IJCTA, 9(10), 2016, pp. 4833-4841

Effective Monte-Carlo Tree Search Strategies for Gomoku AI

***Jun Hwan Kang *Hang Joon Kim**

Abstract : The Artificial Intelligence for Game AI has been evolved during decades of years. Many solutions for Game AI suggested and it overcomes many complicated games such as Tic-Tac-Toe, Chess, and Go. After the suggestion of the Monte-Carlo Tree Search, Game AI got significant results in Go. Recent case of Go AI shows that it is almost time to overcome the whole Game AI. But still it needs additional consideration to fulfil the required performance yet. It also has a special objective with constraints and it needs innumerable learning data. So, it has to concentrate on practical and proper ways to improve the Monte-Carlo Tree Search for the Game AI in the deterministic game Gomoku. This game has different strategy with Go but still hard to explore whole game tree and hard to pruning unnecessary cases. At first, we adjust the improved game AI to Gomoku that needs widely different approach ways from Go.

Keyword : Monte-Carlo Tree Search, Gomoku, Game AI, General Game Playing

1. INTRODUCTION

The importance of the Artificial Intelligence Technology is getting higher recently with new algorithms and advanced computing power. Recent days, AI technology shows progressive outcomes that solving the sticking points such as image recognition, voice recognition and natural language process that worth challenging to human brain area. The field of recognition has developed by stochastic algorithms and machine learning methodology. On the other hand, algorithms that handling the practical problems are still hard to solve perfectly with efficient optimum resources. In case of Game AI, the computer beats human champion of Chess with hyper computing power by computing most of possible cases that is called 'brute force' method. And recent Game AI in Go has shown outstanding performance versus high-rank human player by learning numerous data. It makes sense to achieve the special goals, but in resource management perspective, it is reasonable that better approaches are still needed to solving practical problems with optimum resources and constraints. This type of problems in Game AI field are well known as General Game Playing (GGP).

This paper proposes the practical approaches to complement these drawbacks. The Monte-Carlo Tree Search (MCTS) is the most well-known search algorithm for these general problems in Game AI [2], [3]. It is differing from former tree search algorithm as Minimax Tree Search with Alpha-Beta Pruning [4]. MCTS is best-first search, and it is faster than depth-first search trees. But basic form of MCTS has some drawbacks to get perfect performance. To handle this, we choose proper progressive strategies to make it more effective improvements, and suitable game to adjust it. And the chosen game is Gomoku that is similar but different with Go [5]. And it is combined with progressive strategies used in Go program MANGO but uses different heuristic knowledge and roll-out policy [1]. Our research is focusing on what is better way to apply the practical algorithm with no hyper computing or massive learning.

* School of Computer Science and Engineering, Kyungpook National University Daegu 41566, Republic of Korea Corresponding Author E-mail: kimhj@knu.ac.kr

2. BACKGROUND

A. Monte-Carlo Tree Search

Monte-Carlo Tree Search is tree search which based on Monte-Carlo simulation methods. It is best-first search by using results which come from lots of random simulation. Therefore, it can get high accuracy with random sampling and it is faster than any other full-search trees. The MCTS uses fast and simple simulation policy, also called roll-out policy, and minimize time cost that spends for calculating evaluation values [2].

