International Core Journal of Engineering Volume 7 Issue 6, 2021
ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

Project Report: Improvement of Surakarta Game AI

Junxi Chen[1], Jiaxun Gao[2], Jiacheng Sun[3], Xiao Lin[4], Xiangyu Ren[5], Zihu Xu[6]

[1]Faculty of Information Technology, Beijing University of Technology, 100 Pingleyuan, Chaoyang District, Beijing 100124, China;

[2]School of Electrical Engineering and Computer Science, University of Ottawa, 75 Laurier Ave. East, Ottawa ON K1N 6N5 Canada;

[3]Department of Information and Control Engineering, Xi'an University of Architecture and Technology, No. 13 Yanta Road, Xi'an, Shaanxi, 710055, P.R. China;

[4]SWJTU-Leeds Joint School, Southwest Jiaotong University, No. 999 Xi'an Road, West High-Tech Zone, Chengdu, Sichuan, China;

[5]School of Electrical Engineering and Computer Science, Pennsylvania State University, 201 Old Main, University Park, Pennsylvania 16802, USA;

[6]Rutgers Business School, Rutgers University, 1 Washington Park, Newark, NJ 07102 United States.

Abstract

The significant development of AI has rendered old traditions energetic again at this modern age. While AlphaGo shining on the field of Go chess, other board games are also taking advantages from AI. This report covers the topic of a project which use a variety to improve the performance of AI playing Surakarta game. In the project, we optimize the minimax search algorithm with alpha-beta pruning by Null-Move strategy, Transposition Table and we also use the method of deep reinforcement learning to let the AI to take advantage of moves it made in the past. In general, the project makes the Surakarta AI get a greater winning rate.

Keywords

Artificial Intelligence; Surakarta; Optimization.

1. Introduction

For a long time, it is generally believed that it is completely unrealistic for machines to defeat humans in the field of Go. Although "Deep Blue" defeated Kasparov in 1997, the performance of artificial intelligence on Go is still very weak. At that time, due to complexity of Go, it was considered as the highest achievement of Artificial intelligence to be really performed on Go.

However, in March 2016, ALphaGo developed by google Deepmind defeated Li Shishi with score of 4-1, and a year later, AlphaGo Zero defeated its predecessor with 100-0. The Alpha Go/Zero system combines Monte Carlo Tree Search and Residual Convolutional Neural Network to achieve a great project. In this project, we will do the similar method but on a different chess, Surakarta.

2. Null-Move

The Null-Move strategy is an optimization based on minimax search algorithm to let the AI get much greater search depth within a short time.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

2.1 Maximum and minimum search

In 1950, Professor Claude Shannon first proposed the "maximum-minimum algorithm ".[1,2,3] The "maximum-minimum algorithm" laid the theoretical foundation for computer man-machine games. As shown in Figure 1, if the evaluation score of the game is denoted as the difference between the sum of values in the white part and that in the black part, then the white side can be considered as the maximum value side of the evaluation value, Black is the minimum value of the evaluation value. If the white side is set to move, its choice in the even layer is to find the one with the highest evaluation value among all its child nodes, that is, the "Max search". The opposite side Black is on the odd layer Select the one with the smallest evaluation value among all its child nodes, namely "Min Search", the schematic diagram is shown in Figure 2.

"Maximum-minimum algorithm" belongs to the Brute force without cutting for the game tree. [4] The number of search nodes is BD, which B is the branching factor and D is the number of search layers. According to the existing hardware conditions, it is impossible to search the complete game tree, but it can go through the first few steps in a given situation. In theory, this method can be used to deal with Surakarta chess machine games, but it is far from enough to achieve a competition with human masters and win.

2.2 Alpha-Beta search

The maximum and minimum algorithm is the basis of all search algorithms, and the fundament of all pruning algorithms is Alpha-Beta search. Figure 3,4 shows the schematic diagram of the search and pruning process, and the best path as shown by the thick arrow is obtained. Alpha-Beta pruning is carried out according to the maximum and minimum search rules. Although it does not necessarily traverse some nodes of some search subtrees, it is still based on exhaustive search. The study found that if the nearest path is found as soon as possible, the remaining branches can be pruned, greatly reducing the search nodes. The discovery of pruning skills has laid the foundation for the realization of the computer to defeat human chess players. The flow chart of Alpha-Beta pruning is shown in Figure 5.

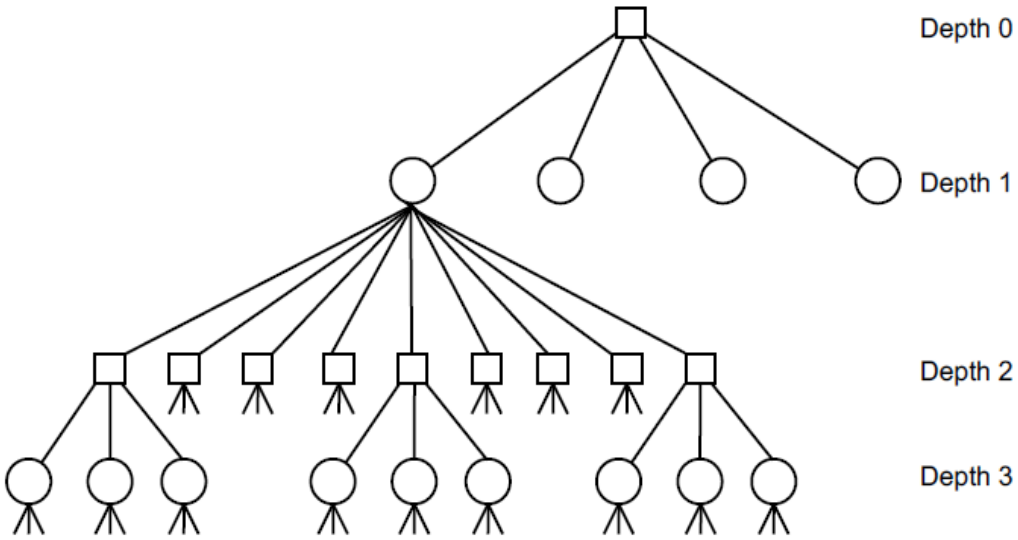

Figure 1. The figure of games tree structure with three-depth

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

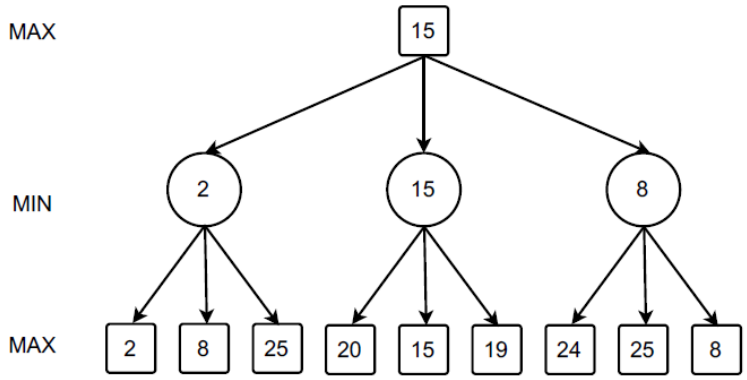

Figure 2. The sketch map of Minmax Algorithm

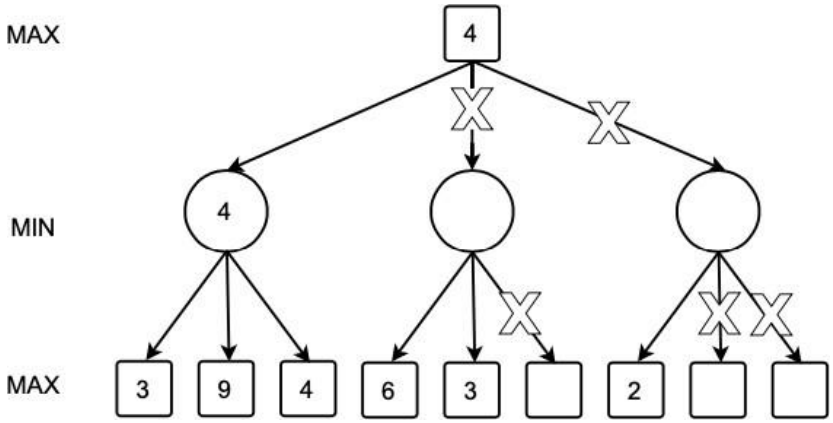

Figure 3. Schematic diagram of Alpha pruning

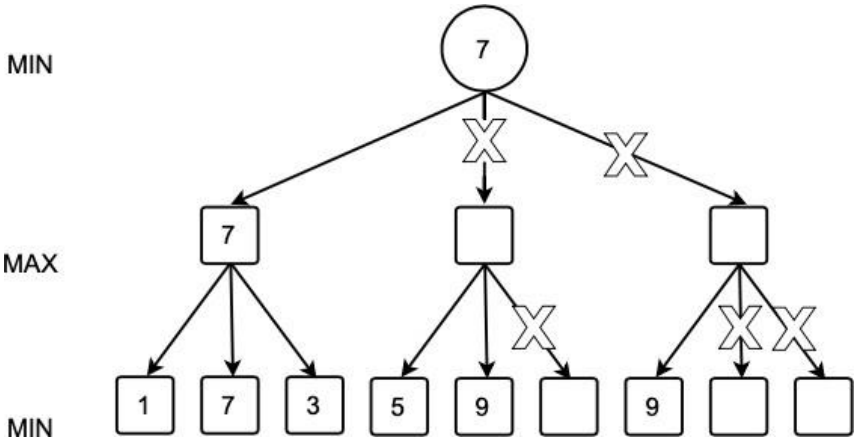

Figure 4. Schematic diagram of beta pruning

The research found that the efficiency of the α-β pruning algorithm is highly related with the search order of the subtree nodes. In the most ideal case (minimal tree), that is, the leftmost branch is the best path, and the number of nodes generated is 8.

