A Persistent Game Graph in Computer GO

Jing Huang and Zhiqing Liu
BUPT-JD Research Institute of Computer GO
School of Software Engineering
Beijing University of Posts and Telecommunications
Beijing, China 100876
Email: flyanjj@gmail.com, zhiqing.liu@gmail.com

***Abstract*—Internal states in a computer GO program are typically organized as a game tree. While simple and convenient, the game tree organization may result in more tree nodes than actual internal states because duplicated tree nodes exist to represent the same gaming state. Modern computer GO programs use the UCT algorithm to conduct game tree search. Duplicated nodes in the game tree representation would make the UCT algorithm less effective. This paper proposes an approach to represent gaming internal states as a directed graph efficiently, and presents a modification of the UCT algorithm to work on the directed graph representation. Furthermore, this paper describes an approach to make the directed game graph persistent so that historical data can be reused to improve the accuracy of the UCT algorithm.**

***Index Terms*—UCT algorithm; Game Tree; Graph;State**

I. Introduction

Computer GO is computer architecture encompassing hardware and software capable of playing GO autonomously without human guidance. Computer GO is an active research area in Artificial Intelligence that aims to create machine intelligence that perceives its environment and takes actions that maximize its chances of success in the game of GO. A kind of competitive activities, the game of GO is a two-person information symmetrical games, in which two players confront each other; each takes turns to walk the pieces; and the both sides not only know each other's previous moves, but also can estimate options the other can make in the future. And the result of the game can be that one person wins and the other loses, or the game go to the draw. A two-person information symmetrical game such as GO is typically represented by a special "And-Or Tree" known as the game tree. Near optimal solutions of such a game are usually computed by conducting a search on its game tree.

In modern computer GO, the search on the game tree is conducted following a novel tree search algorithm called Monte-Carlo Tree Search (MCTS)[1]. MCTS is an extension of a numerical method known as Monte Carlo planning to solve a Markovian Decision Process (MDP)[2] numerically, in which GO problems are represented as a MDP and off-line GO knowledge is heavily employed in on-line Monte Carlo simulation on the MDP. The efficiency of the on-line Monte Carlo simulation has been greatly improved with the use of the UCT algorithm[3].

Even though MCTS has improved game tree search of computer GO significantly, it still has a number of limitations. One of the primary limitation of MCTS is as follows: its performance is severely downgraded when working on a MDP whose number of states is very large as in the case of the game of GO. However, a number of game tree nodes are the same and duplicated representing the same game state reached from different paths. In order to address this problem, this paper proposes a method in which game trees are organized as game graphs such that duplicated game tree nodes are combined into a single game graph node, resulting fewer MDP nodes. This paper also enhances the traditional UCT algorithm such that it works also on directed graphs such that MCTS can be extended as Monte Carlo Graph Search (MCGS) working on directed game graphs. Our initial results indicate that the proposed approach is promising.

The rest of the paper is organized as follows: Section 2 discusses formally both game tree and game graph representations. Section 3 extends the UCT algorithm such that it can work on directed graphs.Section 4 presents an efficient game graph representation in computer such that game graphs can used with little overhead. Section 5 describes an approach to make game graph persistent such that historical information on game graphs can be reused. Section 6 presents and discusses our initial experimental results of game graph on computer GO. This paper is concluded with summary remarks and directions for future work in Section 7.

II. Game Tree and Game Graph

Two-person information symmetrical game can be represented by a game tree, which is a special "And-Or Tree", which is shown in Fig 1. A game tree has the following features:

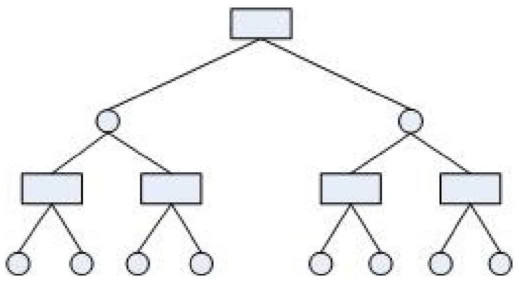


Fig. 1. And-Or Tree.

1) Define the initial state of the game to be the initial note.
2) The "Or" nodes and the "And" nodes in a game tree is layer-alternating. The relationship of the notes in ones own is "Or", and as to the opponent, the relationship of the note is "And". The two sides take turns to expand their nodes. Suppose the two sides of a game is A and B. There are a lot of options for A and B to choose in every step. From the viewpoint of A, the relationship of the options for him to choose is "Or", cause he holds the initiative. On the other hand, the relationship of the options for B to choose is "And", cause the initiative was held by B. Every option can be chosen by B, so A has to avoid this occasion to happen.
3) During the whole process of a game, we stand on our own side, and all the situations that can make we win are primitive issues, and the corresponding nodes are solvable; The situations that can make the opponent win are unsolvable notes.

Generally, we search the game tree to solve the problems. The detail process is generating a game tree with a certain depth and find the best action plan currently. After that, expand a certain depth in the branches selected, and then, select the best action plan, and then repeat the process described before, until there is the winner or the draw. During this solving process, the depth of the game tree is the bigger, the better. But due to the limit of computer memory and its processing time, we should definite the depth according to the actual situation. However, we should at least expand a layer of "Or" notes and a layer of "And" notes every expansion.