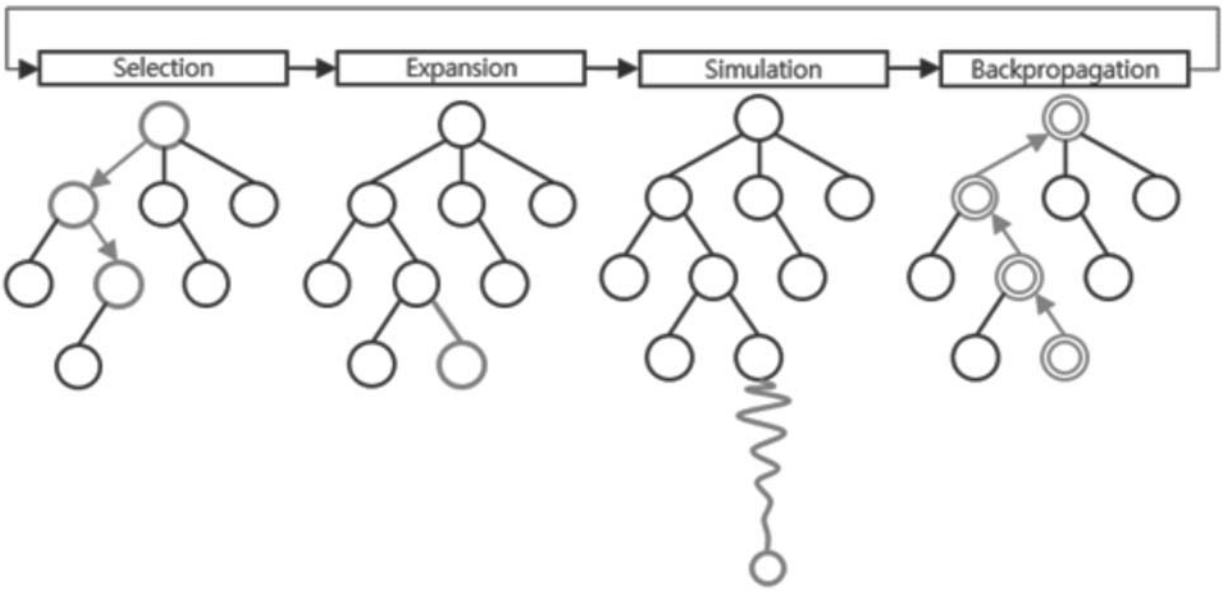

Fig. 1. Structure of Monte-Carlo Tree Search algorithm

The Monte-Carlo Tree Search has 4 phases: 1) Selection, 2) Expansion, 3) Simulation, and 4) Backpropagation. First, in Selection phase, the algorithm selects one existing node that want to expand child nodes. Also it can work with pure random selection policy, but this Flat Monte-Carlo (flat-MC) method can be wrong easily. So, the Upper Confidential Tree (UCT) method was suggested in selection policy which makes proper selection strategy using Upper Confidential Bound (UCB) that have to make balance with exploration and exploitation. The formula (1) is base form of UCB [6], [7].

$$\mathrm{UCB} = v_i + \mathrm{C} \times \sqrt{\frac{\ln \mathrm{N}}{n_i}} \tag{1}$$

Every nodes takes UCB value when the selection algorithm computes it. The variable N is total number of nodes, and n_i is number of child nodes belong to node i.v_i is the value that the total winning rate of node that has all results of child nodes. And coefficient C a constant value that set to $\sqrt{2}$ usually.

The next step, Expansion phase, makes decision that which node is proper to expand the node and make value through the simulation result. The common method which used in flat-MC is that look through the entire possible child nodes and choose one randomly. The Simulation phase is just simulating from the new child node and getting result value that contains just win or lose information. It works well when simulation policy is simple and fast due to the efficiency that can reduce evaluation time and widen the whole tree size. The final phase is Backpropagation. It updates all nodes from expanded node to root node and discovers new best node to make next decision. The child of root node can get measured values that used for what node has best value to choose.

B. Gomoku

The Gomoku is traditional board game originated from Ancient Asia. This game uses same board and objects as another traditional board game Go. But rule is widely different. It uses a square board with m by m array, usually

19 x 19 board. One player takes black stone and he got chance to start with first turn. Another player takes white stone. Each player lay the own stone alternatively until one player gets winning objective. The objective of this game is making a five-in-a-row line vertically, horizontally, or diagonally. It is form that consecutive 5 stones with player's own color. In this paper, we treat the case of over 6 length of line as line of 5. If the board is fully filled with stones, it treats as draw. On the other hand, there is no capture rule compared with Go rule. Once the stone is placed, there are no way to remove or replace it [5].