D is even

$$N_D = 2B^{D/2} - 1 \quad (1)$$

D is odd.

$$N_D = B(D+1)/2 + B(D-1)/2 - 1 \quad (2)$$

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

Where D represents the search depth and B denote the branching factor. When α-β pruning is not used, the number of nodes to be searched is $N_D = B^D$, which is the maximum tree. Therefore, in the ideal case, the number of nodes with a search depth of D for the α-β algorithm is equivalent to the number of nodes with no pruning at a search depth of D/2, so it can be seen that the research on the ordering (branch expansion node) sorting will be an important subject of search [5].

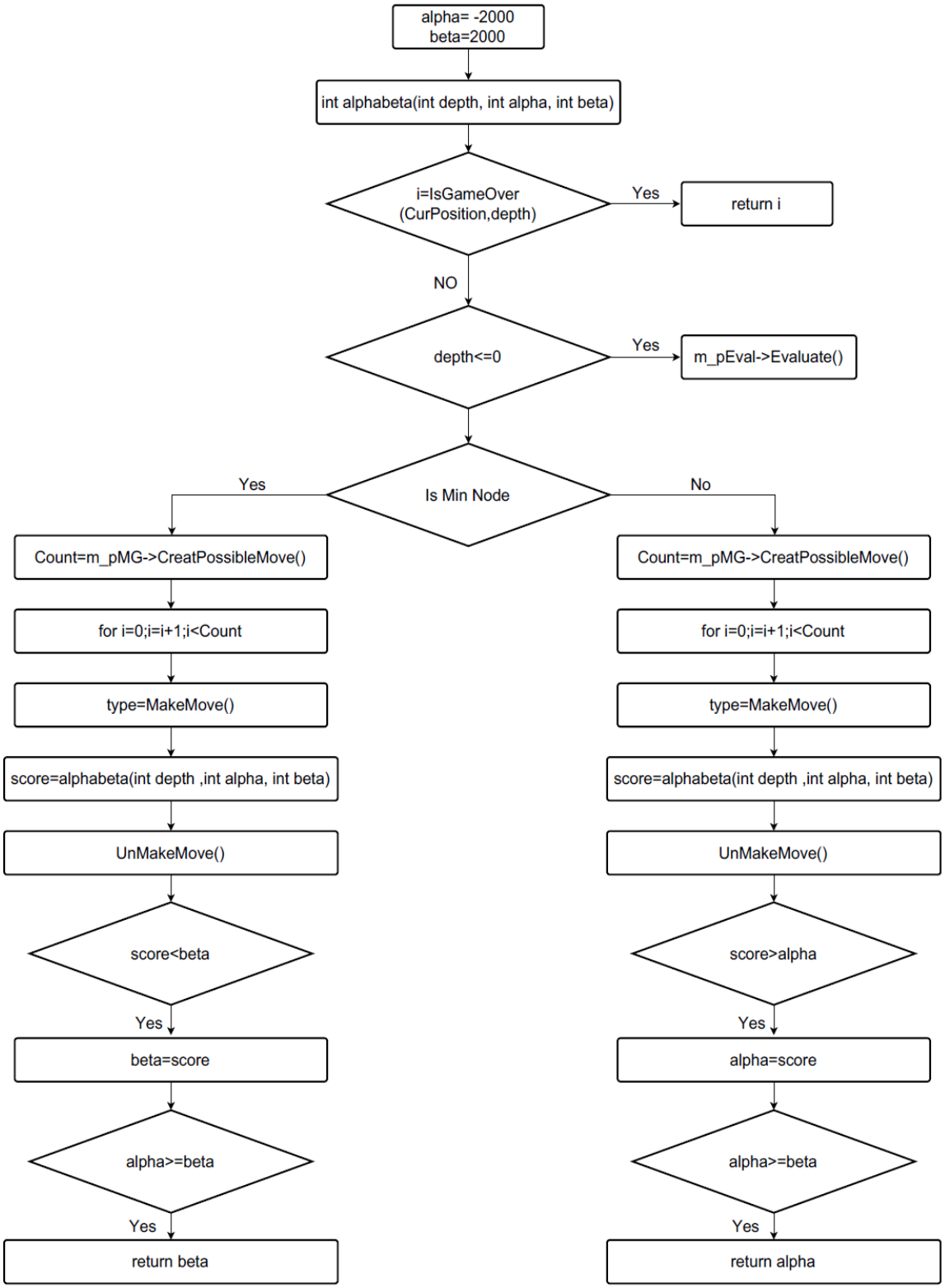

Figure 5. The flow chart of Alpha-Beta Pruning

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

For example, when the move is generated (node expansion), the first move is generated, especially the “big child” move with a high score is generated first, and the resulting move is more likely to be the best.

2.3 Null-Move Strategy

For a node of the search tree, in general, it has fewer good moves. The purpose of pre-prune search is to quickly determine which moves are bad. Null-Move Forward Pruning search is one of the most famous algorithms. It was first proposed by Chrilly Donninger in ICCA Magazine in 1993. It uses an adventurous strategy that may ignore important routes, and the chess search branch factor is rapid reduced, and the search depth is significantly improved.

The idea of null move is to think of a question before searching for any moves: "If I do nothing here, what can the opponent do?" The previous approach was just to find the best move to fight back the opponent, but this is different. Because in most cases, in the game of Surakarta chess, if one side moves continuously, it will gain considerable benefits, or even turn into a victory. So if a certain chess game, even if you let the opponent take one more step in a row, you still have the advantage, then you can do a search to reduce the depth, let the opponent go first, if the result of this search is greater than or equal to Beta, you can simply return to Beta without searching furthermore, so that it avoids the danger of forward pruning based on static functions, and the implementation is relatively simple.

In this method, a reduced depth search is applied to save time. As R represents the depth reduction factor, compared to searching all moves with depth d, the search time of the opponent with d-R is now greatly reduced. If R is equal to 2, then 6 layers of data should be searched, and only 4 layers need to be searched in the end. This saves a lot of time, and practice has proved that it is possible to increase the search by one or two layers without a crop algorithm. The flow chart of NullMove pruning is shown in Figure 6. Null move search plays a very important role in improving the level of the engine. Although the principle of NullMove search seems very simple, it is difficult to make NullMove search perfect. It must be applied with caution. Pay attention to the following issues:

i) NullMove search is just a judgment process, and the Alpha value should not be modified. In addition, Null Move just judges whether there is a threat. It is very dangerous to modify the size of the window with the return value.

ii) About the choice of depth reduction factor. The depth reduction factor is generally taken as 2, but it can also be taken as 3 for higher search engine efficiency. The practice of adjusting the R value according to different situations is called "Adaptive Null-Move Pruning "[6], and its content can be summarized as:

(1) When the depth is less than or equal to 6, use R = 2 to search.

(2) When the depth is greater than 8, use R = 3.

(3) When the depth is 6 or 7, if each piece is greater than or equal to 3, then use R = 3, otherwise use R = 2. Verify whether it is safe to cut empty, because in Surakarta chess, there are also many times that continuous chess moves do not gain an advantage. The research on the security of empty cuts, namely "Verified Null-Move Pruning with Verification "[6], was first published by Tabibi in the 2002 ICGA (former ICCA) magazine. Its content can be summarized as:

(1) Use R = 3 for null move search.

(2) If it is higher than the boundary, then do a shallow search.

