper Confidence bounds applied to Trees
MDP	Markov Decision Process
UCB	Upper Confidence Bounds
LCB	Lower Confidence Bounds
AMAF	All Moves As First

Chapter 1: Introduction

1.1 Background

Simulation is a useful technique in systems engineering because it can be used for verification and to compare alternative systems. Verification is an important part of systems engineering. Common verification methods are inspection, analysis, analogy, demonstration, test, and sample. However, when a system is too expensive to test or a system is too complicated to gain analytic results, simulations might be handy. Also, when comparing alternative systems, simulations can provide results of performance with a relatively low cost compared to building an actual system.

Monte Carlo Tree Search can be used for simulating systems where actors make decisions with random outcomes. The most notable examples are the implementation of AI in computer games or board games.

1.2 Comparison of Monte Carlo Tree Search and Minimax Search

While implementing tree-search based AI to games, evaluation functions are important. Evaluation functions help AI to determine how good states and actions are. Traditional AI, e.g. Deep Blue [1], which defeated the world champion in 1997, utilizes evaluation functions to apply minimax search and alpha-beta pruning. However, when applying AI to the game of Go with traditional methods, the result is not promising. Go AI was easily beaten by amateur players in the early development. This is due to two characteristics of Go:

1. The complexity of Go is much higher than Chess. The game state-space complexity of Go is estimated 10^{170} while chess is estimated 10^{47} [2, 3].
2. There is no well-developed evaluation function for Go.

Go AI showed signs of rising when Monte Carlo Tree Search (MCTS) [4, 5] was proposed. Instead of fixed-depth minimax search, MCTS samples the promising states and actions more. Therefore, the search tree grows larger as the sample size increases. Evaluation functions are replaced by the Monte Carlo method [6], which evaluates a state by running simulations. In a simulation, moves are randomly played until the game reaches the end, and the simulation reports a reward from the end state. The Monte Carlo method estimates a state by averaging rewards of simulations. Thus, MCTS can be implemented without any domain-based knowledge, but the performance can be improved with domain-based knowledge. In 2016, AlphaGo [7], a Go AI which uses MCTS with two deep neural networks, beat the top Go player without handicaps.

Even though MCTS has had great success on games with large branching factor, minimax search with alpha-beta pruning still beats MCTS on games like Chess or Checkers [9]. Since MCTS mostly focuses on the promising actions, if there are a lot of shallow traps [9] in a search space, MCTS is less appropriate than minimax with alpha-beta pruning. A shallow trap is a situation where a player will lose within a few moves if an opponent responds correctly. These traps are common in Chess (capturing the king) but less common in games like Go.

1.3 Goal of Thesis

The goal of this thesis is to test the ability of minimax-combined MCTS to detect shallow traps compared to MCTS. Achieving this goal requires the following tasks:

1. Develop a model for the Go problem.
2. Implement MCTS to the Go problem.
3. Propose a minimax-combined MCTS algorithm.
4. Implement the minimax-combined MCTS to Go problem.
5. Compare the algorithms in terms of accuracy and computation.

1.4 Structure of Thesis

This thesis is structured as follows. Chapter 2 provides the background on minimax search with alpha-beta pruning, MCTS, the relation of MCTS and shallow traps. Chapter 3 provides the model of the Go problem and the algorithms for both the and minimax-combined MCTS. Chapter 4 provides the experimental results. Conclusions and future work are described in Chapter 5.

Chapter 2: Literature Review

2.1 Deep Blue

Deep Blue [1] was a great milestone of AI. Deep Blue won a game against the world champion in chess in 1996, but lost 3 times and drew twice. The next year, Deep Blue won by a score of 3.5-2.5. The basic components of Deep Blue are minimax, alpha-beta pruning, and evaluation functions.

First, Deep Blue has a position generator, which allows Deep Blue to search game trees deeper. Then, once it hits a certain depth of the tree, evaluation functions kick in. There are two types of evaluation functions. The first one is simpler but takes fewer computations, and the second one takes more computations but is more accurate. The first evaluation function is just a sum of values of pieces. If the one player has much more value of pieces compared to the opponent, then no further evaluation is needed. However, if the values of pieces of both players are close, then complex evaluation functions will be applied. The second evaluation function is a sum of feature values. Deep Blue recognizes about 8000 patterns, and there are corresponding values to the patterns.

Second, after the positions are evaluated, the value is backed up by minimax and alpha-beta pruning algorithms. In minimax algorithms, there are max and min nodes. Since the white player moves first in chess, we assume that the higher the score, the better the situation for the white player. Therefore, every node that the white player has the next move is a max node because the white player wants to maximize the score, and every node that the black player has the next move is a min

node because the black player wants to minimize the score. This is a recursive process that continues until the value is backed up to the root.

Third, alpha-beta is an algorithm based on minimax search. The central idea is subtrees that cannot influence the root can be pruned. For example, a root (a max node) has two children (min nodes): the value of child 1 is 5, and child 2 is still being explored. If one of the children of child 2 has a value below 5, then child 2 can be pruned. The reason is that child 2 is a min node, so it only updates values that are lower. If child 2 has a value below 5, then it is impossible to have a value greater than 5. Thus, child 1 will always be greater than child 2, so we can prune child 2.

Fourth, Deep Blue has an extended minimax search due to the nature of chess. If the leaf node is at a forcing position (i.e., checkmate or threat to win), the evaluation functions do not work so well. The evaluating current forcing position is not useful because players are expected to play a few moves responding to the threat. Therefore, the leaf node is expanded one more layer, and the expanded position will be evaluated.

2.2 Monte Carlo Tree Search

The methodology of Monte Carlo Tree Search [2] is the core of this proposal. The observations and experiments of this thesis will be conducted under the framework of MCTS to demonstrate why MCTS does not work well with a large number of shallow traps. Therefore, understanding how the MCTS operates is important. In general, the MCTS consists of 6 parts shown in chapter 2.2.1~2.2.6. In short, how simulation can be estimated, how bounds are created, how bounds are applied to tree search will be discussed as below:

2.2.1 Markov Decision Process

Markov decision process models sequential decision problems in fully observable environments using four components:

• S: set of states, with s0 being the initial state.

• A: set of actions.

• T (s, a, s'): transition model that determines the probability of reaching state s' if action a is taken at state s.

• R(s): reward function.

The goal for an MDP problem is to find and optimal policy π which maps states to actions. In other words, a policy specifies what actions should be taken in a given state. Optimal policy means the reward is maximized when decisions are made by optimal policy.

2.2.2 Monte Carlo Method

The Monte Carlo method [8] approximates the analytic value by repeated random sampling. By the law of large number, the empirical mean approximates the expected value as the number rises. Therefore, a reliable estimate can be generated by Monte Carlo method.

2.2.3 Multi-armed Bandit Problem

A multi-armed bandit problem [5, 10] is a sequential decision problem. The player chooses among K arms for each iteration and gets a reward. The goal of the problem is to maximize the accumulated reward. The difficulty is that the distribution of each arm is unknown, so the player estimates the reward by pulling an arm. This

leads to an exploration-exploitation tradeoff problem. Exploitation means pulling the currently best performing arm, and exploration means pulling sub-optimal arms. One wants to do exploitation to maximize the reward but also wants to do exploration in case the current believed best arm is actually sub-optimal.

To deal with multi-armed bandit problems, a concept called regret is introduced. Whenever a player pulls an arm that is not optimal, there is a corresponding regret defined by $R = \mu^* - \mu_i$, where R is the regret, μ^* is the mean of the best arm, and μ_i is the mean of the chosen arm i which is not the best arm. Many bandit algorithms aim to minimize the regret.

2.2.4 Upper Confidence Bounds

Upper confidence bounds [10] (UCB) are useful for multi-armed bandit problems. Since the width of confidence bounds decreases as the sample size increases, upper confidence bounds of sub optimal arms fall below the mean of the best arm as sample sizes increase. When the sample sizes are close, the best arm should be most likely to be pulled. When one or more arms have much smaller sample sizes due to their bad performance, their confidence bounds are wide, so they should have chances to be pulled. Therefore, the exploitation-exploration tradeoff can be applied by choosing the highest UCB for each iteration.

There are many ways to generate upper confidence bounds. UCB1, which sets confidence bounds by Hoeffding's inequality, is one of the well-known ways because of its ease of application and its ability to minimize regret. The algorithm is as follow:

Assume there are K arms with unknown identical independent distributions within [0, 1]. One pulls the arm that maximizes the formula $\bar{X}_j + \sqrt{\frac{2 \ln n}{n_j}}$, where $j \in 1 \ldots, K$, $\bar{X}_j$ is the average reward from arm j, n_j is the times of arm j has been pulled, and n is the overall number of pulls.

2.2.5 Monte Carlo Tree Search Algorithm

Monte Carlo Tree Search [2, 3] is a best-first tree search algorithm. MCTS relies on two concepts for the best-first characteristic. One is that the value of an action can be approximated by random simulations, and the other one is that the value of simulations is useful for the best first policy. MCTS repeats the following four steps until it reaches the stopping condition. The condition can be limited time, memory, or iteration.

1. Selection:

Start from the root node, a child is selected recursively until an expandable node is reached. A node is expandable if it is a non-terminal state and has children unvisited.

2. Expansion:

Add one or several of unexplored children nodes of a leaf node to the tree.

3. Simulation:

Run a simulation from a newly added node to produce an outcome.

4. Backpropagation:

Back up the result of simulation to the parent recursively.

MCTS is a popular algorithm for its following characteristics:

1. Heuristic

Although full-depth minimax tree search does not require any domain-based knowledge, it is quite computation-consuming. If fixed-depth minimax tree search is applied, then evaluation functions are required. On the other hand, MCTS does not require any domain-based knowledge, but the performance of MCTS can be improved with specific knowledge. In short, fixed-depth minimax and MCTS both work with domain-specific knowledge, but only MCTS is workable without any knowledge.

2. Anytime

The search tree is built incrementally in MCTS, and the results are propagated immediately after simulations. This allows MCTS to give a current best solution anytime.

3. Asymmetric

The selection policy allows MCTS to search more on promising nodes. Therefore, the shape of the tree tends to be asymmetric.

2.2.6 Upper Confidence Bounds Applied to Trees

UCT algorithm [5] is a selective policy in MCTS. A child is recursively selected by UCB1 until a terminal or expandable node is reached. UCT keeps the exploration-exploitation tradeoff characteristic from UCB1, and UCT is proved to converge to minimax [11].

2.3 Shallow Traps in MCTS

A player is at risk falling into a shallow trap [9] if there exist a sequence of actions that are guaranteed for the opponent to win the game. Figure 2.3.1 shows an example of shallow traps. If the white player chooses the middle action, the black player can win the game with a correct response. The definition of a level-k shallow trap is that after the player falls into a shallow trap, the opponent has a k-move winning strategy. Therefore, the levels of shallow traps are typically odd numbers, because it is assumed that players do not lose a game on their own move. The study [9] shows that MCTS is able to identify level-3 shallow traps, but it takes an extremely long time to identify level-5 or higher shallow traps.

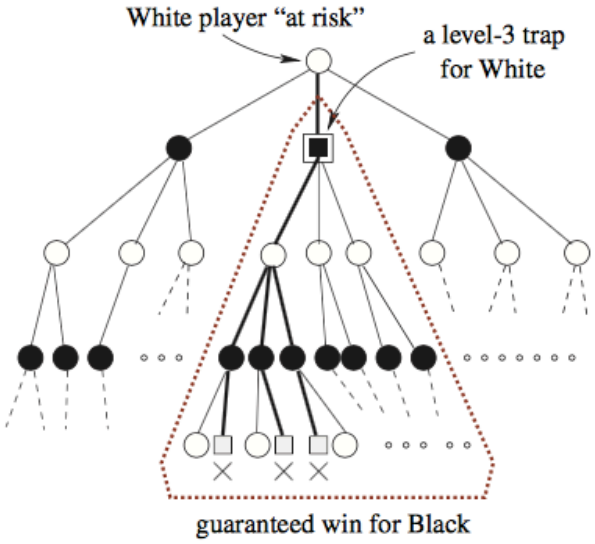


Figure 2.3.1: A demonstration of shallow traps [9]

The frequency of shallow traps varies from game to game. For example, shallow traps occur quite often in Chess compared to the game of Go. The games stop when the king is captured in chess whereas there is no particular rule or pattern to determine the end of Go. Therefore, it is possible that the king will be captured inevitably in a few moves. In contrast, the ending of Go means all positions are either someone's territories or impossible to become territories, which cannot happen in a few moves.

Therefore, shallow traps barely happen in Go. Figure 2.3.2 shows the frequency of different level shallow traps in different board depths. The upper part of the figure is the result of semi-random generated games, and the lower part is the result of games played by grandmasters. In semi-random games, moves are played randomly with probability 1/3 and played with GNU Chess (http://www.gnu.org/software/chess) heuristic with probability 2/3. Here we can see when shallow traps occur more often in end games than the opening (comparing depth 63 to 15). Also, the deeper shallow traps are, the more frequently the shallow traps happen. Finally, grandmasters are good at avoiding shallow traps in the opening, compared to the semi-random generated games.

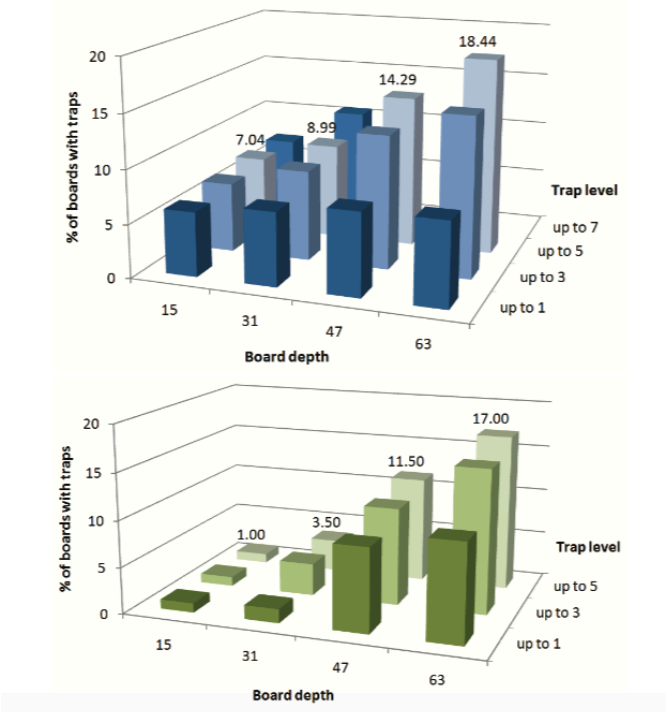


Figure 2.3.2: Frequency of shallow traps in different scenarios. [9]

The ability of UCT to avoid shallow traps is not promising. Figure 2.3.3 shows how many iterations with UCT are needed to detect shallow traps compared to the number of minimax nodes. The result shows that UCT is able to detect level-3 shallow traps given roughly 10 times the number of iterations, but it takes extremely long to detect any shallow traps at higher levels (in some cases, 50 times is not enough). The result shows that 95% of nodes explored are 7 levels deeper when level 7 shallow traps occur.

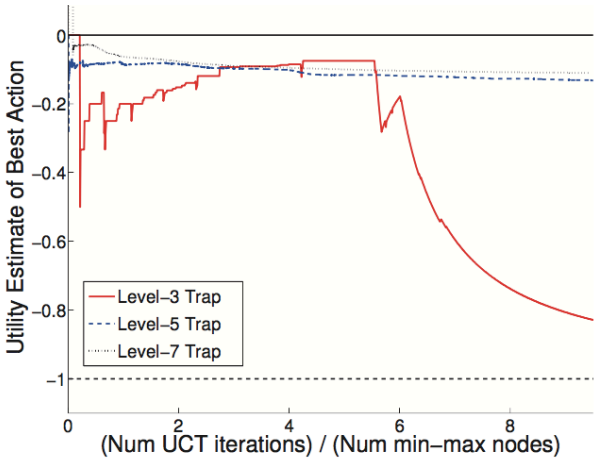


Figure 2.3.3: Avoiding shallow traps with UCT [9]

2.4 Related Work Combining MCTS and Minimax Search

2.4.1 MCTS Solver

MCTS solver [15] finds the theoretical value under the framework of MCTS. When running MCTS solver, not only the results of simulations but also the proven wins and losses are propagated. If the expanded nodes are not in the end state, then the procedure is same as MCTS. However, if the expanded nodes are at the end state, then the proven wins and losses are propagated by the following rules:

If the node is a max node, then

A proven win is backpropagated if one of the children is a proven win.

A proven loss is backpropagated if all of the children are proven losses.

Otherwise, nothing is backpropagated.

If the node is a min node, then

A proven loss is backpropagated if one of the children is proven loss.

A proven win is backpropagated if all of the children are proven wins. Otherwise, nothing is backpropagated.

By the algorithm above, the theoretical values can be backpropagated. Also, proven nodes are no longer searched to improve the efficiency. The experiments show that MCTS solver has a win rate of 65% against MCTS in the game of Connect 4.

2.4.2 MCTS and Minimax Hybrids

To improve the performance of MCTS when shallow traps exist, MCTS with minimax hybrid algorithm has been proposed in [8]. The minimax can be embedded in all four phases of MCTS.

1. Minimax in simulation phases:

In simulation phases, a fixed-depth minimax search is done before every random move. Since no evaluation is given, the minimax can only detect proven wins or losses. Thus, the random simulation will find forced wins or avoid forced losses.

2. Minimax in selection and expansion phases:

In selection and/or expansion phases, a shallow-depth full width minimax search is done. This improves the MCTS by checking immediate descendants of a subset of tree nodes.

3. Minimax in backpropagation phases:

MCTS backpropagates simulation results to parents. What minimax does is to backpropagate proven wins and losses.

2.5 Monte Carlo Tree Search Extension

2.5.1 UCB1-Tuned

UCB1-tuned [10] is a variation of UCB1 which tunes the bounds more finely than UCB1. It replaces the formula of upper confidence bounds $\sqrt{\frac{2\ln n}{n_j}}$, with

$$\sqrt{\frac{\ln n}{n_j}\min\{\frac{1}{4}, V_j(n_j)\}}$$

where

$$V_j(s) = \left(\frac{1}{2}\sum_{\tau=1}^{s} X_{j,\tau}^2\right) - \overline{X}_{j,s}^2 + \sqrt{\frac{2\ln t}{s}}$$

which means that machine j, which has been played s times during the first t plays, has a variance that is at most the sample variance plus $\sqrt{\frac{2\ln t}{s}}$. Although there is no analytical way to prove a regret bound for UCB1-tuned, experiments show that UCB1-tuned performs better than UCB1.

2.5.2 Best Arm identification algorithm

UCB is an accumulated regret minimizing technique. A suitable example is medical treatment. There are several treatments, and their effectiveness is unknown. Therefore, the objective is to do as little accumulated damage as possible to the patients. The regrets happen during the exploration. However, in the case of making decisions in the game of Go, the regrets happen after the exploration. Only the final

decision matters. Best arm identification algorithm [12] has a highly exploring policy [13], UCB-E. The following is the algorithm:

$$For\ i \in \{1, \dots, K\}, let\ B_{i,s} = \hat{X}_{i,s} + \sqrt{\frac{a}{s}}\ for\ s \geq 1\ and\ B_{i,0} = +\infty$$

$$For\ each\ iteration\ t = 1, \dots, n: Draw\ I_t \in argmax\ B_{i,s}$$

where the parameter satisfies: $0 \leq a \leq \frac{\frac{25}{36}(n-K)}{H_1}$, and machine i has been played s times, and K is the number of arms, and n is the total times of plays.

$$H_1\ is\ defined\ as: \sum_{i=1}^{K} \frac{1}{\Delta_i^2}\ and\ \Delta_i = \mu^* - \mu_i$$

2.5.3 All Moves As First (AMAF)

ALL MOVES AS FIRST [16] is a history heuristic which uses the information in simulations for the selection phase. In MCTS, the simulation result will only be backpropagated to the node triggering the simulation and all of its parent nodes. AMAF backpropagates the simulation result to siblings of the node triggering the simulation and all of the parent nodes. Take Figure 2.5.1 for example. Actions C1 and A1 are selected by UCT, and B1, A3, and C3 are the actions of the simulation. UCT only backpropagates the result to C1 and A1, but AMAF backpropagates the result to C1, A1, B1, A3, and C3.

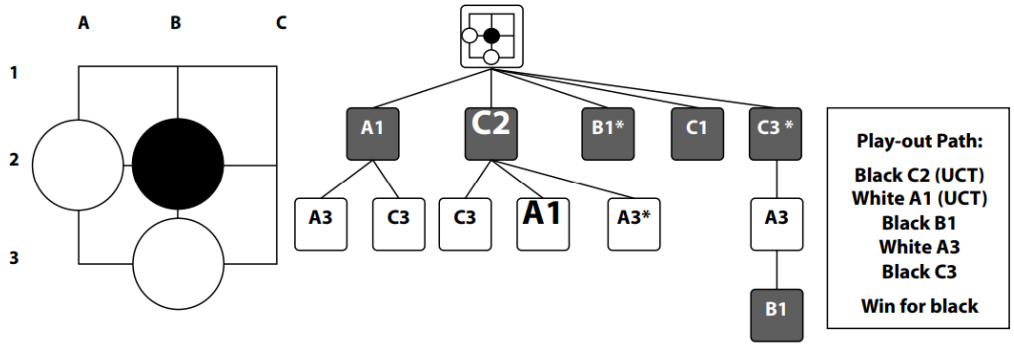


Figure 2.5.1: An example of AMAF [16].