Since the expansion of Game Tree was processed in branches, so the branch itself doesn't check whether the note that will be added has already been added in other branches. Therefore, there will be the same notes in different branches. The Figure below shows a part of "Tic-Tac-Toe" Game[4]. From the figure, we can see that the Game Trees expanded through A and B can generate the notes with the same state (boxed with red lines). But in the real practise, there will be a lot of same notes in the expansion of the Game Tree. If we just distinguish the notes by state, the Game Tree will show the characteristics of the directed graph. The notes in the tree will degenerate to the vertices of directed graph. Thus, the State Space which may appear in the Game will be compressed.

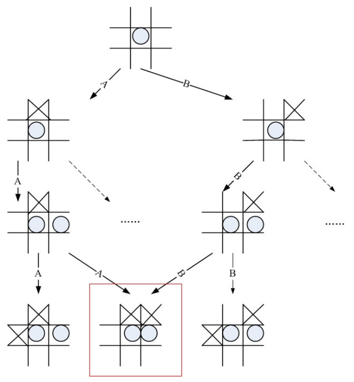


Fig. 2. Tic-Tac-Toe.

Thinking another way, it is good to consider Game Tree from the perspective of graph. The information of the same notes generated in different branches can not be shared. But actually, the development later will base on the state currently regardless of how we arrived the current status. That is, if the state A appear frequently in the same branch, we will give the assessment to the A note in a way. If we find note A in other branches by searching, we now have some knowledge of state A, cause we have the assessment of A before. Therefore, from this perspective, it is good for us to utilize the searching and assessing results if we consider the Game Tree by the method of Graph. It also can improve the efficiency of the searching.

III. Modified UCT Algorithm for Game Tree

In the development of Go System, UCT algorithm is considered to be the most suitable search algorithm in Go Game. The table below shows the pseudo-code of the UCT algorithm. We can see from the pseudo-code that UCT algorithm is a dynamic Game Tree searching algorithm based on Monte Carlo algorithm (the details not described here). According to UCT algorithm, the expansion of the notes is guided by the UCB1 formula, as to the UCB [5]value of each individual node, can be expressed as:

$$VALUE_i = \begin{cases} \infty, the\ node\ is\ not\ visited \\ \frac{W_i}{V_i} + \sqrt{\frac{2logN}{V_i}}, the\ node\ has\ been\ visited \end{cases} \tag{1}$$

In this formula, i represents the number of the optional nodes, where $i \in [1, k]$, k represents the total number of the optional nodes. So the UCB value of the number i node is related with these three value: V_i represents the times i-node is visited by now. W_i represents the winning times during the V_i times visits. N represents the total visited times of the whole k optional notes, that is $N = V_1 + V_2 + \ldots + V_k$.

Until now, we can know that if the visit times and winning times of every optional node are big enough, we can make the reasonable action order accurately by the calculation of UCB value, and thus improve the efficiency of the searching. However, generally, after we finished searching a Game Tree, all the information will be deleted for the next search, simulation or assessment. And moreover, during a complete searching of Game Tree, the value of W and V of the same state in different branches don't share. Therefore, the value of the W and V is not big enough if we use the traditional method, which, to some extent, limits the accuracy of the UCB guidance.

So we need to make some improvement to the dynamic search algorithm of Game Tree. Define the state to be a note in the Graph, regardless of which branch the state appear in. Use the same W value and V value as long as the state is the same. Moreover, visits from every branch can modify the total W value and V value. So, we can store the searching process of Game Tree according to the theory of Graph.

IV. Computer GO Game Graph Representation

In some ways, Replacement Table is a method that connects the searching tree and graph.. Replacement Table is based on the theory that sacrificing the space for time. The basic principle is to use a table to record the information searched, and if a note is already recorded in the table, use the record directly and recite it into the current search, that it, use the replacement to reduce the times of search.

1) The Representation of Go Searching Graph

The Representation of Vertices: Since every note in Go Searching Tree can represents the status of the whole board, when we transfer it to searching graph, it should still record the information of the whole board. Here we use the way of Zobrist hash coding to number the status of the board. In Zobrist hash coding, assign a different random number to every state of each location on the board, that is, every state of each location has a hash value. As to calculate the Zobrist hash value of a specific state of the board, it is to do bitwise XOR process of the state hash value of each location, and therefore we can get the Zobrist hash value of this board state. The conflict rate is very small, as for the coding of 19*19 or 9*9 board, the conflicts can be ignored. Moreover, We can learn from the characters of XOR operation that XOR operation is commutative. So, if the status of the board changes, we don't need to recalculate, and we just need to take the XOR operation of the corresponding state value. Thue, when the requirements of the calculation speed is high, the coding won't make much impact on the efficiency. In specific implementations, Zobrist hash needs to generate random numbers to three states (Black pawn, White pawn, Empty) of each location. As to the status of the whole board, Zobrist hash value is take the Zobrist hash value of each location, and do XOR operation of the previous Zobrist hash value. In this representing process, in order to avoid conflicts, we usually use a 4-byte unsigned integer to represent a board status.