(3) When doing a shallow search, use Null move without verification and set R = 3 for the child nodes;

(4) If the shallow search is higher than the boundary, the null move search with verification is successful, otherwise the full search is necessary.

iii) Two Null Move searches cannot be used consecutively. This does not mean that Null Move can only be used once at the beginning of the search and can no longer be used in the future, but that Null Move search can be used multiple times on a branch, but it cannot be used continuously. There must be a normal search in between.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

iv) Do not pruning when depth=1, avoid using Null Move. Because if depth=1, the opponent's chess checks you, the consequence of using Null Move is that the checked side avoids the checked and eats the opponent's chess piece, It will make the judgment of the computer far from the actual situation.

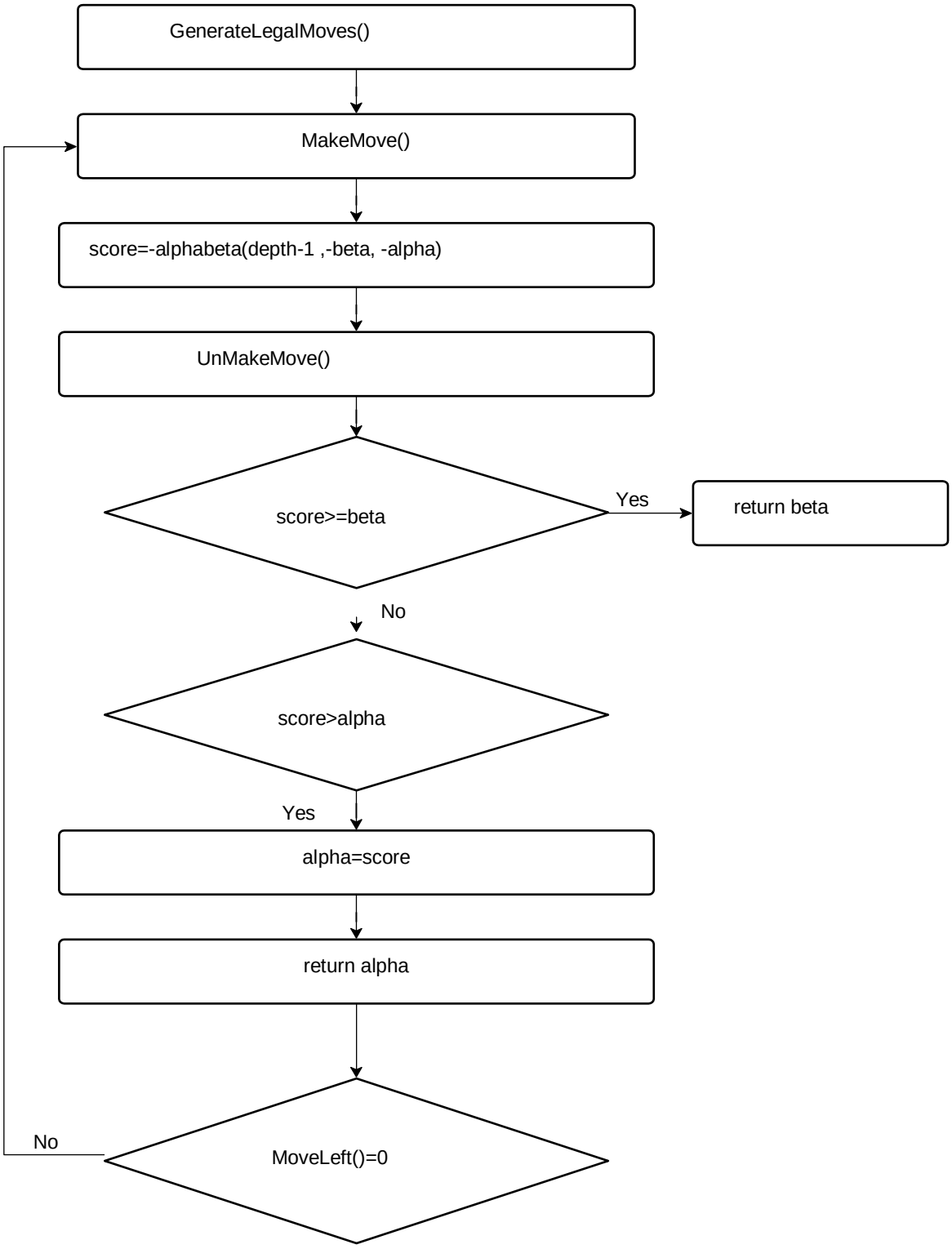

Figure 6. The flow chart of NullMove pruning

v) Judgment of the "Zugzwang" situation.[7] "Zugzwang" is a special situation in chess. In this case, it is often more advantageous not to play chess. At this time, the use of NullMove will bring great problems.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

3. Transposition Table

Since a search tree for chess game usually generates many nodes, it would cause a large amount of time to search all nodes. Alpha-Beta pruning algorithm is the most common and effective method to make the search process faster. With the implement of Alpha-Beta pruning algorithm, a large quantity of nodes will not be searched. However, this algorithm can still be optimized, and transposition table is one of the common techniques. Since there might be some nodes with the same chessboard state in a search tree, transposition table is designed to avoid such a repetitive search. Consequently, the time consumption could be further reduced.

3.1 Theory

3.1.1 Basic principle

All the chessboard state that has been searched will be saved in the Transposition table and their chessboard data can be used directly when some chessboard state appears again. Therefore, there is no need to search the state again. More specifically, when a chessboard state is going to be evaluated, checking the transposition table first to see if there is a record with the same chessboard state. If is, the evaluation result can be used directly. [8] Since there is a large amount of state that need to be stored in transposition table, it requires a large space. Therefore, the main idea of transposition table is using more space to reduce the time cost, then make the search process faster.

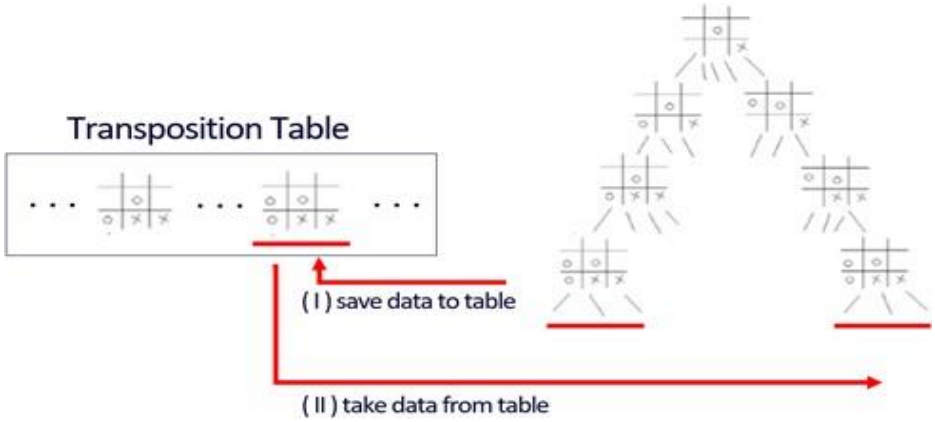

Figure 7. An example of using Transposition Table

3.1.2 Hash Table

Transposition table is a data structure based on hash table. By using hash technique, a specific record can be located immediately, thus saving a lot of time searching in the transposition table. Hash table is a data structure accessed directly according to a series of Key-Value records and each record has a unique address that depends on the key. The relation between addresses and keys is called hash function. There are some methods to construct this function but the most common one is division method.

The expression of division method is not difficult, but it has something need to be noticed that m here is the length of the hash table and p usually is a prime number that less than m.

3.1.3 Time complexity and Collision

With the usage of hash function, the address of a specific chessboard state can be calculated directly by the given key, then go to the address and take the value. This process only cost O (1) time, where a traversal search would take O(n) time. It is possible that different key would get the same address, but one address cannot store more than one value at once, this is called collision. Although there are many solutions about collision, they will not be applied to the transposition table. To guarantee the search speed, if there is a collision, the previous data would be overwritten if some conditions are met.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

3.1.4 Data structure and The working process

In the transposition table, each record contains 4 kinds of data for a chessboard state. In addition to the necessary (1) key and (2) value, there is a (3) type and a (4) depth.

i) More specifically, key is a 64-bit integer used to uniquely identify the chessboard state. In fact, the key is generated according to the chessboard state. The process can be summarized as 3 following steps. First, constructing a function which can return a 64-bit integer randomly. Second, using this function to give a random number for each chess piece. Since Surakarta game only has two kinds of chess and has 36 accessible position on the chessboard, there are 72 random numbers in total. Third, setting an initial key = 0 and let key do the XOR operation with each piece on chessboard. After that, the final key is able to identify the chessboard state because all chess pieces have been included in the XOR operation.

ii) Type is used to represent the value's type. Because in the search process, the exact value of the node is hard to obtained, and in most cases, it would return the upper bound value or lower bound value of the chessboard state. Consequently, there are three kinds of type, HashExact is defined to indicate the value is exact, HashAlpha is defined as upper bound and HashBeta is defined as lower bound. For example, suppose there is a chessboard with value 16, if type is HashAlpha, it means the value is at most 16, if type is HashBeta, it means the value is at least 16.

iii) (4) Depth is used to indicate how deep a chessboard has been searched, it is obvious that the value is more accurate if the depth is deeper. Because the deeper level in a search tree is closer to the result. Suppose the current chessboard A with depth D, and there is a same chessboard A' in transposition table with depth D'. Only when D ' ≥ D can A directly use the value of A'. Because the value in transposition table is more accurate.

