國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University

Master Thesis

有效使用記憶體的小盤面圍棋求解演算法及實作

Memory efficient algorithms and implementations for solving small-board-sized Go

林弘承
Hung-Cheng Lin

指導教授：薛智文教授
Advisor: Chih-Wen Hsueh, Ph.D.
共同指導教授：徐讚昇教授
Co-Advisor: Tsan-sheng Hsu, Ph.D.

中華民國 107 年 8 月
August, 2018

doi:10.6342/NTU201802448

doi:10.6342/NTU201802448

Acknowledgements

The author thanks his advisor, Prof. Tsan-sheng Hsu, and his co-advisor, Prof. Chih-Wen Hsueh, for their continuous support for his graduate study and research and for their patience and the motivation and knowledge that they have given. The thesis would not have been completed without their help.

The author would also like to thank his colleague, Cheng Yueh, for his insightful thoughts during discussions.

The author would also like to thank the members of his family for their support throughout his life.

doi:10.6342/NTU201802448

doi:10.6342/NTU201802448

摘要

之前的研究已有弱解的小盤面圍棋解法，在此基礎上，我們希望能找到小盤面圍棋的強解，做爲更深入的分析。我們應用圍棋的特性來壓縮狀態的儲存空間，並用回溯分析以找出所有可能狀態的最佳解，也設計出一個儲存於硬碟的資料庫以供後續存取。爲了儲存及更新大量的狀態，使用壓縮後並儲存分割的資料區塊於記憶體而非硬碟，在需要使用的時候再解壓，可達到速度和記憶體用量的平衡。此方法亦可應用於巨量資料的處理。此外，我們由結果觀察並證明正確一路圍棋的最佳簡單著手規則。

關鍵字： 小盤面圍棋, 狀態空間搜尋, 回溯分析, 記憶體內運算

doi:10.6342/NTU201802448

doi:10.6342/NTU201802448

Abstract

Previously studies have weakly solved the problem of playing small-board-sized Go, but this study determines a strongly-solved solution and a database to access afterward.

State reduction is applied by the features of Go; and then retrograde analysis is used to find the optimal answer of every possible state of small-board-sized Go.

Dealing with large state information, an in-memory method is used to search the states for small-board-sized Go. Saving separated compressed data in the memory, instead of on a disk, and decompressing this data on demand, to balance performance and memory usage, in order to solve the problem efficiently. This method can also be applied to large scale data processing. A method is also determined that obtains the optimal result for boards with a single row.

Keywords: Small-board-Go, State Space Search, Retrograde Analysis, In-Memory Computing

doi:10.6342/NTU201802448

doi:10.6342/NTU201802448

Contents

doi:10.6342/NTU201802448

doi:10.6342/NTU201802448

doi:10.6342/NTU201802448

doi:10.6342/NTU201802448

List of Figures

doi:10.6342/NTU201802448

doi:10.6342/NTU201802448

List of Tables

doi:10.6342/NTU201802448

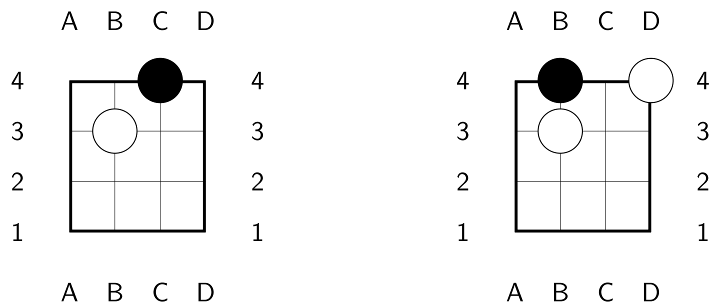
doi:10.6342/NTU201802448

Chapter 1

Motivation

This study determines all possible states for a small rectangular Go board. It is treated as a sub problem of a 9×9 Go or 19×19 Go board, so the search method for small boards may be applied to larger boards. In addition, the fair Komi and opening (see Figure1.1) are determined by solving the small-board-sized Go problem, the optimal result of fair opening is the draw. It is a better way to study the strategy and properties for small-board-sized Go, and may extend to bigger board. By solving small-board-sized Go, we can find the relation between boundary and intersections, which is the important feature of Go.

In order to solve all possible states for small-board-sized Go, a great number of state information is needed to be accessed. A memory-efficient method with acceptable performance is required to solve this task.



(a) Black's turn (b) Black's turn

Figure 1.1: Fair opening examples of 4×4 Go board

doi:10.6342/NTU201802448

doi:10.6342/NTU201802448

Chapter 2

Related Work

2.1 Previous Works

Small boards are used for beginners to learn Go. Because 19×19 Go requires a long search time, many studies address a smaller Go board and extend the result to larger boards.

Using an Alpha-Beta search with optimization, the problem can be weakly solved for a small rectangular board with less than 30 intersections. For example, the optimal game result and the optimal first move are found in 5×5, the optimal first move is the center of the board (C3 as Go board coordinate) and the optimal game result is Black fully win (25 points) [1, 2].

Using Meta-MonteCarlo-Tree-Search to build a huge opening book for 7×7 Go, and defeat professional Go players [3].

The variation in a game of Go is a challenge, such as atari-Go and kill-all Go. In atari-Go, the winner is the player that first captures the stone(s), and playing pass is prohibited. The 5×5 atari-Go game result is determined as Black's win [4].

Other studies use a proof-number search to determine the result for specific 7×7 kill-all Go opening positions [5]. In kill-all Go, Black plays two stones first, and White wins if there's a white live string, Black wins if there's no legal move for both players.

The number of legal states for square Go boards are calculated for size up to 17×17 and give the boundary of the legal state count for 19×19 Go [6], the result is shown in

doi:10.6342/NTU201802448

Table 2.1.

Board Size	Digit	Legal State Number
1	1	1
2	2	57
3	5	12675
4	8	24318165
5	12	414295148741
6	17	62567386502084877
7	23	83677847847984287628595
8	30	990966953618170260281935463385
9	39	103919148791293834318983090438798793469
10	47	96498428501909654589630887978835098088148177857
11	57	793474866816582266820936671790189132321673383112185151899
12	68	57774258489513238998237970307483999327287210756991189655942651331169
13	80	37249792307686396442294904767024517674249157948208717533254799550970595875237705
14	93	212667732900366224249789357650440598098805861083269127196623872213228196352455447575029701325
15	107	10751464308361383118768413754866123809733788820327844402764601662870883601711298309339239868998337801509491
16	121	4813066963822755416429056022484299646486874100967249263944719599975607459850502222039591149331431805524655467453067042377
17	137	19079388919628199204605726181850465220151058338147922243967269231944059187214767997105992341735209230667288462179090073659712583262087437

Table 2.1: Legal State Count for square board Go

2.2 Retrograde Analysis

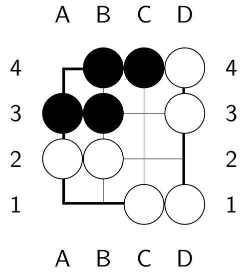
Retrograde analysis is widely used for searching endgames in chess-like game programming [7], such as Chinese chess [8] and shogi [9]. This method can be also applied to solve the full games when the state-space complexity of the game is small, like awari [10]. Previous study use this method to analyze patterns in Go [11].

[12] categorizes four major regretrograde analysis algorithms by their implementations. Our study use the fourth method, which is the refined method to implement the retrograde analysis algorithm.

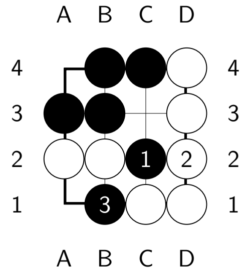
doi:10.6342/NTU201802448

2.3 4×4 Problems

These 4×4 problems and solutions are from [13], as shown at Figure 2.1 and Figure 2.2, which is for beginners to learn Go.

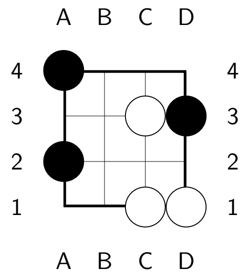


(a) Initial state, black's turn

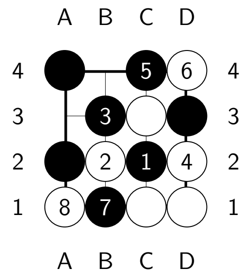


(b) Answer

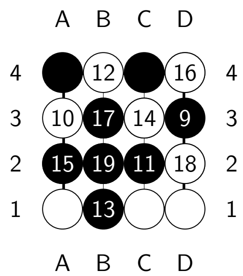
Figure 2.1: 4×4 problem 1



(a) Initial state, black's turn



(b) Answer move 1 to 8



(c) Answer move 9 to 19

Figure 2.2: 4×4 problem 2

doi:10.6342/NTU201802448

doi:10.6342/NTU201802448

Chapter 3

Problem Definition

3.1 The Rules of Go

Go is an ancient game that originated in China. Black and white stones are used to secure territory on the board and to surround the opponent. The basic rules are shown in Figure 3.1.

3.1.1 Basic Rules

Initially, the board is empty and there are vertical and horizontal lines. Each player places black or white stones at the intersections of these lines.

We look the connected set of stones in the same color as a string. The liberty of a string is the number of connected empty intersections. There are many different Go rules, mainly different between rules of Ko and scoring.

We look the connected set of stones in the same color as a string. The liberty of a string is the number of connected empty intersections.

There are many different Go rules, mainly different between rules of Ko and scoring rules.

doi:10.6342/NTU201802448

1. The black player plays first.
2. Black and white players place stones with the corresponding color in order.
3. If the opponent's string is out of liberty, it is captured and moved off the board.
4. A player cannot play a Ko move.
5. A player cannot play a suicidal move, which is the move that makes his or her string liberty become 0, unless this move involves the capture of an opponent's string.
6. A player can pass a move. If two players pass continuously, the game is ended and the score is calculated.

Figure 3.1: Basic rules of Go

3.1.2 Rules of Ko

Ko is the rule that prevents a loop in the game. The some commonly used Ko rules are [14] :

1. Basic Ko
2. Positional Superko
3. Situational Superko

Define the board position to be the positions of stones on the board; and the board configuration to be the board position and player's turn.

Basic Ko

Basic Ko only forbids a move that recreates the position from two moves before, but allows a longer cycle.

Positional Superko

Positional superko prevents the repeat of a board position.

Situational Superko

Situational superko does not allow a play that repeats a board's configuration.

The rules that are used in professional games are a variant of these three Ko rules.

doi:10.6342/NTU201802448

3.1.3 Scoring Rules

The two main scoring methods involve area scoring and territory scoring [15].

Area Scoring

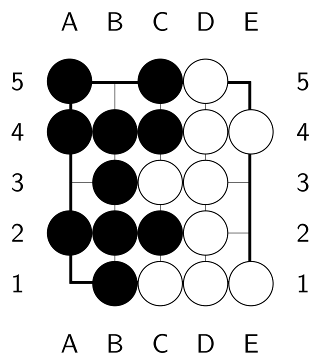
The player's score is the number of empty intersections only his color surround and the number of its stones on the board. Area scoring is used in Chinese rules. It's easier rule to implement in computer Go.

Territory Scoring

Dead stones and stones that are captured are looked as another color's player's prisoners.

The player's score is calculated in terms of his or her territory and the number of prisoners. Territory is the empty intersections that are controlled by one color. Its detailed definition is different for each set of rules. Territory scoring is used in Japanese rule and Korean rule.

In general, area scoring and territory scoring give the same result or one or two points difference. For example, in Figure 3.2, if there's no stone captured in the game, the result of territory scoring is draw (Black 3 points, White 3 points), and the result of area scoring is Black win one point (Black 13 points, White 12 points).



End of the game

Figure 3.2: Different results when applying different scoring rules

doi:10.6342/NTU201802448

3.2 Seki

Seki is a special case for the scoring rule for Go. It literally means mutual life. Two player's strings share liberties, if one player fills the liberty, his or her string is captured by another player. However, these strings are alive if they do not occupy the shared liberties. An example is shown at Figure 3.3. Some rules do not count the score for strings which are Seki [16].

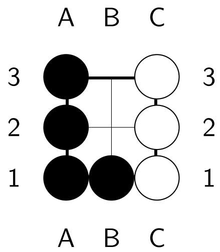


Figure 3.3: An example of Seki

3.3 The Rules that are used for this study

This study uses basic Ko and area scoring without komi. Because komi is not used, the white player does not get compensation when he or she scores.

The basic Ko rule states that when the game falls into a loop, the result of the game is considered to be draw.