2.5.4 Last-Good-Reply Policy

The Last Good Reply Policy [17] views MCTS as a machine learning technique. In each simulation, actions are selected according to similar game states. After the simulation, if the result is successful, the move will be adopted as good reply. If there was a good reply and another good reply appears, the last good reply will be adopted. Figure 2.5.2 illustrates an example of the last good reply policy. The result of the first simulation is black's win. Therefore, all the replies (C replies to B, E replies to D, and G replies) are adopted. The result of the second simulation is white's win. The same procedure is taken. The result of the third simulation is black's win. Two rseplies exist to action B (C to B and D to B). Only the last reply (D to B) will be adopted, and the old one is forgotten.

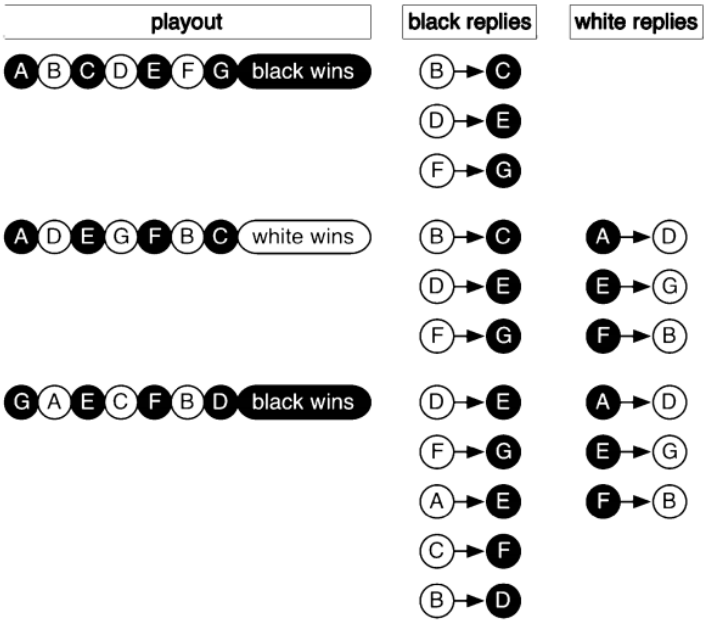


Figure 2.5.2: An example of last good reply policy [17].

Chapter 3: Approach

3.1 Inspiration for minimax-combined MCTS

The idea of minimax-combined MCTS comes from a very simple example, level one trap. In this example, we have a game with branching factor 20. The root node (black player's turn) has 20 children. 19 children nodes are normal moves with a mean of 0.5, and the last child node is a trap. The trap node (white player's turn) has 19 children nodes (wrong moves for white) with a mean of 0.7, and the last child node (correct move for white) has a mean of 0.3. Notice that no proven win or loss is within 2 layers, so the node is not a shallow trap. Therefore, both MCTS and MCTS and minimax hybrid do not work here. Nonetheless, the node is indeed a trap move because the opponent has a response that leads himself to a good situation. The logic is explained in the next paragraph.

If the black player plays a normal move, the result is 0.5. If the black player plays a trap move and the white player plays a wrong move, the result is 0.7. If the black player plays a trap move and the white player plays the correct move, the result is 0.3. The black player should assume the white player will pick the correct move, because it requires only one step to figure out the results. Therefore, the black player should choose the normal moves instead of the trap move.

If one uses the minimax search with a fixed depth = 2, only $20+20^2=420$ nodes are needed to be explored to find the trap. However, it takes a lot of simulations for MCTS to detect the trap. The result is shown in the following tables.

Name of the node	# of simulations	Mean	Upper confidence bounds	Lower confidence bounds
Root	479	0.5117	X	X
Normal moves	23	0.5	0.9828	X
Trap	42	0.6333	0.9906	X
Wrong moves for white	2	0.7	X	-0.5741
Correct move for white	4	0.3	X	-0.6009

Table 3.1.1: Simulation results after 479 simulations

Name of the node	# of simulations	Mean	Upper confidence bounds	Lower confidence bounds
Root	2976	0.5031	X	X
Normal moves	145	0.5	0.7189	X
Trap	221	0.5416	0.7189	X
Wrong moves for white	9	0.7	X	-0.0218
Correct move for white	50	0.3	X	-0.0062

Table 3.1.2: Simulation results after 2976 simulations

Name of the node	# of simulations	Mean	Upper confidence bounds	Lower confidence bounds
Root	7092	0.4999	X	X
Normal moves	355	0.5	0.6473	X
Trap	347	0.4983	0.6472	X
Wrong moves for white	13	0.7	X	0.0746
Correct move for white	100	0.3	X	0.0748

Table 3.1.3: Simulation results after 7092 simulations

From the simulation results, we can see that MCTS works properly. The promising nodes are explored more, and the correct decision is finally made. The only problem is that too many simulations are needed to detect traps. It takes roughly

twenty times more simulations than minimax search. The potential reason for such a low efficiency to detect traps is that MCTS evaluates a node by the average scores of simulations. Therefore, MCTS still takes some time to converge to the theoretical value.

The purpose of minimax-combined MCTS is very simple. The result of MCTS under a subtree can be treated as an evaluation function. Thus, a minimax search can identify not only shallow traps but also good/bad situations. On the other hand, MCTS solver and MCTS and minimax hybrid can only detect proven wins and losses but not good/bad situations.

3.2 Rules of Go

In the game of Go, the board is a plane grid of 19 horizontal lines and 19 vertical lines (figure 3.2.1, an example of 9 by 9 board). Two players (black and white) place one stone in turns, and the stones are placed on the intersections on the board. Once the stone is placed, it cannot be moved, but it can be captured and taken away from the board. The objective is to control more territories than the opponent. Territories are defined as areas enclosed by own stones, where every opponent's stones within the territories are expected to be captured eventually (figure 3.2.2).

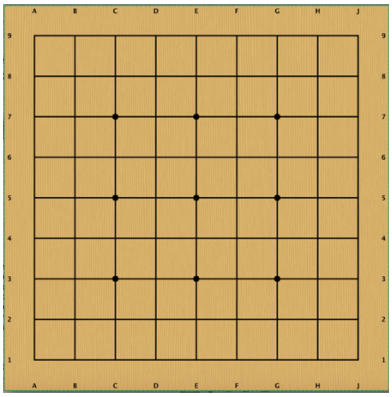


Figure 3.2.1: A 9 by 9 board

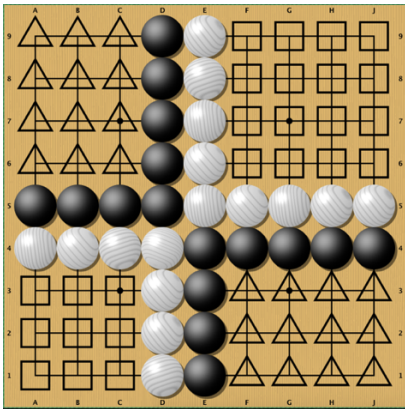


Figure 3.2.2: A finished game. The triangle marks are considered black's territories, and the square marks are considered white's territories.

3.2.1 Connection and Capture

Connected stones are considered to be a group. When a group is captured, all stones are removed. Stones are connected with adjacent stones, and only vertical and horizontal direction counts. Figure 3.2.3 shows examples of connection. Every adjacent empty point to a group is considered its liberty (figure 3.2.4). If a group has no liberty, it is captured (figure 3.2.5). Players cannot fill their last liberty (figure 3.2.6), which is a suicide unless that move is able to capture opponent's groups (figure 3.2.7).

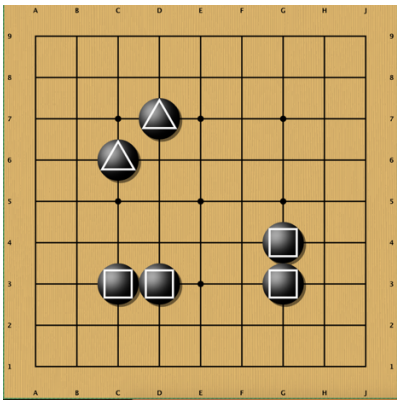


Figure 3.2.3: Examples of connections. The stones marked with triangles are not connected while the stones marked with squares are connected.

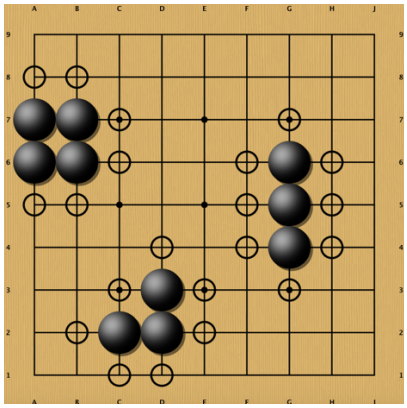


Figure 3.2.4: Examples of liberty. The circle-marked spaces are liberties of black groups.

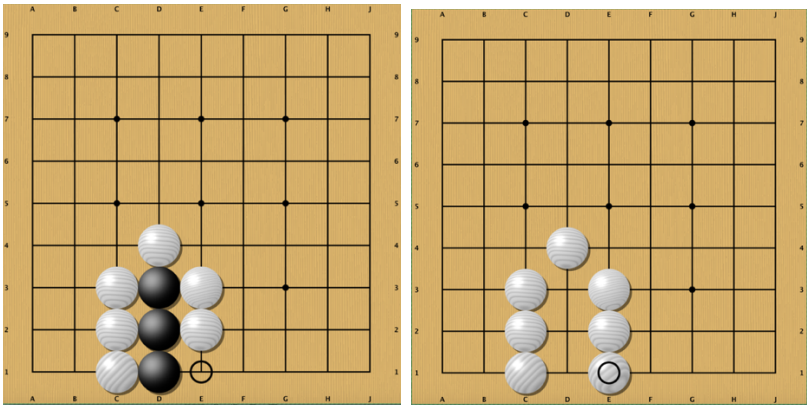


Figure 3.2.5: An example of captures. On the left side, black groups only have one liberty. When white places a stone at the marked space, the black group is captured.

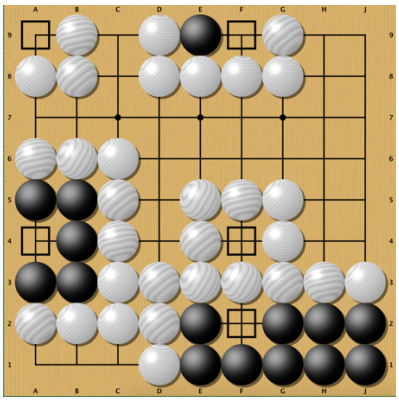


Figure 3.2.6: Examples of suicides. The black player is not allowed to play the positions marked with squares, because the moves make the groups have no liberty.

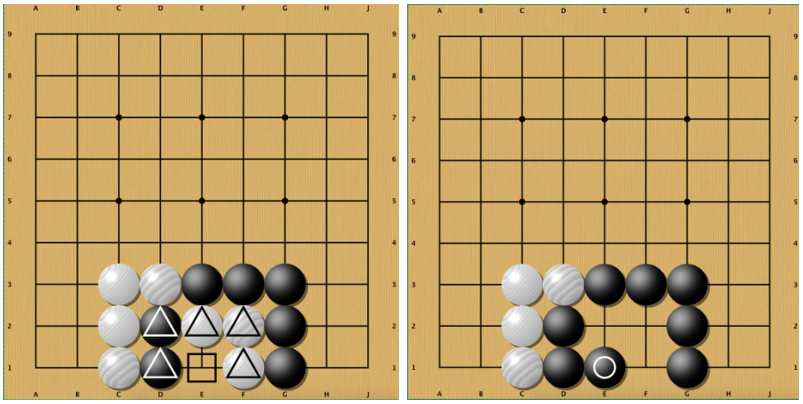


Figure 3.2.7: An example of captures. The stones marked with triangles have only one liberty. If either player plays the square, both groups have no liberty. By the rules of Go, whoever makes the move can capture the opponent's groups. The right-hand board shows the

3.2.2 Ko and Ko fight

A Ko is a special situation in Go where both players can capture opponent's stones and create an endless loop. The rule of Ko is to prevent immediate repetition. If a move creates a position same as the last previous position, then the move is illegal (Figure 3.2.8). Because of this rule, when Ko happens, players will play moves that opponents want to defend, which is called Ko threat. If so, the player can capture the opponent's stone again because the position is changed (Figure 3.2.9). This

process is called Ko fight. In general, a player will win the Ko fight and gain some profit while the other player gains profit from the Ko threat.

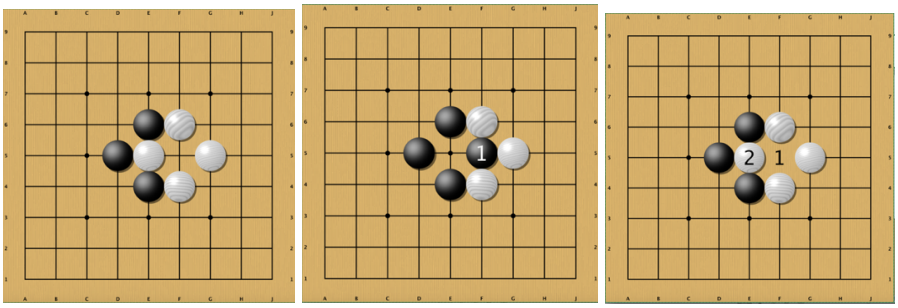


Figure 3.2.8: Endless cycle of Ko. Without the rule of Ko, players can keep playing move 1 and 2, and create a position same as previous one.

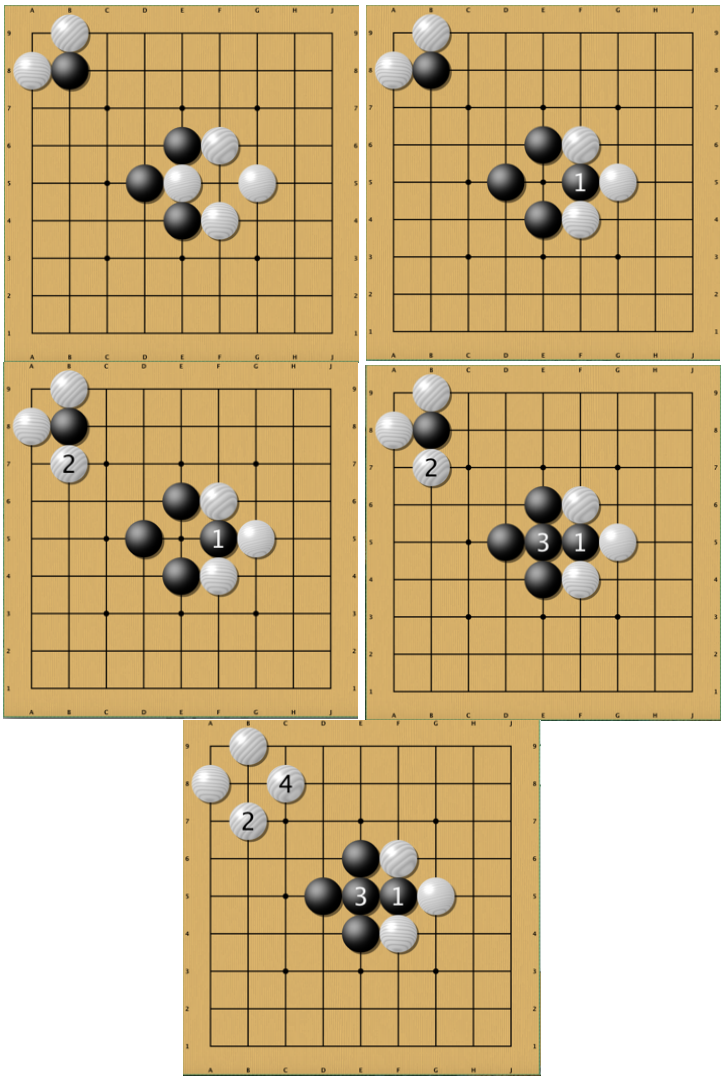


Figure 3.2.9: A Ko fight. Black starts a Ko fight, and white finds a Ko threat. Black ignores the Ko threat, and white gains from the Ko threat.

3.2.3 Alive Groups

Some groups can never be captured, even if opponents can play several moves in a row. These groups are considered alive. Basically, groups with two eyes or more are alive. An eye is defined as a space surrounded by a player's own stones (Figure 3.2.10). When a group has two eyes, its liberty cannot be decreased down to 0 because it is a suicide for the opponent when they put stones in the eyes (Figure 3.2.11).

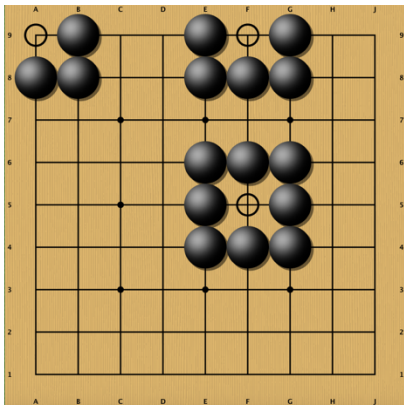


Figure 3.2.10: Examples of eyes. The circle-marked spaces are defined as eyes.

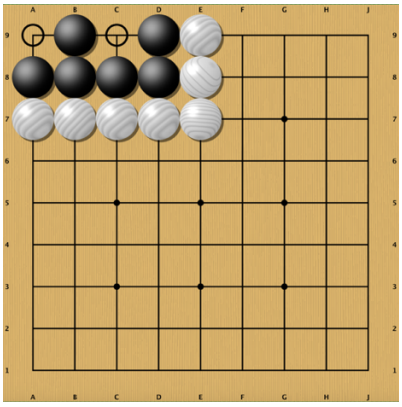


Figure 3.2.11: Example of alive group. The group has two eyes, and white cannot play either spaces because suicide is not allowed.

3.2.4 Seki

Seki, or mutual life, is a situation that groups of both players do not have two eyes, but they are also not able to capture the opponents' stones. The most classic situation of Seki is that both groups have two mutual liberties, so when one player fills a liberty, the other player is able to capture (figure 3.2.12). Therefore, no one wants to take the first move, and no groups can be captured.

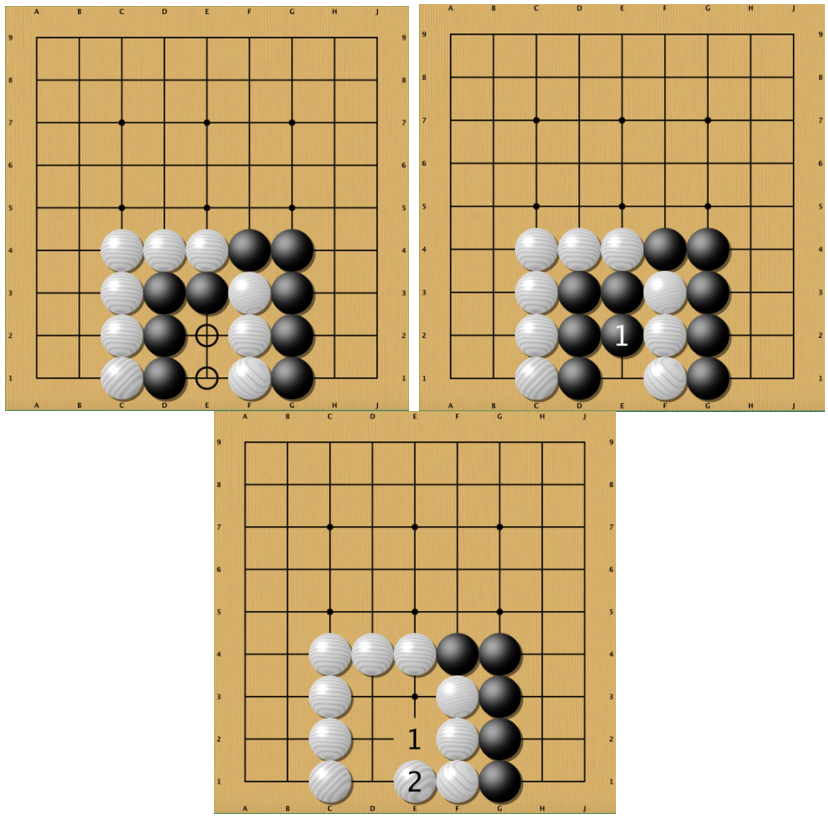


Figure 3.2.12: Examples of Seki. Both players do not want to play the marked spaces. If one plays there, his group will be captured, and vice versa.