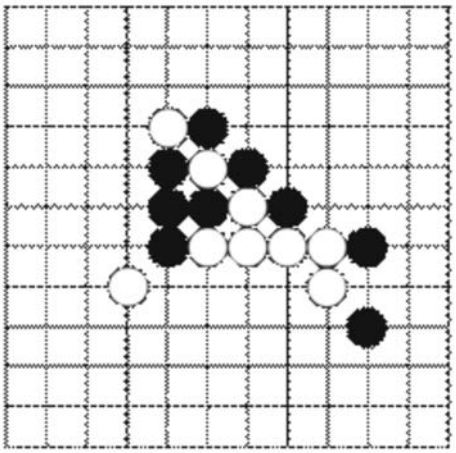

Fig. 2. Example of Gomoku game. The case of white player wins

C. Background knowledge before making Gomoku AI

This game is often referred as advanced tic-tac-toe. But the size of board is larger as more than 20 times. So, the full-search case is innumerable as case of Go. It is easy to approach to this game with MCTS. Go is full playing game that the result of this game depends on the score when it ends. And in case of Go, it is hard to find the best position to get more score when it ends even well-playing human player can't have confidence during most of playing time. Therefore, many advanced methods that based of MCTS with efficient simulation method can be powerful in Go more than any other algorithms [7]. But in Gomoku, MCTS needs a general approach like GGP.

There are three points to construct this AI. First, the way to win this game is focused on short-term object, not long-term score as Go. The objective of Gomoku is making the winning condition like Chess, not like Go. Moreover, it has more crisis condition than Chess dynamically. It means that most of choices are too sensitive to make winning condition [9]. Although you find the key position to absolute win after a bit of turns, it is enough to lose the game if the opponent can make winning condition faster than your turn. In the normal form of MCTS, it is easy to miss this problem [8], [10]. So, the evaluation method with turn order factor or reasonable tree search policy are needed.

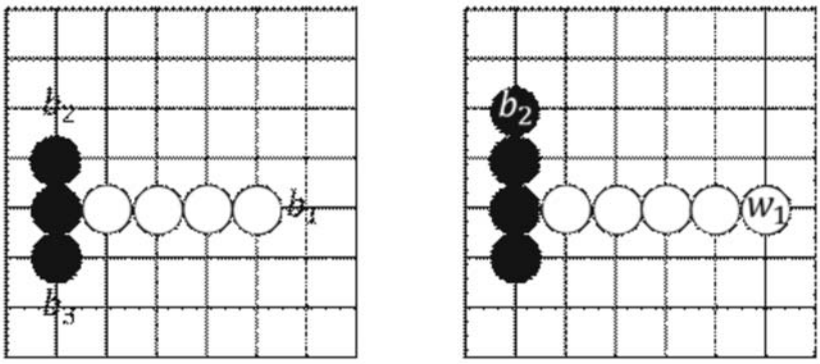

Fig. 3. The case of random playout fails in black player's turn.
MCTS of black player could detect three key chances but missed b_1 and lost the game.

Second, the total case is so enormous as Go in 19 × 19 board but there are few cases to make real moves in Gomoku. Because lots of unnecessary moves are existing in each turn. So, the classical solution via tree search is evaluating these moves and pruning most of moves to compute more cases as Chess AI [5]. But it shows that this

kind of approach needs inefficient resources. Deep Blue, the Chess AI of IBM, uses hyper computing to search most of cases. So, we still use MCTS method and exclude or downsize the evaluation up to limit to maximize the performance of MCTS. Third, the random simulation in simulation phase is not powerful as in case of Go. It is reasonable that the high performance of Go program uses elaborate simulation policy by using heuristics with multi-level neural networks [12]. But because this AI is focusing on adapting to practical conditions with limited computing resources, it is more important that it has to figure out how to make the simulation policy more simple and significant.

In traditional Gomoku, first starting player gets advantage to win this game. In case of Go, the way to make fair rule for this game is modulate the final score. However, Gomoku can't solve this balance problem by handling the score. So, in international Gomoku league, there are lots of artificial additional rules to make fair winning rate [13]. This paper not treats it and solves it by taking fair chance to play first turn for each AI group instead of using these complex rules.

3. STRATEGIES

The advanced strategies for MCTS has been suggested many times, especially the field of Go AI. There are some strategies to transform it to Gomoku AI. First, there are some better strategies for selection phase by using additional value with UCB instead of using basic form of UCB in selection phase [1], [14], [15]. Also we can control the number of child nodes when we select one parent node and expand all possible child nodes. And, to get more accurate simulation values, it needs proper simulation policy that has more reliable than basic form of random policy [17]. In following sections, we'll explain progressive strategies that take soft transitions to be appropriate for Gomoku AI.

