2009 Third International Symposium on Intelligent Information Technology Application

Backpropagation Modification in Monte-Carlo Game Tree Search

Fan Xie
Jiu-Ding Computer Go Research Institute
Beijing University of Post and Telecommunication
Beijing, China
xiefan198877@gmail.com

Zhiqing Liu
Jiu-Ding Computer Go Research Institute
Beijing University of Post and Telecommunication
Beijing, China
zhiqing.liu@gmail.com

***Abstract*—The Algorithm UCT, proposed by Kocsys et al.[3], which apply multi-armed bandit problem into the tree-structured search space, achieves some remarkable success in some challenging fields[2]. For UCT algorithm, Monte-Carlo simulations are performed with the guidance of UCB1 formula, which are averaged to evaluate a specified action. We observe that, as more simulations are performed , later ones usually lead to more accurate results, partly because the level of the search used in the later simulation is deeper and partly because more results are available to direct subsequent simulations. This paper presents a new method to improve the performance of UCT algorithm by increasing the feedback value of the later simulations. And the experimental results in the classical game Go show that our approach increases the performance of Monte-Carlo simulations significantly when exponential models are used.**

***Keywords*-machine learning; Monte-Carlo tree search; weight factor**

I. INTRODUCTION

The problem of finding a near optimal action in large state-space Markovian Decision Problems (MDPs) is a famous problem in machine learning. There are a lot solutions to this problem, however, only a few viable approaches work, for example the sampling based look ahead search proposed by Kearns et al.[1]. However, according to this method, the amount of work needed to compute just a single near-optional action from a given state is overwhelmingly large. Another solution comes to Monte-Carlo Planning algorithm, whose successful application is in game-tree search. Over the years, Monte-Carlo simulations based search algorithms have been used successfully in many non-determinist and imperfect information games[4,5], and also applied to some challenging games with large branching factors, such as Go. Now, UCT algorithm has been used to the application of many computer games, and has achieved a remarkable success[2].

We observe that Monte-Carlo simulations are largely performed sequentially, which is a natural side-effect of the current von-Neumann computing model, even in a parallel or distributed computing environment. Furthermore, as more simulations are performed, later simulations usually lead to more accurate results than early ones. This is partly because the level of game search trees used in the later simulations is deeper, and partly because more previous results are available to direct subsequent game tree search simulations. Therefore, all simulations are not equal, and this fact shall be reflected in the Monte-Carlo tree search process in order to improve its effectiveness.

Based on this observation, we aim to explore a proper way to adjust feedback values of Monte-Carlo simulations in order to improve their effectiveness. This paper presents FAP (Feedback Adjustment Policy), an extension of UCT algorithm, which would increase feedback values of later Monte-Carlo simulations progressively according to results of previous ones. This algorithm introduces a weight factor for each simulation such that the weight factor for a later simulation can be larger than a previous one. This algorithm also specifies the way in which the weight factors of various simulations are precisely determined.

The main idea of our approach is to regard early simulations more as knowledge to direct the later simulations and enhance effectiveness of the later simulations, more accurate simulations, in evaluating a specified move. In order to motivate our approach, let consider problems with application of UCT in non-determinist and imperfect information games, such as poker, Scrabble and Go. If a good move M unfortunately is calculated as a bad move in the first half of simulations, as the search processes, UCT algorithm detects that it is a good move during the second half simulations. It might result in that this good move M still be abandoned for the results of all simulations are averaged to evaluate a specified move. Obviously, we should prefer the result of the later simulations for they are more accurate. If, we could give later simulations higher weights, for example regarding one later simulation as two independent simulations, the good move M would not be ignored, which might need more simulations to reach. Hence, if the later simulations could be weighted more, performance improvement could be expected.

Only a few related works exist. The most noticeable and related is the AMAF (All Moves As First) method introduced by Gelly[7] and Progressive Strategies proposed by H.J. van den Herik[8]. Both AMAF and Progress strategies improve the effectiveness of Monte-Carlo simulations by exploring within the simulations. However, FAP differs

978-0-7695-3859-4/09 $26.00 © 2009 IEEE
DOI 10.1109/IITA.2009.331

significantly from these two in the following aspects: 1)FAP does not use any specific domain knowledge and is thus a general extension of UCT. 2) AMAF and progressive strategies help UCT overcome its slow start but becomes more and more insignificant as more simulations are performed; FAP has little effect in the beginning of simulations as its effectiveness becomes more evident when more simulations are performed. This is because early simulations are often biased and as they are of low accuracy and that later simulations are meaningful and shall be heavily weighted. 3) AMAF and progressive strategies are mainly performed in the selection step of MCTS(presented in section 2), however, FAP is mainly performed in the backpropagation step. Despite these differences, these two techniques and FAP are rather complimentary and can be used together.