It is worthy of note that the Benson algorithm [17] is not used to check dead strings because it only affects the result for unfinished boards, so both players pass when one of the players makes a legal move that does not fill his or her eye. Dead stones are considered to be alive if the opponent does not capture them before game ends.

In summary, the rules that this study uses allow superko and loops whose lengths are longer than 2, stones that seem dead are considered to be alive and the stones in Seki are also included in the score.

doi:10.6342/NTU201802448

3.4 Goal

This study solves the game result (which player wins the game) and the score (the difference in points between the winner and the loser) for all possible states on a small-board-sized Go. These are strongly solved for small-board-sized Go using the criteria for solving two-player zero-sum games with perfect information [18].

doi:10.6342/NTU201802448

doi:10.6342/NTU201802448

Chapter 4

Method

4.1 Encoding of States

Go's state includes the information about a stone's position, the Ko position, the pass count and the turn. The degree, the game result and the score are required for the search process.

Let the number of rows on the board be R and the number of columns on the board be C. Ternary representation is sued to map intersections on the board. If an intersection is empty, it is assigned a value of 0 and a black stone and a white stone are respectively assigned values of 1 and 2, so the range of board position values is from 0 to $3^{R\times C}$.

The Ko position can be represented using a simple position index. Including the situation where there is no Ko, there are $R \times C + 1$ different Ko positions. The pass count includes $0, 1$ and 2. The turn indicates the player that can play a move in the current state. In this implementation, it is reduced (See Section 4.2).

Ko and pass can be combined to reduce the number of bits that are used. When the pass is 1 or 2, there is definitely no Ko, so Ko and the pass have $R \times C + 3$ different combinations.

The degree (the number of legal next states), the game result and the score are saved and all information is encoded into a bit array, to reduce the amount of memory that is used.

The 5×5 board that shown at Figure 4.1(a) can be encoded to Table 4.2.

doi:10.6342/NTU201802448

Feature	Maximum Different Values	Bit Used	Description
Board Position	$3^{R\times C}$	40	
Ko and Pass	$R \times C + 3$	5	Pass is 0 with all intersections, pass is 1 with Ko is 0, pass is 2 with Ko is 0
Turn	2	0	Reduced
Degree	$R \times C + 1$	5	All possible move in the board and pass
Game Result	4	2	(black)win, draw, lose, and undetermined
Score	$2R \times C + 2$	6	$-R \times C$ to $R \times C$, and undetermined

Table 4.1: List of features and used bits in 5×5 board state

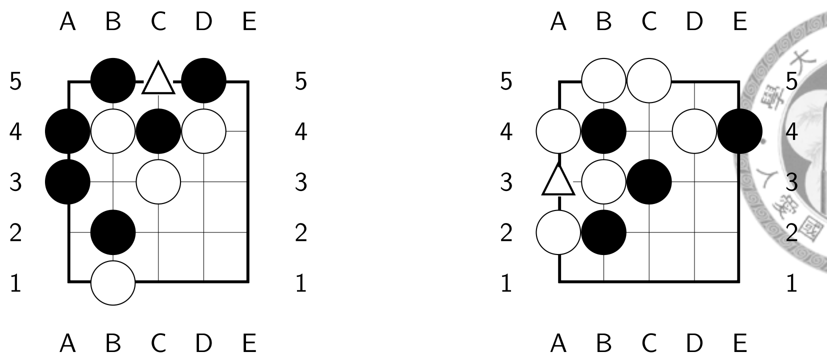
Feature	Value	Binary Representation	Description
Board Position	4091124856	100111000011001101 01101110111000010	$0002000010002010212101010_3$
Ko and Pass	3	00011	Ko at position 3, pass is 0
Degree	13	01101	Empty intersections without position 1 and Ko
Game Result	3	11	Undetermined
Score	51	110011	Undetermined, $2 \times R \times C + 1 = 51$

Table 4.2: Encoding of state in Figure 4.1(a)

Feature	Value	Binary Representation	Description
Board Position	72664314	10001010100110 0010011111010	$0000000012001201201200220_3$
Ko and Pass	11	01011	Ko at position 11, pass is 0
Degree	13	01101	Doesn't change after reduction
Game Result	3	11	Doesn't change after reduction
Score	51	110011	Doesn't change after reduction

Table 4.3: Encoding of state in Figure 4.1(b)

doi:10.6342/NTU201802448



White's turn
(a) A 5×5 state without reduction

Black's turn
(b) A 5×5 state with reduction

Figure 4.1: A 5×5 state reduction example, the triangle symbol is Ko move

A 5×5 board uses 58 bits per state. Because the memory must be aligned, 64 bits are used for each state.

4.2 Legal States and Reduction

Only legal states are searched, which are the states that follow the rules. Illegal states, such as a string with no liberty or an impossible Ko position, are discarded in the preprocessing phase.

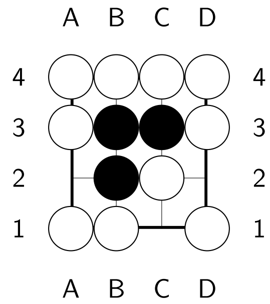
4.2.1 Terminal States

Terminal states are the set of states that can directly determine the result of the game without processing, such as a state whereby one player cannot make any legal move and the other player is currently winning (see Figure 4.2). A move is legal when the position of the move has an empty intersection that can be occupied by the player's stone without breaking the rules.

Terminal states are not saved in the memory because the result can be calculated when it is needed.

Obviously, all states for which pass is 2 are terminal states.

doi:10.6342/NTU201802448



Black Lose

Figure 4.2: A terminal state example. Black has no legal move, so white can simply pass to win the game.

4.2.2 State Reduction

States with symmetrical board configuration have the same result and score. Thus, these states can be reduced into one state.

The state that has the lowest value for symmetrical board positions is specified as the reduced state. There are at most 8 different square board states that are considered to be the same state. For a rectangular board, there are at most 4 different states that can be reduced into one.

The Ko position can also be reduced by specifying the lowest index for Ko positions in all symmetrical Ko positions.

Turn is reduced by changing the turn and the color of the stones on the board together. The states are reduced to black's turn, so it is not necessary to include turn information in the reduced state.

Figure 4.1 is an example of state reduction.

We keep all the states reduced during the search phase. In this way, the number of states that as shown in Table 4.4 must be searched is also reduced.

4.2.3 Sort Order

Serial Order

The states are sorted in terms of features in the order: board position > Ko position > pass count. An example is shown in Figure 4.3.

doi:10.6342/NTU201802448

Property	Value
Possible States	$3^{20} \times 21 \times 3 = 219667417263$
Legal States	1840058693
Legal States Ratio in Possible States	0.837%
Ko States	22418691
Ko States Ratio in Legal States	1.22%
Reduced States	460114319
Reduced States Ratio in Legal States	25.01%

Table 4.4: State statistics of 4×5 board

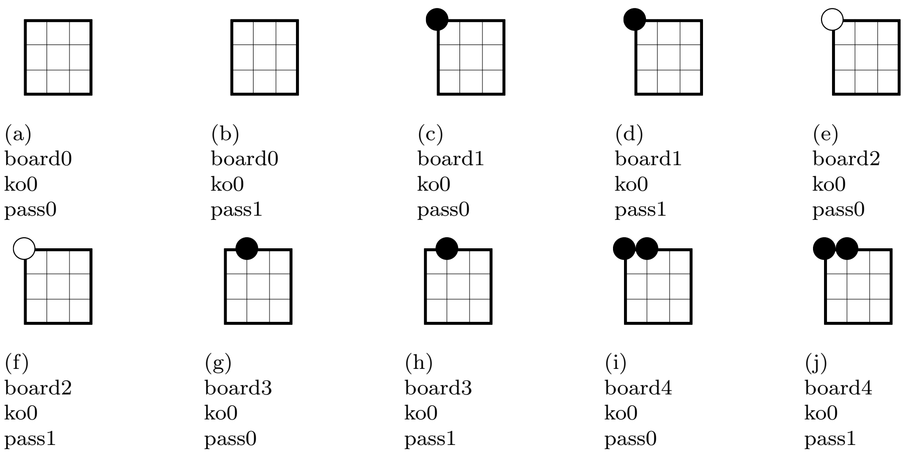


Figure 4.3: First 10 states of 4×4 board in serial order

Piece Order

Another sorting method is used to obtain a better locality in performing sorting.

Sorting using the order: total number of stones on the board > number of black stones on the board > board position > Ko position > pass count. An example is shown in Figure 4.4.

The former sorting method is called "serial order" and the latter sorting method is called "piece order".

Let S_{total} and S_{black} be the total number of stones and the total number of black stones for the current states. The total number of stones and the total number of black stones for previous states are S'_{total} and S'_{black}. If there is no previous capture, $S'_{total} = S_{total} - 1$ and $S'_{black} = S_{black} - 1$. Thus, previous states are concentrated into a smaller range in state array. By using piece order, the frequency of a memory block being compressed and decompressed is then decreased. Details will be described in Section 4.4.

doi:10.6342/NTU201802448

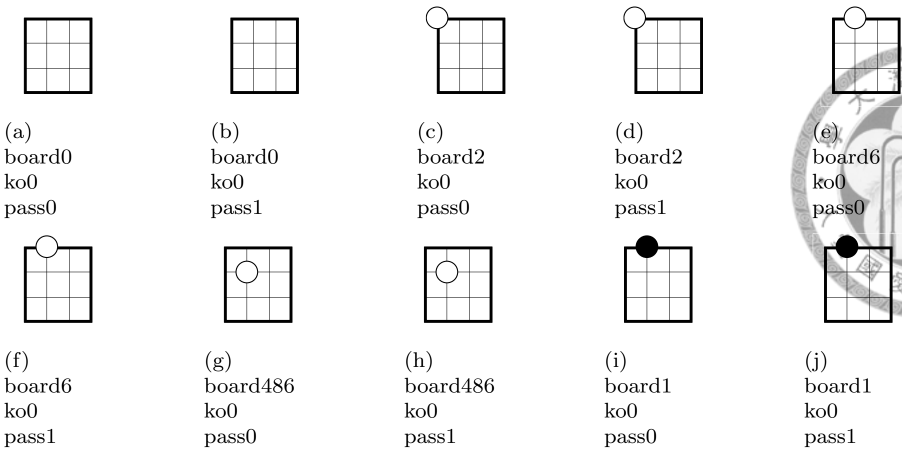


Figure 4.4: First 10 states of 4×4 board in piece order

4.3 Search Algorithm

Algorithm 1 states the main processes for the algorithm to solve small-board-sized Go.

Algorithm 1: Main Processes of Search Algorithm

Function $\mathrm{Main}()$**:**
    Preprocessing()
    // Section 4.3.1
    SearchGameResult()
    // Section 4.3.2
    SearchScore()
    // Section 4.3.3
    SaveSearchResult()
    // Section 4.3.4
    Validation()
    // Section 4.3.5

Preprocessing is required before the search and the process is described in Section 4.3.1. The search game result involves searching the game results for all legal states, as described in Section 4.3.2. The search score involves searching the score for all legal states, as described in Section 4.3.3. The search result is saved by saving the game result and the score into the database, as described in Section 4.3.4. Validation is an optional process that verifies that the result is correct and is described in Section 4.3.5.

doi:10.6342/NTU201802448

4.3.1 Preprocessing

The legal states are calculated, sorted and saved in the preprocessed files. The data can be directly dumped into the memory as an array.

4.3.2 Algorithm to Search the Game Result

Concept

The main concepts of the search result algorithm are:

1. Let S be the state with undetermined result
2. Let S' be the state that supersedes S
3. If there exists a state for which the result is lose in S', the result for S is a win
4. If for all states in S', the result is a win, the result for S is a loss
5. Repeat steps 1 to 4 until there is no undetermined state that can be updated as win or lose
6. The remaining undetermined states are considered to be draws

If there is an undetermined state S, such that the result for its next states has at least one undetermined state S' and has no loss state, then S' is the optimal next state. Because S' is also an undetermined state, there is also an undetermined state S'' in the next states of S', so the game does not end and the result is draw.

Retrograde Analysis Algorithm

Let W_i be the states that wins in i moves, L_i be the states that loses in i moves.

The fourth method in [12], which is called "Layered Backward Propagation with Unknown-children counting", is used to categorize the states in terms of the game result and the depth of the search, and the result is propagated to the previous states, as $W_i \rightarrow L_{i+1}$, and $L_i \rightarrow W_{i+1}$.

In this algorithm, every possible state propagates its value only once. When the result is a loss, it is necessary to check that all of its next states are wins, so information about undetermined next states is retained to determine whether the state is a loss.

doi:10.6342/NTU201802448

The detailed algorithm is shown in Algorithm 2.