A. Progressive bias

The Progressive Bias is heuristic based strategy that can be more accurate and time-expensive than UCB in selection phase. The object of the progressive bias is getting bias in selection phase to choose more significant node to simulate with domain knowledge [1]. It consists of basic form of UCB and add some heuristic values to generate subtle bias. The formula (2) shows it.

$$\mathrm{UCB} = v_i + \mathrm{C} \times \sqrt{\frac{\mathrm{In\ N}}{n_i}} + \frac{\mathrm{H}_i}{n_i + 1} \tag{2}$$

It makes nodes which closed to beginning state to depend on heuristic value H_i rather than UCB value. It makes effect that avoiding the dependency made by few simulated games when UCB value has less confidence with few random sampled nodes. When the number of node has progressed more and more, the effect of heuristic value is decreased by O $(1/n_i)$because the value of n_i is increased. On the other hand, the dependency of exploration-exploitation value in UCB is also decreased by O $(\sqrt{\mathrm{In}\ n_p / n_i})$ but less effective than heuristic value. The origin form of progressive bias uses fixed number M to replace UCB value when it just start with $n_i = 0$. This approach can make algorithm to select every child node at least once and give preference to the child node with high heuristic value [1]. However, there is no need to consider every child in Gomoku that differ from the case of Go. So, the fixed number M is set to zero and progressive bias can consider the limited number of child that has heuristic value at least over zero when $n_i = 0$. If there is no reference information in the board when it starts with first turn, the heuristic function indicates the prospective position that used in common starting point of game.

B. Progressive Unpruning with heuristic values

When MCTS algorithm perform without pruning, it used to search ineffective position because of limited available time and expanding the tree without checking the utility of nodes. The Progressive Unpruning can resolve it by control the proportion between preferred nodes and not preferred nodes. The goal of strategy is that control the branching factor to expand not too much nodes that cause bad efficiency. When the number of simulation of a parent node n_p reaches threshold T, node p stop expanding child nodes randomly and prunes the most of the child

except k_{init} children that has highest heuristic values in ascending order. To make heuristic values, the heuristic function is called and it causes computing time cost. After that, when the sum of simulation in node p exceeds $A \times B^{k-k_{init}}$, the pruned node is unpruned progressively by according to order of k [1]. The value of A, B, k was set to 50, 1.3, 5 empirically by following the base form.