The rests of this paper are organized as follows: In Section 2 we introduce the structure of Monte-Carlo Tree Search. Section 3 presents the concept of weight factor for Monte-Carlo simulations and an extension of the UCT algorithm to incorporate the weight factor in simulation feedback update. Section 4 discusses a number policies of specifying the simulation weight factor. The best policy as well as its parameters is determined experimentally in Section 5. This paper is concluded in Section 6 with a summary of results and a discussion of possible future works.

II. Monte-Carlo Tree Search

A. Bandit problem and UCT algorithm

The UCT algorithm, proposed by [5], is based on the K-armed bandit problem, which is a simple machine learning problem referring to a traditional slot machine but with more than one arm. In this problem, each arm would give a reward when played, and the result would be influenced by the distribution associated to that arm. The object of the player in this problem is to maximize the rewards through a lot of plays. And it is presumed that the player has no initial knowledge about that game. However, as the plays process, the player could focus on the most rewarding arms.

The questions related to bandit problems come to the well-known exploitation-exploration dilemma in reinforcement learning, which means the difficult of balance between exploring to gain more information and taking advantage of information gained so far. Algorithm UCB1, presented in Fig. 1, whose finite-time regret is studied in details by[1] is a simple, yet attractive algorithm that succeeds in resolving the exploration-exploitation tradeoff.

UCT algorithm is a minimax tree search with UCB1 formula. This algorithm regards each node as an independent bandit, and its child-nodes as independent arms. Different from the description in the arm bandit problems, nodes are not played once iteratively. Instead, it plays a sequences of bandits from the root to the leaf nodes. In other words, in UCT all action selections of the every internal nodes are regards as separated multi-armed bandits. And the arms correspond to actions from the state represented by the internal nodes.

- **Initialization:** Play each machine once
- **Loop:** Play machine j that maximizes $\overline{x_j} + \sqrt{\frac{2 \ln n}{n_j}}$, where $\overline{x_j}$ is the average reward obtained from machine j, n_j is the number of times machine j has been played so far, and n is the overall number of plays done so far [7].

Figure 1. the definition of UCB1 algorithm

B. Monte-Carlo Tree Search Structure

The Monte-Carlo Tree Search is based on the main idea of UCT algorithm, a best-rst search algorithm which does not need game-dependent heuristic knowledge. Monte-Carlo Tree Search mainly consists of four different steps, introduced by H.J. van den Herik[8]. (1) The tree is traversed from the root node to a leaf node L, using a selection strategy. (2) An expansion strategy is called to store one (or more) children of L in the tree. (3) A simulation strategy plays moves in self-play until the end of the game is reached. (4) the results of the simulations are backpropagated in the tree according to a backpropagation strategy, which is the main focus of our approach.

III. Weight Factor of Monte-Carlo Simulations

In UCT algorithm, Monte-Carlo simulations are performed when a leaf node is reached in search. The simulations play from the corresponding state of node until an end of the game. And the results of the simulations will be updated back to the search tree subsequently. However, as we have mentioned in the introduction, all simulations are not equal. Different simulations in different periods, which might have different accuracy because the level of the game tree used in the simulations is different and the number of results that could be used to direct next simulations is different also, should be weighted differently. If we can order simulations sequentially by the time in which they are performed, later simulations, having the following characteristics, tend to be more accurate than earlier ones:

1) Starting from a deeper node in the search tree
2) Having fewer simulation steps
3) More importantly, more knowledge being accumulated, reflected by the tree search process

Therefore, we have to distinguish different simulations to reflect their different accuracy. This is naturally done by introducing a weight factor that is used to reflect the relative significance of different simulations. This weight factor is

used in the UCT algorithm to determine how the simulation results are updated in the search tree to be able to reflect the accuracy of different simulations.

In our approach, the weight factor of one simulation decides the feedback value of this playoff. The way we change the feedbacks of simulations is that we regard a simulation with a weight factor K as K different simulations. Our approach, different from the normal implement of UCT algorithm, in which the one simulation give one result, just gets several same results from one simulation. For example, a simulation with a weight factor 8, when it is done, a result would be returned from the evaluator, however, this simulation would be regarded as 8 simulations with a same result.