3.1.5 Transposition strategy

The example above follows one of the transposition strategies, Depth- First. Every time a same chessboard state was found in transposition table, there would exist two chessboard state at once, the current one and the saved one. Then it is necessary to decide which one to save into the transposition table. Chessboard with deeper depth has the priority to read and write. It means that the only factor is depth, because transposition table is designed to save the valuable chessboard. But this strategy also has a defect that some new data cannot be saved because of the depth. Always-Replace is another strategy, it can save new data without any condition. But it may cause the loss of valuable data.

3.2 implementation

3.2.1 Code implementation

The core part of implementing transposition table is the improvement of Alpha-Beta function. In general, a function is added before each return value to store the value in the transposition table, and another function is used to search if there is a same chessboard in the transposition table before each recursive function is executed. More specifically, at the beginning of the function, it will check the search state, if game is over or search is over, then stores the value to transposition table and returns this value. If the search is not finished and it finds a corresponding value in transposition table, then returns the value directly. Otherwise, it will generate move array to store all possible moves. [9] For each possible move in move array, do the recursive function first, and it will return a value after the recursive function finished. If it is going to do the beta pruning, then stores beta to transposition table first. And if this value is bigger than alpha, then assign this value to alpha. After all possible move have been searched, store the final alpha to transposition table. The flow chart is as follows:

3.2.2 Problem after adding transposition table

Theoretically, if a corresponding value can be found in transposition table, then the following recursive function will not be executed, which will save a lot of time. However, the test result was contrary to expectation. After running the test function, the time consumption even increased rather than decreased. Obviously, transposition table did not work, and the reason is that a lot of data needs to be stored to transposition table, but they are rarely used. To improve the utilization of data in

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

transposition table, a text file is created to store these data and next time carry these data from text file to transposition table before starting search.

3.2.3 File implementation

There are four kinds of number that should be stored to file. The first one is key, it is a random 64-bit integer, and the second number is depth, in our project, the depth is at most 7. The third one is type. It has 3 kinds of number to represent 3 different value type. The last one is value, which is used to represent the chessboard situation. There is an instance shown as follows: It is one of the data from text file. It shows that the chessboard with this key could win the game after 7 steps. The value here is 9999 and it means the game is over and the current player is winner. Otherwise if the value is -9999, it means the current player might lose the game after 7 steps.

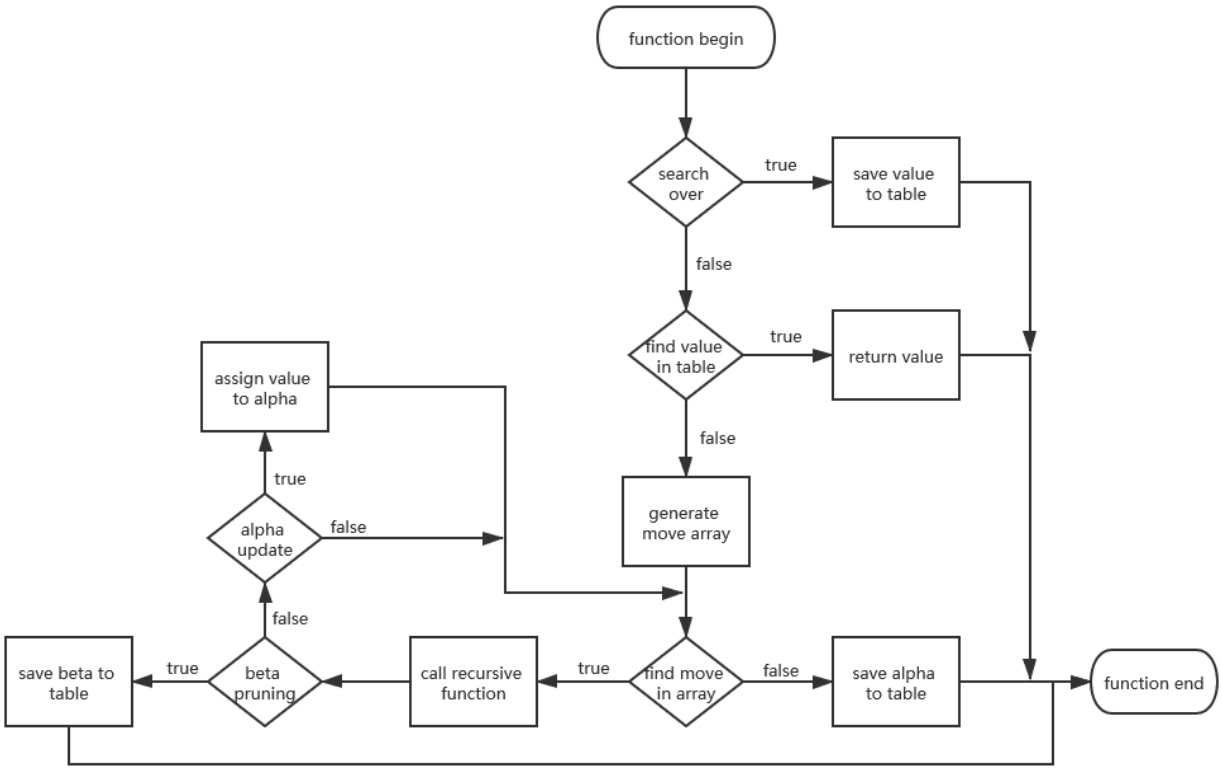

Figure 8. The flow chart of search process.

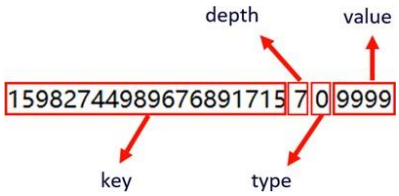

Figure 9. An instance from text file.

3.2.4 Problems after file r/w

After reading data from text file, the time consumption did not make any progress. The reason is that the data from text file could not be mapped because the program generated a new random key for each chessboard. Therefore, to guarantee that each chessboard always gets the same key, another text file is created to store 72 random integers. Because all keys are generated based on these 72 numbers. With these two text files, the time consumption problem has been solved. However, the error number in test result increased sharply from 39 to about 90. The sharp increase is caused by test cases. In each test case, there are 7 different depths from 1 to 7, and each depth will return a value by calling search

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

function. But after using transposition table, each test case could directly get the value with the biggest depth 7. Therefore, all these depths were return the same value which is from depth 7.

3.3 Result

Since only the test cases with depth 7 are effective, there is a table shows the comparison between alpha-beta search and TT with different table size. First, without using TT, the error number is 6 and it cost 10 seconds. By using TT, you can see when the table size is 2 to the power of 9, there was no extra error number, but the time consumption was reduced by only 80 percent. And when the table size is 2 to the power of 11, the time consumption was reduced by nearly 100%, but the error number was increased by about 33%. If the size continues to increase, its efficiency will gradually decrease.

4. Monte Carlo Tree Search

4.1 introduction

Reinforcement Learning is an important field of Machine Learning that trains AI to be more suitable for one and only one specific field in daily usage, such as chess games. A great example is the project from Google, AlphaGo Zero. [10] In this work, we did the following: Write cpp based MCTS programs, and modified rollout policy and weights of attack moves to increase efficiency of MCTS AI; Rebuild MCTS in python for implementing reinforcement learning and neural networks following the extremely-randomly-pick policy of MCTS.

Monte Carlo Tree Search (MCTS) is a well-known and successful algorithm of searching when implementing on board games, has been proven by AlphaGo Zero. In past years, Teams from Google, who wrote AlphaGo, used AlphaGo to beat world players in Chess Go, even mastered it at the end. Since MCTS is a relatively independent algorithm, we don't need to change the code for implementing it on different games. When implementing MCTS, searching trees are built recursively in a loop, and we have 4 steps for each searching: select, expand, simulate, backpropagate.

MCTS uses an important factor UCT to balance between exploration and exploitation. c here is set to $\sqrt{2}$. [11]

$$UCT = w_i/n_i + c\sqrt{\ln N_i}/n_i$$

So, to implement MCTS on Surakarta, at first, we create a root node for the MCTS tree. Then expand the root node with all potential moves from current state as children node. See Figure 10.