3.3 Model of Go Problem

There are several types of Go problems. This model is created to solve the problems that involve capturing opponent's stones or avoiding one's own stones from being captured. When capturing opponent's stones, the player usually gains the

territories where the stones are removed. As a result, capturing stones can be seen as a sub-goal of winning a game, which is very important in the game of Go.

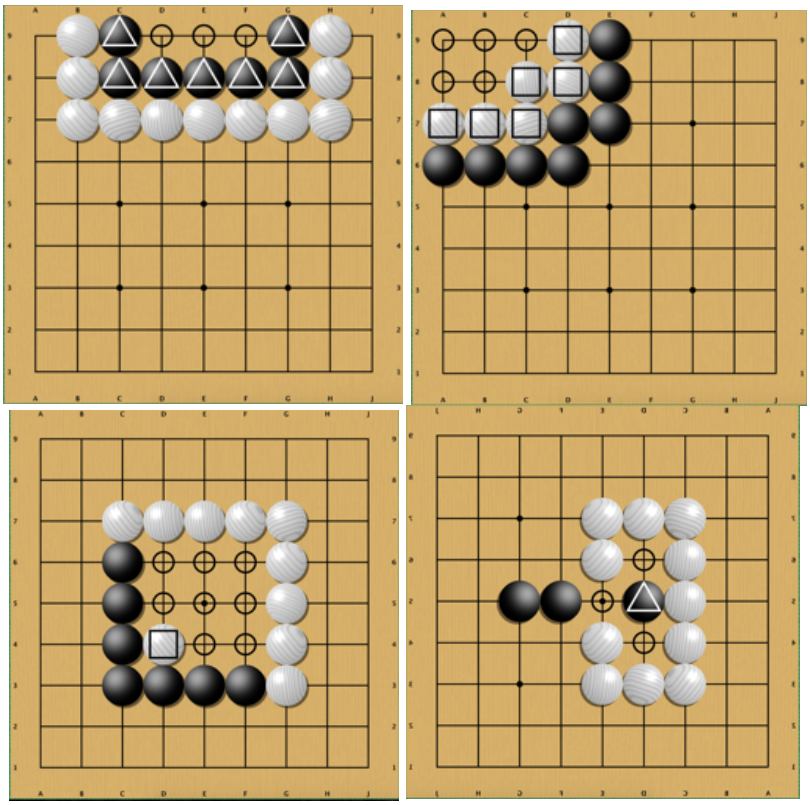
3.3.1 Setup

The Go problem considered in this section is from the black player's perspective. Therefore, actions, objectives, and scores are from the black player's perspective. The black player has the first move. The higher the score is, the better the condition black player is in (on the other hand, the worse condition the white player is in). If the objective is to save groups, the black player has to save groups.

Go problems discussed in this thesis only include two objectives, capturing stones and avoiding stones from being captured. The problems fall into 5 sub-problems: saving groups, killing groups, cutting groups, connecting groups, and capturing race (Figure 3.2.13). In cutting and connecting groups problems, stones are inevitably captured if they lose the connection. In capturing race problems, if opponent's stones are not captured, then one's own stones are inevitably captured. To sum up, the objectives of the five sub-problems are either capturing stones and avoiding stones from being captured, and the only difference is the scenario. In addition, Go problems usually do not involve the entire board, and some groups are assumed alive to simplify the problems. The following are the setup:

1. Decide problem mode: Saving groups or killing groups
2. Set positions: putting black and white stones on the board according to the problem

3. Determine feasible area: Determine area which is legal to play. The problem must have an ending (either groups are saved or captured) just by playing within feasible area.
4. Determine important positions: Determine which stones are supposed to be saved or captured. If important stones at important positions are saved or captured, then the problem is ended.
5. Determine alive groups: For connecting problems, the objective is to connect groups instead of making two eyes. Therefore, some groups are marked as alive, and other groups are considered alive when they connect to alive groups. Thus, connecting problems has the same standard of scoring as other problems: alive (connected), seki, and dead.



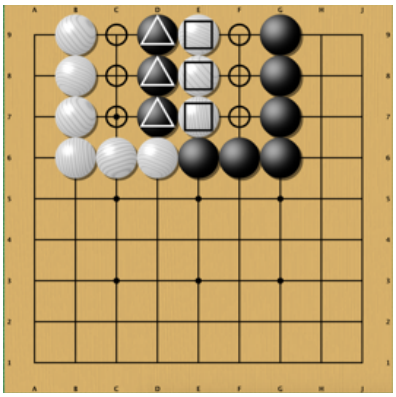


Figure 3.2.13: Five types of Go problem. From left to right, top to bottom: saving groups, killing groups, cutting groups, connecting groups, and capturing race. The triangle marked stones are the groups to be saved, and the square marked stones are the groups to be killed, and the circle marked spaces are the feasible areas.

3.3.2 Scoring

The problem ends when important groups are alive, Seki, or captured. Based on my experience, scores of results are shown in Table 3.3.1. Notice that Seki, or mutual life, seems to be an even result for both players. However, the result is slightly better for the player who is saving groups.

Avoiding being captured	Score		Capturing opponent's groups	Score
Alive	1		Alive	0
Seki	0.7		Seki	0.3
Dead (Captured)	0		Dead (Captured)	1

Table 3.3.1: Scores of results

However, the Ko fight might happen in Go problems. As mentioned, players find Ko threat "somewhere else" and hope their opponents will respond. However, when doing local problems, there is no clue of how opponents will respond. Thus, assumptions are made for scoring. First, when Ko fight happens, players are allowed to fight back immediately, which is against the rule. Second, after a player fights back

immediately, that player can fight back the Ko immediately, but the opponent cannot fight back at all. Third, the player who fights back the Ko has a penalty.

The assumptions are based on the Ko fight pattern. When a player fights back the Ko, we assume the player wins the Ko, but meanwhile winning the Ko means that the opponent can play two moves in a row somewhere else. Therefore, the player has a score penalty because the opponent gains something somewhere else. Based on the assumptions, we have the modified score:

	Black wins the Ko	No one wins the Ko	White wins the Ko
Alive	0.5	1	1
Seki	0.3	0.7	0.85
Dead	0	0	0.5

Table 3.3.2: Scores when saving groups

	Black wins the Ko	No one wins the Ko	White wins the Ko
Alive	0	0	0.5
Seki	0.15	0.3	0.7
Dead	0.5	1	1

Table 3.3.3: Scores when capturing groups

3.3.3 Ending Pattern Recognition

The model has to decide the result of the problem: alive, Seki, or dead.

1. Dead: When important stones are captured, the result is dead.

After a move check each position marked as important

If the purpose is to save groups

If any of important positions is not black stone

Return dead

Else if the purpose is to kill groups

If any of important positions is not white stone

Return dead

If above statements do not return dead

Return not dead

2. Seki: When both players pass continuously over 3 times, the result is Seki.

If continuous passes are over 3 times

Return Seki

Else

Return not Seki

3. Alive: Recognizing alive patterns is the most difficult. According to rules, a group with two eyes is alive, and eyes are defined as a space surrounded by own stones. However, there are true and false eyes which do not have simple ways to determine. Therefore, instead of recognizing eyes, the player who is capturing is given infinite moves in a row. If the important groups can be captured, then the group is not alive.

    Fill the board with stones of the player who is capturing expect positions of eyes of the player who is being captured

    Do

        Fill every position of eyes one at a time

        If important stones are captured

            Return not alive

    While some stones are captured

    Return alive

3.3.4 Extra Rules

The extra rules are not necessary but simplify the model of Go problems and prevent endless cycles.

1. Situations allowing passing: We don't want players to pass in any situation. In fact, we want players to pass when Seki happens. Notice that satisfying this situation does not mean Seki, but situations are always satisfied when Seki happens.

    Pass rule 1: if a player has no place to play, the player is allowed to pass.

Pass rule 2: if every feasible place is either reducing own liberty to 1 or filling own eyes, the player is allowed to pass.

2. Winning by passing: Sometimes opponents can keep playing useless moves and make the game endless. Therefore, if a player passes three times more than the opponent, the player wins.
3. Limited Ko advantages: In the model, when a player wins a Ko, the player is assumed to win Ko in the future. However, this advantage is limited to 5 times. After 5 Ko, the player wins the Ko cannot fight Ko back anymore.
4. Not filling true eyes: Players are not allowed to fill true eyes.

3.4 Algorithm of MCTS

Although the model of MCTS is discussed in chapter 2, the model is for general MDP problems. The main difference between MDP problems and Go problems is that Go problems are two-player zero-sum games or adversarial games. Therefore, the selection phase is different. When the black player has the move, the player wants to pick moves with higher scores. When the white player has the move, the player wants to pick moves with lower scores. This is different from MDP problems where the purpose is either maximizing or minimizing rewards. Also, no policy is applied in this thesis.

3.4.1 Selection

In MCTS, UCT is applied in the selection phase. For max nodes (black player's turn), the child node with the highest upper confidence bound (UCB) is picked. For min nodes (white player's turn), the child node with the lowest lower

confidence bound (LCU) is picked. The formula to calculate UCB and LCB is $\bar{X}_j \pm \sqrt{\frac{2\ln n}{n_j}}$, where $j \in 1 \ldots, K$, $\bar{X}_j$ is the average reward from arm j, and n_j is the times of arm j has been pulled, and n is the overall number of pulls. The algorithm keeps selecting children nodes until it reaches terminated nodes, or it reaches nodes whose children nodes are all unexplored. If unexplored nodes are reached, the algorithm enters expansion phase. If terminated nodes are reached, the algorithm enters backpropagation phase, which propagates the score of the terminated node.

3.4.2 Expansion

When expanding a node, all of its children nodes are expanded. Expanded nodes are added to the search tree, and all of them will be simulated.

3.4.3 Simulation

Moves are uniformly randomly played until reaching the end state. After the simulation, the score of the simulation will be propagated.

3.4.4 Backpropagation

The result of simulations will be returned to parent nodes recursively from leaf nodes to the root node. For nodes on the returning path, the times of simulations are increased by 1, and the total score is increased by the score of simulation.

3.4.5 Decision

The children node of root nodes with the highest mean is chosen as the final decision.

3.5 Algorithm of Minimax-combined MCTS

The framework of minimax-combined MCTS is identical to MCTS. It has four repetitive phases and a final decision. However, the ways algorithms select nodes and backpropagate values are different. Two terms are introduced here:

1. Minimax threshold: The minimax threshold indicates whether a node is included in the minimax search. A node meets the minimax threshold if the node is visited enough times. In experiments, the minimax threshold = minimax threshold parameter * branching factor. For example, a node has 5 children and the minimax threshold parameter =100. It meets the minimax threshold if it is visited 500 times.
2. Minimax node: The best leaf node which can be reached by the minimax search on the mean from a certain node. If the node does not meet the minimax threshold, the minimax node is itself.

3.5.1 Notation

$$N^{C,i}: the\ ith\ child\ of\ the\ node$$

$$N^{M}: the\ minimax\ node\ of\ the\ node$$

$$M_N: the\ mean\ of\ the\ node\ N$$

$$U_N: the\ UCB\ of\ the\ node\ N$$

$$L_N: the\ LCB\ of\ the\ node\ N$$

$$MT: the\ minimax\ threshold$$

3.5.2 Selection

$$NextChild = NULL$$

$$CurrentValue = M_{N^{C,0}M}$$

$$For\ i\ from\ 1\ to\ the\ number\ of\ children - 1$$

$$If\ black's\ turn$$

$$If\ U_{N^{C,i}M} > CurrentValue$$

$$CurrentValue = U_{N^{C,i}M}$$

$$NextChild = N^{C,i}$$

$$If\ white's\ turn$$

$$If\ L_{N^{C,i}M} < CurrentValue$$

$$CurrentValue = L_{N^{C,i}M}$$

$$NextChild = N^{C,i}$$

$$NextChild\ selects\ its\ children\ with\ the\ same\ process$$

3.5.3 Expansion

The expansion process is the same as MCTS.

3.5.4 Simulation

The simulation process is the same as MCTS.

3.5.5 Backpropagation

$$Number\ of\ simulation + +$$

$$Total\ Score += Simulaiton\ result$$
$$Update\ children's\ UCB\ and\ LCB$$

$$If\ V_N < MT$$
$$N^M = N\ (the\ node\ itself)$$
$$Return$$

$$N^M = N^{C,0^M}$$
$$CurrentValue = M_{N^{C,0^M}}$$
$$For\ i\ from\ 1\ to\ number\ of\ children - 1$$
$$If\ black's\ turn$$
$$If\ M_{N^{C,i^M}} > CurrentValue$$
$$CurrentValue = M_{N^{C,i^M}}$$
$$N^M = N^{C,i^M}$$
$$If\ white's\ turn$$
$$If\ M_{N^{C,i^M}} < CurrentValue$$
$$CurrentValue = M_{N^{C,i^M}}$$
$$N^M = N^{C,i^M}$$

3.5.6 Decision

The child node with the highest visit times is picked as the final decision.

Chapter 4: Experiment

4.1 Experiment Process

The experiment evaluates the performance of MCTS and minimax-combined MCTS by the number of simulations needed to consistently pick the correct move. That means only one correct move exists in each problem, and the correct move is known. Every time a backpropagation is done, the program will pick one move. MCTS picks the child with the highest mean, and minimax-combined MCTS picks the most frequently visited child. The definition of consistent pick will be discussed in the last paragraph of this section.

Experiments are conducted in the following scenarios:

1. Level-3 shallow trap
2. Level-5 shallow trap
3. Simple problems
4. Complex problems
5. Complex problems with multiple scores

The experiments are conducted in the following steps:

1. Run a long enough simulation to make sure MCTS converges to correct moves.
2. Run multiple MCTS and minimax-combined MCTS with an exploration parameter = 1.414 and a minimax threshold = 50 * branching factor.

3. Do paired t-tests to see if both algorithms are significantly different from each other.

Determine the consistent correct pick:

The consistent pick involves manual tracking. There are two requirements to decide whether the pick is consistent.

1. The correct move is continuously picked many times.
2. The mean of the correct move asymptotically approaches the theoretical value.

```
Number of Simulation Spent:  7
Chosen Action:               8 6

Number of Simulation Spent:  13
Chosen Action:               8 6

Number of Simulation Spent:  19
Chosen Action:               8 6

Number of Simulation Spent:  25
Chosen Action:               8 6

Number of Simulation Spent:  31
Chosen Action:               8 6

Number of Simulation Spent:  37
Chosen Action:               8 6

Number of Simulation Spent:  43
Chosen Action:               7 5

Number of Simulation Spent:  49
Chosen Action:               7 5

Number of Simulation Spent:  54
Chosen Action:               8 6

Number of Simulation Spent:  59
Chosen Action:               7 5

Number of Simulation Spent:  64
Chosen Action:               6 4

Number of Simulation Spent:  69
Chosen Action:               7 5

Number of Simulation Spent:  74
Chosen Action:               7 5
```

```
Number of Simulation Spent:  2557
Chosen Action:               7 5

Number of Simulation Spent:  2559
Chosen Action:               7 5

Number of Simulation Spent:  2561
Chosen Action:               7 5

Number of Simulation Spent:  2562
Chosen Action:               7 5

Number of Simulation Spent:  2563
Chosen Action:               7 5

Number of Simulation Spent:  2564
Chosen Action:               7 5

Number of Simulation Spent:  2565
Chosen Action:               7 5

Number of Simulation Spent:  2568
Chosen Action:               7 5

Number of Simulation Spent:  2570
Chosen Action:               7 5

Number of Simulation Spent:  2571
Chosen Action:               7 5

Number of Simulation Spent:  2572
Chosen Action:               7 5

Number of Simulation Spent:  2574
Chosen Action:               7 5

Number of Simulation Spent:  2575
Chosen Action:               7 5
```

Figure 4.1.1: List of picks. The picks change often in the beginning of MCTS (left list). MCTS picks action (7,5) every time from 1024 simulations (right list).

Children Action		# of sim.	Mean	UCB	LCB
6	4	139	0.46	0.78	0.14
6	5	160	0.48	0.78	0.18
6	6	61	0.28	0.76	-0.21
7	4	45	0.2	0.76	-0.36
7	5	368	0.59	0.79	0.39
7	6	255	0.55	0.78	0.31
8	6	248	0.54	0.78	0.3

Figure 4.1.2: After 1276 Simulations

Children Action		# of sim.	Mean	UCB	LCB
6	4	139	0.46	0.8	0.12
6	5	160	0.48	0.8	0.17
6	6	61	0.28	0.79	-0.23
7	4	45	0.2	0.79	-0.39
7	5	1934	0.82	0.91	0.72
7	6	255	0.55	0.8	0.3
8	6	248	0.54	0.79	0.29

Figure 4.1.3: After 2842 simulations

Children Action		# of sim.	Mean	UCB	LCB
6	4	139	0.46	0.82	0.1
6	5	160	0.48	0.82	0.15
6	6	61	0.28	0.82	-0.26
7	4	45	0.2	0.83	-0.43
7	5	7103	0.92	0.97	0.87
7	6	255	0.55	0.81	0.28
8	6	248	0.54	0.81	0.27

Figure 4.1.4: After 8011 simulations

Figures 4.1.1 through 4.1.4 show that correct move (7,5) is picked every time from 1024 simulations spent. They also show that the results asymptotically approach the theoretical value 1.

4.2 Level-3 Shallow Trap

Figures 4.2.1 to 4.2.4 illustrate an example of level-3 shallow traps. Figure 4.2.1 shows the position of the stones. The objective is to connect the stones marked with circles. Figure 4.2.2 shows the correct move and the shallow traps. The position marked with a square is the correct move, and the position marked with triangles are the shallow traps. Figures 4.2.3 and 4.2.4 demonstrate how a shallow trap happens.

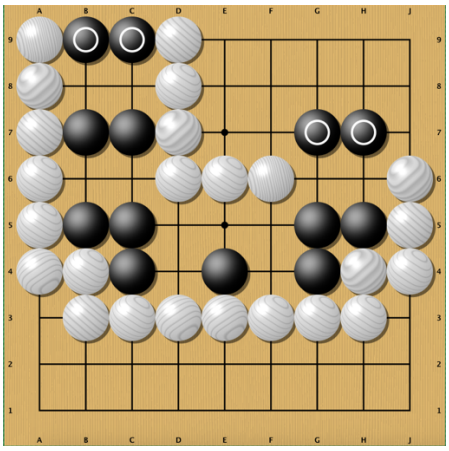


Figure 4.2.1: The objective is to connect marked black stones

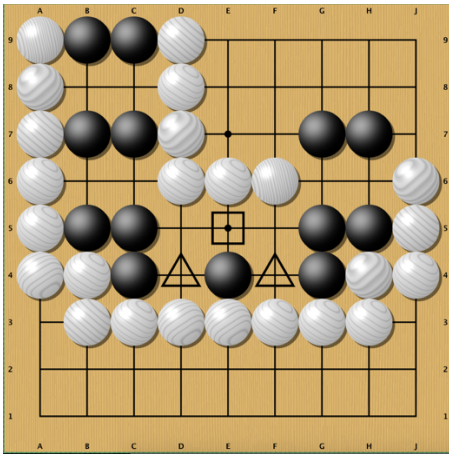


Figure 4.2.2: The square is the correct move. The triangles are the shallow traps.

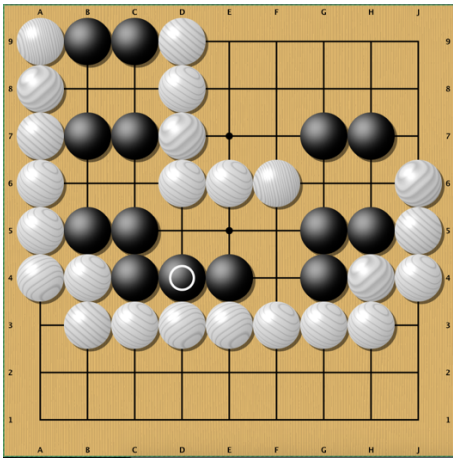


Figure 4.2.3: A demonstration of falling into a shallow trap

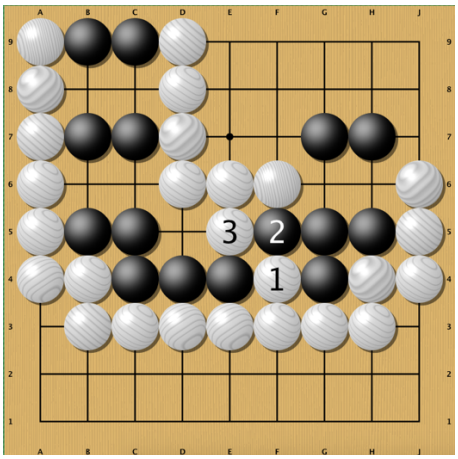


Figure 4.2.4: The white player cuts off black stones in three moves.

The theoretical score of the correct move is 1, and any other moves are 0. The shallow traps are 3 level deep, and the correct move is 11 level deep. Table 4.2.1 shows the computation needed of MCTS and minimax-combined MCTS to pick the correct move (minimax threshold parameter = 50). Table 4.2.2 shows the paired t-test under 95% confidence interval. The confidence interval does not include 0, showing that the minimax-combined MCTS needs significantly fewer computations to pick the correct move.