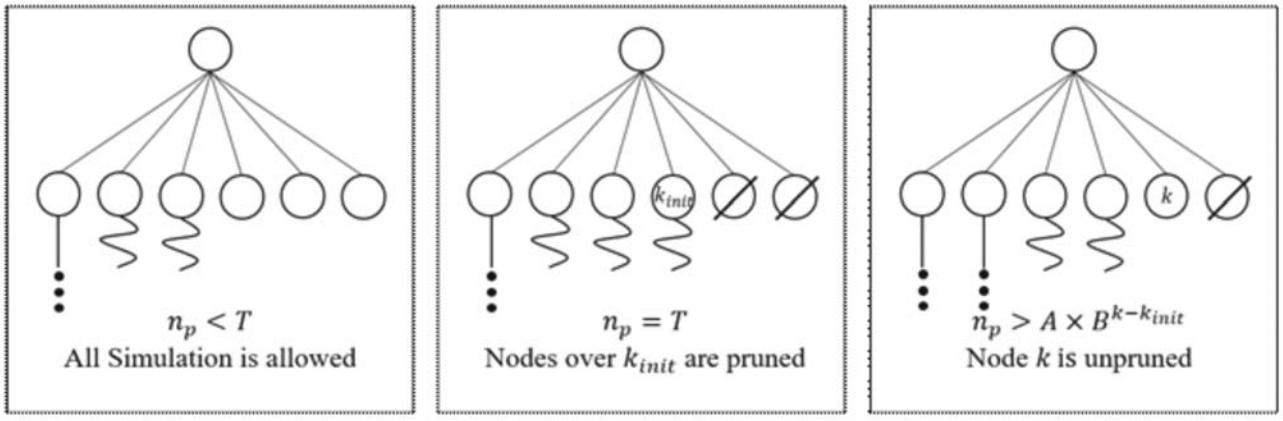

Fig. 4. Progressive unpruning

We add one transition on it to make proper approach for Gomoku. The existing progressive unpruning for Go has to compute for all moves with complicated estimate function for Go. But the case of Gomoku, we don't need to estimate whole moves because the most of moves are unnecessary to compute and have zero value. As we explained at next section, the heuristic function for Gomoku consists of simplified function. So it doesn't cause time-expensive cost than in case of Go. Therefore, this AI calls the heuristic function instantly when the parent node has been selected first time and it has no child to expand yet. Instead of scanning the all moves at first time, it looks over limited number of nodes to expand that have heuristic values at least over zero. It can consider just possible moves rather than consider all nodes randomly as case of Go, so it can reduce useless compute time for unnecessary cases.

C. Heuristic knowledge

The heuristic knowledge is required to define heuristic value H_i in previous two strategies. To reduce most of computing cost in this AI, the efficient heuristic algorithm is required to construct simple and practical form [19]. It is designed a simple heuristic knowledge for Gomoku which different from Go. In case of Go, it is hard to determine the advantage for each moves and evaluate the heuristic values. Most of all, using the pattern analysis in game of Go is necessary, and it often calls heavy computing [1]. So, this can be complicated and needs time-expensive computing easily. But in case of Gomoku, it doesn't need such sophisticated evaluation to determine the current factor to get higher final score than opponent. It just needs final results and better ways to win the game. Therefore, it is possible that set the heuristic function as simply estimated values just because it is used to find the way that leads to win regardless of complexity. This heuristic function is simplified as the estimation formula by using sum of line length that indicates how closer to winning condition, the straight line of 5. The formula (3) shows it.

$$H_i = \sum_{l} \left\{ (L_{\text{open}})^2 + \left(\frac{L_{hclosed}}{2} \right)^2 \right\} \tag{3}$$

The variable L_{open} is length of line that has no opponent's stone at two ends of line and it can be widen to the straight line of 5. $L_{h\text{closed}}$ is half-closed line that has one blocked end of line. The outcome of this formula can be represented by following threat table in Table I. Each type of line threat is classified by common threat evaluation method in Gomoku AI [9].

And we should not forget that the way to interrupt the opponent's winning strategy is also important as our player. So this function has the opponent's sum values, and both sum values are integrated as just one heuristic value table. It still spends time-cost but less than pattern analysis.

Table 1. Threat table with heuristic value

Type of Threat	*Heuristic Value*
Half-Closed 2	1
Half-Closed 3	2.25
Open 2	4
Half-Closed 4	4
Open 3	9
Open 4	16
Closed 5 Half-Closed 5 Open 5	25

The Table II shows the whole combination to check the real case that heuristic values can make. For just one move, it can make maximum 4 lines with vertical, horizontal, and two ways of diagonal cases. Table 2 shows it. For example, three of open 2 can make 12 point that higher than one of open 3, but it isn't critical because sometimes three of open 2 is better than one of open 3. The problem is case of double threat by two of half-closed 4, and single open 4 that can almost finish the game. Because the open 2 have same value as half-closed 4, the threat priority by heuristic value has confusing. And the single open 4 gets 16 point, but two of open 2 and one of open 3 get 17 point, so the selection algorithm select the latter case first. However, two problem cases have just less than 1 point of gap and get the priority easily with high heuristic value. So, the select algorithm can select and simulate both cases and the gap of its child node is widen because one straight line of 5 has created by one of open 4. Thus it covers this a bit of problem.