2) Storage and Query of Hash Table

Internal Storage and Query:

Internal Storage: We use a big list to represent hash table in the memory. Correspondingly, the nodes which connect the list are the header of the small lists. The specific method is: As to a new hash value, through a particular way of transformation, we locate it in a specific group of the big list, and insert it into the end of this grouped list. If the memory is too full, we need to insert the header data to the external file, and make room for the new data.

Internal Query: We should firstly locate the hash value in the group of the big list when querying, and then query the list which the grouped data may be in one by one until the success of the query or to the end of the list. If we don't find the result needed in the list, we can continue to query in the external files.

3) The Usage of Replacement Table

Until now we have got the replacement table, and currently we can take advantage of it. If a specific status appears, we will firstly give an assessment of it, and get the W value and V value of it. If this status has appeared in history, there may be the historic values of W and V, we represent them by W_h and V_h. Since we have brought the theory of Gragh into account, the historic information can also be used. So, when we use the formula UCB, the winning ratio in the first part is not simply calculated by $\frac{W_i}{V_i}$, it is now expressed as:

$$\frac{mW_i + W_{h_i}}{mV_i + V_{h_i}} \tag{2}$$

In the formula above, m is a constant which is greater than 1, and it improves the accuracy of the estimated winning ratio by making amendment to the simulation result. The reason why we do this is obvious. When the historical data is insufficient, the appropriate amendment to m can strengthen the current simulation; When the historical data is sufficient, it is confident to trust the historic information. On the other hand, as to the historical data, we don't simply accumulate the visits and the results each time. We weaken the historical data every time before accumulation. That is, the new data is the sum of the current simulation result and the weakened historical data. The formula is showed as below:

$$V_{h_{newi}} = V_i + nV_{h_{oldi}} \tag{3}$$

In the formula above, n is a constant which is less than 1. The reason why we weaken the historical data is obvious too. The credibility of the historical data reduced along with the evolving of the Go Game. We don't have the historical data at the very beginning, so the data is generated occasionally. And that in turn make the historical data accumulating unreliable. But when the data accumulated in a large scale, the date will be more accurate. The previous data can be weakened by the weighting factor n.

V. Persistency Game Graph

Since the memory is limited, it is impossible to store all the hash code in the memory. When the memory is too full, we store the state of header into the external hash table. Since the board status is represented by a 4-byte (32bits) unsigned integer, meanwhile, the visit times and winning times also require 4-byte unsigned integer, so actually, to record all the information of the whole board, we need the space of 12 bytes. As the 4-byte unsigned integer value range from 0 to $2^{32} - 1$, it is impossible to store them in single file if all the states appear. In order to not only distinguish the states, but also store all the states, we use the first 5 bits of hash code as the file number and the latter 27 bits as the serial number of this state in the document, and then store every state.

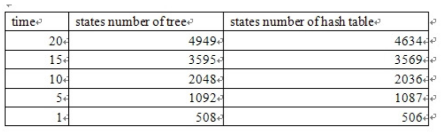
The principle of query is the same as the external storage. We find the location of the file according to the hash code firstly, and find the specific serial number in the document to read the correspond data. If the result of external query is NULL, it proves that there is no data bits here, we need to insert data into internal storage.

VI. Experimental Results

Game Tree expands through its branches, so there are as many state spaces as notes. The huge scale of the state space is the key limit factor of the Game Tree's searching accuracy.

However, after the combination with the graph theory, the only thing need to consider is the number of the real state. This to some extent reduces the number of the state space. Moreover, due to the sharing of data, we can assess the state more accurately.

When I used game graph instead of game tree, I got the results following. It shows that we can reduce the number of the states from using game graph.



time	states number of tree	states number of hash table
20	4949	4634
15	3595	3569
10	2048	2036
5	1092	1087
1	508	506

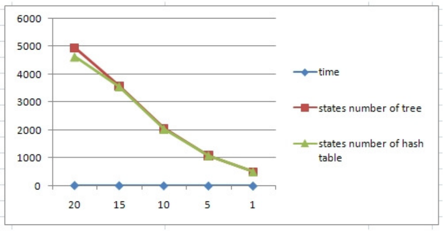


Fig. 3. Result.

VII. Conclusions

The results shows the relationship between the time and the number of the state. We haven't find other advantage in other property. In the future, we have to do more other work to prove that game graph is excellent in Go game.

References

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

[2] L. Kocsis and C. Szepesvari. Bandit based monte-carlo planning. In 15Th European Conference on Machine Learning(ECML), pages 282-293,2006.

[3] Sylvain Gelly, Yizao Wang, Rémi Munos, Olivier Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA, France, November 2006.

[4] Edward Kaplan. Interactive tic-tac-toe slot machine. United States Patent.Patent number 5927714

[5] F. Van Lishout, G. Chaslot, and Jos W.H.M. Uiterwijkm. Monte-Carlo Tree Search in Backgammon. Computer Games Workshop :175-184, 2007.