	Average computations needed	Standard deviation
MCTS	48803	7792
Minimax-combined MCTS	7304	1415

Table 4.2.1: The average computations needed of MCTS and minimax-combined MCTS

Lower bound	Upper bound
34662	48336

Table 4.2.2: Paired t-test (MCTS – Minimax MCTS)

4.3 Level-5 Shallow Trap

Figures 4.3.1 to 4.3.4 illustrate an example of level-5 shallow traps. Figure 1 shows the position of the stones. The objective is to connect the stones marked with circles. Figure 4.3.2 shows the correct move and the shallow traps. The position marked with a square is the correct move, and the position marked with triangles are the shallow traps. Figures 4.3.3 and 4.3.4 demonstrate how a shallow trap happens.

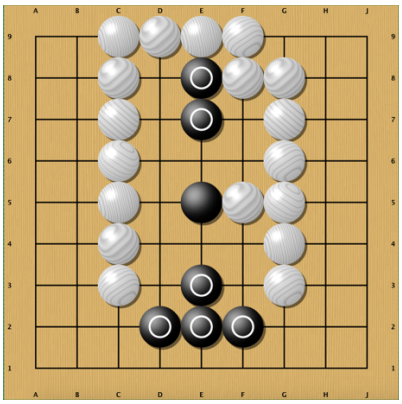


Figure 4.3.1: The objective is to connect marked black stones.

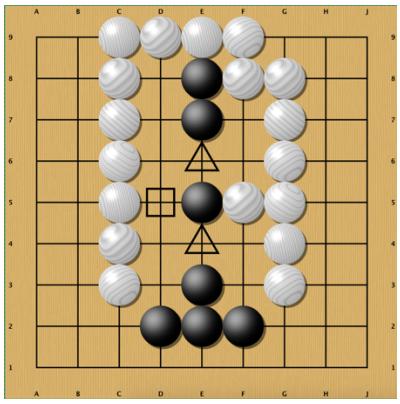


Figure 4.3.2: The square is the correct move. The triangles are the shallow traps.

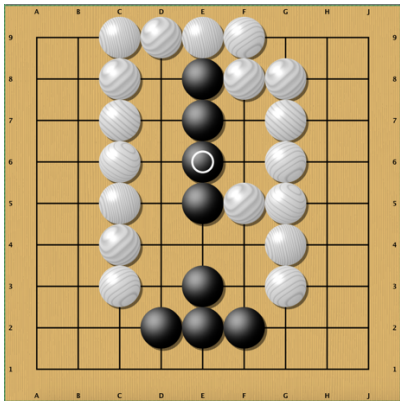


Figure 4.3.3: A demonstration of falling into a shallow trap.

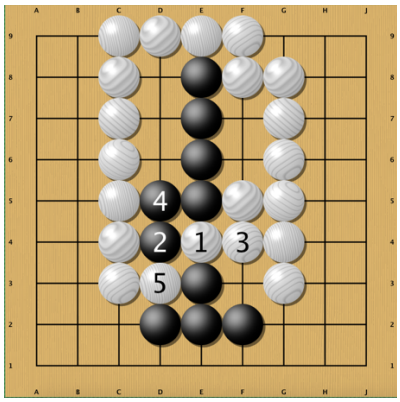


Figure 4.3.5: The white player can cut off black stones in 5 moves.

The theoretical score of the correct move is 1, and any other moves are 0. The shallow traps are 5 level deep, and the correct move is 13 level deep. Table 4.3.1 shows the computation needed of MCTS and minimax-combined MCTS to pick the correct move (minimax threshold parameter = 50). Table 4.3.2 shows the paired t-test

under 95% confidence interval. The confidence interval does not include 0, showing that the minimax-combined MCTS needs significantly fewer computations to pick the correct move.

	Average computations needed	Standard deviation
MCTS	49786	7761
Minimax-combined MCTS	23067	7384

Table 4.3.1: The average computations needed of MCTS and minimax-combined MCTS

Lower bound	Upper bound
19363	34076

Table 4.3.2: Paired t-test (MCTS – Minimax MCTS)

4.4 Simple Problems

This section discusses the performance of minimax-combined MCTS when shallow traps do not exist. Simple problems are defined as problems that need 500~5000 computations for MCTS to pick the correct move.

Problem 1:

The shallow traps are 5 level deep, and the correct move is 9 level deep.

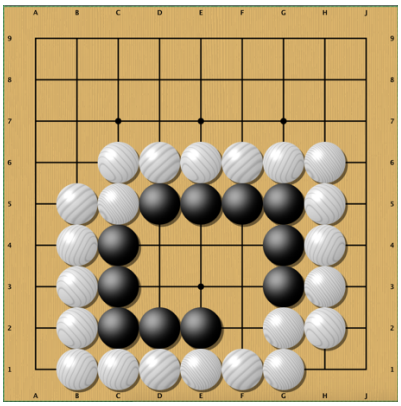


Figure 4.4.1: The position of problem 1

	Average computations needed	Standard deviation
MCTS	1153	250
Minimax-combined MCTS	1016	303

Table 4.4.1: The average computations needed of MCTS and minimax-combined MCTS

Lower bound	Upper bound
-80	354

Table 4.4.2: Paired t-test (MCTS – Minimax MCTS)

Problem 2:

The shallow traps are 3 level deep, and the correct move is 9 level deep.

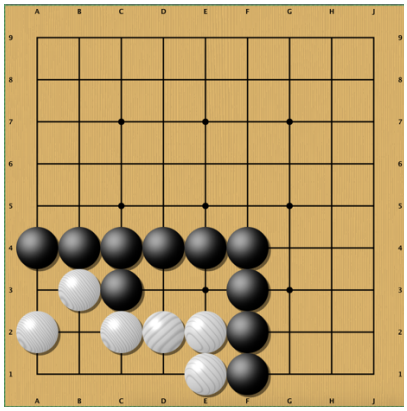


Figure 4.4.2: The position of problem 2

	Average computations needed	Standard deviation
MCTS	1906	235
Minimax-combined MCTS	2026	359

Table 4.4.3: The average computations needed of MCTS and minimax-combined MCTS

Lower bound	Upper bound
-315	76

Table 4.4.4: Paired t-test (MCTS – Minimax MCTS)

Problem 3:

The shallow traps are 5 level deep, and the correct move is 7 level deep.

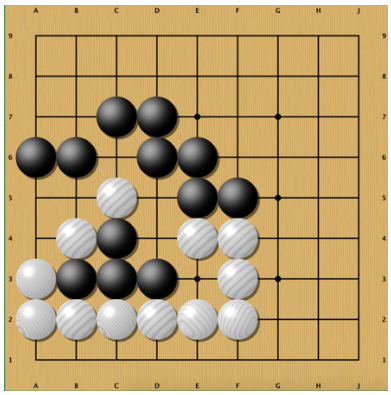


Figure 4.4.3: The position of problem 3

	Average computations needed	Standard deviation
MCTS	1074	294
Minimax-combined MCTS	825	418

Table 4.4.5: The average computations needed of MCTS and minimax-combined MCTS

Lower bound	Upper bound
-16	514

Table 4.4.6: Paired t-test (MCTS – Minimax MCTS)

Problem 4:

The shallow traps are 5 level deep, and the correct move is 5 level deep.

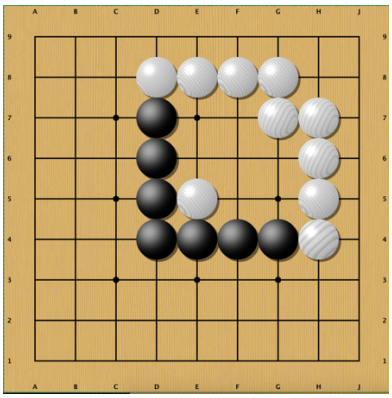


Figure 4.4.4: The position of problem 4

	Average computations needed	Standard deviation
MCTS	2046	395
Minimax-combined MCTS	1917	279

Table 4.4.7: The average computations needed of MCTS and minimax-combined MCTS

Lower bound	Upper bound
-118	376

Table 4.4.8: Paired t-test (MCTS – Minimax MCTS)

Problem 5:

The shallow traps are 9 level deep, and the correct move is 9 level deep.

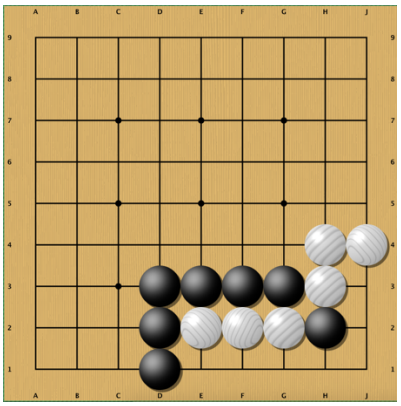


Figure 4.4.5: The position of problem 5

	Average computations needed	Standard deviation
MCTS	2396	346
Minimax-combined MCTS	2107	369

Table 4.4.9: The average computations needed of MCTS and minimax-combined MCTS

Lower bound	Upper bound
23	555

Table 4.4.10: Paired t-test (MCTS – Minimax MCTS)

4.5 Complex Problems

This section discusses the performance of minimax-combined MCTS when shallow traps do not exist. Complex problems are defined as problems that need 50000 or more computations for MCTS to pick the correct move.

Problem 1:

The shallow traps are 9 level deep, and the correct move is 9 level deep.

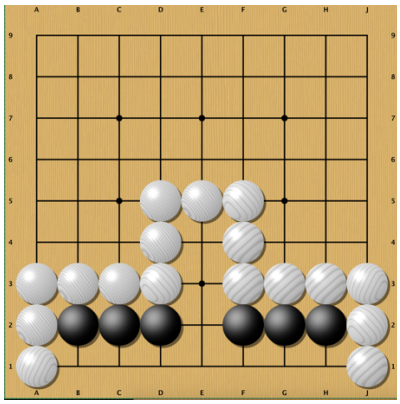


Figure 4.5.1: The position of problem 1

	Average computations needed	Standard deviation
MCTS	78395	4344
Minimax-combined MCTS	17643	2023

Table 4.5.1: The average computations needed of MCTS and minimax-combined MCTS

Lower bound	Upper bound
56855	64649

Table 4.5.2: Paired t-test (MCTS – Minimax MCTS)

Problem 2:

The shallow traps are 9 level deep, and the correct move is 13 level deep.

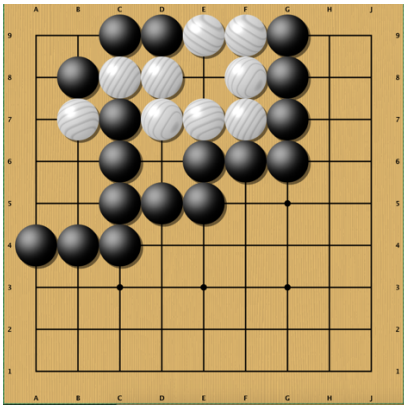


Figure 4.5.2: The position of problem 2

	Average computations needed	Standard deviation
MCTS	52966	2984
Minimax-combined MCTS	35254	15631

Table 4.5.3: The average computations needed of MCTS and minimax-combined MCTS

Lower bound	Upper bound
4300	31122

Table 4.5.4: Paired t-test (MCTS – Minimax MCTS)

Problem 3:

The shallow traps are 9 level deep, and the correct move is 9 level deep.

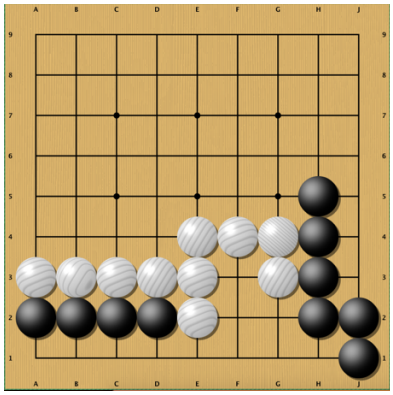


Figure 4.5.3: The position of problem 3

	Average computations needed	Standard deviation
MCTS	37692	6180
Minimax-combined MCTS	29536	6259

Table 4.5.5: The average computations needed of MCTS and minimax-combined MCTS

Lower bound	Upper bound
-146	16457

Table 4.5.6: Paired t-test (MCTS – Minimax MCTS)

Problem 4:

The shallow traps are 7 level deep, and the correct move is 7 level deep.

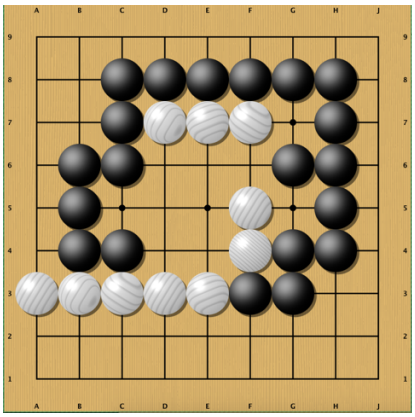


Figure 4.5.4: The position of problem 4

	Average computations needed	Standard deviation
MCTS	36677	9654
Minimax-combined MCTS	31260	8223

Table 4.5.7: The average computations needed of MCTS and minimax-combined MCTS

Lower bound	Upper bound
-5482	16317

Table 4.5.8: Paired t-test (MCTS – Minimax MCTS)

Problem 5:

The shallow traps are 9 level deep, and the correct move is 11 level deep.

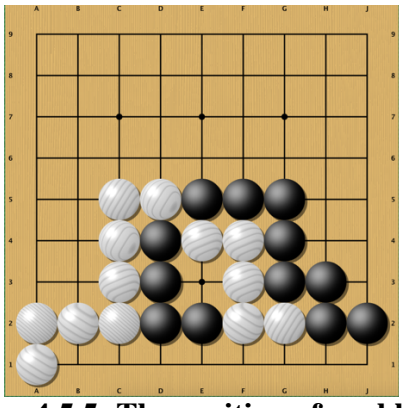


Figure 4.5.5: The position of problem 5

	Average computations needed	Standard deviation
MCTS	43118	15449
Minimax-combined MCTS	15838	9397

Table 4.5.9: The average computations needed of MCTS and minimax-combined MCTS

Lower bound	Upper bound
16522	38039

Table 4.5.10: Paired t-test (MCTS – Minimax MCTS)

4.6 Complex Problems with Multiple results

This section does not relate to shallow traps. In fact, it is an unsolved problem for either or minimax-combined MCTS. When a complex problem has multiple results, MCTS tends to figure out a good move without a lot of simulations, but it might take very long to find out the best move. Figure 4.6.1 shows an example of a problem that MCTS take very long to solve (more than 500,000 simulations). The position marked with the square is the correct move, which has a theoretical score of 1. The position marked with the triangle is the second-best move which has a theoretical score of 0.7. The position marked with the circle is the third-best move which has a theoretical score of 0.5. The positions unmarked are bad moves which have a theoretical score of 0. Tables 4.6.1 to 4.6.3 show the results when the number of total simulations increases.

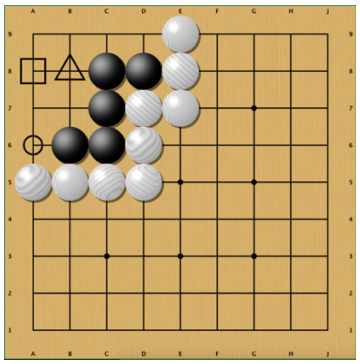


Figure 4.6.1: A complex multi score problem

Moves	Number of simulations	Mean	UCB
Best	714	0.37	0.52

Second best	1347	0.41	0.52
Third best	870	0.38	0.52
Bad move - 1	131	0.16	0.52
Bad move - 2	725	0.37	0.52
Bad move - 3	390	0.31	0.52
Bad move - 4	687	0.36	0.52
Bad move - 5	167	0.2	0.52
Bad move - 6	382	0.31	0.52

Table 4.6.1: After 5413 simulations

Moves	Number of simulations	Mean	UCB
Best	1013	0.35	0.49
Second best	10956	0.51	0.55
Third best	1251	0.36	0.49
Bad move - 1	173	0.16	0.49
Bad move - 2	1075	0.35	0.49
Bad move - 3	700	0.32	0.49
Bad move - 4	1292	0.36	0.49
Bad move - 5	233	0.2	0.49
Bad move - 6	601	0.31	0.49

Table 4.6.2: After 17294 simulations

Moves	Number of simulations	Mean	UCB
Best	1013	0.35	0.5
Second best	86077	0.72	0.74
Third best	1251	0.36	0.5
Bad move - 1	173	0.16	0.5
Bad move - 2	1075	0.35	0.5
Bad move - 3	700	0.32	0.5
Bad move - 4	1292	0.36	0.5
Bad move - 5	233	0.2	0.5
Bad move - 6	601	0.31	0.5

Table 4.6.3: After 92415 simulations

From the tables, we can see that the second-best move is not significantly different from other moves at the beginning. After 20000 simulations, all simulation budgets are allocated to the second-best move. The second-best move converges to its theoretical score 0.7, and the UCB of the second-best increases from 0.52 to 0.74, which raises a question: If the UCB of a move was 0.52 and it converges to 0.7, is it

possible that other moves with UCB of 0.5 converge to a score higher than 0.7? The answer is positive. In this example, the best move has a theoretical score of 1. Table 4.6.4 shows what will happen if the best move is simulated more.

Number of simulations	Mean
572	0.39
1066	0.36
3022	0.37
7833	0.45
13636	0.59
19428	0.68
51133	0.84
102159	0.91

Table 4.6.4 The number of simulation and mean of the best move

Table 4.6.5 shows the result of comparing the theoretical scores and the confidence bounds after 10000 simulations:

Moves	Mean	UCB	LCB	Theoretical value	Fall in confidence bounds or not
Best	0.35	0.48	0.21	1	No
Second best	0.42	0.49	0.36	0.7	No
Third best	0.34	0.48	0.2	0.5	No
Bad move - 1	0.18	0.48	-0.13	0	Yes
Bad move - 2	0.34	0.48	0.2	0	No
Bad move - 3	0.29	0.48	0.09	0	No
Bad move - 4	0.31	0.48	0.15	0	No
Bad move - 5	0.35	0.48	0.22	0	No
Bad move - 6	0.24	0.48	0	0	Yes

Table 4.6.5: Theoretical values vs. confidence bounds

Here we can see most of the confidence bounds do not contain the theoretical score. This is due to the following fact:

Bandit problems, Hoeffding's inequality, UCB1, and UCT all rely on an assumption: the distributions of samples are identical independent distribution (iid),

which is helpful when generating confidence bounds. However, in MCTS, samples under a node are not identical and independent distributed. Based on previous simulations, MCTS tends to search more promising nodes, so the later simulations are dependent on previous simulations and not identical to previous simulations.

If we applied these theories to tree search games while assuming the samples are iid, we get overconfident bounds. Like the example, UCB indicates other moves are confidently under 0.5, which is not true.

Chapter 5: Conclusion and Future Work

5.1 Conclusion

In this thesis, I develop a model of Go problems and propose minimax-combined MCTS. Then, I successfully implement MCTS and minimax-combined MCTS to the model of Go problems.

According to the results of the experiments, minimax-combined MCTS performs significantly better than MCTS when level-3 and level-5 shallow traps exist in complex problems. The MCTS spends 670% as many computations as minimax-combined MCTS in the level-3 shallow trap problem, and MCTS spends 216% as many computations as minimax-combined MCTS in the level-5 shallow trap problem.

However, when the problem is simple, the minimax-combined MCTS is significantly better in only 1 out of 5 scenarios (and the difference is not much), and the performance is not significantly different among the two algorithms in the rest of the problems.

On the other hand, in 3 out of 5 complex problems without level-3 or 5 shallow traps, MCTS spends more computations (440%, 150%, and 272% as many computations as minimax-combined MCTS), and the performance is not significantly different among the two algorithms in the rest of the problems.

Also, the iid assumption allows MCTS to find out good moves fast. Nonetheless, when several outcomes exist in the problem, it might only able to find out the second or third best move instead of the best.

5.2 Future Work

First, I would like to do more experiments. The current experiments are insufficient in many ways. The sample size is small, and the number of problems is low, and the parameters are fixed. The experiment should be conducted under different parameters. Also, I would like to test the effectiveness of minimax-combined MCTS on different domains of games such as chess, Go (the entire game, not only restricted to local problems). The performance might vary from games to games.

Second, I would like to investigate more ways to define the minimax threshold. Currently, the minimax threshold is picked arbitrarily. A more statistics-based rule should be applied. The UCB and LCB of parent and child nodes may give some clues.

Third, I would like to incorporate opponent policy play into MCTS, which is not considered in this thesis. With such a policy, the minimax-combined MCTS should show substantial improvement, as in AlphaGo and AlphaGo Zero.

Fourth, I would like to apply various MCTS extensions as well, e.g., as mentioned in the literature review, the MCTS solver and three ways of MCTS and minimax hybrid, and compare them to minimax-combined MCTS. Furthermore, I would like to apply UCB1-tuned instead of UCB1, because UCB1-tuned takes variance of samples into account, which might improve the performance. Also, best-arm identification algorithm only considers the regret of a final decision, which is a more suitable assumption than accumulated regrets.