Algorithm 2: Retrograde Analysis Search Result Algorithm

/* W_i is the set of states that will win in i turn */
/* L_i is the set of states that will lose in i turn */
$W_0 \leftarrow$ terminal states that result is win
$L_0 \leftarrow$ terminal states that result is lose
$i \leftarrow 0$
while $W_i \neq \phi$ *or* $L_i \neq \phi$ **do**
    $S \leftarrow W_i$
    $S' \leftarrow findPreviousStates(S)$
    sort S'
    foreach *State* $s \in S$ **do**
        if s.*result* $\neq$ *undetermined* **then**
            S.remove(s)
    remove duplicate states in S
    foreach *State* $s \in S$ **do**
        s.result $\leftarrow lose$
    $L_{i+1} \leftarrow S$

    $S \leftarrow L_i$
    $S' \leftarrow findPreviousStates(S)$
    sort S'
    foreach *State* $s \in S$ **do**
        if s.*result* $\neq$ *undetermined* **then**
            S.remove(s)
    count the number of duplicate states in S
    foreach *State* $s \in S$ **do**
        s.degree $\leftarrow$ s.degree - count of s in S
        if s.*degree* $= 0$ **then**
            s.result $\leftarrow win$
            add s into W_{i+1}
foreach *State* $s \in S_{arr}$ **do**
    if s.*result* $=$ *undetermined* **then**
        s.result $\leftarrow draw$

Determining previous States

$PreviousState$ is the class that stores the intermediate data structure. This is temporarily saved in a cache in order to determine actual states. One $PreviousState$ can represent multiple legal states that have the same board position, but a different Ko or

doi:10.6342/NTU201802448

pass.

Algorithm 3: Find Previous States

```
Function findPreviousStates(State s):
    if s has Ko then
        similar to the part of s.pass = 0, but check previous states can generate Ko
    else if s.pass = 0 then
        foreach white stone w ∈ s do
            find possible captured strings from 4 directions
            foreach combination c ∈ possible capture strings combination do
                previousBoard ← s.board + captured strings in c − w
                append previousBoard to possibleBoardSet
        return (PreviousState(possibleBoard, pass = 0 or 1)) for each possibleBoard ∈ possibleBoardSet
    else if s.pass = 1 then
        return PreviousState(s, pass = 0 or 1)
    else if s.pass = 2 then
        s.pass ← 1
        return s
```

4.3.3 Algorithm to Find the Score

Retrograde analysis is also used to search the score, but a different algorithm is used. The states for which pass is 2 and which win and lose the most are initially determined. Their score is then propagated to their previous states. This search method is repeated to search for the states in decreasing order of the absolute value of the score. Figure 4.5 shows the wrong result for a search in non-decreasing order of the absolute value of the score.

This method ensures for a state S, all of its next states S' are propagated to S no more than once. This implies that the state does not get a score that is better than its real score during the search phase. The details are shown in Algorithm 4 and Algorithm 5.

4.3.4 Data Structure used to save the Search Result

Because many differently sized boards are saved in the database, not all of the information is saved in the memory when there is a database query. The state results are saved

doi:10.6342/NTU201802448

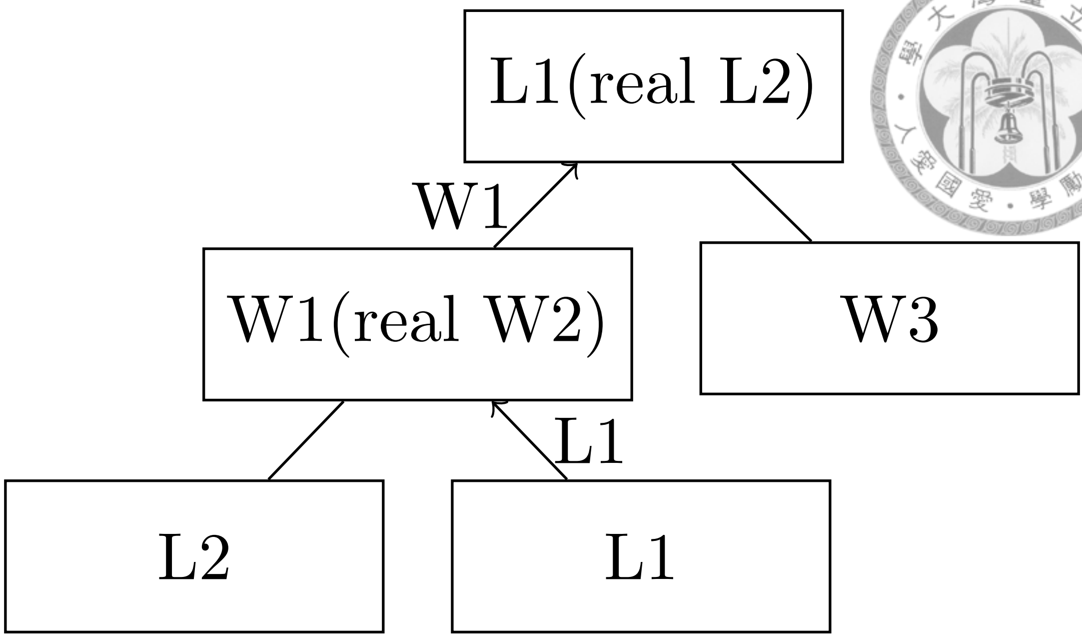


Figure 4.5: Using increasing order of score to update cause wrong result.

Algorithm 4: Search Score Algorithm

Input : State array which result is known
Output: State array which result and score is known

State queue q
/* from win $R \times C$ and lose $R \times C$ to win 1 and lose 1 */
for *score* $\leftarrow R \times C$ **to** *1* **do**
    /* find terminal states which score equals to current score */
    foreach *State* $s \in$ *compressedStates* **do**
        if s*.pass* $= 2$ *and* s*.score* $=$ *score* **then**
            q.push(s)
    while $q \neq \phi$ **do**
        state $s \leftarrow$ queue.top()
        q.pop()
        foreach *State* $ps \in$ *previous states of* s **do**
            if *updateScore(*ps*, s.result, s.score)* **then**
                // if update success, ps.score = score, so it should also propagate to its previous states
                q.push(ps)

---

doi:10.6342/NTU201802448

Algorithm 5: Update Score

Input : State and current score to update

Output: Update success or not

Function *updateScore(State s, int $result$, int $score$)*

  if s *is already updated* **then**

    return false

  if *result* $=$ *lose and s.result* $=$ *win* **then**

    // Because we search from high score to low score and state result is win, so it is the best result, the score of s is determined

    s.degree $\leftarrow 0$

  else

    s.degree $\leftarrow$ s.degree-1

  if *-score* $>$ *s.score or s.score* $=$ *undetermined* **then**

    // update to currently best score

    setScore(-score)

  // degree is 0 means score is found

  return $s.degree = 0$

on disk to save time.

The scores for each state's next states are stored in a file. The result is accessed by specifying the index of the state and then reading the information directly from the file.

Each state has at most $(R \times C + 1)$ next states, including pass. Each score is saved using 5 bits (see Section 4.1). Let the number of legal states be N. The size of the result file is $5 \times N \times (R \times C + 1)$ bits.

The following is the process to determine the result for a state.

1. When there is a query, the state is transformed to the reduced state.
2. The index of the state in the state file is then obtained. The state file contains the same data as the legal state array.
3. The result is then accessed from the result file using the index.

For each query of a state, a binary search of the file and random access disk is conducted to determine the game result for the state and its next states.

The state file is separated into multiple files, which have the same number of states as the memory blocks, so less time is required for a binary search.

doi:10.6342/NTU201802448

4.3.5 Validation

This is an optional process to check whether the search result is reasonable, as shown in Algorithm 6.

Algorithm 6: Function for validating the result

```
Output: the final result is reasonable or not
Function validate():
    foreach legal state S do
        S' ← possible next states of state S
        flag ← false
        if S.result is draw then
            foreach state s ∈ S' do
                if s.result is win then
                    return false
                if s.result is draw then
                    flag ← true
        else if S.result is win then
            k ← S.score
            foreach state s ∈ S' do
                if s.result is win then
                    if s.score > k then
                        return false
                    if s.score is k then
                        flag ← true
        else if S.result is loss then
            foreach state s ∈ S' do
                if s.result is win or draw then
                    return false
                if s.result is loss then
                    if s.score < k then
                        return false
                    if s.score is k then
                        flag ← true
        if flag is false then
            return false
```


doi:10.6342/NTU201802448

4.3.6 Misc

The in-memory data is transferred to the disk after each iteration, so the process can be continued if an error occurs.

4.4 In-Memory Method

When the size of board increases, the number of legal states increases exponentially. Thus, the state data cannot all be stored in the memory. Unused states can be temporarily saved to disk, but the I/O operation is time-consuming, so an in-memory method is used for the proposed program.

This study uses zlib [19, 20] library to compress or decompress data in the memory. The state array is divided into several fixed-sized memory blocks and these blocks are compressed by zlib if they are not in use.

In general, the greater the number of uncompressed memory blocks the better. The number of uncompressed memory blocks depends on the size of the available memory. Larger memory blocks are used to allow more uncompressed states at a given time.

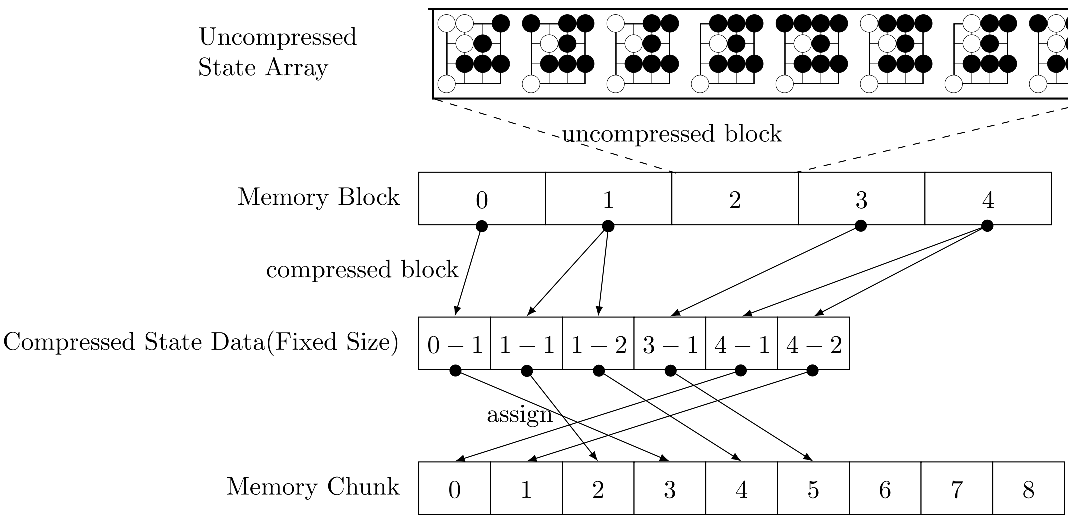


Figure 4.6: Memory usage of state array using memory block and memory chunk

Data is compressed or decompressed at the level of the memory block and data is saved or read at the level of the memory chunk. The relationship is shown in Figure 4.6.

doi:10.6342/NTU201802448

The algorithm is shown as Algorithm 7.

4.5 Memory Issues

As well as the legal state array, additional information is stored in the memory to accelerate the search process.

A state is represented by its index in the state array. Flags are saved in the memory as Boolean arrays to allow quick access. The information includes the result of whether state has already been searched, or current states in W_i or L_i. These Boolean arrays are the same size as the number of legal states. For a 5×5 Go board, each Boolean array uses 19.59 GB. Directly saving the states into the array requires a maximum of 187 GB, which is too large to save several these data in the memory.

The following are lists of Boolean arrays.

- $hasResult$: already searched states ($W_k, L_k, k < i$)
- win: currently searched win states (W_i)
- $lose$: currently searched lose states (L_i)
- $newwin$: win states to be searched in the next iteration (W_{i+1})
- $newlose$: lose states to search in next iteration (L_{i+1})

A temporary state array is used as a cache to save states and will be transformed into array indices on batch when the cache is full. The size of cache is flexible, but the larger the cache, the better the performance. In general, the cache must be able to contain at least one uncompressed memory block.

4.6 Performance Issues

State data can be used to find previous states that have Ko, to obtain or to set a result, a score or a degree of a state.