IV. Feedback Adjustment Policy

In the last section, we have extended the UCT algorithm to incorporate weight factor for differentiating simulations with different accuracy. In this section, we will explore several policies for specifying the weight factor for a specific simulation. We shall refer to these policies as Feedback Adjustment Policies, or FAP in short. The following two issues must be fully addressed in order to specify a FAP:

1) **Partitioning simulations into segments**. All simulations must be partitioned into segments and all simulations within the same segment have the same weight factor. This is because it is impossible to assign a floating number in the update process of Monte-Carlo tree search so that the weight factor must be a fixed point integer number. (if one simulation has a weight factor K, we just regard one simulation as K simulations)
2) **Assign weight factors for each simulation segments**. In our approach, all simulations within the same segments will share the same weight factor. We have developed some schemes for specifying a weight factor for each simulation segment.

A. Simulation Partition

Our goal here is to divide simulations to several segments based on their order in which they are performed, where the weight factor of each of the simulations in the segment should be the same segment. Additionally weight factors of simulations from different segments would be different, with the weight factor of later simulations larger than that of earlier ones.

We have designed two schemes for this purpose: We shall refer to the first scheme as equal partition, in which all simulations are divided into K segments of equal size, where K is a parameter that will be determined experimentally in the next section. We shall refer to the second scheme as exponential partition, in which all simulations are divided into K segments by an exponential function, where the nth segment consists of the simulations whose order is between

Figure 2. Two schemes for simulation partition. The top is the equal partition scheme where the bottom is the exponential partition scheme. The parameter K is 5 in both cases.

$M(2^{n-1}/2^K)$ and $M(2^n/2^K)$, assuming that M is the total number of simulations in consideration. These two schemes are shown in Fig. 2.

B. Weight Factor Specification

This subsection presents schemes to specify a weight factor for simulations in a segment. In a similar way in which the schemes for simulation partition are designed, we have designed two schemes for weight factor specification: We shall refer to the first weight factor specification scheme as Linear Weight, in which weight factors for segments in the time order will grow linearly. In other words, the weight factor of simulations in the nth segment is n. We shall refer to the second scheme as Exponential Weight, in which the weight factor for simulations in the nth segment is 2^{n-1}. While both schemes assign larger weight factors to later simulations, Exponential weight is more aggressive to take advantages of later simulations.

V. Experiments with Random Game Tree

Monte-Carlo simulation based search algorithms have been used successfully in many non-deterministic and imperfect information games, including backgammon [9], poker [5] and Scrabble [10].Computer game playing is one of the core of artificial intelligence (AI) with computer Go as one of its grand challenges[11]. Recently, the application of UCT in the more challenging game Go achieves a remarkable success[2]. A P-game tree is a minimax tree that is meant to model games where at the end of the game the winner is decided by a global evaluation of the board position where some counting method is employed (examples of such games include Go, Amazons and Clobber). Accordingly, rewards are only associated with transitions to terminal states. We have performed several experiments for comparison between the performance of UCT algorithm with FAP and the original algorithm on the game Go.

A. Comparison of Different Combinations

To test the performance of the different combinations, they will play in 9X9 board against a default engine of only the UCT search tree and a random simulation policy, which have no FAP. There are four combination shown below, which are all using a random simulation policy but with different feedback adjustment algorithm. To simplify the expression, we regard AB as the two simulation classification formulas and XY as the two weight adjustment formula:

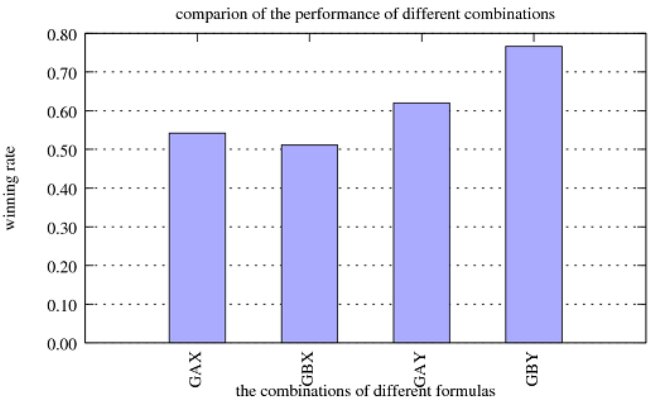

Figure 3. This is the experiment result of the four version TestGo playing against the default engine of only UCT search tree and a random simulation policy with the default K value 5 and the upper limit on the number of simulations 5000 per move. comparison of the 4 combinations