Fifth, I would like to apply heuristic algorithms that particularly fit the game of Go. All Moves As First (AMAF) is a tree policy enhancement. The board of Go is big compared to other games, so when the positions of some places are changed, the rest of the board are not influenced that much. Therefore, a position which leads to good results under a sub-tree is often a good position in other sub-trees. This concept can be extended to the Last Good Reply heuristic. For example, if the sequence of moves (A1-A2-B1-B2) leads to a good result. Then, when opponents play B1, B2 might be a good response.

Last, I would like to do research on the influence of the non-iid characteristic in MCTS. When having poor policies or no policy, the influence is huge. The problem is whether the characteristic has great or little influence when the policy is as good as AlphaGo. Also, I would like to explore the algorithms that can deal with the non-iid characteristic.

Appendices

A. Experiment Data

A.1 Level-3 Shallow Trap

X_1	X_2	X_3	X_4	X_5
52848	58821	54372	37941	42139
X_6	X_7	X_8	X_9	X_{10}
43128	55132	43259	42146	58240

Table A.1.1: The number of computations needed of MCTS to pick the correct move (the exploration parameter = 1.414)

X_1	X_2	X_3	X_4	X_5
5572	7544	5167	8243	7982
X_6	X_7	X_8	X_9	X_{10}
5176	8647	8266	7979	8462

Table A.1.2: The number of computations needed of minimax-combined MCTS to pick the correct move (the exploration parameter = 1.414 and minimax threshold parameter =50)

A.2 Level-5 Shallow Trap

X_1	X_2	X_3	X_4	X_5
52811	46902	51639	43533	69134
X_6	X_7	X_8	X_9	X_{10}
44109	42579	52095	49536	45526

Table A.2.1: The number of computations needed of MCTS to pick the correct move (the exploration parameter = 1.414)

X_1	X_2	X_3	X_4	X_5
28138	15428	20815	38386	31840
X_6	X_7	X_8	X_9	X_{10}
19470	19660	20827	20302	15806

Table A.2.2: The number of computations needed of minimax-combined MCTS to pick the correct move (the exploration parameter = 1.414 and minimax threshold parameter =100)

A.3 Simple Problems without Shallow Trap

Problem 1:

X_1	X_2	X_3	X_4	X_5

1167	1443	1255	1152	981
X_6	X_7	X_8	X_9	X_{10}
932	1528	1028	1340	1009
X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
878	1211	1090	1645	1327
X_{16}	X_{17}	X_{18}	X_{19}	X_{20}
746	1158	1438	1003	727

Table A.3.1: The number of computations needed of MCTS to pick the correct move (the exploration parameter = 1.414)

X_1	X_2	X_3	X_4	X_5
943	1387	988	993	709
X_6	X_7	X_8	X_9	X_{10}
1075	1182	1186	831	1302
X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
1123	529	1586	1220	192
X_{16}	X_{17}	X_{18}	X_{19}	X_{20}
1012	1069	1008	869	1107

Table A.3.2: The number of computations needed of minimax-combined MCTS to pick the correct move (the exploration parameter = 1.414 and minimax threshold parameter =50)

Problem 2:

X_1	X_2	X_3	X_4	X_5
1916	2003	1470	1720	2417
X_6	X_7	X_8	X_9	X_{10}
1856	2350	1738	1867	1777
X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
2143	2021	2137	1899	1893
X_{16}	X_{17}	X_{18}	X_{19}	X_{20}
1850	1542	1985	1702	1835

Table A.3.3: The number of computations needed of MCTS to pick the correct move (the exploration parameter = 1.414)

X_1	X_2	X_3	X_4	X_5
1608	1127	1803	2202	2152
X_6	X_7	X_8	X_9	X_{10}
2150	2648	1957	2121	1747
X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
2406	2188	2314	1759	1841
X_{16}	X_{17}	X_{18}	X_{19}	X_{20}
2683	1921	1978	2124	1786

Table A.3.4: The number of computations needed of minimax-combined MCTS to pick the correct move (the exploration parameter = 1.414 and minimax threshold parameter =50)

Problem 3:

X_1	X_2	X_3	X_4	X_5
937	1804	1201	1064	1384
X_6	X_7	X_8	X_9	X_{10}
748	1470	960	1013	1027
X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
1076	477	658	907	830
X_{16}	X_{17}	X_{18}	X_{19}	X_{20}
1157	1245	1266	1116	1131

Table A.3.5: The number of computations needed of MCTS to pick the correct move (the exploration parameter = 1.414)

X_1	X_2	X_3	X_4	X_5
382	420	1024	663	900
X_6	X_7	X_8	X_9	X_{10}
1208	1176	1094	257	533
X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
280	194	1019	1039	1076
X_{16}	X_{17}	X_{18}	X_{19}	X_{20}
1156	849	1668	1260	297

Table A.3.6: The number of computations needed of minimax-combined MCTS to pick the correct move (the exploration parameter = 1.414 and minimax threshold parameter =50)

Problem 4:

X_1	X_2	X_3	X_4	X_5
2033	1597	1967	1842	1669
X_6	X_7	X_8	X_9	X_{10}
2168	1489	2610	2533	2200
X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
2231	2257	1536	1712	2099
X_{16}	X_{17}	X_{18}	X_{19}	X_{20}
1498	1814	2350	2571	2747

Table A.3.7: The number of computations needed of MCTS to pick the correct move (the exploration parameter = 1.414)

X_1	X_2	X_3	X_4	X_5
2316	2015	1991	2250	1934
X_6	X_7	X_8	X_9	X_{10}
2155	1799	1897	1770	2151
X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
1972	1866	1769	1177	1435
X_{16}	X_{17}	X_{18}	X_{19}	X_{20}
2186	1784	1716	1916	2245

Table A.3.8: The number of computations needed of minimax-combined MCTS to pick the correct move (the exploration parameter = 1.414 and minimax threshold parameter =50)

Problem 5:

X_1	X_2	X_3	X_4	X_5
2648	2595	2785	1826	2851
X_6	X_7	X_8	X_9	X_{10}
1777	2497	2253	2357	2374
X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
2807	2285	2194	1672	2684
X_{16}	X_{17}	X_{18}	X_{19}	X_{20}
2131	2372	2458	2594	2754

Table A.3.9: The number of computations needed of MCTS to pick the correct move (the exploration parameter = 1.414)

X_1	X_2	X_3	X_4	X_5
2372	2334	1878	2215	1913
X_6	X_7	X_8	X_9	X_{10}
2088	2219	2072	1576	1941
X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
1849	2194	1899	2123	1876
X_{16}	X_{17}	X_{18}	X_{19}	X_{20}
2301	1676	2322	1930	3360

Table A.3.10: The number of computations needed of minimax-combined MCTS to pick the correct move (the exploration parameter = 1.414 and minimax threshold parameter =50)

A.4 Complex Problems without Shallow Trap

Problem 1:

X_1	X_2	X_3	X_4	X_5
82338	81734	70378	79007	74319
X_6	X_7	X_8	X_9	X_{10}
83975	79494	79836	73289	79581

Table A.4.1: The number of computations needed of MCTS to pick the correct move (the exploration parameter = 1.414)

X_1	X_2	X_3	X_4	X_5
16527	16267	17740	16414	15791
X_6	X_7	X_8	X_9	X_{10}
18012	20085	16389	17017	22191

Table A.4.2: The number of computations needed of minimax-combined MCTS to pick the correct move (the exploration parameter = 1.414 and minimax threshold parameter =100)

Problem 2:

X_1	X_2	X_3	X_4	X_5
50899	49774	58950	51618	56656
X_6	X_7	X_8	X_9	X_{10}
54415	50883	54154	51218	51089

Table A.4.3: The number of computations needed of MCTS to pick the correct move (the exploration parameter = 1.414)

X_1	X_2	X_3	X_4	X_5
60320	33934	37281	27826	19976
X_6	X_7	X_8	X_9	X_{10}
58647	12298	38391	41156	22714

Table A.4.4: The number of computations needed of minimax-combined MCTS to pick the correct move (the exploration parameter = 1.414 and minimax threshold parameter =100)

Problem 3:

X_1	X_2	X_3	X_4	X_5
34519	30172	35242	38399	47751
X_6	X_7	X_8	X_9	X_{10}
46638	38477	28536	37295	39893

Table A.4.5: The number of computations needed of MCTS to pick the correct move (the exploration parameter = 1.414)

X_1	X_2	X_3	X_4	X_5
36104	37460	22037	31771	21091
X_6	X_7	X_8	X_9	X_{10}
36588	32016	29274	22552	26470

Table A.4.6: The number of computations needed of minimax-combined MCTS to pick the correct move (the exploration parameter = 1.414 and minimax threshold parameter =50)

Problem 4:

X_1	X_2	X_3	X_4	X_5
27433	26469	44101	43254	27051
X_6	X_7	X_8	X_9	X_{10}
28315	29114	48490	48197	44346

Table A.4.7: The number of computations needed of MCTS to pick the correct move (the exploration parameter = 1.414)

X_1	X_2	X_3	X_4	X_5

41871	36937	31965	28147	35115
X_6	X_7	X_8	X_9	X_{10}
31452	11069	32699	35446	27897

Table A.4.8: The number of computations needed of minimax-combined MCTS to pick the correct move (the exploration parameter = 1.414 and minimax threshold parameter =50)

Problem 5:

X_1	X_2	X_3	X_4	X_5
58978	33232	33013	33910	69404
X_6	X_7	X_8	X_9	X_{10}
67056	32853	34205	36054	32478

Table A.4.9: The number of computations needed of MCTS to pick the correct move (the exploration parameter = 1.414)

X_1	X_2	X_3	X_4	X_5
23214	6543	28460	5695	28505
X_6	X_7	X_8	X_9	X_{10}
20363	13198	9766	3378	19257

Table A.4.10: The number of computations needed of minimax-combined MCTS to pick the correct move (the exploration parameter = 1.414 and minimax threshold parameter =50)

B. Source Code

The program is written is C++, including 6 files.

1. Main.cpp: Where the experiment process is defined.
2. Node.h: Define the algorithm of MCTS. This is the only different part between MCTS and minimax-combined MCTS.
3. Board.h: Define the model of Go.
4. Position.h: An interface between Node.h and Board.h.
5. Function.h: Supporting functions.
6. Go.txt: input files.

Main.cpp (MCTS):

```
#include "Node.h"
#include <iostream>
using namespace std;
int main(){

    srand((unsigned int)time(NULL));
    rand();
    rand();
    rand();
    rand();
    rand();

    char fileName[]="Go.txt";
    Board board(fileName);
```

```
    Node root(&board,1500,1.414);
    root.MonteCarloTreeSearch();
    root.viewTree();

    return 0;
}
```

Main.cpp (minimax-combined MCTS);

```
#include "Node.h"

#include <iostream>
using namespace std;

int main(){

    srand((unsigned int)time(NULL));
    rand();
    rand();
    rand();
    rand();
    rand();

    char fileName[]="Go.txt";
    Board board(fileName);

    Node root(&board,1000,1.414,50);
    root.MonteCarloTreeSearch();
    root.viewTree();

    return 0;
}
```

Node.h (MCTS):

```
//
//  Node.h
//  Thesis
//
//  Created by Jonathan Lin on 6/19/17.
//  Copyright © 2017 Jonathan Lin. All rights reserved.
//

#ifndef Node_h
#define Node_h

#include "Board.h"

class Node{

private:

    Board* board;
    Node* parent;
    int action[2];
    vector<Node*> children;

    vector<int>decisionTimeList;
    vector<int>decisionListRow;
    vector<int>decisionListColumn;

    int simulationBudget;
    int budgetUsed;
    float explorationParameter;

    float numberOfSimulation;
    float totalScore;
    float mean;
    float UCB;
```

```
    float LCB;

    void updateDecisionList();
    void decision(int functionAction[2]);
    void selection();
    void expansion();
    void simulation(float* score);
    void backPropagation(float score);

    void randomPlay(Board* copy);

    Node* searchChild(int functionAction[2]);
    void getAction(int functionAction[2]);

    void calculateUCBLCB();

public:

    void test();
    Node(Board* input, int functionSimulationBudget, float
functionExplorationParameter); //for root
    Node(Node* parentNode, int functionAction[2]); //for child
    ~Node();
    void MonteCarloTreeSearch();
    void giveExtraSearchBudget(int budget);
    void viewDecision();
    void viewData();
    void viewTree();

};

//PUBLIC

void Node::test(){

    //*
    //board->placeStone(2, 1);
    MonteCarloTreeSearch();
    viewTree();
    //*/

    /*
    int i=0;
    while(i<100){
        float score=0;
        simulation(&score);
        backPropagation(score);
        i++;
    }
    viewData();
    //*/

}

Node::Node(Board* input, int functionSimulationBudget, float
functionExplorationParameter){

    Board* copy=new Board(input);
    board=copy;
    parent=NULL;
    action[0]=-1;
    action[1]=-1;

    simulationBudget=functionSimulationBudget;
    budgetUsed=0;
    explorationParameter=functionExplorationParameter;

    numberOfSimulation=0;
```

```
    totalScore=0;
    mean=0;
    UCB=0;
    LCB=0;

}

Node::Node(Node* parentNode, int functionAction[2]){

    parentNode->children.push_back(this);
    parent=parentNode;
    Board* copy=new Board(parentNode->board);
    copy->placeStone(functionAction[0], functionAction[1]);
    board=copy;
    action[0]=functionAction[0];
    action[1]=functionAction[1];

    simulationBudget=0;
    budgetUsed=0;
    explorationParameter=parent->explorationParameter;

    numberOfSimulation=0;
    totalScore=0;
    mean=0;
    UCB=0;
    LCB=0;

}

Node::~Node(){

    delete board;

}

void Node::MonteCarloTreeSearch(){

    while(budgetUsed<=simulationBudget){

        selection();
        budgetUsed++;

        if(budgetUsed!=0&&budgetUsed%1000==0){
            cout<<budgetUsed<<" of budget is used."<<endl;
        }

    }

}

void Node::giveExtraSearchBudget(int budget){
    simulationBudget+=budget;
    MonteCarloTreeSearch();
    viewTree();
}

void Node::viewDecision(){

    int convergeTime=-1;
    int converageMove[2]={-1,-1};

    int i=0;
    while(i<decisionListRow.size()) {

        if(!(decisionListRow[i]==converageMove[0]&&decisionListColumn[i]==converageMo
ve[1])
           ){
            converageMove[0]=decisionListRow[i];
```

```
            converageMove[1]=decisionListColumn[i];
            convergeTime=decisionTimeList[i];
        }

        //cout<<decisionTimeList[i]<<endl;
        //cout<<decisionListRow[i]<<" "<<decisionListColumn[i]<<endl;
        i++;
    }

    cout<<endl;
    cout<<"Convergent time: "<<convergeTime<<endl;
    cout<<"Convergent move: "<<converageMove[0]<<" "<<converageMove[1]<<endl;
    cout<<endl;

}

void Node::viewData(){

    board->showBoard();

    if(board->getOwnColor()==1) {
        cout<<"Black to move."<<endl<<endl;
    }
    else{
        cout<<"White to move."<<endl<<endl;
    }

    cout<<setw(30)<<"Last move"<<action[0]<<" "<<action[1]<<endl<<endl;

    cout<<setw(30)<<"Number of simulation"<<numberOfSimulation<<endl<<endl;

    cout<<setw(30)<<"Total score"<<rounding(totalScore, 2)<<endl<<endl;

    cout<<setw(30)<<"Mean"<<rounding(mean, 2)<<endl<<endl;

    cout<<setw(30)<<"UCB"<<rounding(UCB, 2)<<endl<<endl;

    cout<<setw(30)<<"LCB"<<rounding(LCB, 2)<<endl<<endl;


    cout<<"Children"<<endl;

    if(children.size()==0) {
        cout<<"No child"<<endl<<endl;
    }
    else{

        cout<<setw(15)<<"Action";
        cout<<setw(15)<<"# of sim.";
        cout<<setw(15)<<"Mean";
        cout<<setw(15)<<"UCB";
        cout<<setw(15)<<"LCB";
        cout<<endl;

        int i=0;
        while(i<children.size()){
            cout<<setw(5)<<children[i]->action[0];
            cout<<setw(10)<<children[i]->action[1];
            cout<<setw(15)<<children[i]->numberOfSimulation;
            cout<<setw(15)<<rounding(children[i]->mean,2);
            cout<<setw(15)<<rounding(children[i]->UCB,2);
            cout<<setw(15)<<rounding(children[i]->LCB,2);
            cout<<endl;
            i++;
        }
        cout<<endl;

    }
```

```
}

void Node::viewTree(){

    Node* currentNode=this;
    int control1=0, control2=0;

    while(currentNode!=NULL){

        currentNode->viewData();

        cout<<"Go to child: (row, col). Go to parent node: -1. Exit: -2. Give more
budget: -3. View decision: -4."<<endl;
        cin>>control1;

        if(control1==-1) {
            currentNode=currentNode->parent;
            continue;
        }

        else if(control1==-2) {
            break;
        }

        else if(control1==-3) {
            break;
        }

        else if(control1==-4){
            viewDecision();
            continue;
        }

        else{

            cin>>control2;
            int move[2]={control1,control2};
            if(currentNode->searchChild(move)==NULL){
                cout<<"Child not found. Input a key to continue."<<endl;
                int temp=0;
                cin>>temp;
                continue;
            }

            currentNode=currentNode->searchChild(move);
            continue;

        }

    }

    if(control1==-3) {
        int extraBudget=0;
        cout<<"Enter extra budget."<<endl;
        cin>>extraBudget;
        giveExtraSearchBudget(extraBudget);
    }



}

//PRIVATE

void Node::updateDecisionList(){

    int move[2]={-1,-1};
```

```
    decision(move);
    decisionTimeList.push_back(numberOfSimulation);
    decisionListRow.push_back(move[0]);
    decisionListColumn.push_back(move[1]);

}

void Node::decision(int functionAction[2]){

    if(children.size()==0){
        return;
    }

    float value=children[0]->mean;
    functionAction[0]=children[0]->action[0];
    functionAction[1]=children[0]->action[1];

    int i=1;
    while(i<children.size()) {
        if(children[i]->mean>value){
            value=children[i]->mean;
            functionAction[0]=children[i]->action[0];
            functionAction[1]=children[i]->action[1];
        }
        i++;
    }

}

void Node::selection(){

    Node* currentNode=this;

    while(currentNode!=NULL) {

        //if the node is at end state
        if(currentNode->board->getScore()!=-2) {
            currentNode->backPropagation(currentNode->board->getScore());
            updateDecisionList();
            return;
        }

        //get legal positions
        Position position(currentNode->board->getBoardSize());
        currentNode->board->computerInterface(&position);

        //if the node is expandable
        if(currentNode->children.size()==0) {
            currentNode->expansion();
            updateDecisionList();
            return;
        }

        //if the node is not expandable
        else{

            //if the node is MAX node
            if(currentNode->board->getOwnColor()==1){
                int selectedChild=0;
                float maxUCB=currentNode->children[selectedChild]->UCB;
                int i=1;
                while(i<currentNode->children.size()){
                    if(currentNode->children[i]->UCB>maxUCB){
                        selectedChild=i;
                        maxUCB=currentNode->children[selectedChild]->UCB;
                    }
                    i++;
                }
```

```
                currentNode=currentNode->children[selectedChild];
            }

            //if the node is MIN node
            else if(currentNode->board->getOwnColor()==2) {
                int selectedChild=0;
                float minLCB=currentNode->children[selectedChild]->LCB;
                int i=1;
                while(i<currentNode->children.size()){
                    if(currentNode->children[i]->LCB<minLCB){
                        selectedChild=i;
                        minLCB=currentNode->children[selectedChild]->LCB;
                    }
                    i++;
                }
                currentNode=currentNode->children[selectedChild];
            }
        }

    }

}

void Node::expansion(){

    if(children.size()!=0){
        cout<<"Expansion error, the node is already expanded."<<endl;
        return;
    }

    Position position(board->getBoardSize());
    board->computerInterface(&position);

    int i=0;
    while(i<position.legalPositionColumn.size()) {

        int move[2]={position.legalPositionRow[i],position.legalPositionColumn[i]};
        Node* expandedChild=new Node(this,move);
        float score=0;
        expandedChild->simulation(&score);
        expandedChild->backPropagation(score);

        i++;
    }

}

void Node::simulation(float* functionScore){

    float score=0;
    Board copy(board);

    while(copy.getScore()==-2){
        randomPlay(&copy);
    }

    score=copy.getScore();
    *functionScore=score;

}

void Node::backPropagation(float score){

    Node* currentNode=this;

    while(currentNode!=NULL){

        currentNode->numberOfSimulation++;
```

```
        currentNode->totalScore+=score;
        currentNode->mean=currentNode->totalScore/currentNode->numberOfSimulation;

        int i=0;
        while(i<currentNode->children.size()) {
            currentNode->children[i]->calculateUCBLCB();
            i++;
        }

        currentNode=currentNode->parent;

    }

}

void Node::randomPlay(Board* copy){

    Position position(copy->getBoardSize());
    copy->computerInterface(&position);

    //randomly pick position

    int random=(int)(zeroToOne()*position.getNumberOfLegalMove());

    int pickedRow=position.legalPositionRow.at(random);
    int pickedColumn=position.legalPositionColumn.at(random);

    copy->placeStone(pickedRow, pickedColumn);

}

Node* Node::searchChild(int functionAction[2]){

    Node* result=NULL;

    int counter=0;

    while(counter<children.size()){

        int childMove[2]={0};
        children.at(counter)->getAction(childMove);
        if(functionAction[0]==childMove[0]&&functionAction[1]==childMove[1]){
            result=children.at(counter);

        }

        counter++;
    }

    return result;

}

void Node::getAction(int functionAction[2]){

    functionAction[0]=action[0];
    functionAction[1]=action[1];

}

void Node::calculateUCBLCB(){

    if(parent==NULL) {
        return;
    }

    float exploration=explorationParameter*sqrt(log(parent-
>numberOfSimulation)/numberOfSimulation);
```

```
    UCB=mean+exploration;
    LCB=mean-exploration;

}

#endif /* Node_h */
```