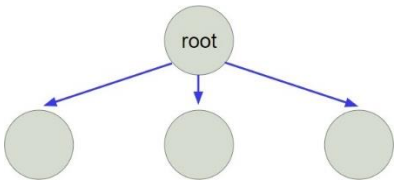

Figure 10. Expanding the root

Since all children nodes aren't visited, we select the first node, plus 1 to visit count of the node, and do simulation following specified rollout policy: randomly choose from current player's potential moves then swap player until game is in terminal state, then return and update to all parent nodes 1 if AI wins, 0 if loses. Take wins as an example, See Figure 11.

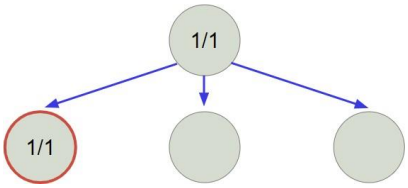

Figure 11. Simulation and Backpropagation

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

After visited all leaf nodes, the MCTS tree with w_i/n_i labeled might look like Figure 12. Then compute UCT of every node to decide which node to select and expand.

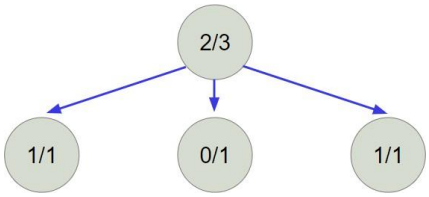

Figure 12. Simulation and Backpropagation

We can get following equations: for the First node,

$$UCT_1 = 1 + \sqrt{ln3/1}$$

for the Second node,

$$UCT_2 = 0 + \sqrt{ln3/1}$$

and for the Third node.

$$UCT_3 = 1 + \sqrt{ln3/1}$$

Depends on maximizing UCT, select the first node again, and we expand the node with potential moves. All expanded nodes are not visited, select the first leaf node, and do simulation.

Starting from the root node, Compute and Maximize UCT again. Third child node of the root node has max UCT this time, See Figure 13. Select the node and expand.

The 4 processes will keep looping until hit the preset terminal stage, in this project, it is 1600 times.

4.2 Multi-threading Monte Carlo Tree Search

MCTS is easier to be implemented with multi-thread than other traditional tree search algorithms, such as alpha-beta algorithm, because of its own 4 separate processes. Also, programs running on python are relatively slow on its efficiency. With an idea of parallel search, implementing MCTS with multi-threading properly set up can increase the efficiency by at least 30%. [12,13]

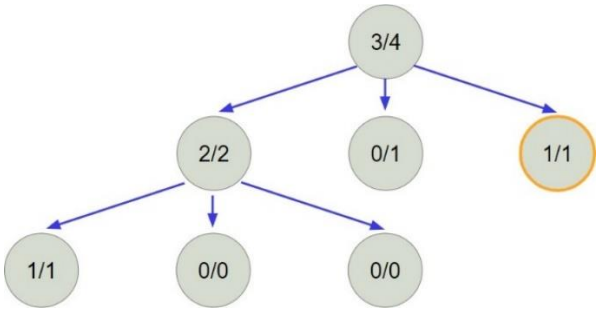

Figure 13. Simulation and Backpropagation

A possible approach is to add buffers to the MCTS using pipeline. With the MCTS algorithm introduced in Section 1.1, instead of simulating for one leaf node, simulate for all leaf node of the selected node which has the max UCT. [14]

4.3 Implementation

4.3.1 Working on MCTS using cpp

While preparing MCTS using cpp for Reinforcement Learning, we met the following problems.

When we exactly followed the policy of MCTS, which is to randomly choose potential moves when simulating, the MCTS AI performs badly – could not even win a rookie to the game.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

For the first problem, we modify our rollout policy, which is used in the simulation process, to randomly pick potential attack moves first, then other potential moves. Also, we change the weight of the UCT of attack moves by multiplying 1.4 when AI is making its final move. With these two changes, the AI performs more aggressive and a little smarter, however it still has 47% percent of losing rate when competing with other Surakarta AIs, See Figure 14.

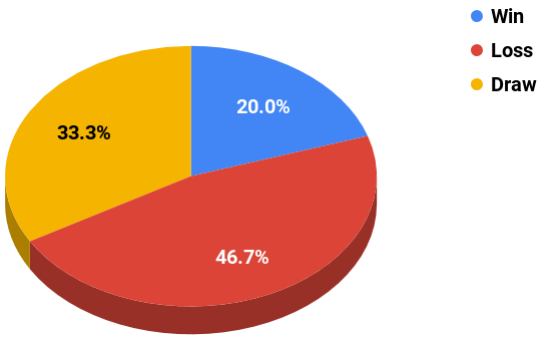

Figure 14. MCTS AI vs. Other Surakarta AIs

We also add a destroy function to free memory of the tree that was built when the function terminated to avoid any memory leaks.

4.3.2 Datasets input for Reinforcement Learning

In the dataset for each game, it has 17 boards and a list of UCTs of all potential moves for each round of the game, which is each chess play in our project. See Figure 15, A has 1 board to determine the side: a 6×6 matrix filled with 1 or -1; B has 8 enemy's history boards: eight 6×6 matrices filled with 1 and 0; C has 8 AI's history boards: eight 6×6 matrices filled with 1 and 0. So we wrote an output function to write in data each round. Starting from the initial state, which is when the game begins, We initialized the stack with one identity board and 2 eight-same boards:

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
1	1	1	1	1	1
1	1	1	1	1	1

and

1	1	1	1	1	1
1	1	1	1	1	1
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

At each play, push or insert the latest board into the stack. If the size of one side is over 8, pop that oldest board from that side.

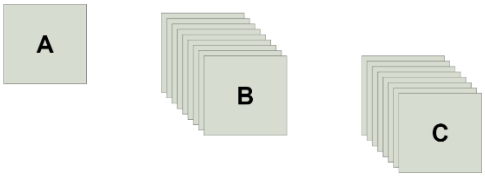

Figure 15. 17 boards in dataset

4.3.3 Migrating MCTS to python

We let MCTS AI self-play a few games, and generate the datasets we need, we found that We misunderstood Reinforcement Learning and confused with Supervised Learning. For this problem, with our previous understanding of Reinforcement Learning, which is actually Supervised Learning, using the old setup of the neural networks can only give us an approximately same performance of the provided AI, might increase the time efficiency but not make the AI more intelligent or play better on Surakarta game. So, we migrate the MCTS to python, rewrite it following the randomly-pick rollout policy, rebuild the policy setup of the neural networks and train it using Google Cloud Service.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

5. Reinforcement Learning

To improve the performance of the AI, we use the method of deep reinforcement learning to let AI learn the strategy from the history of his own moves. The reinforcement learning requires no data from others than the AI itself. The advantage of this strategy is that reinforcement learning requires no human intelligence to guide the search and there is no need to write the evaluation function compare to the minimax search due to the algorithm is a MCTS based method.

5.1 Process of Reinforcement Learning

First, we need to build a deep neural network fθ with initial parameter θ. The input of the network will be a stack of gameboards information includes the current gameboard and its history of 7 gameboards. The deep neural network will output both a policy head and a value. The policy head is a vector which contains the probability of selecting each move and it can be represented as pa = Pr (a|s). And the value is a scalar evaluation which indicates the probability of the current player winning after game from the current position s. This neural network uses the architecture of ResNet which has been testified to have a great outcome in the chess-like game.

There are three steps of the reinforcement learning, which are self-play, policy update and policy evaluation. As it's mentioned, a reinforcement learning AI learn to play the game by its own. Thus, we need to generate the training data from self-play before we train the deep neural network. The AI use Monte Carlo Tree Search as the search method to do the search of the current position s. However, the MCTS process has been modified with the output of the neural network. In each position s, and MCTS search is executed and the MCTS search outputs probabilities π of playing each move. These search probabilities usually select much stronger moves than the raw move probabilities p of the neural network fθ (s); MCTS may therefore be viewed as a powerful policy improvement operator. The play from the AI against itself will use the improves version of MCTS search to place each move and then use the game winner z as a sample of the value – may be viewed as a powerful policy evaluation operator. The main idea of the reinforcement learning algorithm is to use the search operators repeatedly in policy iteration procedure. The intentions of the training is to make move probabilities and value (p, v) more closely match the improved search probabilities given by MCTS and self-play winner(π, z); these new parameters are used int the next iteration of self-play to make the search stronger. The final goal of the reinforcement learning is to maximize the future reward.