There is a bottleneck because a binary search of the uncompressed memory block is required to obtain the index of a state in the array. If the state is in the compressed memory block, it must be decompressed before it can be used.

doi:10.6342/NTU201802448

When the cache is full, a binary search is used to determine the indices, or a segmental scan over the array is needed to determine the index. If the number of states in a memory block is more than $\log_2(\text{state per block})$, this method is better than the binary search method in terms of time complexity.

doi:10.6342/NTU201802448

Algorithm 7: Memory Related Functions

```
In-memory block index queue q
/* fixed size memory array to read/write, pre-allocated */
MemoryChunk array chunks
MemoryBlock array blocks
State array blockFirstState
/* record first state of each memory blocks */
Function decompressBlock(blockindex)
    if blockindex ∈ q then
        return
    if number of uncompressed block is up to limit then
        block lastUsedBlock ← blocks[q.front()]
        q.pop()
        lastUsedBlock.compress()
    blocks[blockindex].decompress()
    q.push(blockindex)
Function getStateIndex(State s)
    blockIndex ← blockFirstState.binarySearch(s)
    if blocks[blockIndex] not uncompressed then
        decompressBlock(blockIndex)
    stateIndex ← blocks[blockIndex].binarySearch(s)
    return blockIndex × STATEPERBLOCK + stateIndex
Function getState(index)
    blockIndex ← index ÷ STATEPERBLOCK
    stateIndex ← index mod STATEPERBLOCK
    if blocks[blockIndex] not uncompressed then
        decompressBlock(blockIndex)
    return blocks[blockIndex].getState(stateIndex)
Function compress()
    /* compress a memory block */
    compressedArray ← zlib.compress(stateArray)
    stateArray.clear()
    distribute compressArray to multiple memory chunks
Function decompress()
    /* decompress a memory block */
    combine memory chunks to form compressedArray
    clear used memory chunks
    stateArray ← zlib.decompress(compressedArray)
    compressedArray.clear()
```


doi:10.6342/NTU201802448

Chapter 5

Result

5.1 Configuration

The following experiments were run on FreeBSD Clang version 3.4.1 using cpu Intel(r) Xeon(r) cpu E5-2699 v3 @ 2.30 ghz (2300.05-mhz K8-class cpu), 64 cores with an available memory of 512 GB. Experiments for smaller boards are run on Linux 4.14.15-1-arch using cpu Intel(r) Xeon(r) cpu E5-2620 0 @ 2.00ghz, 24cores with an available memory of 128 GB. The former is called Machine 1 and the latter is called Machine 2.

All experiments that were run on Machine 1 are specified and the other experiments were run on Machine 2. Table 5.4 shows the configurations of these two machines.

This study used g++ 8.1.0 as the compiler and zlib 1.2.11 to compress and decompress memory blocks.

Machine Number	Operating System	CPU	Core Numbers	Memory
1	FreeBSD Clang version 3.4.1	Intel(r) Xeon(r) cpu E5-2699 v3 @ 2.30ghz (2300.05-mhz K8-class cpu)	64	512 GB
2	Linux 4.14.15-1-arch	Intel(r) Xeon(r) cpu E5-2620 0 @ 2.00ghz	24	128 GB

Table 5.1: List of machines used for the experiment

doi:10.6342/NTU201802448

5.2 Experimental Results

Define Edge Count to be the sum of the number of the next states for all states. The results for rectangular Go boards are listed in Table 5.2, the biggest size board searched is 2×11.

The relation between search time and edge count is positive relative. Because we use less ratio of cache in 2×11, it cost far more time to search.

5.3 Performance Considerations

Experiments were conducted to determine the factors that affect search performance.

5.3.1 Memory Block Size and Memory Chunk Size

The memory block must be neither too small nor too large. If the memory block is too small, the frequency of compression and decompression is increased. If the memory block is too large, the time to compress and decompress is increased.

Because the size of the memory chunk only affects the segmentation of compressed blocks, it has no obvious effect on performance. The results are shown in Table 5.3 and Table 5.4.

In order to optimize the search process, the maximum number of uncompressed states not exceeding the memory limit is first computed and then the best memory block size is decided to minimize the running time.

If the memory block size is larger, the total number of states that must be decompressed during an iteration is greater, but there is a greater compression ratio. If the memory block size is smaller, the total number of states that must be decompressed during an iteration is smaller because it is not necessary to access some compressed memory block. However, we need to use more total memory.

doi:10.6342/NTU201802448

Size	Depth	Compressed State Number	Edge Count	Time	Best Result	Best First Move
1×1	–	–	–	–	$draw$	pass
1×2	–	–	–	–	$draw$	pass
1×3	–	–	–	–	$B + 03$	B1
1×4	–	–	–	–	$B + 04$	B1
$1 \times n$ $n \leq 20$ and $n \geq 5$	–	–	–	–	$draw$	–
2×2	2	26	72	0.164 second	$draw$	A1
2×3	11	293	963	0.167 second	$draw$	A1, B1
2×4	18	2169	8591	0.234 second	$B + 08$	A1, B1
2×5	30	18205	84906	0.791 second	$B + 10$	B1, C1
2×6	32	152887	821342	3.944 second	$B + 12$	B1, C1
2×7	35	1304472	7934582	1.1 minute	$B + 14$	C1, D1
2×8	41	11122653	75585864	10.61 minute	$B + 16$	D1
2×9	49	141646333	713466331	107.43 minute	$B + 18$	E1
2×10	54	1206719025	6673830049	18.0 hour	$B + 04$	E1
2×11	63	6941794698	61972960096	32.6 day	$B + 06$	F1
3×3	26	3696	15884	0.462 second	$B + 09$	B2
3×4	46	166358	884980	4.121 second	$B + 04$	B2
3×5	46	4200206	26752878	3.8 minute	$B + 15$	B2, C2
3×6	54	106590386	790892022	152.2 minute	$B + 18$	B3
3×7	59	2715285034	23000866733	1.93 day	$B + 21$	B3
4×4	56	9276006	41786050	5.9 minute	$B + 01$	B2
4×5	70	1402761648	7637285055	951.4 minute	$B + 20$	C2

Table 5.2: Result of small-board-sized Go, result $B + n$ means Black win n stones, best move is the coordinate label on the board

doi:10.6342/NTU201802448

Size	Block Size	Number of Block	Number of Uncompressed Block	Time (minutes)	Used Memory of Memory Blocks
2×9	256 MB	3	3	94.7	768 MB
2×9	256 MB	3	1	410.4	376 MB
3×6	1024 MB	1	1	99.1	1024 MB
3×6	256 MB	4	1	407.1	427 MB

Table 5.3: Performance and memory usage with different memory size used

Size	Block Size	State per Block	Number of Block	Number of Uncompressed Block	Time (minutes)
2×9	256 MB	32000000	3	1	410.4
2×9	64 MB	8000000	12	4	380.2
2×9	16 MB	2000000	48	16	380.9
3×6	256 MB	32000000	4	1	407.1
3×6	64 MB	8000000	16	4	389.6
3×6	16 MB	2000000	64	16	389.6

Table 5.4: Performance in different memory block size with the same memory used

5.3.2 Sort Order

We describe piece order in Section 4.2.3. The result is shown in Table 5.5.

Piece ordering achieves a better data locality when sorting is performed, but much more time is required for sorting than is required to determine the original sort order because of the additional time to calculate the number of stones, compared to serial order.

Size	Piece Order	Total Blocks	Avg. of iterate blocks	Std. of iterate blocks	Time (minutes)
4×5	No	30	26.02	6.25	951.4
4×5	Yes	30	19.24	9.79	3372.9
3×6	No	13	11.82	2.72	152.2
3×6	Yes	13	9.99	4.31	310.2

Table 5.5: Influence of sort locality and performance by applying piece order

doi:10.6342/NTU201802448

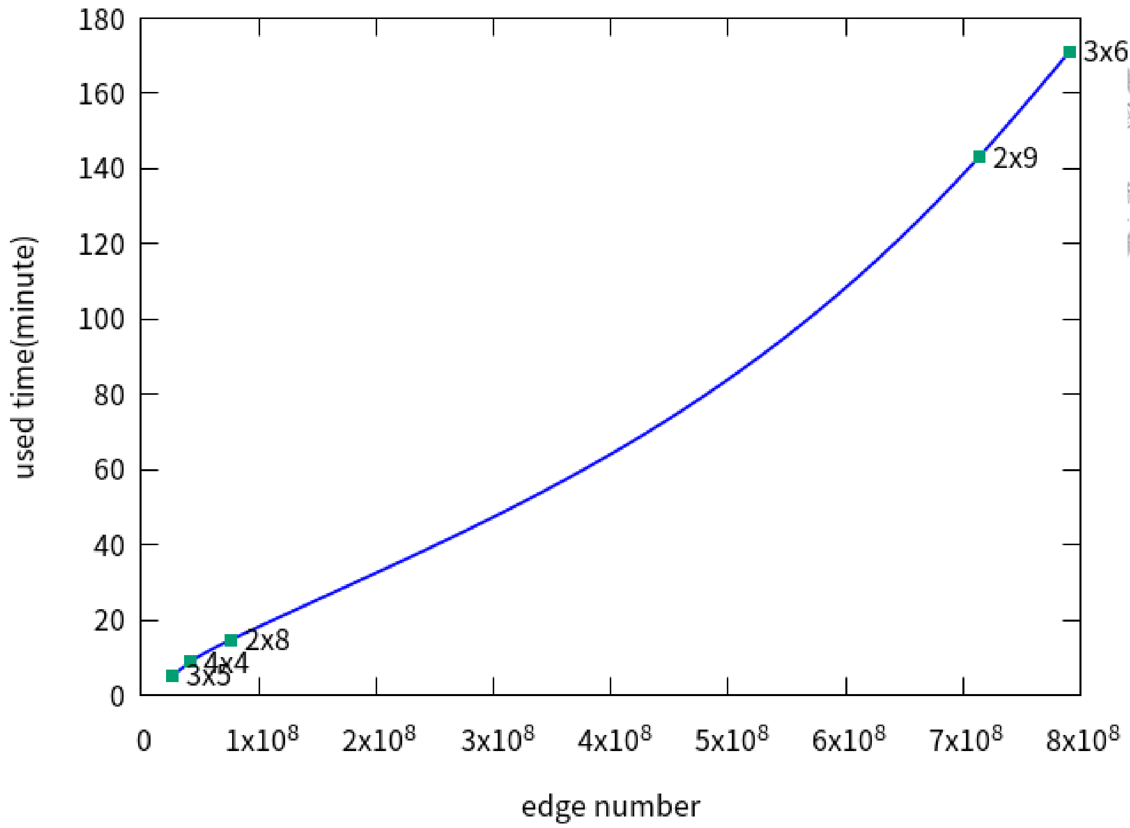


Figure 5.1: Correlation between time and edge count

5.3.3 Number of Edges

Figure 5.1 shows the correlation between time and edge count (total number of legal next states) for a Go board with between 15 and 18 intersections. We know each edge at most update one time by using Algorithm 2, there is a high positive correlation between edge count and time.

5.3.4 Search Depth

Figure 5.2 shows that most of the states can be determined in less than $R \times C$ moves.