Node.h (minimax-combined MCTS);

```
//
//  Node.h
//  Thesis
//
//  Created by Jonathan Lin on 6/19/17.
//  Copyright © 2017 Jonathan Lin. All rights reserved.
//

#ifndef Node_h
#define Node_h

#include "Board.h"

class Node{

private:

    Board* board;
    Node* parent;
    Node* minimaxChild;
    int action[2];
    vector<Node*> children;

    vector<int>decisionTimeList;
    vector<int>decisionListRow;
    vector<int>decisionListColumn;

    int simulationBudget;
    int budgetUsed;
    float explorationParameter;
    int backupThresholdParameter;
    int backupThreshold;

    float numberOfSimulation;
    float totalScore;
    float mean;
    float UCB;
    float LCB;

    void updateDecisionList();
    void decision(int functionAction[2]);
    void selection();
    void expansion();
    void simulation(float* score);
    void backPropagation(float score);

    void randomPlay(Board* copy);

    Node* searchChild(int functionAction[2]);
    void getAction(int functionAction[2]);

    void calculateUCBLCB();
    void calculateMinimax();
    bool checkMinimax();

public:

    void test();
```

```
    Node(Board* input, int functionSimulationBudget, float
functionExplorationParameter, int functionBackupThresholdParameter); //for root
    Node(Node* parentNode, int functionAction[2]); //for child
    ~Node();
    void MonteCarloTreeSearch();
    void giveExtraSearchBudget(int budget);
    void viewDecision();
    void viewData();
    void viewTree();

};

//PUBLIC

void Node::test(){



}

Node::Node(Board* input, int functionSimulationBudget, float
functionExplorationParameter, int functionBackupThresholdParameter){

    Board* copy=new Board(input);
    board=copy;
    parent=NULL;
    minimaxChild=this;
    action[0]=-1;
    action[1]=-1;

    simulationBudget=functionSimulationBudget;
    budgetUsed=0;
    explorationParameter=functionExplorationParameter;

    backupThresholdParameter=functionBackupThresholdParameter;
    Position position(board->getBoardSize());
    board->computerInterface(&position);
    backupThreshold=backupThresholdParameter*position.getNumberOfLegalMove();

    numberOfSimulation=0;
    totalScore=0;
    mean=0;
    UCB=0;
    LCB=0;

}

Node::Node(Node* parentNode, int functionAction[2]){

    parentNode->children.push_back(this);
    parent=parentNode;
    minimaxChild=this;
    Board* copy=new Board(parentNode->board);
    copy->placeStone(functionAction[0], functionAction[1]);
    board=copy;
    action[0]=functionAction[0];
    action[1]=functionAction[1];

    simulationBudget=0;
    budgetUsed=0;
    explorationParameter=parent->explorationParameter;

    backupThresholdParameter=parent->backupThresholdParameter;
    Position position(board->getBoardSize());
    board->computerInterface(&position);
    backupThreshold=backupThresholdParameter*position.getNumberOfLegalMove();

    numberOfSimulation=0;
```

```
    totalScore=0;
    mean=0;
    UCB=0;
    LCB=0;

}

Node::~Node(){

    delete board;

}

void Node::MonteCarloTreeSearch(){

    while(budgetUsed<=simulationBudget){

        selection();
        budgetUsed++;

        if(budgetUsed!=0&&budgetUsed%100==0){
            cout<<budgetUsed<<"/"<<simulationBudget<<" of budget is used."<<endl;
        }

    }

}

void Node::giveExtraSearchBudget(int budget){
    simulationBudget+=budget;
    MonteCarloTreeSearch();
    viewTree();
}

void Node::viewDecision(){

    int convergeTime=-1;
    int converageMove[2]={-1,-1};

    int i=0;
    while(i<decisionListRow.size()) {

        if(!(decisionListRow[i]==converageMove[0]&&decisionListColumn[i]==converageMo
ve[1])
           ){
            converageMove[0]=decisionListRow[i];
            converageMove[1]=decisionListColumn[i];
            convergeTime=decisionTimeList[i];
        }

        //cout<<decisionTimeList[i]<<endl;
        //cout<<decisionListRow[i]<<" "<<decisionListColumn[i]<<endl;
        i++;
    }

    cout<<endl;
    cout<<"Convergent time: "<<convergeTime<<endl;
    cout<<"Convergent move: "<<converageMove[0]<<" "<<converageMove[1]<<endl;
    cout<<endl;

}

void Node::viewData(){

    board->showBoard();

    if(board->getOwnColor()==1) {
        cout<<"Black to move."<<endl<<endl;
```

```
    }
    else{
        cout<<"White to move."<<endl<<endl;
    }

    cout<<setw(30)<<"Last move"<<action[0]<<" "<<action[1]<<endl<<endl;

    if(minimaxChild==this){
        cout<<"Don't have minimax child."<<endl<<endl;
    }
    else{
        cout<<"Have minimax child."<<endl<<endl;
    }

    cout<<setw(30)<<"Number of simulation"<<numberOfSimulation<<endl<<endl;

    cout<<setw(30)<<"Total score"<<rounding(totalScore, 2)<<endl<<endl;

    cout<<setw(30)<<"Mean"<<rounding(mean, 2)<<endl<<endl;

    cout<<setw(30)<<"UCB"<<rounding(UCB, 2)<<endl<<endl;

    cout<<setw(30)<<"LCB"<<rounding(LCB, 2)<<endl<<endl;

    cout<<"Children"<<endl;

    if(children.size()==0) {
        cout<<"No child"<<endl<<endl;
    }
    else{

        cout<<setw(15)<<"Action";
        cout<<setw(15)<<"# of sim.";
        cout<<setw(15)<<"Mean";
        cout<<setw(15)<<"UCB";
        cout<<setw(15)<<"LCB";
        cout<<setw(15)<<"# of sim.";
        cout<<setw(15)<<"Mean";
        cout<<setw(15)<<"UCB";
        cout<<setw(15)<<"LCB";
        cout<<endl;

        int i=0;
        while(i<children.size()){
            cout<<setw(5)<<children[i]->action[0];
            cout<<setw(10)<<children[i]->action[1];
            cout<<setw(15)<<children[i]->numberOfSimulation;
            cout<<setw(15)<<rounding(children[i]->mean,2);
            cout<<setw(15)<<rounding(children[i]->UCB,2);
            cout<<setw(15)<<rounding(children[i]->LCB,2);

            cout<<setw(15)<<children[i]->minimaxChild->numberOfSimulation;
            cout<<setw(15)<<rounding(children[i]->minimaxChild->mean,2);
            cout<<setw(15)<<rounding(children[i]->minimaxChild->UCB,2);
            cout<<setw(15)<<rounding(children[i]->minimaxChild->LCB,2);
            cout<<endl;
            i++;
        }
        cout<<endl;

    }

}

void Node::viewTree(){

    Node* currentNode=this;
    int control1=0, control2=0;
```

```
    while(currentNode!=NULL){

        currentNode->viewData();

        cout<<"Go to child: (row, col). Go to parent node: -1. Exit: -2. Give more
budget: -3. View decision: -4."<<endl;
        cin>>control1;

        if(control1==-1) {
            currentNode=currentNode->parent;
            continue;
        }

        else if(control1==-2) {
            break;
        }

        else if(control1==-3) {
            break;
        }

        else if(control1==-4){
            viewDecision();
            continue;
        }

        else if(control1==-5){
            currentNode=currentNode->minimaxChild;
            continue;
        }

        else if(control1==-6){
            currentNode->calculateMinimax();
            continue;
        }

        else{

            cin>>control2;
            int move[2]={control1,control2};
            if(currentNode->searchChild(move)==NULL){
                cout<<"Child not found. Input a key to continue."<<endl;
                int temp=0;
                cin>>temp;
                continue;
            }

            currentNode=currentNode->searchChild(move);
            continue;

        }

    }

    if(control1==-3) {
        int extraBudget=0;
        cout<<"Enter extra budget."<<endl;
        cin>>extraBudget;
        giveExtraSearchBudget(extraBudget);
    }

}

//PRIVATE

void Node::updateDecisionList(){
```

```
    int move[2]={-1,-1};
    decision(move);
    decisionTimeList.push_back(numberOfSimulation);
    decisionListRow.push_back(move[0]);
    decisionListColumn.push_back(move[1]);

}

void Node::decision(int functionAction[2]){

    if(children.size()==0){
        return;
    }

    float value=children[0]->numberOfSimulation;
    functionAction[0]=children[0]->action[0];
    functionAction[1]=children[0]->action[1];

    Position position(board->getBoardSize());
    board->computerInterface(&position);

    int i=1;
    while(i<children.size()) {
        if(children[i]->numberOfSimulation>value){//&&children[i]-
>numberOfSimulation>numberOfSimulation/position.getNumberOfLegalMove()
            value=children[i]->numberOfSimulation;
            functionAction[0]=children[i]->action[0];
            functionAction[1]=children[i]->action[1];
        }
        i++;
    }

}

void Node::selection(){

    Node* currentNode=this;

    while(currentNode!=NULL) {

        //if the node is at end state
        if(currentNode->board->getScore()!=-2) {
            currentNode->backPropagation(currentNode->board->getScore());
            updateDecisionList();
            return;
        }

        //get legal positions
        Position position(currentNode->board->getBoardSize());
        currentNode->board->computerInterface(&position);

        //if the node is expandable
        if(currentNode->children.size()==0) {
            currentNode->expansion();
            updateDecisionList();
            return;
        }

        //if the node is not expandable
        else{

            //if the node is MAX node
            if(currentNode->board->getOwnColor()==1){
                int selectedChild=0;
                float maxUCB=currentNode->children[0]->minimaxChild->UCB;
                int i=1;
                while(i<currentNode->children.size()){
                    if(currentNode->children[i]->minimaxChild->UCB>maxUCB){
```

```
                        selectedChild=i;
                        maxUCB=currentNode->children[i]->minimaxChild->UCB;
                    }
                    i++;
                }
                currentNode=currentNode->children[selectedChild]->minimaxChild;
            }

            //if the node is MIN node
            else if(currentNode->board->getOwnColor()==2) {
                int selectedChild=0;
                float minLCB=currentNode->children[0]->minimaxChild->LCB;
                int i=1;
                while(i<currentNode->children.size()){
                    if(currentNode->children[i]->minimaxChild->LCB<minLCB){
                        selectedChild=i;
                        minLCB=currentNode->children[i]->minimaxChild->LCB;
                    }
                    i++;
                }
                currentNode=currentNode->children[selectedChild]->minimaxChild;
            }
        }

    }

}

void Node::expansion(){

    if(children.size()!=0){
        cout<<"Expansion error, the node is already expanded."<<endl;
        return;
    }

    Position position(board->getBoardSize());
    board->computerInterface(&position);

    int i=0;
    while(i<position.legalPositionColumn.size()) {

        int move[2]={position.legalPositionRow[i],position.legalPositionColumn[i]};
        Node* expandedChild=new Node(this,move);
        float score=0;
        expandedChild->simulation(&score);
        expandedChild->backPropagation(score);

        i++;
    }

}

void Node::simulation(float* functionScore){

    float score=0;
    Board copy(board);

    while(copy.getScore()==-2){
        randomPlay(&copy);
    }

    score=copy.getScore();
    *functionScore=score;

}

void Node::backPropagation(float score){
```

```
    Node* currentNode=this;

    while(currentNode!=NULL){

        currentNode->numberOfSimulation++;
        currentNode->totalScore+=score;
        currentNode->mean=currentNode->totalScore/currentNode->numberOfSimulation;

        currentNode->calculateMinimax();

        int i=0;
        while(i<currentNode->children.size()) {
            currentNode->children[i]->calculateUCBLCB();
            i++;
        }
        currentNode=currentNode->parent;

    }

}

void Node::randomPlay(Board* copy){

    Position position(copy->getBoardSize());
    copy->computerInterface(&position);

    //randomly pick position

    int random=(int)(zeroToOne()*position.getNumberOfLegalMove());

    int pickedRow=position.legalPositionRow.at(random);
    int pickedColumn=position.legalPositionColumn.at(random);

    copy->placeStone(pickedRow, pickedColumn);

}

Node* Node::searchChild(int functionAction[2]){

    Node* result=NULL;

    int counter=0;

    while(counter<children.size()){

        int childMove[2]={0};
        children.at(counter)->getAction(childMove);
        if(functionAction[0]==childMove[0]&&functionAction[1]==childMove[1]){
            result=children.at(counter);

        }

        counter++;
    }

    return result;

}

void Node::getAction(int functionAction[2]){

    functionAction[0]=action[0];
    functionAction[1]=action[1];

}

void Node::calculateUCBLCB(){
```

```
    if(parent==NULL) {
        return;
    }

    float exploration=explorationParameter*sqrt(log(parent-
>numberOfSimulation)/numberOfSimulation);

    UCB=mean+exploration;
    LCB=mean-exploration;

}

void Node::calculateMinimax(){

    if(children.size()==0){
        minimaxChild=this;
        return;
    }

    if(numberOfSimulation<backupThreshold){
        minimaxChild=this;
        return;
    }

    float maxValue=-99;
    float minValue=99;
    int i=0;
    while(i<children.size()){

        if(board->getOwnColor()==1){
            if(children[i]->minimaxChild->mean>maxValue){
                maxValue=children[i]->minimaxChild->mean;
                minimaxChild=children[i]->minimaxChild;
            }
        }

        if(board->getOwnColor()==2){
            if(children[i]->minimaxChild->mean<minValue){
                minValue=children[i]->minimaxChild->mean;
                minimaxChild=children[i]->minimaxChild;
            }
        }

        i++;
    }
}

#endif /* Node_h */
```