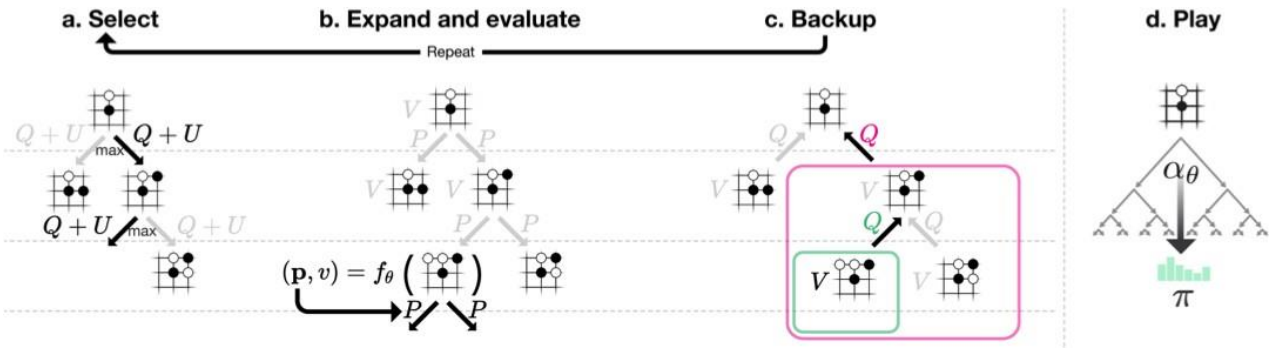

Figure 16. Modified MCTS

5.2 Self-play

The program plays games against itself from the beginning state to terminate state. In each position st, a Monte-Carlo tree search (MCTS) αθ is executed (see Figure 17) using the latest neural network fθ. Moves are selected according to the search probabilities computed by the MCTS, at πt. The terminal position st is scored according to the rules to compute winner. The Monte-Carlo tree search uses the neural network fθ to guide its simulations (see Figure 17). Each edge (s,a) in the search tree stores a prior probability P(s,a), a visit count N(s,a), and an action-value Q(s, a). Each simulation

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

starts from the root state and iteratively selects moves that maximize an upper confidence bound Q (s,a) + u (s,a), where u (s,a) $\propto$ P(s,a)/(1 + N (s,a)), until a leaf node s' is encountered. [15] This leaf position is expanded and evaluated just once by the network to generate both prior probabilities and evaluation, (P (s,), V (s,)) = fθ (s,). Each edge (s, a) traversed in the simulation is updated to increment its visit count N (s, a), and to update its action-value to the mean evaluation over these simulations, Q (s, a), where s, as, indicates that a simulation eventually reached s' after taking move a from positions. MCTS may be viewed as a self-play algorithm that, given neural network parameters θ and a root position s, computes a vector of search probabilities recommending moves to play, π = αθ (s), proportional to the exponentiated visit count for each move, πa $\propto$ N (s, a)1/r, where r is a temperature parameter. If the temperature is big than it encourages the MCTS to explore the game tree, otherwise the algorithm will stick to the search result. [15]

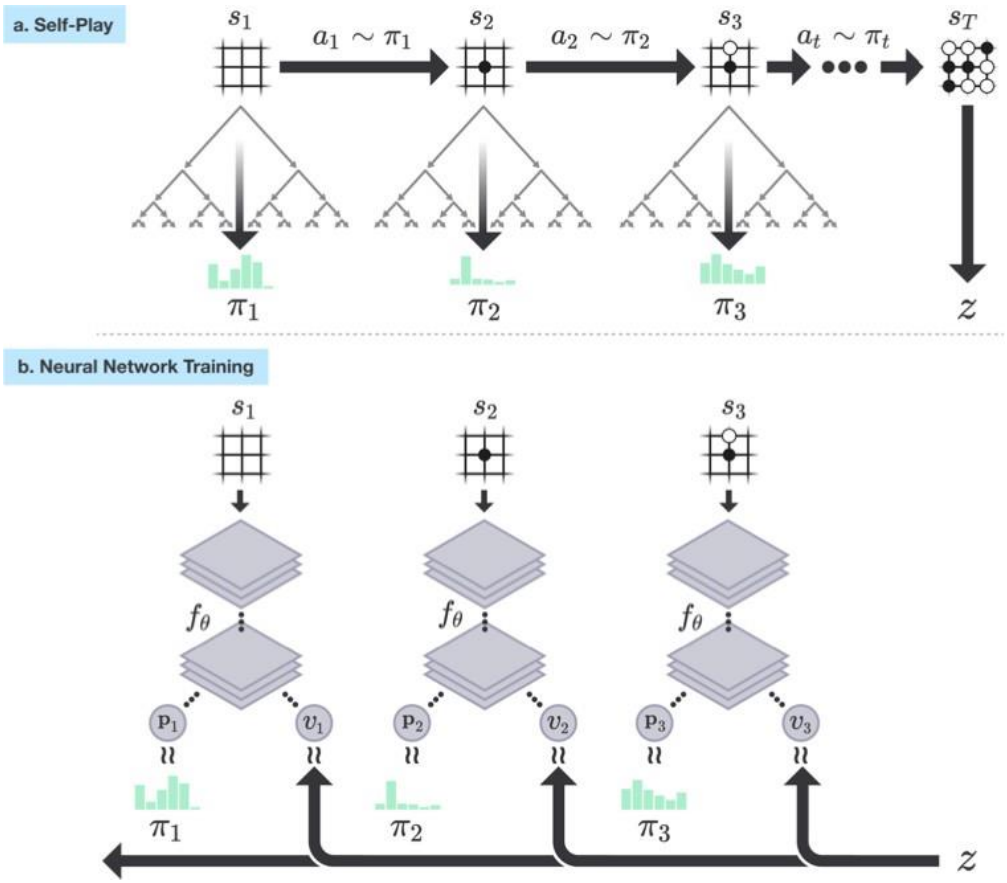

Figure 17. Training Process

Then we can use the following formula to calculate $u(s, a)$

$$u(s, a) = c_{puct} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)} \quad (3)$$

we select the best move with max $Q + U$

$$at = \text{argmax} (Q (st , a) + u (st , a)) \quad (4)$$

With the best action, we can explore or expand the following branch and continue the MCTS. If we have met a state s which is not in the game tree. Instead of doing the simulation, we can simply use the value predicted from the deep neural network to backpropagate. The two system is combined closely together. The prediction from the neural network will redirect the search of the neural network. As we know the reinforcement learning process is not stable. However, the MCTS will stable the process of reinforcement learning by its regulation.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

5.3 Policy Update

The neural network is trained by a self-play reinforcement learning algorithm that uses MCTS to play each move. First, the neural network is initialized to a random parameter theta. The MCTS will follow the current policy to do the search. The data of each time-step is stored as a tuple (s, π, z). In parallel, new network parameters θi are

trained from data (s, π, z) sampled uniformly among all time-steps of the last iteration of self-play. The goal of the

training is to minimize the difference between the prediction from the neural network and data generated by MCTS. The parameter theta are adjusted by gradient descent on a loss function which includes three entry, mean-squared error and cross-entropy losses and L2 weight regulation.

The deep neural network will give its prediction by each round. However, the reward z is given after the terminate state of the game by each game. Therefore, we can only train the network and update the policy after each game.

5.4 Policy Evaluation

Finally, it comes to the evaluation of the new policy. We let the AI that has been updated to play against the old version of AI. If the wining rate is greater than 55%, we will replace the AI with the new version which has a better policy. In conclusion, we will only keep one AI to play and iterate.

5.5 Analyze

Different than the supervised learning, the deep reinforcement learning requires a great amount of computation resource. We need at least half an hour on a 2-core xeon virtual machine with an Tesla K80 GPU. It is been tested, that there are barely no intelligence of AI under 4000 times of training. The training will take a relatively long time because both the process of MCTS and prediction will take a great amount of time. So at the end of the project, we can not tell the real power of reinforcement learning after carrying out 100 training. Yet, we believe that the true ability of reinforcement learning is far more powerful. We simply need more training steps. We can address the problem by developing the multi-thread MCTS further. Therefore, we can decrease the time of MCTS by a lot. In the future research, we believe we can address the training problem.