There's some difference between Black-non-fully-win Go and Black-fully-win Go in Figure 5.3 and Figure 5.4. Black-non-fully-win Go will take more ratio of time at deeper search depth, and Black-fully-win Go will mostly solved at the front part of the searching depth.

doi:10.6342/NTU201802448

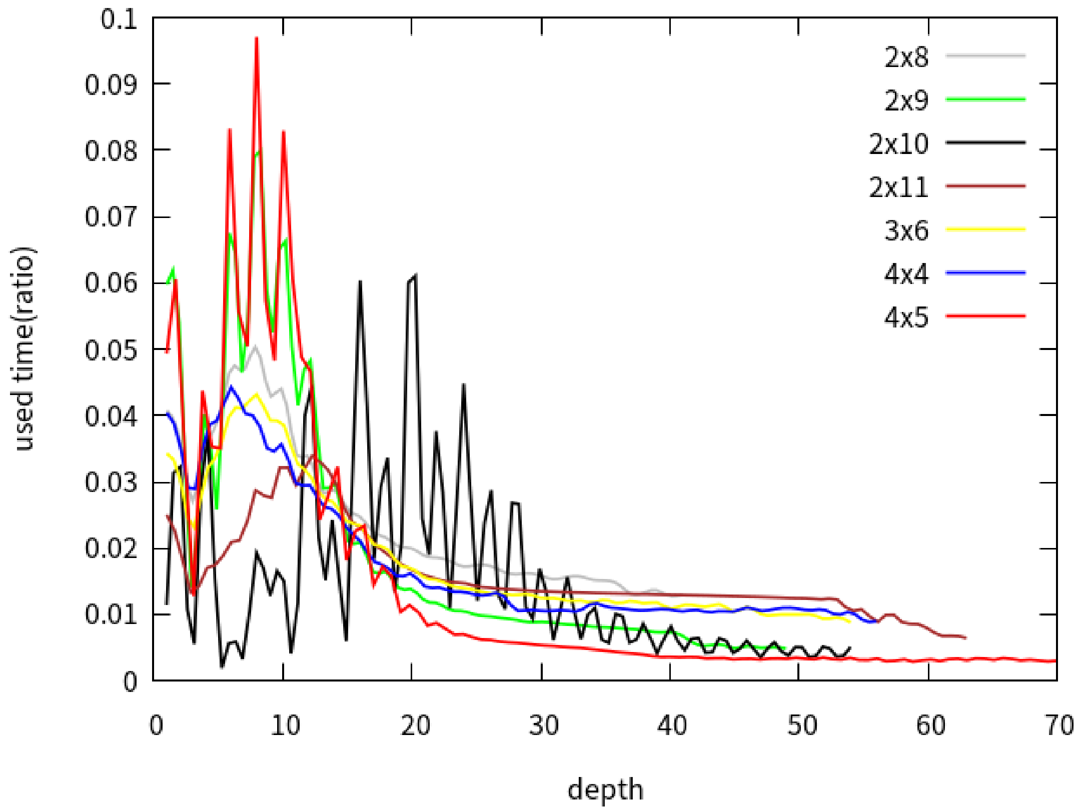


Figure 5.2: Time distribution of search depth in different size of Go boards

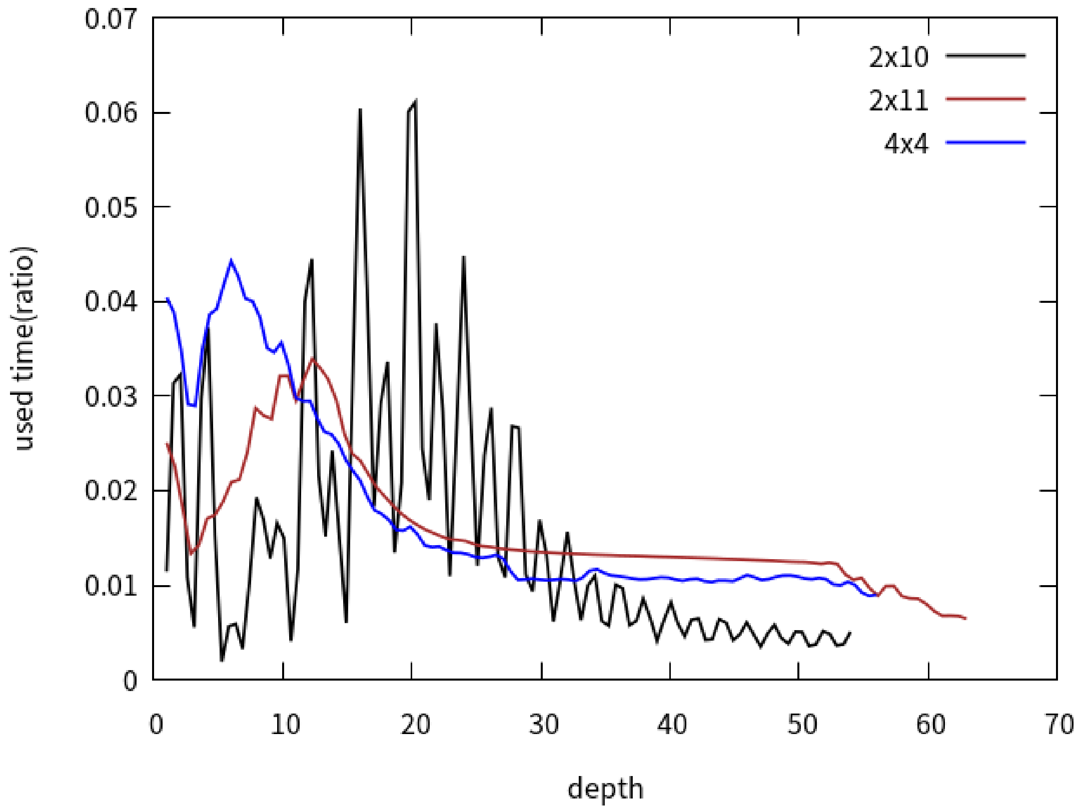


Figure 5.3: Time distribution of search depth in different size of Black-non-fully-win Go boards

doi:10.6342/NTU201802448

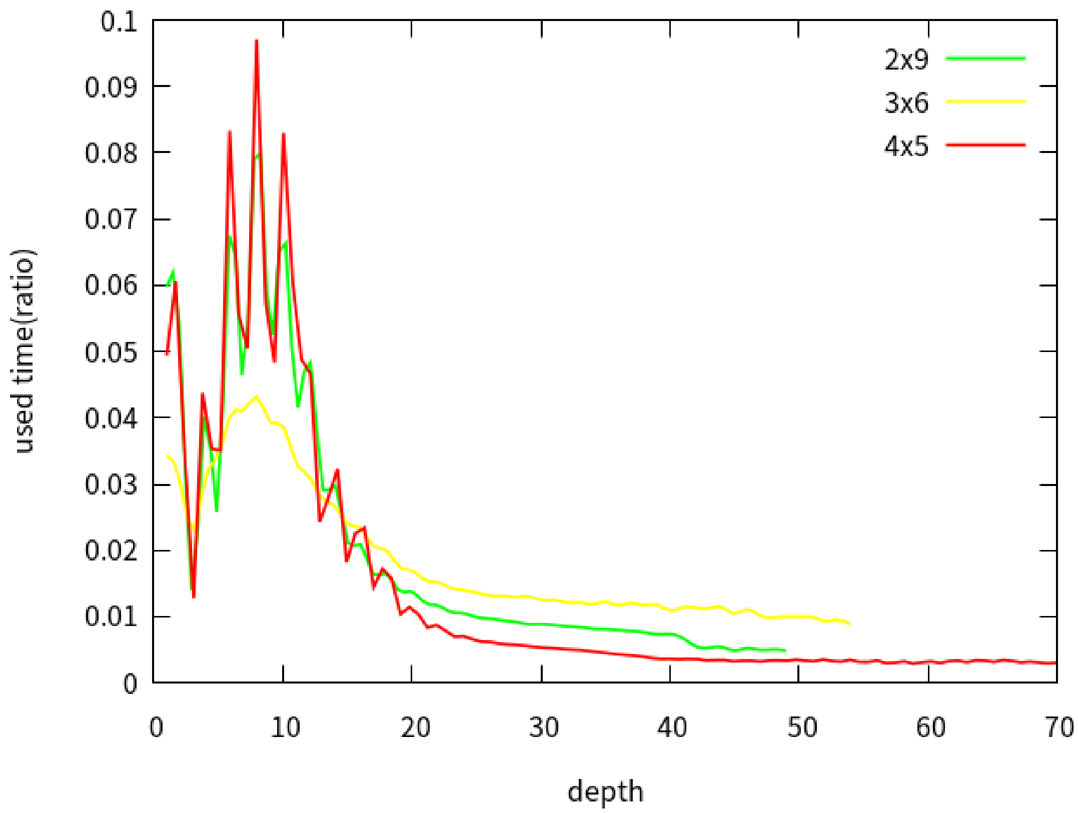


Figure 5.4: Time distribution of search depth in different size of Black-fully-win Go boards

doi:10.6342/NTU201802448

5.3.5 Data Saving Method

Figure 5.5, Table 5.6 and Table 5.7 show the performance for different zlib compression levels and for I/O and zlib for different sizes of block. This implementation has almost the same number of compressions and decompressions during the search because the number of uncompressed memory blocks is constant. Although the compression time for zlib is slow, the total time required is less than that for the I/O. For different implementations in compression levels 1..3 and levels 4..10 [21], we found zlib may run faster at level 4 than level 3. In general, we use level 1 compression for better performance.

Size	File Size	Method	Write(compress)time (seconds)	Read(decompress)time(seconds)	Compress level
5×5	32 GB	I/O	5920.6	111.4	none
5×5	32 GB	zlib	4379.1	1105.1	1
5×5	16 GB	I/O	2810.6	47.5	none
5×5	16 GB	zlib	2200.5	535.4	1
5×5	8 GB	I/O	1404.2	24.3	none
5×5	8 GB	zlib	1101.0	275.4	1
5×5	4 GB	I/O	680.7	12.5	none
5×5	4 GB	zlib	550.1	132.8	1
5×5	2 GB	I/O	318.4	6.5	none
5×5	2 GB	zlib	274.1	66.2	1

Table 5.6: Average I/O and zlib performance by testing 13 different memory blocks which is 5×5

Size	File Size	Method	Compressed Ratio $(\frac{originalsize}{compressedsize})$	Write(compress)time (seconds)	Read(decompress)time(seconds)	Compress level
5×5	32 GB	I/O	1	5920.6	111.4	none
5×5	32 GB	zlib	< 1	387.7	247.6	0 (no compression)
5×5	32 GB	zlib	4.541	4379.1	1105.1	1
5×5	32 GB	zlib	4.761	4735.9	1056.2	2
5×5	32 GB	zlib	4.921	7520.9	1020.3	3
5×5	32 GB	zlib	4.975	6589.2	1033.5	4
5×5	32 GB	zlib	4.975	11878.8	1034.5	5
5×5	32 GB	zlib	5.104	22732.9	1014.4	6
5×5	32 GB	zlib	5.136	33677.1	988.7	7

Table 5.7: Average zlib performance with different compression levels in 5×5 Go.

doi:10.6342/NTU201802448

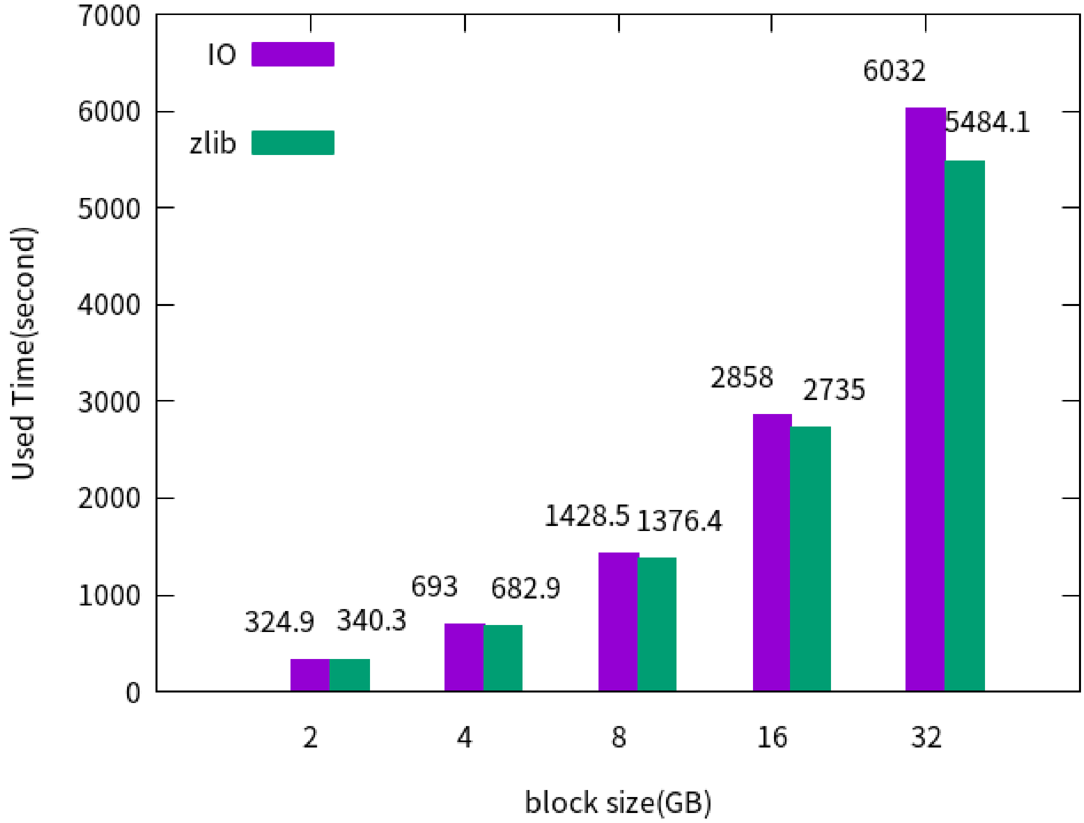


Figure 5.5: Comparison of zlib and I/O performance in different memory block size.