Board.h:

```
#ifndef Board_h
#define Board_h

#include "Functions.h"
#include "Position.h"

class Board{

private:

    int gameType; // 1 black to live 2 black to kill
    int boardSize; // n*n board

    int **space; // 0 not playable 1 playable
    int **important; // 0 not important 1 important
    int **alive; // 0 not alive 1 alive
    int colorOfOwn; // 1 black to play 2 white to play
    int colorOfOpponent; // 1 black to play 2 white to play
    int positionOfKo[2]; // [0] row [1] column
    int whoWinKo; // 0 no one 1 black 2 white
    int remainingKoAdvantage; // default 5
    int continuousPass; // if increase to 3, game ends.
```

```
    int accumulatedPass; // if up to 4``, black wins, if down to -4, white wins
    int **stone; // 0 empty 1 black 2 white -1 border
    int countLiberty(int row, int column);
    void subCountLiberty(int **liberty, int row, int column);
    void capture(int row, int column, int color); // color default 0
    bool checkKo(int row, int column); // true if move is a ko
    bool checkLegalMove(int row, int column); // true if move is legal
    void getLegalMove(Position* position);
    bool checkFillSpace(int row, int column); //true if fill space
    bool checkFillEye(int row, int column); // true if fill eye
    bool checkPass(); //if allow to pass 1. if not 0.

    bool checkDead(); // true if dead
    bool checkAlive(); // true if alive
    void updateAlive(int row, int column); // if the move connects to alive groups, all stones connected
are alive and not important
    void changePlayer();

public:
    void test();
    Board(char fileName[]);
    Board(Board* copyBoard);
    ~Board();
    void humanInterface();
    void computerInterface(Position* position);
    void placeStone(int row, int column);
    float getScore(); // have nine states: 0 undecided, 1 lose ko and alive, 2 alive, 3 lose ko and seki,
4 seki, 5 win ko and alive, 6 lose ko and dead, 7 win ko and seki, 8 dead, 9 win ko and dead
    int getBoardSize();
    int getOwnColor();
    void showBoard();
    void viewData();

};

//---------------------PUBLIC----------------------------

void Board::test(){

    while(getScore()==-2){

        humanInterface();

    }

    cout<<"The score is "<<getScore()<<endl;

         showBoard();

}

Board::Board(char fileName[]){

    int counter1=0,counter2=0;
    int buffer=0;
    string input="";

    //if file not read

    ifstream readFile;
    readFile.open(fileName);
    if(!readFile.is_open()){
        cout<<"Board() error: file is not read."<<endl;
        return;
    }
    readFile.close();

    //set value

    readFile.open(fileName);

    while(!readFile.eof()){

        readFile>>input;

        if(input=="size"){
         readFile>>buffer;
         boardSize=buffer;
         //create array
                     counter1=0,counter2=0;
                stone=new int *[boardSize+2];
                while(counter1<boardSize+2){
                stone[counter1]=new int[boardSize+2];
                counter1++;
```

```
                }
                counter1=0,counter2=0;
                space=new int *[boardSize+2];
                while(counter1<boardSize+2){
                space[counter1]=new int[boardSize+2];
                counter1++;
                }
                counter1=0,counter2=0;
                important=new int *[boardSize+2];
                while(counter1<boardSize+2){
                important[counter1]=new int[boardSize+2];
                counter1++;
                }
                counter1=0,counter2=0;
                alive=new int *[boardSize+2];
                while(counter1<boardSize+2){
                alive[counter1]=new int[boardSize+2];
                counter1++;
                }
                }

                if(input=="type"){
                         readFile>>buffer;
                         gameType=buffer;
                }

                if(input=="Stone"){
                         counter1=0,counter2=0;
        while(counter1<boardSize+2){
            counter2=0;
            while(counter2<boardSize+2){
                readFile>>buffer;
                stone[counter1][counter2]=buffer;
                counter2++;
            }
            counter1++;
        }
               }

               if(input=="Space"){
        counter1=0,counter2=0;
        while(counter1<boardSize+2){
            counter2=0;
            while(counter2<boardSize+2){
                readFile>>buffer;
                space[counter1][counter2]=buffer;
                counter2++;
            }
            counter1++;
        }
    }

    if(input=="Important"){
        counter1=0,counter2=0;
        while(counter1<boardSize+2){
            counter2=0;
            while(counter2<boardSize+2){
                readFile>>buffer;
                important[counter1][counter2]=buffer;
                counter2++;
            }
            counter1++;
        }
    }

    if(input=="Alive"&&!readFile.eof()){
        counter1=0,counter2=0;
        while(counter1<boardSize+2){
            counter2=0;
            while(counter2<boardSize+2){
               readFile>>buffer;
                alive[counter1][counter2]=buffer;
                counter2++;
            }
            counter1++;
        }
    }
}

colorOfOwn=1;
colorOfOpponent=2;
positionOfKo[0]=0;
positionOfKo[1]=0;
whoWinKo=0;
```

```
    remainingKoAdvantage=5;
    continuousPass=0;
    accumulatedPass=0;

    return;
}

Board::Board(Board* copyBoard){

    int counter1=0,counter2=0;

    //set board size

    boardSize=copyBoard->boardSize;

    //create array

    counter1=0,counter2=0;
    stone=new int *[boardSize+2];
    while(counter1<boardSize+2){
        stone[counter1]=new int[boardSize+2];
        counter1++;
    }
    counter1=0,counter2=0;
    space=new int *[boardSize+2];
    while(counter1<boardSize+2){
        space[counter1]=new int[boardSize+2];
        counter1++;
    }
    counter1=0,counter2=0;
    important=new int *[boardSize+2];
    while(counter1<boardSize+2){
        important[counter1]=new int[boardSize+2];
        counter1++;
    }
    counter1=0,counter2=0;
    alive=new int *[boardSize+2];
    while(counter1<boardSize+2){
        alive[counter1]=new int[boardSize+2];
        counter1++;
    }

    //set array

    counter1=0,counter2=0;
    while(counter1<boardSize+2){
        counter2=0;
        while(counter2<boardSize+2){
            stone[counter1][counter2]=copyBoard->stone[counter1][counter2];
            space[counter1][counter2]=copyBoard->space[counter1][counter2];
            important[counter1][counter2]=copyBoard->important[counter1][counter2];
            alive[counter1][counter2]=copyBoard->alive[counter1][counter2];
            counter2++;
        }
        counter1++;
    }

    //set other attributes

    gameType=copyBoard->gameType;
    colorOfOwn=copyBoard->colorOfOwn;
    colorOfOpponent=copyBoard->colorOfOpponent;
    positionOfKo[0]=copyBoard->positionOfKo[0];
    positionOfKo[1]=copyBoard->positionOfKo[1];
    whoWinKo=copyBoard->whoWinKo;
    remainingKoAdvantage=copyBoard->remainingKoAdvantage;
    continuousPass=copyBoard->continuousPass;
    accumulatedPass=copyBoard->accumulatedPass;

}

Board::~Board(){

    int counter1=0;

    while(counter1<boardSize+2){
        delete stone[counter1];
        delete space[counter1];
        delete important[counter1];
        delete alive[counter1];
        counter1++;
    }

    delete stone;
```

```
    delete space;
    delete important;
    delete alive;

}

void Board::humanInterface(){

    showBoard();

    if(colorOfOwn==1){
        cout<<"Black's turn."<<endl;
    }
    else{
        cout<<"White's turn."<<endl;
    }

    cout<<"row, column"<<endl;
    int row=0, column=0;
    cin>>row>>column;

    placeStone(row, column);

}

void Board::computerInterface(Position* position){

    getLegalMove(position);

    if(checkPass()){
        position->position[0][0]=1;
    }

    //if fill eye, set position to 0

    int counter1=1,counter2=1;
    while(counter1<boardSize+1){
         counter2=1;
         while(counter2<boardSize+2){
                  if(checkFillEye(counter1, counter2)){
                           position->position[counter1][counter2]=0;
                           }
                  counter2++;
                  }
         counter1++;
         }

    position->getLegalPosition();

}

void Board::placeStone(int row, int column){

    //if pass

    if(row==0&&column==0){
        if(!checkPass()){
            cout<<"Place stone error, not able to pass."<<endl;
            return;
        }

        if(colorOfOwn==1){
            accumulatedPass++;
        }
        else{
            accumulatedPass--;
        }

        continuousPass++;
        positionOfKo[0]=0;
        positionOfKo[1]=0;

        //change player
        changePlayer();

        return;
    }

    //if not legal

    if(!checkLegalMove(row, column)){
        cout<<"Place stone error, illegal move."<<endl;
        return;
    }
```

```
    else{
        continuousPass=0;
    }

    //if ko

         bool ifkohappen=false;

    if(checkKo(row, column)){

        ifkohappen=true;

        //if take ko back

        if(positionOfKo[0]==row&&positionOfKo[1]==column){
            whoWinKo=colorOfOwn;
            remainingKoAdvantage--;
        }

    }

    //place stone

    stone[row][column]=colorOfOwn;

    //set ko position

    if(ifkohappen){
        if(countLiberty(row-1, column)==0){
            positionOfKo[0]=row-1;
            positionOfKo[1]=column;
        }
        if(countLiberty(row+1, column)==0){
            positionOfKo[0]=row+1;
            positionOfKo[1]=column;
        }
        if(countLiberty(row, column-1)==0){
            positionOfKo[0]=row;
            positionOfKo[1]=column-1;
        }
        if(countLiberty(row, column+1)==0){
            positionOfKo[0]=row;
            positionOfKo[1]=column+1;
        }
    }
    else{
        positionOfKo[0]=0;
        positionOfKo[1]=0;
    }

    //if capture opponent stone

    if(stone[row-1][column]==colorOfOpponent&&countLiberty(row-1, column)==0){
        capture(row-1, column, 0);
    }
    if(stone[row+1][column]==colorOfOpponent&&countLiberty(row+1, column)==0){
        capture(row+1, column, 0);
    }
    if(stone[row][column-1]==colorOfOpponent&&countLiberty(row, column-1)==0){
        capture(row, column-1, 0);
    }
    if(stone[row][column+1]==colorOfOpponent&&countLiberty(row, column+1)==0){
        capture(row, column+1, 0);
    }

    //update alive

    if(
         (stone[row-1][column]==colorOfOwn&&alive[row-1][column]==1)||
         (stone[row+1][column]==colorOfOwn&&alive[row+1][column]==1)||
         (stone[row][column-1]==colorOfOwn&&alive[row][column-1]==1)||
         (stone[row][column+1]==colorOfOwn&&alive[row][column+1]==1)
         ){
        updateAlive(row, column);
    }

    //change player

    changePlayer();

}

float Board::getScore(){
```

```
//if accumulated pass is over 4

if(accumulatedPass>=4) {
    return 1;
}
else if(accumulatedPass<=-4){
    return -1;
}

//if seki

if(continuousPass==3){
    if(whoWinKo==0){
        if(gameType==1){
            return 0.7;
        }
        if(gameType==2){
            return 0.3;
        }
    }
    if(whoWinKo==1){
        if(gameType==1){
            return 0.3;
        }
        if(gameType==2){
            return 0.15;
        }
    }
    if(whoWinKo==2){
        if(gameType==1){
            return 0.85;
        }
        if(gameType==2){
            return 0.7;
        }
    }
}

//if dead

if(checkDead()){
    if(whoWinKo==0){
        if(gameType==1){
            return 0;
        }
        if(gameType==2){
            return 1;
        }
    }
    if(whoWinKo==1){
        if(gameType==1){
            return 0;
        }
        if(gameType==2){
            return 0.5;
        }
    }
    if(whoWinKo==2){
        if(gameType==1){
            return 0.5;
        }
        if(gameType==2){
            return 1;
        }
    }
}

//if alive

if(checkAlive()){
    if(whoWinKo==0){
        if(gameType==1){
            return 1;
        }
        if(gameType==2){
            return 0;
        }
    }
    if(whoWinKo==1){
        if(gameType==1){
            return 0.5;
        }
        if(gameType==2){
            return 0;
```

```
            }
        }
        if(whoWinKo==2){
            if(gameType==1){
                return 1;
            }
            if(gameType==2){
                return 0.5;
            }
        }
    }

    return -2;
}

int Board::getBoardSize(){
    return boardSize;
}

int Board::getOwnColor(){
    return colorOfOwn;
}

void Board::showBoard(){
    int counter1=1,counter2=1;
    cout<<endl;

    //top numbers

    counter1=1;
    cout<<setw(2)<<" "<<setw(2)<<" ";
    while(counter1<boardSize+1){
        cout<<left<<setw(4)<<counter1;
        counter1++;
    }
    cout<<endl<<endl;

    //board

    counter1=1;
    while(counter1<boardSize+1){
        counter2=1;

        //left numbers

        cout<<setw(2)<<counter1<<setw(2)<<" ";

        //line with stones

        while(counter2<boardSize+1){
            if(stone[counter1][counter2]==0){
                  //cout<<setw(2)<<(char)250;
                cout<<setw(2)<<"+";
            }
            else if(stone[counter1][counter2]==1){
                  cout<<"▓"<<setw(1)<<"";
                //cout<<setw(2)<<"X";
            }
            else if(stone[counter1][counter2]==2){
                  cout<<"▒"<<setw(1)<<"";
                //cout<<setw(2)<<"O";
            }
            if(counter2!=boardSize){
                  cout<<setw(2)<<"";
                //cout<<setw(2)<<"+";
            }

            counter2++;
        }

        //right numbers

        cout<<setw(2)<<" "<<setw(2)<<counter1;

        cout<<endl;

        //line without stones

        if(counter1!=boardSize){
            counter2=1;
            cout<<setw(2)<<" "<<setw(2)<<" ";
            while(counter2<boardSize+1){
                  cout<<setw(2)<<"";
                //cout<<setw(2)<<"+";
```

```
                if(counter2!=boardSize){
                    cout<<setw(2)<<"";
                }
                counter2++;
            }
        }
        cout<<endl;
        counter1++;
    }

    //bottom numbers

    counter1=1;
    cout<<setw(2)<<" "<<setw(2)<<" ";
    while(counter1<boardSize+1){
        cout<<left<<setw(4)<<counter1;
        counter1++;
    }
    cout<<endl<<endl;

}

void Board::viewData(){

    int counter1=0,counter2=0;

    cout<<"Stone"<<endl;
    counter1=1,counter2=1;
    while(counter1<boardSize+1){
        counter2=1;
        while(counter2<boardSize+1){
            cout<<setw(3)<<stone[counter1][counter2];
            counter2++;
        }
        cout<<endl;
        counter1++;
    }
    cout<<endl;

    cout<<"Space"<<endl;
    counter1=1,counter2=1;
    while(counter1<boardSize+1){
        counter2=1;
        while(counter2<boardSize+1){
            cout<<setw(3)<<space[counter1][counter2];
            counter2++;
        }
        cout<<endl;
        counter1++;
    }
    cout<<endl;

    cout<<"Important"<<endl;
    counter1=1,counter2=1;
    while(counter1<boardSize+1){
        counter2=1;
        while(counter2<boardSize+1){
            cout<<setw(3)<<important[counter1][counter2];
            counter2++;
        }
        cout<<endl;
        counter1++;
    }
    cout<<endl;

    cout<<"Alive"<<endl;
    counter1=1,counter2=1;
    while(counter1<boardSize+1){
        counter2=1;
        while(counter2<boardSize+1){
            cout<<setw(3)<<alive[counter1][counter2];
            counter2++;
        }
        cout<<endl;
        counter1++;
    }
    cout<<endl;

    cout<<"Legal positions"<<endl;

    Position position(boardSize);
    getLegalMove(&position);
    position.viewData();
    cout<<endl;
```

```
    cout<<"gameType "<<gameType<<endl<<endl;
    cout<<"colorOfOwn "<<colorOfOwn<<endl<<endl;
    cout<<"colorOfOpponent "<<colorOfOpponent<<endl<<endl;
    cout<<"positionOfKo "<<positionOfKo[0]<<" "<<positionOfKo[1]<<endl<<endl;
    cout<<"whoWinKo "<<whoWinKo<<endl<<endl;
    cout<<"remainingKoAdvantage "<<remainingKoAdvantage<<endl<<endl;
    cout<<"continuousPass "<<continuousPass<<endl<<endl;

}

//----------------------PRIVATE----------------------------

int Board::countLiberty(int row, int column){

    //precondition
    //the row and column are in the range of 1~boardsize
    //the selected space must be a black or white stone

    //if out of board

    if(row<0||row>boardSize+1||column<0||column>boardSize+1){
        cout<<"Count liberty error: count liberty out of board."<<endl;
        return -1;
    }

    //if not stone

    if(!(stone[row][column]==1||stone[row][column]==2)){
        return -1;
    }

    //create array

    int **liberty;
    int counter1=0,counter2=0;
    liberty=new int *[boardSize+2];
    counter1=0;
    while(counter1<boardSize+2){
        liberty[counter1]=new int[boardSize+2];
        counter1++;
    }

    counter1=0,counter2=0;
    while(counter1<boardSize+2){
        counter2=0;
        while(counter2<boardSize+2){
            liberty[counter1][counter2]=stone[counter1][counter2];
            counter2++;
        }
        counter1++;
    }

    //mark liberty with 3

    subCountLiberty(liberty, row, column);

    //count number of 3

    int countLiberty=0;
    counter1=0;
    while(counter1<boardSize+2){
        counter2=0;
        while(counter2<boardSize+2){
            if(liberty[counter1][counter2]==3){
                countLiberty++;
            }
            counter2++;
        }
        counter1++;
    }

    //delete array

    counter1=0;
    while(counter1<boardSize+2){
        delete liberty[counter1];
        counter1++;
    }
    delete[] liberty;

    //return value

    return countLiberty;
```

```
}

void Board::subCountLiberty(int **liberty, int row, int column){

    //variables

    bool up=false,down=false,left=false,right=false;

    //check down

    if(liberty[row+1][column]==liberty[row][column]&&liberty[row][column]!=4){
        down=true;
    }
    if(liberty[row+1][column]==0){
        liberty[row+1][column]=3;
    }

    //check up

    if(liberty[row-1][column]==liberty[row][column]&&liberty[row][column]!=4){
        up=true;
    }
    if(liberty[row-1][column]==0){
        liberty[row-1][column]=3;
    }

    //check left

    if(liberty[row][column-1]==liberty[row][column]&&liberty[row][column]!=4){
        left=true;
    }
    if(liberty[row][column-1]==0){
        liberty[row][column-1]=3;
    }

    //check right

    if(liberty[row][column+1]==liberty[row][column]&&liberty[row][column]!=4){
        right=true;
    }
    if(liberty[row][column+1]==0){
        liberty[row][column+1]=3;
    }

    //prevent infinite calling

    liberty[row][column]=4;

    //calling

    if(down){
        subCountLiberty(liberty, row+1, column);
    }

    if(up){
        subCountLiberty(liberty, row-1, column);
    }

    if(left){
        subCountLiberty(liberty, row, column-1);
    }

    if(right){
        subCountLiberty(liberty, row, column+1);
    }

}

void Board::capture(int row, int column, int color){

    //if out of board

    if(row<0||row>boardSize+1||column<0||column>boardSize+1){
        cout<<"Capture error: out of board."<<endl;
        return;
    }

    //initialize

    if(color==0) {

        color=stone[row][column];

        //if the stone has liberty
```

```
        if(countLiberty(row, column)!=0){
         //cout<<row<<" "<<column<<" has liberty"<<endl;
            return;
        }

    }

    //capture

    if(stone[row][column]==color){
        stone[row][column]=0;
        capture(row-1, column, color);
        capture(row+1, column, color);
        capture(row, column-1, color);
        capture(row, column+1, color);
    }
    else{
        return;
    }

}

bool Board::checkKo(int row, int column){

    //if out of board

    if(row<1||row>boardSize||column<1||column>boardSize){
         cout<<"Check ko error: out of board."<<endl;
         return false;
         }

    //if not check empty space

    if(stone[row][column]!=0){
         cout<<"Check ko error: not check empty space."<<endl;
        return false;
    }

    Board copy(this);

    //suppose the position is played

    copy.stone[row][column]=colorOfOwn;

    //if the position is surrounded by either opponent's stones or border

    if(copy.countLiberty(row, column)==0){

        //if only one opponent stone has 0 liberty

        int zeroLibertyCounter=0;
        int zeroLibertyPosition[2]={0};
        if(copy.stone[row-1][column]==copy.colorOfOpponent&&copy.countLiberty(row-1, column)==0){
            zeroLibertyCounter++;
            zeroLibertyPosition[0]=row-1;
            zeroLibertyPosition[1]=column;
        }
        if(copy.stone[row+1][column]==copy.colorOfOpponent&&copy.countLiberty(row+1, column)==0){
            zeroLibertyCounter++;
            zeroLibertyPosition[0]=row+1;
            zeroLibertyPosition[1]=column;
        }
        if(copy.stone[row][column-1]==copy.colorOfOpponent&&copy.countLiberty(row, column-1)==0){
            zeroLibertyCounter++;
            zeroLibertyPosition[0]=row;
            zeroLibertyPosition[1]=column-1;
        }
        if(copy.stone[row][column+1]==copy.colorOfOpponent&&copy.countLiberty(row, column+1)==0){
            zeroLibertyCounter++;
            zeroLibertyPosition[0]=row;
            zeroLibertyPosition[1]=column+1;
        }

        //if only one stone has 0 liberty

        if(zeroLibertyCounter==1){

            //if the 0 liberty stone are surronded by opponent's stones

            if((copy.stone[zeroLibertyPosition[0]-
1][zeroLibertyPosition[1]]==copy.colorOfOwn||copy.stone[zeroLibertyPosition[0]-
1][zeroLibertyPosition[1]]==-
1)&&(copy.stone[zeroLibertyPosition[0]+1][zeroLibertyPosition[1]]==copy.colorOfOwn||copy.stone[zeroLibert
```

```
yPosition[0]+1][zeroLibertyPosition[1]]==-1)&&(copy.stone[zeroLibertyPosition[0]][zeroLibertyPosition[1]-
1]==copy.colorOfOwn||copy.stone[zeroLibertyPosition[0]][zeroLibertyPosition[1]-1]==-
1)&&(copy.stone[zeroLibertyPosition[0]][zeroLibertyPosition[1]+1]==copy.colorOfOwn||copy.stone[zeroLibert
yPosition[0]][zeroLibertyPosition[1]+1]==-1)){

                return true;
            }
        }

    }

    //remove the move
    return false;
}

bool Board::checkLegalMove(int row, int column){

    //if out of board

    if(row<0||row>boardSize+1||column<0||column>boardSize+1){
         cout<<"Check legal move error: out of board."<<endl;
        return false;
    }

    //if out of space

    if(space[row][column]!=1){
        return false;
    }

    //if not empty

    if(stone[row][column]!=0){
        return false;
    }

    //if ko

    if(checkKo(row, column)){

        //if take ko back

        if(row==positionOfKo[0]&&column==positionOfKo[1]){

            //if no one wins ko

            if(whoWinKo==0){
                return true;
            }

            //if opponents wins ko

            if(whoWinKo==colorOfOpponent) {
                return false;
            }

            //if ownself wins ko

            if(whoWinKo==colorOfOwn){
                if(remainingKoAdvantage!=0){
                    return true;
                }
                else{
                    return false;
                }
            }
        }

        //if not take ko back

        else{
            return true;
        }
    }

    //make a copy board and assume the position is played

    Board copy(this);
    copy.stone[row][column]=copy.colorOfOwn;

    //if has liberty

    if (copy.countLiberty(row, column)!=0) {
        return true;
```

```
    }

    //if no liberty

    if (copy.countLiberty(row, column)==0){

        //if any surrounding opponent's stones have no liberty

        if(
                    (copy.stone[row-1][column]==copy.colorOfOpponent&&copy.countLiberty(row-1,
column)==0)||
                    (copy.stone[row+1][column]==copy.colorOfOpponent&&copy.countLiberty(row+1,
column)==0)||
                    (copy.stone[row][column-1]==copy.colorOfOpponent&&copy.countLiberty(row, column-
1)==0)||
                    (copy.stone[row][column+1]==copy.colorOfOpponent&&copy.countLiberty(row,
column+1)==0)){

            return true;
        }

        //if all surrounding opponent's stones have liberty

        else{
            return false;
        }

    }

    return true;
}

void Board::getLegalMove(Position* position){

    int counter1=0, counter2=0;

    //set value
    counter1=0, counter2=0;
    while(counter1<boardSize+2){
        counter2=0;
        while(counter2<boardSize+2){

            if(checkLegalMove(counter1, counter2)){
                position->position[counter1][counter2]=1;
            }
            else{
                position->position[counter1][counter2]=0;
            }

            counter2++;
        }
        counter1++;
    }

}

bool Board::checkFillSpace(int row, int column){

    bool result=false;

    if((stone[row-1][column]==colorOfOwn||stone[row-1][column]==-1)&&
       (stone[row+1][column]==colorOfOwn||stone[row+1][column]==-1)&&
       (stone[row][column-1]==colorOfOwn||stone[row][column-1]==-1)&&
       (stone[row][column+1]==colorOfOwn||stone[row][column+1]==-1)
       ){
            result=true;
       }

    return result;

}

bool Board::checkFillEye(int row, int column){

         //if out of board
         if(row<1||column<1||row>boardSize+1||column>boardSize+1){
                  cout<<"Check fill eye error: out of board"<<endl;
                  return false;
         }

         int countBorder=0;
         if(stone[row-1][column]==-1){
                  countBorder++;
         }
```

```
if(stone[row+1][column]==-1){
        countBorder++;
}
if(stone[row][column-1]==-1){
        countBorder++;
}
if(stone[row][column+1]==-1){
        countBorder++;
}

//if at cornor

if(countBorder==2){
        int countSurroundingOwnStones=0;
        if(stone[row-1][column-1]==colorOfOwn){
                countSurroundingOwnStones++;
        }
        if(stone[row][column-1]==colorOfOwn){
                countSurroundingOwnStones++;
        }
        if(stone[row+1][column-1]==colorOfOwn){
                countSurroundingOwnStones++;
        }
        if(stone[row-1][column]==colorOfOwn){
                countSurroundingOwnStones++;
        }
        if(stone[row+1][column]==colorOfOwn){
                countSurroundingOwnStones++;
        }
        if(stone[row-1][column+1]==colorOfOwn){
                countSurroundingOwnStones++;
        }
        if(stone[row][column+1]==colorOfOwn){
                countSurroundingOwnStones++;
        }
        if(stone[row+1][column+1]==colorOfOwn){
                countSurroundingOwnStones++;
        }

        if(countSurroundingOwnStones==3){
                return true;
        }
        else{
                return false;
        }
}

//if at side

if(countBorder==1){
        int countSurroundingOwnStones=0;
        if(stone[row-1][column-1]==colorOfOwn){
                countSurroundingOwnStones++;
        }
        if(stone[row][column-1]==colorOfOwn){
                countSurroundingOwnStones++;
        }
        if(stone[row+1][column-1]==colorOfOwn){
                countSurroundingOwnStones++;
        }
        if(stone[row-1][column]==colorOfOwn){
                countSurroundingOwnStones++;
        }
        if(stone[row+1][column]==colorOfOwn){
                countSurroundingOwnStones++;
        }
        if(stone[row-1][column+1]==colorOfOwn){
                countSurroundingOwnStones++;
        }
        if(stone[row][column+1]==colorOfOwn){
                countSurroundingOwnStones++;
        }
        if(stone[row+1][column+1]==colorOfOwn){
                countSurroundingOwnStones++;
        }

        if(countSurroundingOwnStones==5){
                return true;
        }
        else{
                return false;
        }
}
```

```
            //if at center

            if(countBorder==0){

                    int countSurroundingOwnStones=0;

                    //check surrounding (side to the space)

                    if(stone[row][column-1]==colorOfOwn){
                            countSurroundingOwnStones++;
                    }
                    else{
                            return false;
                    }
                    if(stone[row-1][column]==colorOfOwn){
                            countSurroundingOwnStones++;
                    }
                    else{
                            return false;
                    }
                    if(stone[row+1][column]==colorOfOwn){
                            countSurroundingOwnStones++;
                    }
                    else{
                            return false;
                    }
                    if(stone[row][column+1]==colorOfOwn){
                            countSurroundingOwnStones++;
                    }
                    else{
                            return false;
                    }

                    //check surrounding (diagnal to the space)

                    if(stone[row-1][column-1]==colorOfOwn){
                            countSurroundingOwnStones++;
                    }
                    if(stone[row+1][column-1]==colorOfOwn){
                            countSurroundingOwnStones++;
                    }
                    if(stone[row-1][column+1]==colorOfOwn){
                            countSurroundingOwnStones++;
                    }
                    if(stone[row+1][column+1]==colorOfOwn){
                            countSurroundingOwnStones++;
                    }

                    if(countSurroundingOwnStones>=7){
                            return true;
                    }
                    else{
                            return false;
                    }
            }

        return false;

}

bool Board::checkPass(){

        //create position

        Position legalPosition(boardSize);

        //get legal move

        getLegalMove(&legalPosition);

        //count legal space

        int legalSpace=0;
        int counter1=0,counter2=0;
        while (counter1<boardSize+2) {
            counter2=0;
            while(counter2<boardSize+2){
                if(legalPosition.position[counter1][counter2]==1){
                    legalSpace++;
                }
                counter2++;
            }
            counter1++;
        }
```

```
    //if no position is available, player can pass

    if(legalSpace==0){
        return true;
    }

    //if all available positions are either:
        //1.Take back ko.
    //2.connected to own groups and have only one liberty after play.
        //Player can pass.

    else{

        counter1=0,counter2=0;

        while(counter1<boardSize+2){

            counter2=0;

            while(counter2<boardSize+2){

                if(legalPosition.position[counter1][counter2]==1){

                   bool ifFillSpace=false;
                   bool ifTakeKoBack=false;
                   bool ifHaveOneLiberty=false;
                   bool ifConnectedToOwnStones=false;
                     Board copy(this);
                     copy.placeStone(counter1, counter2);

                     //if fill eye
                     ifFillSpace=checkFillSpace(counter1, counter2);

                     //if the position played in copy board is ko position in original board
                     ifTakeKoBack=(counter1==positionOfKo[0]&&counter2==positionOfKo[1]);

                     //if the position played in copy board has one liberty
                     ifHaveOneLiberty=copy.countLiberty(counter1, counter2)==1;

                     //if the place is connected to own stones,
                     //colorOfOwn is from original board
                                          //because already place a stone on copy board
                                          //so the color is changed
                     ifConnectedToOwnStones=
                                          copy.stone[counter1-1][counter2]==colorOfOwn||
                                          copy.stone[counter1+1][counter2]==colorOfOwn||
                                          copy.stone[counter1][counter2-1]==colorOfOwn||
                                          copy.stone[counter1][counter2+1]==colorOfOwn;

                                          if(
                                          !(
                                          ifFillSpace||
                                          ifTakeKoBack||
                                          (ifHaveOneLiberty&&ifConnectedToOwnStones)
                                          )
                                          ){
                                                  return false;
                                          }
                }
                counter2++;
            }
            counter1++;
        }

        return true;
    }
}

bool Board::checkAlive(){

    int counter1=0, counter2=0;
    bool ifPlaceNewStone=true;

    //if no important stone, then return true

    counter1=1,counter2=1;
    int countImportant=0;
    while (counter1<boardSize+1) {
        counter2=1;
        while(counter2<boardSize+1){
            if(important[counter1][counter2]==1){
                countImportant++;
            }
```

```
        counter2++;
    }
    counter1++;
}
if(countImportant==0){
    return true;
}

//create a board

Board copy(this);

while(ifPlaceNewStone){
    ifPlaceNewStone=false;

    //fill the board with killing side stone if the space is not one space eye of living side

    counter1=1;
    while(counter1<copy.boardSize+1){
        counter2=1;
        while(counter2<copy.boardSize+1){

            //if the space is empty

            if(copy.stone[counter1][counter2]==0){

                //if black to live

                if(gameType==1){

                    //if space is not one space eye

                    if(
                                                !(
                                                (copy.stone[counter1-1][counter2]==1||
                                                copy.stone[counter1-1][counter2]==-1)&&
                                                (copy.stone[counter1+1][counter2]==1||
                                                copy.stone[counter1+1][counter2]==-1)&&
                                                (copy.stone[counter1][counter2-1]==1||
                                                copy.stone[counter1][counter2-1]==-1)&&
                                                (copy.stone[counter1][counter2+1]==1||
                                                copy.stone[counter1][counter2+1]==-1))
                                                ){
                        copy.stone[counter1][counter2]=2;
                        ifPlaceNewStone=true;
                    }
                }

                //if black to kill

                if(gameType==2){

                    //if space is not one space eye

                    if(
                                                !(
                                                (copy.stone[counter1-1][counter2]==2||
                                                copy.stone[counter1-1][counter2]==-1)&&
                                                (copy.stone[counter1+1][counter2]==2||
                                                copy.stone[counter1+1][counter2]==-1)&&
                                                (copy.stone[counter1][counter2-1]==2||
                                                copy.stone[counter1][counter2-1]==-1)&&
                                                (copy.stone[counter1][counter2+1]==2||
                                                copy.stone[counter1][counter2+1]==-1))
                                                ){
                        copy.stone[counter1][counter2]=1;
                        ifPlaceNewStone=true;
                    }
                }
            }
            counter2++;
        }
        counter1++;
    }

    //capture the zero liberty stones of living side

    counter1=1, counter2=1;
    while(counter1<copy.boardSize+1){
        counter2=1;
        while(counter2<copy.boardSize+1){

            //if black to live
```

```
                if(gameType==1){
                    if(copy.stone[counter1][counter2]==1){
                        copy.capture(counter1, counter2,0);
                    }
                }

                //if black to kill

                if(gameType==2){
                    if(copy.stone[counter1][counter2]==2){
                        copy.capture(counter1, counter2,0);
                    }
                }
                counter2++;
            }
            counter1++;
        }

        //place stone in one space eye. try to capture. if not able, take the stone away

        counter1=1, counter2=1;
        while(counter1<copy.boardSize+1){
            counter2=1;
            while(counter2<copy.boardSize+1){

                //if space is empty

                if(copy.stone[counter1][counter2]==0){

                    //if black to live

                    if(gameType==1){

                        //if is one space eye

                        if(
                                                    (copy.stone[counter1-1][counter2]==1||
                                                    copy.stone[counter1-1][counter2]==-1)&&
                                                    (copy.stone[counter1+1][counter2]==1||
                                                    copy.stone[counter1+1][counter2]==-1)&&
                                                    (copy.stone[counter1][counter2-1]==1||
                                                    copy.stone[counter1][counter2-1]==-1)&&
                                                    (copy.stone[counter1][counter2+1]==1||
                                                    copy.stone[counter1][counter2+1]==-1)
                                                    ){

                            copy.stone[counter1][counter2]=2;
                            bool ifcapture=false;

                            if(copy.stone[counter1-1][counter2]==1&&
                                                        copy.countLiberty(counter1-1,
counter2)==0){

                                                                copy.capture(counter1-1,
counter2, 0);
                                ifcapture=true;

                            }

                            if(copy.stone[counter1+1][counter2]==1&&
                                                        copy.countLiberty(counter1+1,
counter2)==0){

                                copy.capture(counter1+1, counter2, 0);
                                ifcapture=true;

                            }
                            if(copy.stone[counter1][counter2-1]==1&&
                                                        copy.countLiberty(counter1, counter2-
1)==0){

                                copy.capture(counter1, counter2-1, 0);
                                ifcapture=true;

                            }
                            if(copy.stone[counter1][counter2+1]==1&&
                                                        copy.countLiberty(counter1,
counter2+1)==0){

                                copy.capture(counter1, counter2+1, 0);
                                ifcapture=true;

                            }
```

```
                            //if not capture anything

                            if(!ifcapture){
                                copy.stone[counter1][counter2]=0;
                            }

                        }
                    }

                    //if black to kill

                    if(gameType==2){

                        //if is one space eye

                        if(
                                                    (copy.stone[counter1-1][counter2]==2||
                                                    copy.stone[counter1-1][counter2]==-1)&&
                                                    (copy.stone[counter1+1][counter2]==2||
                                                    copy.stone[counter1+1][counter2]==-1)&&
                                                    (copy.stone[counter1][counter2-1]==2||
                                                    copy.stone[counter1][counter2-1]==-1)&&
                                                    (copy.stone[counter1][counter2+1]==2||
                                                    copy.stone[counter1][counter2+1]==-1)
                                                    ){

                            copy.stone[counter1][counter2]=1;
                            bool ifcapture=false;

                            if(copy.stone[counter1-1][counter2]==2&&
                                                            copy.countLiberty(counter1-1,
counter2)==0){
                                copy.capture(counter1-1, counter2, 0);
                                ifcapture=true;
                            }
                            if(copy.stone[counter1+1][counter2]==2&&
                                                            copy.countLiberty(counter1+1,
counter2)==0){
                                copy.capture(counter1+1, counter2, 0);
                                ifcapture=true;
                            }
                            if(copy.stone[counter1][counter2-1]==2&&
                                                            copy.countLiberty(counter1, counter2-
1)==0){
                                copy.capture(counter1, counter2-1, 0);
                                ifcapture=true;
                            }
                            if(copy.stone[counter1][counter2+1]==2&&
                                                            copy.countLiberty(counter1,
counter2+1)==0){
                                copy.capture(counter1, counter2+1, 0);
                                ifcapture=true;
                            }

                            //if not capture anything

                            if(!ifcapture){
                                copy.stone[counter1][counter2]=0;
                            }

                        }
                    }
                }
                counter2++;
            }
            counter1++;
        }

        //if important stones are capture, then return false

        counter1=0, counter2=0;
        while(counter1<copy.boardSize+2){
            counter2=0;
            while(counter2<copy.boardSize+2){

                //if black to live

                if(gameType==1){
                    if(copy.important[counter1][counter2]==1&&copy.stone[counter1][counter2]!=1){
                        return false;
                    }
                }

                //if black to live
```

```
                if(gameType==2){
                    if(copy.important[counter1][counter2]==1&&copy.stone[counter1][counter2]!=2){
                        return false;
                    }
                }
                counter2++;
            }
            counter1++;
        }

    }
    return true;
}

bool Board::checkDead(){

    int counter1=0, counter2=0;

    //if no important stone, then return false

    counter1=0,counter2=0;
    int countImportant=0;
    while (counter1<boardSize+2) {
        counter2=0;
        while(counter2<boardSize+2){
            if(important[counter1][counter2]==1){
                countImportant++;
            }
            counter2++;
        }
        counter1++;
    }
    if(countImportant==0){
        return false;
    }

    // if important stones are taken, stones are dead

    counter1=0;
    while(counter1<boardSize+2){
        counter2=0;
        while(counter2<boardSize+2){

            //black to live

            if(gameType==1){
                if(important[counter1][counter2]==1&&stone[counter1][counter2]!=1){
                    return true;
                }
            }

            //black to kill

            if(gameType==2){
                if(important[counter1][counter2]==1&&stone[counter1][counter2]!=2){
                    return true;
                }
            }
            counter2++;
        }
        counter1++;
    }

    return false;
}

void Board::updateAlive(int row, int column){
    if(!(stone[row][column]==colorOfOwn&&alive[row][column]==0)){
        return;
    }
    else{
        important[row][column]=0;
        alive[row][column]=1;
        updateAlive(row-1, column);
        updateAlive(row+1, column);
        updateAlive(row, column-1);
        updateAlive(row, column+1);
    }

}

void Board::changePlayer(){
```

```
          if(colorOfOwn==1){
                    colorOfOwn=2;
                    colorOfOpponent=1;
                    return;
          }

          if(colorOfOwn==2){
                    colorOfOwn=1;
                    colorOfOpponent=2;
                    return;
          }

}

#endif /* Board_h */
```