6. The Architecture of the Neural Network

6.1 Basic Introduction of ResNet

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which takes in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. [15] By applying relevant filters, a ConvNet can successfully capture the Spatial and Temporal dependencies in an image. The image dataset can be better fitted due to the reduction in the number of parameters involved and reusability of weights. In other words, through the training process, the network can understand the sophistication of the image better. A ConvNet on the whole is comprised of three layers: Convolutional Layer, Pooling Layer and Fully Connected Layer. [16]

The convolutional layer is central to the convolutional neural network that provides the network its name. In this layer, an operation called a “convolution” is performed. A convolution is a linear operation that includes the multiplication between set of weights and the input data, similar to the traditional neural network. Since this technique is initially invented for input in two dimension, the multiplication is performed between an array of input data and weights in two-dimensional array, called a filter or a kernel. The output of multiplication between the filter with the input array in one time is a single value. As the filter is applied multiple times to the input array, the outputs can be collected in a two-dimensional array that represents a filtering of the input. As such, the two-dimensional output array from this operation is called a “feature map”. We can pass each value in the feature map through a nonlinearity for example a ReLU, just like what we do for the outputs of a fully connected layer. [17]

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

The Pooling layer plays a role in reducing the spatial size of the Convolved Feature furthermore. By reducing the dimension, it can achieve decreasing computational power required to process the data. The pooling layer has two branches: Max Pooling and Average Pooling. Max Pooling returns the maximum value from the portion of the image covered by the Kernel. In contrast, Average Pooling returns the average values from the portion of the image covered by the Kernel. In Max Pooling, an operation called Noise Suppressant is performed. It discards the noisy activations altogether and performs de-noising during dimensionality reduction. Average Pooling just simply performs dimensionality reduction as a noise suppressing mechanism. Therefore, it is reasonable to state that Max Pooling performs a lot better than Average Pooling. The Convolutional Layer and the Pooling Layer cooperate with each other and compose the i-th layer of a Convolutional Neural Network. We can increase the number of such layers to capture low-levels details even further, but the requirement of computational power also increased. [16]

Adding a Fully-Connected layer is a comparatively cheap way of learning non-linear combinations of the high- level features as represented by the output of the convolutional layer. The Fully-Connected layer is learning a possibly non-linear function in that space. Now that the input image has been converted into a suitable form for our Multi-Level Perception, we shall transform the image into a column vector. The transformed output is fed to neural network and back propagation is applied to every iteration of training. Over a series of epochs, the model is able to distinguish between principle and certain low-level features in images. [16]

6.2 Application to Surakarta

When applied to Surakarta chess, the input of neural network is the board status st in form of two-dimensional vector which represents position of each chess stone. Neural network passes it through convolutional layers with parameters θ, and then outputs both the probability distribution over possible moves pt and a scalar value vt, representing the probability of the current player winning in position st . The neural network parameters θ are updated in order to maximize the similarity of the policy vector pt to the search probabilities πt, and to minimize the error between the predicted winner vt and the game winner z (see Equation 1). The new parameters are used in the next iteration of self-play a.

$$(p, v) = f(\theta) \quad l = (z - v)^2 - \pi^T \log p + c\theta^2 \tag{5}$$

We applied a deep neural network f0 with parameters 0 in this method. This neural network takes the raw board status s of the position and its history as an input, and outputs both move probabilities and a wining value, (p, v) = fθ (s). The move probabilities p represents the probability of making each move, pa = Pr (a|s). The value v is a scalar evaluation which estimates the wining probability of current player at position s. This neural network plays the roles of both policy network and value network. The neural network consists of many residual blocks of convolutional

layers, with batch normalization and rectifier non-linearities. The input to the neural network is a 6×6 ×17 image stack comprising 17 binary feature planes. The input features st are processed by a residual tower that consists of a single convolutional block followed by either 19 or 39 residual blocks 4.

The following modules are applied sequentially to its input in each residual block:

1) A convolution of 256 filters of kernel size 3 × 3 with stride 1

2) Batch normalization

3) A rectifier non-linearity

4) A convolution of 256 filters of kernel size 3 × 3 with stride 1

5) Batch normalization

6) A skip connection

7) A rectifier non-linearity

The policy and value are computed by two different heads separately, in each of them output of residual tower is taken as input.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

The modules below are applied in policy head:

1) A convolution of 2 filters of kernel size 1 × 1 with stride 1

2) Batch normalization

3) A rectifier non-linearity

4) A fully connected linear layer that outputs a vector of size 36 + 1 = 37 corresponding to logit probabilities for all intersections and the pass move

The modules below are applied in value head:

1) A convolution of 1 filter of kernel size 1 × 1 with stride 1

2) Batch normalization

3) A rectifier non-linearity

4) A fully connected linear layer to a hidden layer of size 256

5) A rectifier non-linearity

6) A fully connected linear layer to a scalar

7) A tanh non-linearity outputting a scalar in the range [-1,1]

To visually show the flowchart, figure Neural Network Diagram is attached. The overall network depth, in the 20 or 40 block network, is 39 or 79 parameterized layers respectively for the residual tower, plus an additional 2 layers for the policy head and 3 layers for the value head. We use tensorflow packages to implement the whole Neural Network.

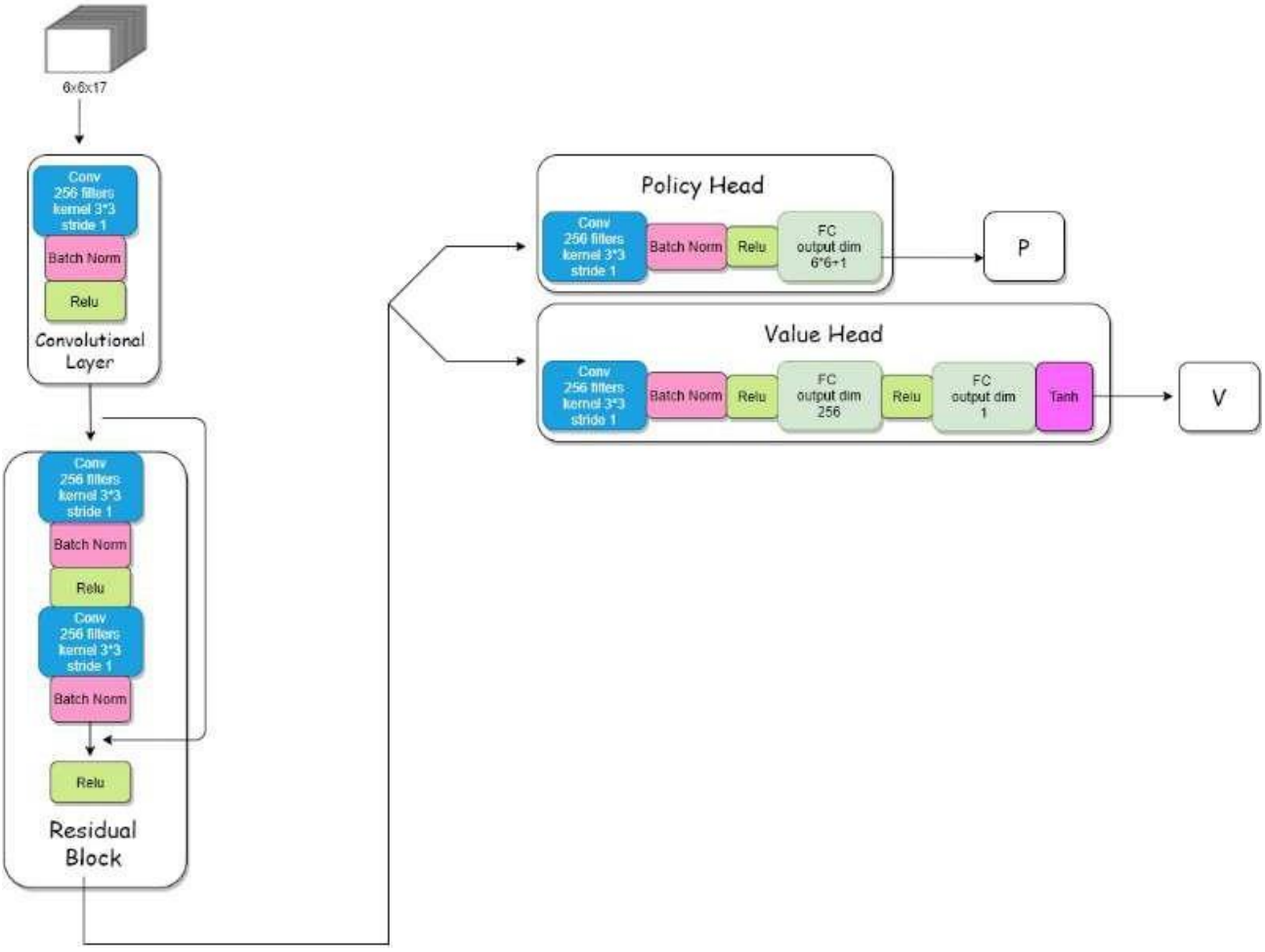

Figure 18. Neural Network Diagram

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

7. Input Processing

In this section, code will be implemented to process data imported from Mote Carlo Tree Search. Raw data is a txt file contains magnanimous Games played by two MCTS bots. By processing it, we can provide sufficient data to train neural network.

7.1 Definitions

For the convenience of interpretation, this section will introduce definitions required for processing data input which derived from Mote Carlo Tree Search.

7.1.1 Board

A board is a 6*6 2-dimension vector to represent real life chess board. But different from real chess board, it only contains 1 in corresponding position of current player and 0 for position of opponent chess stone or blank position.

1	1	1	1	1	1
1	1	1	1	1	1
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Figure 19. Origin Board for Black Stone

7.1.2 State

A state is an Array List that collects 17 Boards to represent current state of game. It works like a vector to identify chess style of two players.