5.4 Variation: Circular Board

The rules for Go are also varied. We define the circular board have the rule that the first column of the board be the neighbor of the last column in the same row. For this rule, there are a maximum of $2 \times 2 \times C$ symmetry boards in a rectangular board. An example of a 2×3 board is shown in Figure 5.6.

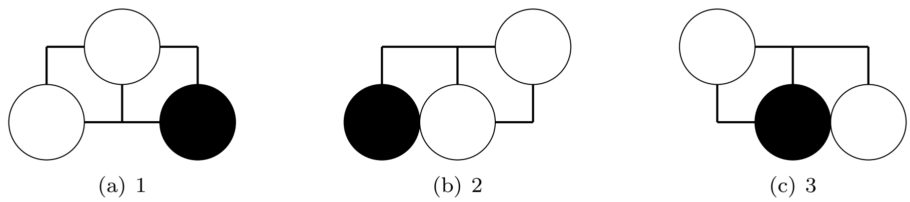


Figure 5.6: Three board positions can be compressed to one in circular board

There are special cases when the column size is 7, for $1 \times N$ and $2 \times N$ circular boards. The results are different to those for other boards with the same row size, as shown in Table 5.8.

doi:10.6342/NTU201802448

Size	Result
1×2 to 1×6	$draw$
1×7	$B + 01$
1×8 to 1×20	$draw$
2×2 to 2×5	$draw$
2×6	$B + 01$
2×7	$B + 14$
2×8	$B + 02$
2×9	$B + 04$
2×10	$B + 03$

Table 5.8: Result of small-board-sized circular Go board

5.5 Properties of Go

5.5.1 Seki

Seki is determined by searching the states for which

1. Current state has no pass.
2. Playing pass is the only best move (or it is defeated).
3. There is more than one legal move (as when liberties are shared).
4. The next states for which pass is 1 must satisfy the following rules:

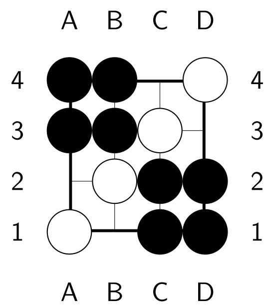
    (a) Pass is the only best move.
    (b) There is more than one legal move.

By these constraints, there are 1887 Sekis for a 4×4 Go board. Figure 5.7 shows several such examples.

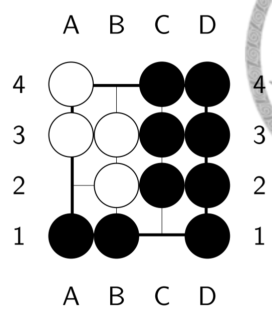
5.5.2 Strongly Connected Component

There are cycles in the game tree for Go that exist in basic Ko rules. In state graph, these cycles can be represented as strongly connected components (SCC) with more than one states. For example, Figure 5.8 shows a superko and Figure 5.9 shows a cycle example.

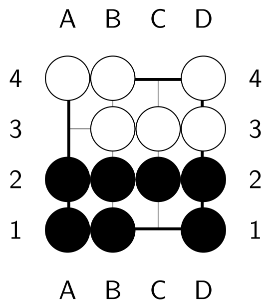
doi:10.6342/NTU201802448



(a) Example 1

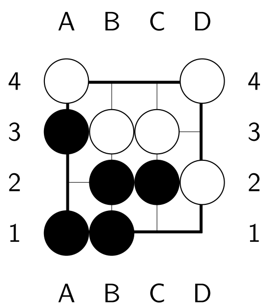


(b) Example 2



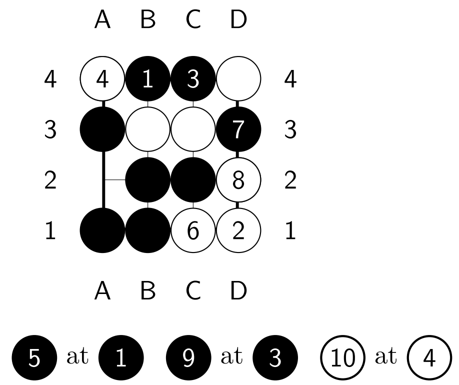
(c) Example 3

Figure 5.7: Seki example in 4×4 Go board



Black to play

(a) SCC: initial state



(b) Repeat the state after 10 moves, the state is still the same as Figure 5.8(a)

Figure 5.8: cycle example 1: superKo

doi:10.6342/NTU201802448

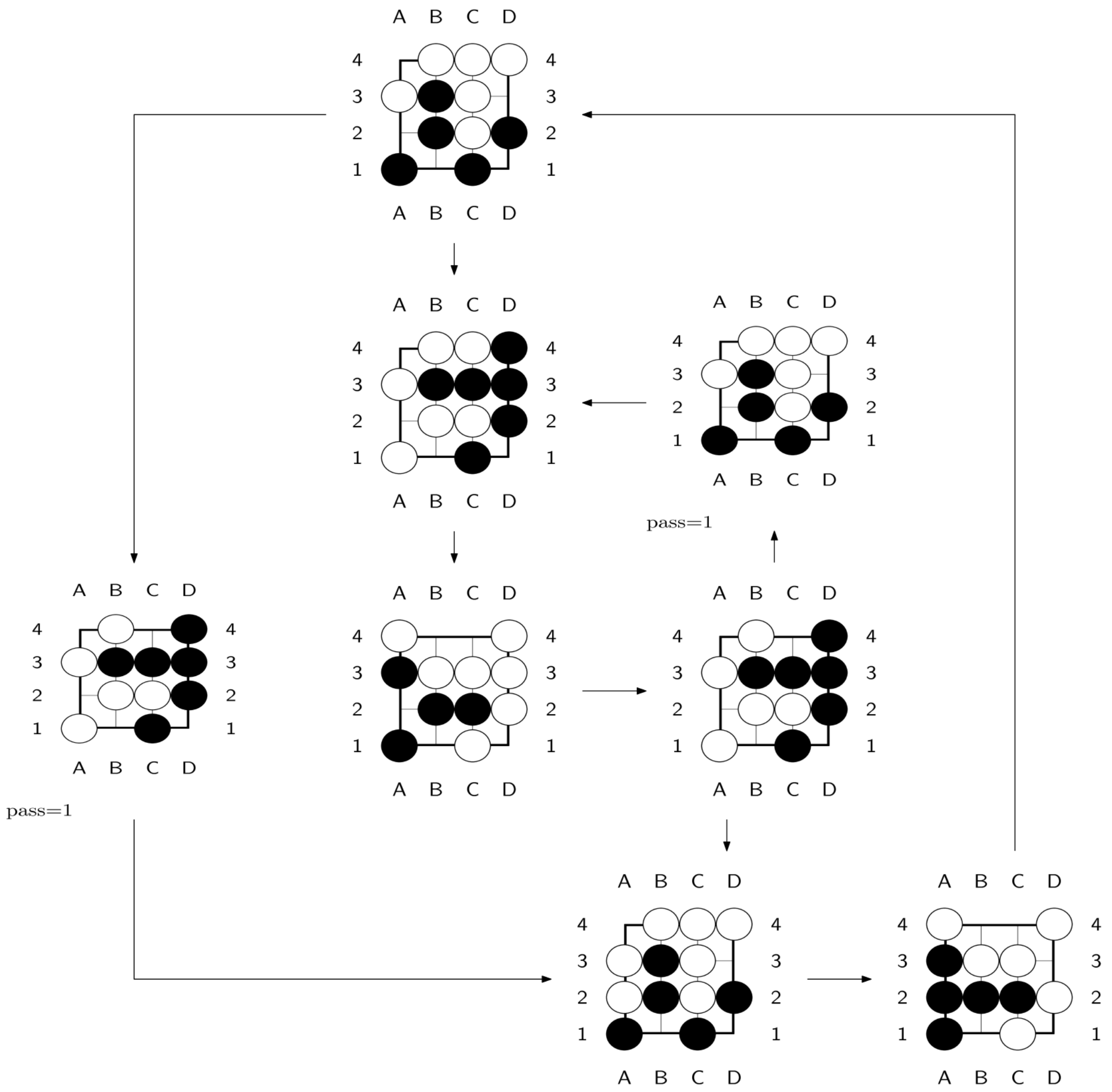
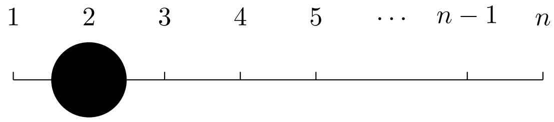
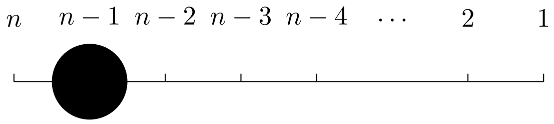


Figure 5.9: cycle example 2: states in SCC

doi:10.6342/NTU201802448



(a) Black plays at position 2



(b) Count position from another side, this state is the same as Figure 5.10(a)

Figure 5.10: Position can be count from another side.

5.6 Strategy for $1 \times n$ Go

A general move-generating rule for all $1 \times n$ boards is verified for 1×6 to 1×20 Go boards. Let the number of board intersections for $1 \times n$ be $\{1, 2, 3, \cdots, n-1, n\}$. Define $board[i]$ as the stone color at position i. $board[i] \in \{black, white, empty\}$. $board[i] = empty$ means that neither a black nor a white stone occupies position i. A special case is defined for which position 0 and position $n+1$ are a fixed boundary and $board[0] = board[n+1] = boundary$.

White's Move-Generating Rule. *Find the first black stone ($firstblack$) from position 4 to position n. If first black stone exists at position w, $firtblack = w$; otherwise, $firstblack = n + 1$. If $board[w-1] = white$ and $board[w-2] = empty$, the move at position $w - 2$ is played.*

Otherwise, play at the first empty intersection k from position 2 to position n. When it is White's first move, changing the order of position ensures that $board[2] \neq black$ and $board[3] \neq black$ (Figure 5.10) . If k does not exist, play pass.

For example, in Figure 5.11, White plays at position k, using step 1 which is described in Algorithm 8.

By applying White's Move-Generating Rule, which is described in Algorithm 8, the result for White is at least a draw (Theorem 4).

Define S_w be the set of states that are White's turn, and can be accessed by applying White's Move-Generating Rule.

doi:10.6342/NTU201802448

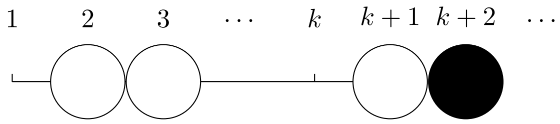


Figure 5.11: An example of White's Move-Generating Rule

Algorithm 8: White's Move Generating Rule

Input : A state S that is in S_w

Output: A legal move that can play at S which generates optimal result

Determine the position by the fact that $board[2] = white$, or $board[2] \neq black$ and $board[3] \neq black$

firstblack $\leftarrow$ minimum position that $board[firstblack] = black$

if *firstblack not exist* **then**

    firstblack $\leftarrow n + 1$

if $firstblack > 3$ *and* $board[firstblack - 1] = white$ *and* $board[firstblack - 2] = empty$ **then**

    // Step 1

    return *firstblack-2*

else

    firstempty $\leftarrow$ first empty intersection from position 2 to position n which is legal

    if *firstempty exist* **then**

        // Step 2: if there's no suitable move in step 1

        return *firstempty*

// Step 3: if there's no suitable move in step 1 and 2

return *pass move*

---

doi:10.6342/NTU201802448

Define $R(s)$ as the move that is generated by White's Move-Generating Rule when the state is s.

$R(s) : S \to N, S = S_w, N = [1, n]$.

By the definition in the basic Go rules (Section 3.1 and Section 3.3):

Suicidal move and capture move is defined in Figure 3.1.

Definition 1. *An empty intersection is a legal move if and only if it is neither a suicidal move nor a Ko move.*

Definition 2. *Let the initial state be S_0, the current state to be S_c, the previous state to be S_{c-1}, and so on. Let S_{c+1} be the state after a player plays at position k in state S_c. When $c \geq 1$, position k is a Ko move if and only if $S_{c-1} = S_{c+1}$.*

Definition 3. *A move k is not a suicidal move if it is a capture move. A move k is not a capture move if it is a suicidal move.*

Lemma 1. *If position k is a Ko move, playing move k can only result in the capture of one stone that the opponent has recently played.*

Proof. If move k captures multiple stones, then without loss of generality, if move k is played by White and it captures m black stones, $m \geq 2$. By Definition 2, $S_{c-1} = S_{c+1}$. Let the number of black stones in each state be B_{c-1}, B_c, and B_{c+1}, corresponding to S_{c-1}, S_c and S_{c+1}. Because $S_{c-1} = S_{c+1}$,

$$B_{c-1} = B_{c+1}. \tag{5.1}$$

In S_{c-1}, black plays one stone, so

$$B_{c-1} + 1 = B_c. \tag{5.2}$$

White captures m stones when playing move k in S_c,

$$B_c - m = B_{c+1}. \tag{5.3}$$

doi:10.6342/NTU201802448

By Equation 5.1, Equation 5.2, and Equation 5.3,

$$B_c = B_{c+1} + m = B_{c-1} + 1. \tag{5.4}$$

, so $m = 1$, results in a conflict. Therefore, a Ko move does not capture multiple stones.