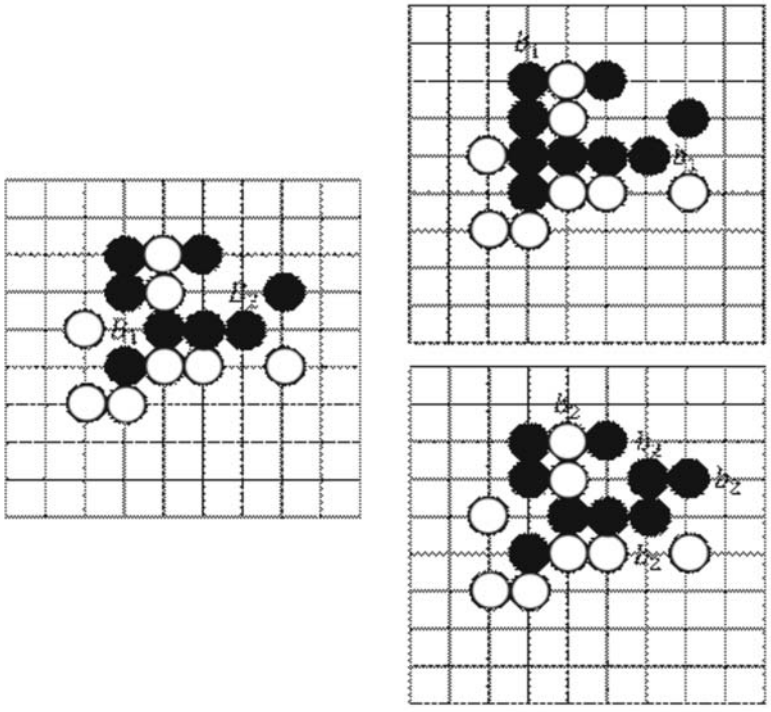

Fig. 5. The example of multiple threat problem.

In first step, B_1 gets 8 and B_2 gets 12. But next step, each of two b_1 can get 25 and each of all b_2 can get 9

Table 2. The threat case of multiple threats

Type of Threat	*Heuristic Value*
Half-Closed 4, Half-Closed 4	8
Open 2, Open 2, Open2	12
Open 2, Half-Closed 4, Half-Closed 4	12
Open 3, Half-Closed 4	13
Open 2, Open 2, Open2, Open 2	16
Open 2, Open 2, Open 3	17
Open 3, Open 3	18
Open 3, Open 4 Closed 5, Half-Closed 5, Open 5	25

D. Sequence-like Simulation with limited depth

The sequence-like simulation which used in Monte-Carlo Go is based on pattern knowledge. It depends on pattern data to solve local problems with sub-optimal simulation [17]. It is a kind of simulation policy that simulate with following local patterns that already learned by pattern knowledge. And it should avoid the critical shortcomings that all simulated results are confined to local patterns. It is reasonable that the simulation policy in Go become complicated more and more to have higher significant simulated values. On the other hand, in case of Gomoku, the simulation policy can be defined by local knowledge. It has to focus on local competition, not on probability that exists on the distant case path. The global winning strategy should be found by expanding progress of whole search tree, not by roll-out progress in temporary simulations. The way to absolute winning, also called double threat in Gomoku, is best way to beat the game. The local competition with simple policy is enough to search this double threat. But if just one simulation tried at one node, it is easy to miss double threat. And it returns not enough result in backpropagation phase. So this node has lower value even though it has the hidden chance to make double threat.