The first 8 Boards is representing for current steps taken by black chess and 8-th Board is the latest step. It is similar for second 8 Boards for white chess. For first 16 boards, it only contains 1 in the position of corresponding chess. For example, in first 8 Boards for black chess, it only shows position for black stone.

One last Board is used to identify current player. If current player is black stone, it only contains -1, otherwise, it only contains 1.

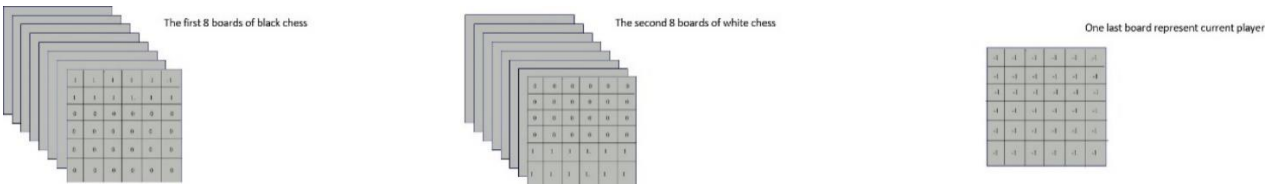

Figure 20. State at Start of Game

7.1.3 Game

Game is an Array List that contains several States from the start to end of game. It simulates whole procedure of chess moving.

7.2 Data Sample

Data is imported from Monte Carlo Tree Search (MCTS) and play a role in neural network part. It contains several Games played by two bots implemented by MCTS. Like figure 3, the whole data file can be divided into sections. Each section contains data of one Game. Sections are separated with two blank lines. In each Game, we can also keep dividing it into States. Each State is composed of 17 Boards and all possible moves in current State.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

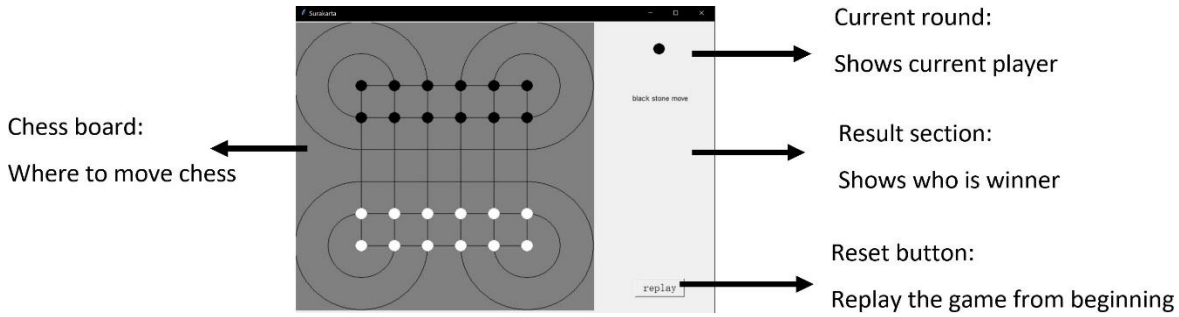

Figure 21. visualization of data file

7.3 Methods to Process Data

A function read_game() with a cursor as input was applied to read one Game from file. Each function call will return a list of States contained in a whole Game.

From the code, a while loop and for loop is applied to achieve the iteration of file. The for loop takes charge of reading one State and iterate 104 times. The integer 104 here is number of valid lines needed to store data of a State. As we have explored previously, a State contains 17 Boards and each Board takes 6 lines in file. Therefore, the number of lines is 17*6 plus 2 extra lines for possible moves equal to 104. While loop is at the outside of for loop to help reading all States in a Game and store it in a list, which will be returned in the end of function.

7.4 Challenge and Solution

One main challenge in processing input is to detect the end of Game in the whole file. As what has mentioned before, Games are separated by 2 consecutive blank lines. Therefore, our first idea is to read two lines in a time and check whether they are blank. If true, break the while loop. However, there will be another problem: where to put the detection part of two lines.

If it was put in the inner for loop, it can not break the outer while loop when there are two consecutive blank lines. If it was put in the outer while loop, the detection part will disturb the inner for loop when we are not at the end of Game, since 104 times of iteration is fixed. There will be one line missing at the beginning of iteration of State if we read two lines after previous iteration to detect end.

Our solution is to attach detection part in the for loop and break it when there are two consecutive lines. In this way, an empty list we be processed and caused IndexError. We can catch the IdexError to break the while loop. By implementing the detection scheme, the function can automatically detect the end of Game and returns a list of all states in the Game.

7.5 GUI Implementation

In order to visualize outcome of neural network part, we also implement a GUI of Surakarta chess with tkinter package in python. The whole board is built on a Canvas object. By compositing different modulus on Canvas, we can achieve lots of functions commonly used in chess games. Chess board sections provides current position of all chess stone, user can interact here to achieve chess moving. Current round section gives notifications of current player. User can only move chess stone corresponding to current player.

Result section will present winner at the end of game. And reset button could reset the whole game. User can also achieve chess moving by modifying 4 global variable to represent move coordinates and then call function judgecoor ().

8. Conclusion

In this project, we combined Monte Carlo Tree Search and Residual Convolutional Neural Network together and get a AlphaGo-like solution for Surakarta chess. On the first half, only Monte Carlo Tree Search is implemented. The chess program written with it can make reasonable decisions. However,

ISSN: 2414-1895 DOI: 10.6919/ICJE.202106_7(6).0064

in the testing process, the program can only maintain a winning rate of 40%. It is not wise enough as an AI currently. On the second half, after using Residual Convolutional Neural Network, program's winning rate increased significantly. This is because in Surakarta, the number of possible choices in each step is similar to Go, which is larger than normal chesses. Therefore, Monte Carlo Tree Search require much more searching time in each step. With limited time slot, the performance of Monte Carlo Tree Search is not as good as in other chess game. In the second part, using Residual Convolutional Neural Network can optimize the simulation step of Monte Carlo Tree Search, reduce the searching time and increase winning rate. More than that, we can extend the idea of combining Monte Carlo Tree Search and Residual Convolutional Neural Network derived from AlphaGo to all situation where the required computation of each step is large.

References

[1] Marsland T A. Computer Chess and Searc [M] S. Shapiro (editor), J. Wiley & Sons, 2nd edition, 1992: 224-241

[2] Plaat A. Research Re: Search and Research[D] Ph.D. thesis, Erasmus University, 1996.

[3] Rivest R.L. Game Tree Searching by MinMax Approximation[J]. Artificial Intelligence, 1998, Vol. 34, No. 1

[4] Berliner H J. An Examination of Brute Force Intelligence[C]. International Joint Conference on Artificial Intelligence, 1981:581-587.

[5] Knuth D. E.,Moore R.W. An analysis of alpha-beta pruning[J]. ICCA Journal 22(3). pp. 123-132. Sep. 1999.

[6] Omid David Tabib, Nathan S. Netanyahu. VERIFIED NULL-MOVE PRUNING[J]. ICGA 2002

[7] Donninger C. Null move and deep search: Selective search heuristics for obtuse chess programs [J] ICCA Journal, 1993; 16(3):137-143.

[8] Bruce Moreland. The Main Transposition Table. 2001

[9] Sen Cao. Improvement of the performance of the alpha-beta pruning algorithm 2012.

[10] S. Gelly. A Contribution to Reinforcement Learning; Application to Computer-Go PhD thesis, Universite Paris-Sud, 2007

[11] Levente Kocsis, Csaba Szepesvári, Jan Willemson. Improved Monte-Carlo Search MTA SZTAKI, Kende u. 13-17, Budapest, Hungary, 2006.

[12] Monte-Carlo Tree Search Algorithm Department of Computing Science, University of Alberta, 2009.

[13] A.L. Brudno. Bounds and valuations for shortening the search of estimates[J]. Problems of Cybernetics. 1963. Vol. 10:225-241.

[14] Structured Parallel Programming for Monte Carlo Tree Search Leiden Centre of Data Science, Leiden University, 2017.Omid David Tabib, Nathan S. Netanyahu. VERIFIED NULL-MOVE PRUNING[J]. ICGA 2002

[15] Lock-free Algorithm for Parallel MCTS Leiden Centre of Data Science, Leiden University, 2018.

[16] Bruce Moreland. The Main Transposition Table. 2001.

[17] David Silver, Julian Schrittwieser*, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, Demis Hassabis. Mastering the Game of Go without Human Knowledge DeepMind, 5 New Street Square, London EC4A 3TW.

[18] Sumit Saha. (2018) A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. Towards Data Sciene.

[19] Jason Brownlee (2019)How Do Convolutional Layers Work in Deep Learning Neural Networks? Deep Learning for Computer Vision.