Position.h:

```
class Position{

private:

    int size;

public:

    int** position;
    vector<int>legalPositionRow;
    vector<int>legalPositionColumn;

          Position(int boardSize);
          ~Position();
    void getLegalPosition();
    int getNumberOfLegalMove();
          void viewData();

};

Position::Position(int boardSize){

          //create array

          int counter1=0, counter2=0;
          position=new int*[boardSize+2];
          while(counter1<boardSize+2){
                    position[counter1]=new int [boardSize+2];
                    counter1++;
          }

          //initialize

          size=boardSize;

          counter1=0, counter2=0;
          while(counter1<boardSize+2){

                    counter2=0;
                    while(counter2<boardSize+2){

                              position[counter1][counter2]=0;

                              counter2++;
                    }

                    counter1++;
          }

}

Position::~Position(){

          int counter=0;
          while(counter<size+2){
                    delete position[counter];
                    counter++;
          }
          delete [] position;

}

void Position::getLegalPosition(){

    //get legal positions
    int counter1=0, counter2=0;
```

```
    while(counter1<size+2){
        counter2=0;
        while(counter2<size+2){
            if(position[counter1][counter2]==1){
                legalPositionRow.push_back(counter1);
                legalPositionColumn.push_back(counter2);

            }
            counter2++;
        }
        counter1++;
    }

}

int Position::getNumberOfLegalMove(){

    int result=0;

    int counter1=0, counter2=0;

    while(counter1<size+2) {

        counter2=0;
        while(counter2<size+2){

            if(position[counter1][counter2]==1){
                result++;
            }

            counter2++;
        }

        counter1++;
    }

    return result;

}

void Position::viewData(){

          int counter1=1, counter2=1;

          cout<<endl;

          if(position[0][0]==1){
                    cout<<"Allow to pass"<<endl;
          }
          else{
                    cout<<"Not allow to pass"<<endl;
          }

          while(counter1<size+1){

                    counter2=1;
                    while(counter2<size+1){

                              cout<<position[counter1][counter2];

                              counter2++;

                    }
                    cout<<endl;

                    counter1++;

          }

          cout<<endl;

}
```

Function.h:

```
#ifndef Functions_h
#define Functions_h

#include <time.h>
#include <stdlib.h>
#include <math.h>
#include <vector>
#include <fstream>
#include <iomanip>
```

```
#include <iostream>
using namespace std;

float zeroToOne(){

    float result=1;
    while(result==1){
        result=(float)rand()/(float)RAND_MAX;
    }
    return result;

}

float rounding(float number, int to){

    float result=0;
    float digit=10;
    int i=0;
    while(i<to){
        digit*=10;
        i++;
    }

    bool ifNegative=false;
    if(number<0){
        number=-number;
        ifNegative=true;
    }

    number*=digit;
    number=(int)((number+5)/10);
    number=number/digit*10;

    result=number;

    if(ifNegative){
        return -result;
    }
    else{
        return result;
    }

}


#endif /* Functions_h */
```

Go.txt:

Figure B.1 is the example of the input data.

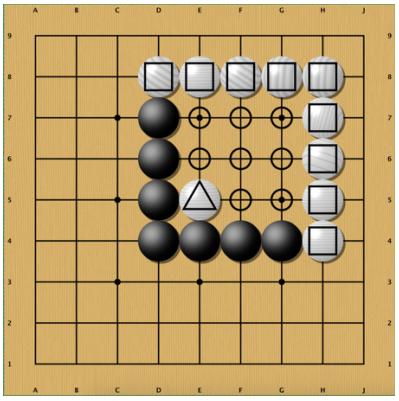


Figure B.1: The position of input data.

The objective for the black player is to capture the important stone (marked with a triangle). The position marked with circles are feasible spaces to play. The

stones marked with squares are alive groups, which the player want to prevent the important stone connect to.

The input file is the following.

Board size

9

Game type

1

Stone

-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 0 0 0 2 2 2 2 2 0 -1

-1 0 0 0 1 0 0 0 2 0 -1

-1 0 0 0 1 0 0 0 2 0 -1

-1 0 0 0 1 2 0 0 2 0 -1

-1 0 0 0 1 1 1 1 2 0 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

Space

-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 0 0 0 0 1 1 1 0 0 -1

-1 0 0 0 0 1 1 1 0 0 -1

-1 0 0 0 0 0 1 1 0 0 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

Important

-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 0 0 0 0 1 0 0 0 0 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

Alive

-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

-1 0 0 0 0 0 0 0 0 0 -1

-1	0	0	0	1	1	1	1	1	0	-1
-1	0	0	0	0	0	0	0	1	0	-1
-1	0	0	0	0	0	0	0	1	0	-1
-1	0	0	0	0	0	0	0	1	0	-1
-1	0	0	0	0	0	0	0	1	0	-1
-1	0	0	0	0	0	0	0	0	0	-1
-1	0	0	0	0	0	0	0	0	0	-1
-1	0	0	0	0	0	0	0	0	0	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Bibliography

[1] Campbell, M., Hoane, A. & Hsu, F. “Deep Blue.” Artif. Intell. 134, 57–83 (2002).

[2] Claude Shannon. “Programming a Computer for Playing Chess.” Philosophical Magazine, Ser.7, Vol. 41, No. 314 - March 1950.

[3] John Tromp and Gunnar Farnebäck. “Combinatorics of Go.” In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the Fifth International Conference on Computer and Games, Turin, Italy, 2006.

[4] Coulom, R. “Efficient selectivity and minimax operators in Monte-Carlo tree search.” In 5th International Conference on Computers and Games, 72–83 (2006)

[5] . Kocsis, L. & Szepesvári, C. “Bandit based Monte-Carlo planning.” In 15th European Conference on Machine Learning, 282–293 (2006)

[6] Bruce Abramson. “Expected-outcome: A general model of static evaluation.” IEEE Transactions on Pattern Analysis and Machine Intelligence, 12(2):182–193, February 1990.

[7] Silver, David, Huang, Aja, Maddison, Chris J., Guez, Arthur, Sifre, Laurent, van den Driessche, George, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, Dieleman, Sander, Grewe, Dominik, Nham, John, Kalchbrenner, Nal, Sutskever, Ilya, Lillicrap, Timothy, Leach, Madeleine, Kavukcuoglu, Koray, Graepel, Thore, and Hassabis, Demis. “Mastering the game of go with deep neural networks and tree search.” Nature, 529(7587):484–489, Jan 2016. Article.

[8] H. Baier and M. H. M. Winands. “Monte-Carlo tree search and minimax hybrids,” In IEEE Conference on Computational Intelligence and Games (CIG), pages 129–136, 2013.

[9] R. Ramanujan, A. Sabharwal, and B. Selman, “On Adversarial Search Spaces and Sampling-Based Planning,” in Proc. 20th Int. Conf. Automat. Plan. Sched., Toronto, Canada, 2010, pp. 242–245.

[10] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. “Finite-time analysis of the multi armed bandit problem,” Machine Learning, 47(2/3):235–256, 2002.

[11] L. Kocsis, C. Szepesvari, and J. Willemson, “Improved Monte-Carlo Search,” Univ. Tartu, Estonia, Tech. Rep. 1, 2006.

[12] J.-Y. Audibert, S. Bubeck, and R. Munos, “Best arm identification in multi-armed bandits,” in Proceedings of the Annual Conference on Learning Theory (COLT), 2010.

[13] S. Bubeck, R. Munos, and G. Stoltz, “Pure exploration in multi-armed bandits problems,” in Proceedings of the International Conference on Algorithmic Learning Theory (ALT), 2009.

[14] Lanctot, M., Winands, M. H. M., Pepels, T. & Sturtevant, N. R. “Monte Carlo tree search with heuristic evaluations using implicit minimax minimaxs.” In IEEE Conference on Computational Intelligence and Games, 1–8 (2014).

[15] M. H. M. Winands, Y. Bjornsson, and J.-T. Saito, “Monte-Carlo Tree Search Solver,” in Proc. Comput. and Games, LNCS 5131, Beijing, China, 2008, pp. 25–36.

[16] D. P. Helmbold and A. Parker-Wood, “All-Moves-As-First Heuristics in Monte-Carlo Go,” in Proc. Int. Conf. Artif. Intell., Las Vegas, Nevada, 2009, pp. 605–610.

[17] H. Baier and P. D. Drake, “The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go,” IEEE Trans. Comp. Intell. AI Games, vol. 2, no. 4, pp. 303–309, 2010.