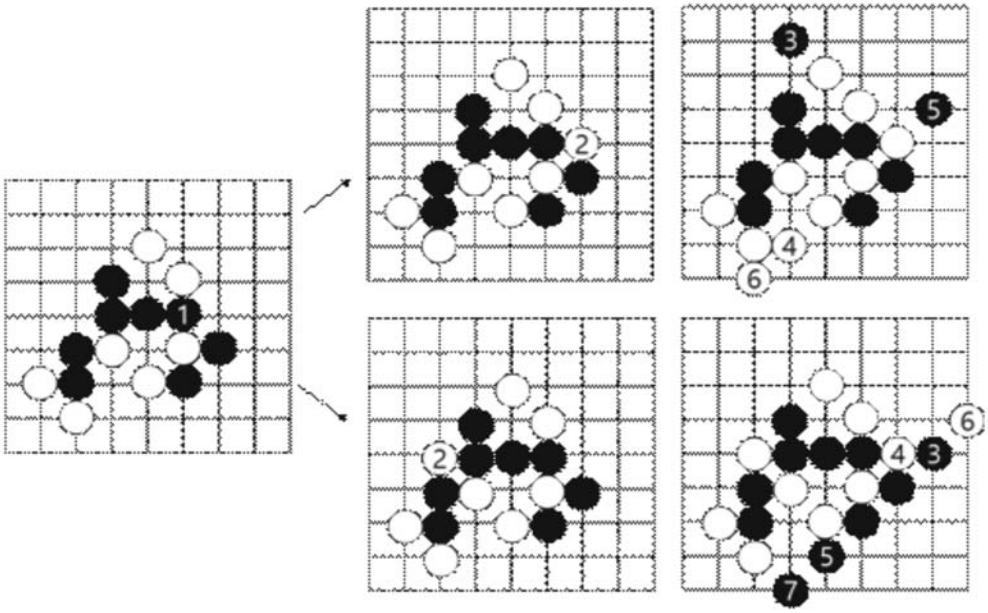

Fig. 6. Different simulation cases show missing the hidden chance and returning opposite results

According to this case, the overall simulated results should have practicality that simulation results with high accuracy quality, not just mass results of procedural progress. So, the simulation policy is designed as normal AI that follows a quick response to instant situation. It seems to common tree search with depth = 1, like 'do attack' when it can attack except when it has to defend. And this simulates all cases when it can make just one threat to find hidden chance to make double threat. It returns +1 when it found double threat to win, and returns -1when it loose. At this point, this algorithm can cause time-expensive simulation by computing all cases of possibilities with unlimited number of game playing. But because we already know that the winning cases found by simulation can often useless when its simulation depth value goes deeper, so the proper depth value is required that is practical and shows high efficiency between winning rate and time-cost empirically. When the simulation result failed to distinguish win or lose, it treats as draw and returns zero. But most of simulated games end up with draw with limited depth, especially early time of game. To complement this problem, we use the heuristic function with the value but less than 1. If there is no value to return simulated result, the heuristic value H_i can replace v_i value. The heuristic value should be normalized to set the value under the variable range of H_i getting equal to v_i in progressive bias formula (2). It can lead the moves to proper way by depending on heuristic knowledge despite the most of simulation results returns draw with limited search depth.

As it runs lots of simulation with the heuristics and confined simulation policy, it is easy to find that the most of simulation has duplicated one and it brings unnecessary time cost to compute duplicated simulation. So, to treat this problem, this AI chooses the time efficiency instead of saving memory space. The simulation progress isn't thrown away after returning result and it can be attached to main tree as nodes with informal condition like invisible node to whole tree. When the MCTS expands one node and uses existing simulated path, the simulation algorithm reuses it and continues the simulation at the last existing node. This reusing method is possible to simple game that can make short winning path by using shallow depth simulation.

4. OPTIMIZATION

There are some important values which should be handled to control the performance. This AI needs three customized values treated in previous sections. The threshold T in progressive unpruning, proper settings of heuristic value, and optimal value of simulation depth to stop the simulation and to return the result on its way to end game.

First, the values of progressive unpruning are started with basic value settings of progressive unpruning. Threshold Tvalue is set to 50 andthe value of A, B, k in $A \times B^{k-k_{init}}$ was set to 50, 1.3, 5. And to find optimal value setting, it should be changed empirically.

The depth value for sequence-like simulation can exploited by competition with different depth simulations. If it has shallow depth, it can easily expand whole game tree but the most of simulation results return the default value as following the heuristics. While the depth value goes deep, it is hard to expand game tree and it spends lots of time to compute simulations. But it takes higher possibilities to find useful simulation results than the former case. The optimal depth value should be exploited concurrently with former other values. Because the optimal computing proportion of selection phase and simulation phase to make the best performance is hard to figure out. So, the optimal value case can be developed empirically and the test process should be taken with gradual different settings.