1) GAX: dividing simulations into equivalent segments and increasing the feedback of Nth segment to N
2) GBX: dividing simulations according to exponential function and increasing the feedback of Nth segment to N
3) GAY: dividing simulations into equivalent segments and increasing the feedback of Nth segment to 2^{N-1}
4) GBY: dividing simulations according to exponential function and increasing the feedback of Nth segment to 2^{N-1}

Because the Feedback Adjustment Policy do not make any modification in the UCT code, we can apply it easily into the original UCT code , where the results are updated into the search tree. Though there need to be a calculation on the weight, the times of this calculation is as much as the simulation number and so fast that make no extra time needed. We combine the FAP code into the original code of UCT introduced by Gelly [6].

To test the performance of the different combinations, each version played at least 1000 9X9 games against the default engine said in the previous paragraph with a default K value 5 and the upper limit on the number of simulations 5000 per move. Each program plays Black half of the games and White for the other half. Fig. 3 show the performance of different versions.

From the Fig. 3, we can see that the different combinations influence the winning rate significantly. And the policy, which classify simulations and adjust weight according the exponential function, improves the winning rate against the default random engine into 76.6% when the constant value K = 5. Without any speeding consuming, this approach increases the winning rate by over 26.6%.

VI. Conclusion

In this article we introduce a new extension of UCT algorithm, FAP, that applies the weight factor to the MCTS to enhance the effectiveness of the more accurate latter simulations. The performance of the FAP was tested experimentally in the random game tree of the classical game Go. In the experiments, we have found that a significant improvement when exponential model is applied in FAP is investigated, and the extension of UCT performed much better than the original algorithm.

The main contributions of this paper could be summarized into these three points: (1) the first is that this paper introduces the weight factor into UCT algorithm; (2) the second is that this paper extended the UCT algorithm with weight factors; (3) at last, this paper represents a new approach called Feedback Adjustment Policy which could be easily combined with MCTS.This paper is only an early work towards finding the most suitable mathematical model for Feedback Adjust Policy. More theoretical analysis is need and A more general policy in adjusting the weight factor of MC simulations would be investigated in the future.

References

[1] M. Kearns, Y. Mansour, and A.Y. Ng: A sparse sampling algorithm for near optimal planning in large Markovian decision processes. In Proceedings of IJCAI'99, pages 1324-1331, 1999.

[2] Gelly,S.,Wang,Y.: Exploration exploitation in go: UCT for Monte-Carlo go. In:NIPS-2006:On-line trading of Exploration and Exploitation Workshop, Whistler Canada (2006)

[3] Kocsis, L., Szepesvari, C.: Bandit Based Monte-Carlo Planning. In: 15th European Conference n Machine Learning (ECML), pages 282 293, 2006

[4] A.G. Barto, S.J. Bradtke, and S.P. Singh. Real-time learning and control using asynchronous dynamic programming. Technical report 91-57, Computer Science Department, University of Massachusetts, 1991.

[5] D. Billings, A. Davidson, J. Schaeer, and D. Szafron. The challenge of poker. Articial Intelligence, 134:201-240, 2002.

[6] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite time analysis of the multiarmed bandit problem. Machine Learning, 47(2-3):235-256, 2002.

[7] Gelly, S., Silver, D.: Combining Online and Offine Knowledge in UCT. In: Ghahramani, Z.(ed.)the International Conference of Machine Learning (ICML 2007), pp. 273C280 (2007)

[8] H.J. van den Herik, and B. Bouzy. Progressive strategies for Monte-Carlo Tree Search. New Mathematics and Natural Computation, 4(3), 2008.

[9] G. Tesauro and G.R. Galperin. On-line policy improvement using Monte-Carlo search. In M.C. Mozer, M.I. Jordan, and T. Petsche, editors, NIPS 9, pages 1068-1074, 1997.

[10] B. Sheppard. World-championship-caliber Scrabble. Artificial Intelligence, 134(1-2):241-275, 2002.

[11] M. Müller, Computer Go, Artificial Intelligence, 134, 2002, pp. 145-179.