If move k does not capture a stone that was recently played by the opponent, let move k capture position m, so the opponent recently played position p, $p \neq m$. By Definition 2, $S_{c-1} = S_{c+1}$. $board[p] = empty$ in S_{c-1} and S_{c+1}. In S_c, $board[p] \neq empty$, because move k only captured move m, $board[p] \neq empty$ in S_{c+1}, which results in a conflict. Therefore, move k captures a stone that has been recently played by the opponent. □

Define $Step(s)$ as the step that generate a move in Algorithm 8.

$Step(s) : S \to N$, $S = S_w$, $N = \{1, 2, 3\}$

We now prove properties of $Step(s)$ by case analysis.

5.6.1 $Step(s) = 1$

Theorem 1. *$\forall s \in S_w$, if $Step(s) = 1$, $R(s)$ is legal.*

Proof. Let $R(s) = k$, because $k = firstblack - 2$, there is no black stone near position k, so move k is not a capture move, by Definition 3, so it is not a Ko move.

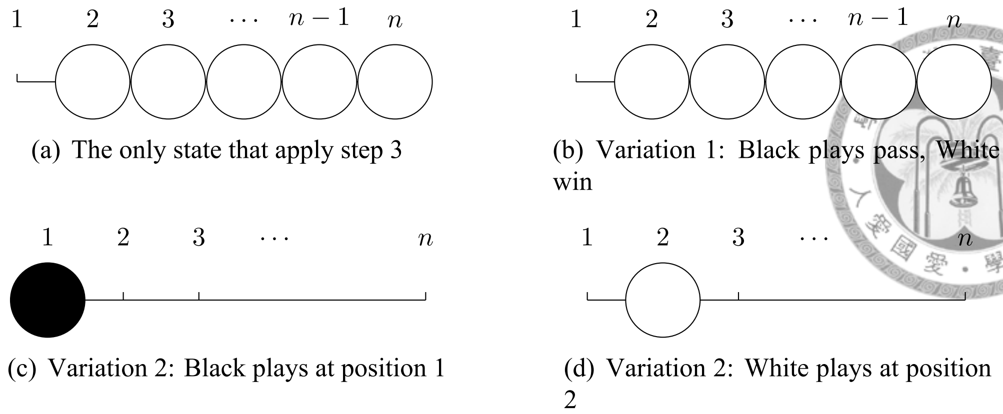
After playing move k, let move L be the minimum value for a position in the string that includes k. It is obvious that $board[L-1] \neq white$ by definition of L and $board[L-1] \neq black$ because $L - 1 < firstblack$ and because by White's Move-Generating Rule, White never plays at position 1, $board[1] \neq white$ and $L \neq 1$, so $board[L-1] \neq boundary$. Therefore, $board[L-1] = empty$, move k is not a suicidal move because there is a liberty in position $L - 1$.

By Definition 1, because move k is neither a Ko move nor a suicidal move, move k is legal. □

5.6.2 $Step(s) = 3$

Theorem 2. *If $Step(s) = 3$, White does not lose after playing pass.*

doi:10.6342/NTU201802448



(a) The only state that apply step 3

(b) Variation 1: Black plays pass, White win

(c) Variation 2: Black plays at position 1

(d) Variation 2: White plays at position 2

Figure 5.12: State that apply step 3 and it's variations

Proof. If $Step(s) = 3$, state s does not satisfy step 1 and step 2, so there is no empty intersection from position 2 to position n, otherwise, step 2 is applied.

The only possible state is $board[1] = empty$ and $board[i] = white$ for $i = 2..n$, in Figure 5.12. After White plays pass in this state, White can win when Black plays pass, or White can continue playing using White's Move-Generating Rule when Black plays position 1. □

5.6.3 $Step(s) = 2$

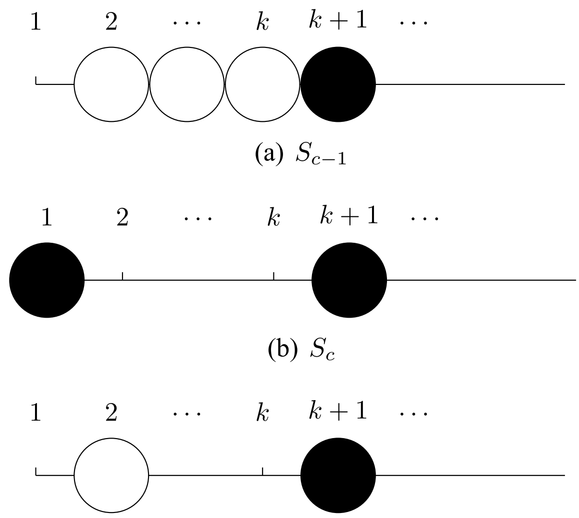
Lemma 2. *$\forall s \in S_w$, if $Step(s) = 2$, $board[1] = empty$ and $board[2] = white$ after White plays by White's Move-Generating Rule.*

Proof. After playing White's first move, the statement is true. It is true that $board[1] = empty$ if $board[2] = white$, because by White's Move-Generating Rule, White never plays at position 1, and $board[1] \neq black$ if $board[2] = white$ because position 1 has no liberty.

If black's move does not capture a white stone at position 2, this statement is always true, so consider a move by Black to capture the white string that includes position 2. This move can only be made position 1.

Figure 5.13 shows that White can capture back by playing position 2 using Step 2. If Black plays at position 1 only one stone is captured, as shown in Figure 5.14. $board[2] \neq$

doi:10.6342/NTU201802448



(a) S_{c-1}

(b) S_c

(c) S_{c+1}

Figure 5.13: Black plays at position 1 which capture multiple stones

$white$ in S_{c-2} because if $board[2] = white$, the black stone in position 3 is captured in S_{c-1}.

We have already been shown that if Black does not capture a white stone in position 2, $board[2] = white$, so a white stone in position 2 is captured to become S_{c-2}. The possible states for S_{c-2} are shown in Figure 5.14(a).

In Figure 5.14(a), $S_{c-2} = S_c$, so Black plays a Ko move at S_{c-1}, which results in a conflict.

There is no legal S_{c-2}, so the situation whereby the black move plays at position 1 and only captures one stone does not exist.

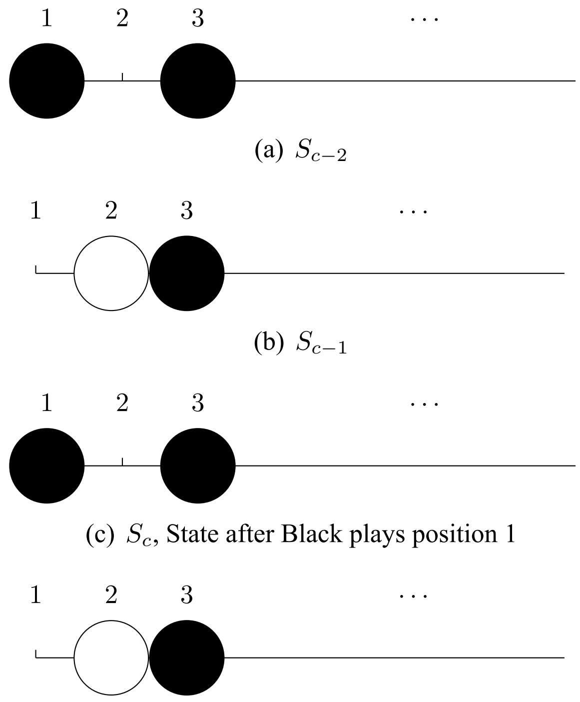
□

Lemma 3. *$\forall s \in S_w$, if $Step(s) = 2$, $R(s) = k$ and $k > 2$, $board[i] \neq empty$ for $i = [2, k-1]$; if $\exists\, i$ $board[i] = black$ such that $i = [2, k-1]$, there is at most one black string between positions 2 to $k-1$, which includes position $k-1$.*

Proof. If there exist m, such that $0 < m < k$ and $board[m] = empty$, White plays at m using step 2 of White's Move-Generating Rule, which results in a conflict, so position 2 to $k-1$ are all non-empty.

Because $k > 2$, by Lemma 2, if $board[2] = black$, $board[2] \neq white$ after White plays a move, which results in a conflict, so $board[2] \neq black$, and $board[i] \neq empty$ for

doi:10.6342/NTU201802448



(a) S_{c-2}

(b) S_{c-1}

(c) S_c, State after Black plays position 1

(d) S_{c+1}, State after White plays by White's Move-Generating Rule

Figure 5.14: Black plays at position 1 and capture only one stone

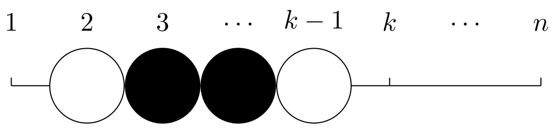


Figure 5.15: More than two strings from position 2 to $k-1$. In this example, black string start from position 3 has no liberty

$i = [2, k-1]$, so $board[2] = white$.

If there are more than two strings, position 2 to $k-1$ are all non-empty. The second string that is in the middle has no liberty, so the board position is not legal, which results in a conflict (Figure 5.15). There's already a white string that includes position 2, so there is at most one black string.

If there are two strings from position 2 to $k-1$, then the black string includes position $k-1$ because the white string includes position 2. □

Lemma 4. *$\forall s \in S_w$, if $Step(s) = 2$, $R(s)$ is not suicidal move.*

Proof. If move k is a capture move, by Definition 3, it's not a suicidal move.

If move k is not a capture move, by Lemma 3, there are two possible situations:

1. There are are only white stones from position 2 to $k-1$

doi:10.6342/NTU201802448

By Lemma 2, after playing move k, there is still a liberty at position 1, so move k is not a suicidal move.

2. Positions from m to $k-1$ are black, $m < k-1$

   The liberty of the black string from positions m to $k-1$ is 1, so move k is a capture move, which results in a conflict.

Therefore, move k is not a suicidal move. □

Define $NextState(s, k)$ as the state that is obtained from state s after making the move at k.

$NextState(s, k) : S \times K \to S$, $S = \{s \mid$ s is legal state$\}$, $K = [1, n]$

Lemma 5. $\forall s \in S_w$, *if* $Step(s) = 2$, $R(s)$ *is not a Ko move.*

Proof. Let the initial state be S_0, the previous state be S_{c-1} and the current state be S_c. Let S_{c+1} be the state whereby a player plays move k in state S_c.

Assume that move k is a Ko move. By Definition 2 and Lemma 1, only the situation whereby move k captures exactly one move is considered and $S_{c-1} = S_{c+1}$.

Let $nextState(S_{c-1}, m) = S_c$, $m = k-1$ or $m = k+1$.

If $m = k-1$, $board[k-1] = black$ and $board[k] = empty$ in S_c. $board[k-2] = white$, $board[k-1] = empty$ and $board[k] = white$ in S_{c+1}. Because $S_{c-1} = S_{c+1}$, these states are shown in Figure 5.16. Because $board[k] = empty$ in S_c, $board[k+1] = black$ in S_{c-1}, S_c, S_{c+1}.

Consider S_{c-2}. If $nextState(S_{c-2}, k) = S_{c-1}$, because $board[k-1] = empty$ in S_{c-1}, then S_{c-2} is shown in Figure 5.16(b) or 5.16(c).

However, in Figure 5.16(b), $S_{c-2} = S_c$, so Black plays a Ko move in S_{c-1} before White plays move k, which results in a conflict. In Figure 5.16(c), White plays at position $k-1$, which results in a conflict.

If $nextState(S_{c-2}, k) \neq S_{c-1}$, then $board[k] = white$ and $board[k-1] = empty$ in S_{c-2} as shown at Figure 5.16(a), so $Step(S_{c-2}) = 1$ and $R(S_{c-2}) = k-1$, but $board[k-1] = empty$ in S_{c-1}, which results in a conflict.

doi:10.6342/NTU201802448

If $k = n$, in Figure 5.16(d), Figure 5.16(e) and Figure 5.16(f), there is no legal S_{c-2} that generates S_{c-1} using White's Move-Generating Rule, which results in a conflict.