5. CONCLUSION

This game AI for Gomoku is constructed by applying the progressive MCTS strategies for game of Go. But in adaptation process, the MCTS strategies for Go is not enough to make the Gomoku AI and it needs customizing. By applying soft transition in each strategy, it becomes hardly different from Go AI. But it can reduce the time cost of algorithm by changing heuristic functions and concentrating on win or lose like most of game AI, not on the score based game as in case of Go. So, it can close to solving general problems as GGP by reducing the computing cost. Moreover, the efficiency of this AI can get higher by improving heuristic functions that use prepared knowledge data for each target case.

6. REFERENCES

1. G. M. J.-B. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy, "Progressive Strategies for Monte-Carlo Tree Search," New Mathematics and Natural Computation, vol. 4, no. 3, pp. 343–357, 2008.
2. R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in 5th International Conference on Computers and Games (CG 2006). Revised Papers, ser. Lecture Notes in Computer Science, H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds., vol. 4630. Springer, 2007, pp. 72–83.
3. C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," IEEE Trans. Comput. Intell. AI Games, vol. 4, no. 1, pp. 1-43, Mar 2012.
4. H. Baier and M. H. M. Winands. Monte-Carlo tree search and minimax hybrids. In IEEE Conference on Computational Intelligence and Games (CIG), pp. 129–136, 2013.
5. Brumagen, Kegan, Jeffrey Emenheiser, and Seung-youn Choi. "Constructing an AI Gomoku Game Player." Journal of the Pennsylvania Governor's School for the Sciences, pp. 129-141, 2009.
6. L. Kocsis and C. Szepesv´ari, "Bandit Based Monte-Carlo Planning," in 17th European Conference on Machine Learning, ECML 2006, ser. Lecture Notes in Computer Science, J. F¨urnkranz, T. Scheffer, and M. Spiliopoulou, Eds., vol. 4212. Springer, 2006, pp. 282–293.
7. Chaslot, G. M. J.-B., 2010. Monte-Carlo Tree Search. Ph.D. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands.
8. C. Browne, "The dangers of random playouts," ICGA Journal, vol. 34, no. 1, pp. 25–26, 2011.
9. K. L. Tan, C. H. Tan, K. C. Tan and A. Tay, "Adaptive game AI for Gomoku", Proc. 4th Conf. Autonom. Robots Agents, pp. 507-512, 2009.
10. C. Browne, "A problem case for UCT," IEEE Transactions on Computational Intelligence and AI in Games, vol. 5, no. 1, pp. 69–74, 2013.
11. M. Campbell, A. J. Hoane, Jr., and F.-H. Hsu, "Deep blue," Artificial Intelligence, vol. 134, no. 1–2, pp. 57–83, 2002.
12. David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. Nature, (529):484–489, 2016.
13. Wágner, János, and István Virág. "Solving Renju," ICGA journal, vol. 24, no. 1, pp. 30-35,2001.
14. HS Park, HT Kim, KJ Kim, "GreedyUCB1 based Monte-Carlo Tree Search for General Video Game Playing Artificial Intelligence," in KIISE Transactions on Computing Practices, vol. 21, no. 8, pp. 572-577, 2015.
15. T. Vodopivec and B. Ster, "Enhancing upper confidence bounds for trees with temporal difference values," in Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG), pp. 1–8, August 2014.
16. H. Finnsson, "Simulation-Based General Game Playing," Ph.D. dissertation, Reykjav´ýk University, Reykjav´ýk, Iceland, 2012.
17. Wang, Yizao, and Sylvain Gelly. "Modifications of UCT and sequence-like simulations for Monte-Carlo Go." Computational Intelligence in Games (CIG), pp 175-182, 2007.
18. A. Yee, A. and M. Alvarado, "Pattern Recognition and Monte-CarloTreeSearch for Go Gaming Better Automation," in Proc. Advances inArtificial Intelligence, Springer Berlin Heidelberg, pp. 11–20, 2012.
19. Jegadeshwari, S., and D. Jaisree. "Heuristic Algorithm for Constrained 3D Container Loading Problem: A Genetic Approach,"International Journal of Computing Algorithm, vol. 3, no. 1, pp. 1016–1020, 2014.
20. M. Genesereth, N. Love, and B. Pell, "General Game Playing: Overview of the AAAI Competition," AI Mag., vol. 26, no. 2, pp. 62-72, Mar. 2005.