If $k = 2$, $board[2] = empty$ in S_c. By Lemma 2, if $board[2] \neq white$, the initial state must be S_0, or Black captures it in previous move, but $board[3] = black$ in S_c, so $S_{c-2} \neq S_0$. So Black capture white stone in position 2 at S_{c-1}, $S_{c-2} = S_c$, as shown in Figure 5.16(g). Therefore, Black plays a Ko move at S_{c-1}, which results in a conflict.

If $m = k + 1$, $board[k] = empty$ and $board[k+1] = black$ in S_c, and $board[k] = white$, $board[k+1] = empty$ in S_{c+1}. Because a black stone in position $k+1$ is captured, $board[k+2] = white$ in S_c, S_{c+1}. Because $S_{c-1} = S_{c+1}$, these states are shown in Figure 5.17.

By S_{c-1} and S_{c+1}, the position k is captured in S_c, so $board[k-1] \in \{empty, black\}$ in S_c. If $board[k-1] = empty$ in S_c, by White's Move-Generating Rule, when $Step(S_c) = 2$, $R(S_c) \neq k$, which results in a conflict. If $board[k-1] = black$ in S_c, because $Step(S_c) = 2$, the black string in position $k-1$ is captured by playing move k in S_c, so move k captures multiple stones in S_c, which results in a conflict.

If $k = n - 1$, then in Figure 5.17(d) and Figure 5.17(e), there is no legal S_{c-2} to generate S_{c-1} by White's Move-Generating Rule, which results in a conflict.

□

Theorem 3. *$\forall s \in S_w$, if $Step(s) = 2$, $R(s)$ is a legal move.*

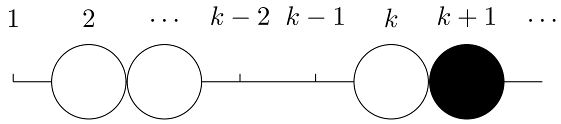
Proof. By Definition 1, Lemma 4 and Lemma 5, move k is a legal move. □

5.6.4 Conclusion

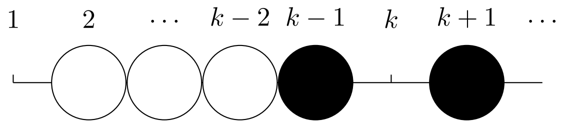
Theorem 4. *$\forall s \in S_w$, if White plays the move using White's Move-Generating Rule, White will lose.*

Proof. By Theorem 1, Theorem 2, and Theorem 3, except in step 3 – White may win in some situations. There always exists at least one legal move, so the game does not finish and White never loses. □

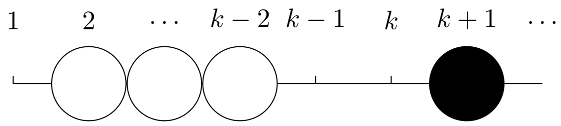
doi:10.6342/NTU201802448



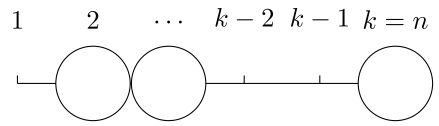
(a) one example of S_{c-2} when $nextState(S_{c-2}, k) \neq S_{c-1}$



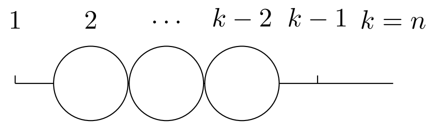
(b) possible S_{c-2} when $nextState(S_{c-2}, k) = S_{c-1}$



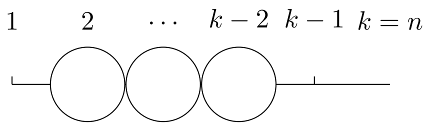
(c) possible S_{c-2} when $nextState(S_{c-2}, k) = S_{c-1}$



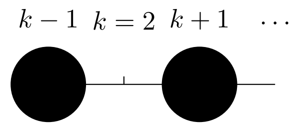
(d) possible S_{c-2} when $k = n$



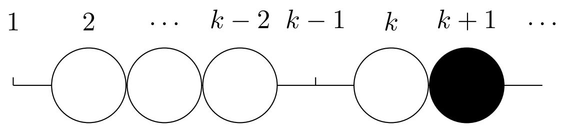
(e) possible S_{c-2} when $k = n$



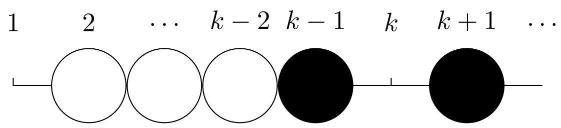
(f) possible S_{c-2} when $k = n$



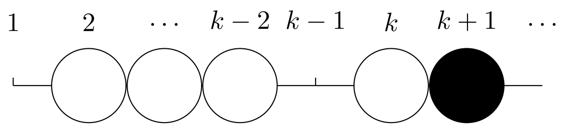
(g) possible S_{c-2} when $k = 2$



(h) S_{c-1}



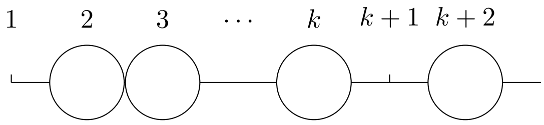
(i) S_c



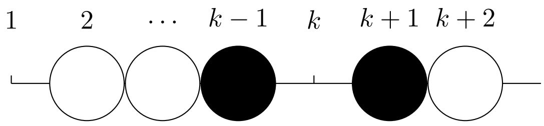
(j) S_{c+1}

Figure 5.16: $m = k - 1$

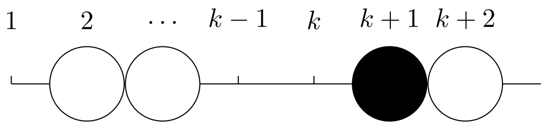
doi:10.6342/NTU201802448



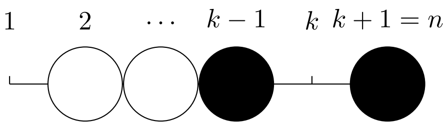
(a) S_{c-1}



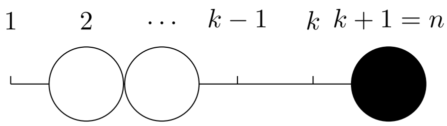
(b) S_c when position $k-1$ is black



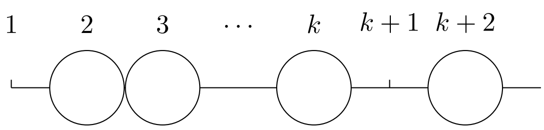
(c) S_c when position $k-1$ is empty



(d) S_c when $k=n-1$ and position $k-1$ is black



(e) S_c when $k=n-1$ and position $k-1$ is empty



(f) S_{c+1}

Figure 5.17: $m=k+1$

doi:10.6342/NTU201802448

doi:10.6342/NTU201802448

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Compared to previous works about weakly-solved small-board-sized Go problem by alpha-beta search, we strongly solve the small-board-sized Go problem by retrograde analysis. The biggest Go board solved is 2×11. And the final result can be accessed using database. Using state reduction, changing the sort order and memory block size to make the better performance.

A retrograde analysis method requires a vast amount of memory. Previous approach solves this problem by either using parallelism [22], storing on disk [12] or advanced indexing method [10].

This thesis proposed to use efficient in-memory compression scheme. With saving separated compressed data in the memory instead on the disk, and decompressing this data on demand, there is a balance between performance and memory usage that allows the problem to be solved efficiently. This method can also be applied to large scale data processing.

A method is also determined that obtains the optimal result for boards with a single row.

doi:10.6342/NTU201802448

6.2 Future Work

6.2.1 Rule-Based $2 \times N$ Go

We successfully find the rule of the boards which row size is 1 and show that their results are all draw when column size ≥ 5. But we cannot find a rule-based method that allows an optimal result for a $2 \times N$ Go by simply taking features from board, can not even simply get the rule of the final result for $2 \times N$ Go. This method may be complicated or it may have incomprehensible rules. We know that 2×2 to 2×9 are Black's full win; but Black only partially win in 2×10 and 2×11. If there's a method can generate optimal result for $2 \times N$ Go, the method must to be applied in the boards which are bigger than 2×9.

6.2.2 Other Sorting Criteria

This study uses serial order and piece order for sorting, but there may be some more efficient sorting methods. The performance of a sorting method is significantly affected by the data locality. The sorting performance is also the key to improve the performance of the search algorithm.

For example, if we sort the states by the pass count at first, because the previous states of state which pass is 2 always have pass = 1, and the previous states of state which pass is 1 always have pass = 0, it may have better data locality to search.

doi:10.6342/NTU201802448

Bibliography

[1] Erik CD van der Werf, H Jaap Van Den Herik, and Jos WHM Uiterwijk. Solving go on small boards. *Icga Journal*, 26(2):92–107, 2003.

[2] Erik CD van der Werf and Mark HM Winands. Solving go for rectangular boards. *Icga Journal*, 32(2):77–88, 2009.

[3] Cheng-Wei Chou, Ping-Chiang Chou, Hassen Doghmen, Chang-Shing Lee, Tsan-Cheng Su, Fabien Teytaud, Olivier Teytaud, Hui-Ming Wang, Mei-Hui Wang, Li-Wen Wu, et al. Towards a solution of 7x7 go with meta-mcts. In *Advances in Computer Games*, pages 84–95. Springer, 2011.

[4] Erik CD van der Werf, Jos WHM Uiterwijk, and H Jaap van den Herik. Solving ponnuki-go on small boards. 2002.

[5] Chia-Chuan Chang, Ting-Han Wei, and I-Chen Wu. Job-level computing with boinc support. In *Technologies and Applications of Artificial Intelligence (TAAI), 2016 Conference on*, pages 200–206. IEEE, 2016.

[6] John Tromp and Gunnar Farnebäck. Combinatorics of go. In *International Conference on Computers and Games*, pages 84–99. Springer, 2006.

[7] Ken Thompson. Retrograde analysis of certain endgames. *ICCA journal*, 9(3):131–139, 1986.

[8] Haw-ren Fang, Tsan-sheng Hsu, and Shun-chin Hsu. Construction of chinese chess endgame databases by retrograde analysis. In *International Conference on Computers and Games*, pages 96–114. Springer, 2000.

doi:10.6342/NTU201802448

[9] Hiroyuki Iida, Jin Yoshimura, Kazuro Morita, and Jos WHM Uiterwijk. Retrograde analysis of the kgk endgame in shogi: Its implications for ancient heian shogi. In *International Conference on Computers and Games*, pages 318–335. Springer, 1998.

[10] John W Romein and Henri E Bal. Solving awari with parallel retrograde analysis. *Computer*, 36(10):26–33, 2003.

[11] Bruno Bouzy. Go patterns generated by retrograde analysis. *Evaluation*, 1(3):9.

[12] Ping-hsun Wu, Ping-Yi Liu, and Tsan-sheng Hsu. An external-memory retrograde analysis algorithm. In *International Conference on Computers and Games*, pages 145–160. Springer, 2004.

[13] 日本棋院. 囲碁パズル「黒猫のヨンロ」. Retrieved Apr 16, 2018, from the World Wide Web: https://www.nihonkiin.or.jp/teach/app/yonro/index.html/, 2011.

[14] Ko at sensei's library. Retrieved Mar 30, 2018, from the World Wide Web: https://senseis.xmp.net/?Ko, 2017.

[15] Scoring at sensei's library. Retrieved Mar 30, 2018, from the World Wide Web: https://senseis.xmp.net/?Scoring, 2017.

[16] Seki at sensei's library. Retrieved Mar 30, 2018, from the World Wide Web: https://senseis.xmp.net/?Seki, 2017.

[17] David B Benson. Life in the game of go. *Information Sciences*, 10(1):17–29, 1976.

[18] Louis Victor Allis et al. *Searching for solutions in games and artificial intelligence*. Rijksuniversiteit Limburg, 1994.

[19] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, 1996.

[20] zlib. Retrieved May 11, 2018, from the World Wide Web: http://www.zlib.net/, 2018.

doi:10.6342/NTU201802448

[21] zlib 1.2.11 manual. Retrieved May 31, 2018, from the World Wide Web: https://www.zlib.net/manual.html, 2017.

[22] Henri Bal and Victor Allis. Parallel retrograde analysis on a distributed system. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pages 73–73. IEEE, 1995.

doi:10.6342/NTU201802448