Study on Loop Problem in Opening Database for Chinese Chess Programs

Jr-Chang Chen[1] Shi-Jim Yen[2] Shun-Chin Hsu[3]

[1]Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan.
[2]Department of Computer Science and Information Engineering, National Dong-Hwa University, Hualien, Taiwan.
[3]Department of Information Management, Chang Jung Christian University, Tainan, Taiwan.

Abstract

A Chinese chess program systematically constructs a large tree-based opening database by collecting knowledge from chess masters, books and game records in the opening phrase. However, those games with loops are not managed properly in the database. The perpetual situations are not recorded correctly in the database, and therefore the program will play a draw in an advantageous position and a loss in a draw position. This study describes a solution to the loop problem in opening database.

Keywords: Computer Chinese chess, opening database, loop problems, perpetual situations.

1. Introduction

The opening phase of Chinese chess refers to the first 8~12 rounds after a game begins. During this phase, the main objective is to occupy better positions, to establish an advantageous position for later combat, and furthermore, to achieve command over the pieces of opponents and thus a strong battle array. Thus, the opening phase is the foundation of the game [1]. How well the opening phase is played will directly and strongly influence the mid-game and endgame.

The current design of the opening phase of Chinese chess is based on an effective and sound opening database system. The database is constructed by gathering games from numerous books and game records, and by extracting and adjusting the abundant master knowledge [2]. Based on the opening database, each move is statistically analyzed, and the optimal move is thus identified. Combined with mid-game programs, the opening database can clearly improve the strength of Chinese chess programs [3]. Certain methods of designing chess programs automatically increase the quality of opening databases [4], take the habitual behaviors of opponents into consideration, and then adopt an appropriate response strategy [5].

In game records of Chinese chess, the same boards frequently appear. Loops due to repetitive boards affect the opening database structure, and create difficulty in gathering statistics about moves. In this study, we develop an efficient data structure and update algorithm to maintain database integrity. This data structure and algorithm solve the problem in gathering statistics about moves in a tree-structured opening database.

2. Opening Database

Computers simulate human thinking in order to get the best move to attack and defend based on the current situation, and create a game tree during the process. When a game tree expands to a specific depth, each leaf node is assigned a score calculated from an evaluation function, and the best move is identified by the mini-max method [6]. Because the main task during the opening phase is to move pieces to advantageous positions, no obvious fighting occurs, and thus the limited difference between the two sides produces a blind spot in the evaluation function. [7]

To overcome this problem, an opening database system was designed using techniques such as framework, tree structure and hash function [2]. Mass good quality games records played by masters from internet are collected and analyzed to retrieve statistical information, including the usage rate and winning rate of each move from each board. The strength of each move can now be compared, and this information can be used as a reference during mid-game play [3].

The architecture of the proposed knowledge-base system has a hash area and an overflow area. The hash function calculates a hash address using move plies and piece positions on board as keys. The address corresponds to a specific slot in the hash area if the slot is available. Otherwise, the board is saved in a slot in the overflow area, and this slot and the specific slot in the hash area are linked.

Each node in the database is assigned a specific board status, which has three types of information – position, statistics and links. The position records the ply and the distribution of all pieces used by a hash function. The statistical information includes winning ratio, usage rate, and the differences among win, draw and lose. The links maintain the connection between the hash and overflow areas, and record all moves but alter the board from its previous to its current status [2].

3. Influence of the Loop Problem

In Chinese chess, a repetition occurs when the same position reappears for the third time. Sometimes this occurs when both players are making non-attacking moves (perhaps aimless ones), in which case the game is a draw. More commonly, a repetition occurs because one side is perpetually attacking (threatening) a specific opponent piece (e.g., the King). The loop resulting from repetitions deeply influences the way that opening databases operates. While updating scores using the mini-max method or gathering statistics of game information related to the node, the program will get an inaccurate result, or even worse, will run into an infinite loop. Therefore, loops must be fully recorded in the database so that programs can suggest the correct move.

To simplify the process of updating the score using the mini-max method, we delete loops in the opening database in our previous Chinese chess program. When a game contains a loop, the same positions, nodes A and B, are identified; then all nodes between A and B are deleted, and all descendent of B now become those of A. Therefore, before a game is added to the database, all loops in the game are deleted, and the structure of the database becomes a tree without any loop. Although the loop-deletion method simplifies the loop problem, if the program uses scores as a guide in choosing moves, the result will be a draw which actually should be a win, and a loss which actually should be a draw, because loops are not recorded in the database.

4. Data Structure of Loop

To record a loop accurately and completely, a mark is made on each node of the loop with the shortest distance to the leaf. The marked nodes can not have any other child nodes besides those belonging to the loop.

Figure 1 indicates that the node sequence **ABCDA** exists in the database, in which nodes **BCDA** form a marked loop, and the two **A**s nodes are repetitive nodes with the same position. If a loop node has a child node whose position is different from all of the loop nodes, then the same board with different plies appears on different nodes. Figure 1 shows that nodes **E** and **G** are the child nodes of the two nodes **A**s, respectively. This situation affects the operation of the database in two ways:

First, when updating score using the mini-max method, one node will have two different scores representing the scores of a loop node and that of a non-loop node respectively. The program must design another mechanism to choose between the two different scores. Figure 1 shows that node **A** chooses the best score based on three scores from child node **G** (score 60), loop node **B** (score -20), and loop **ABCDA** (score 0, the score can not be shown in Figure 2 because it is an infinite loop since both players repeat the same moves.).

Second, the move the opponent chooses will affect the score obtained with mini-max method. An ancestor node in the tree shadows a descendent node such that the nodes with the same position have inconsistent scores. Figure 2 shows that nodes **E** and **G** are both child nodes of the two **A**s nodes by playing different moves, and have scores of 100 and 60 respectively. When updating the score, since node **B** chooses the move to node **F**, which has a score of -20, the ancestor node **A** must choose the move to node **G** which has a score of 60. Hence, although both **E** and **G** are child nodes of node **A**, worse move to node **G** (score 60) is chosen rather than that to node **E** (score 100), which is wrong.

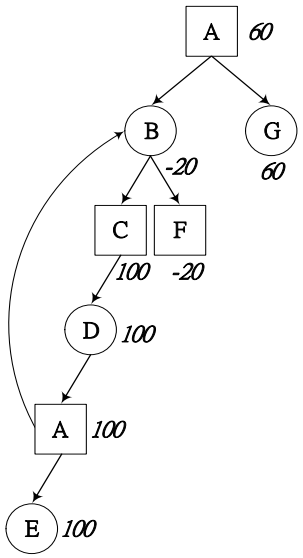

Fig. 1: An example of a loop in the opening database.

In the tree structure of the database system designed here, all loop nodes are stored at the lowest level of the game tree, and child nodes of loop nodes are moved to upper corresponding nodes with identical positions. Figure 2 shows that loop nodes **CDAB** have no other child node, and the ancestor node **A** can choose the better move to node **E** whose score is 100.

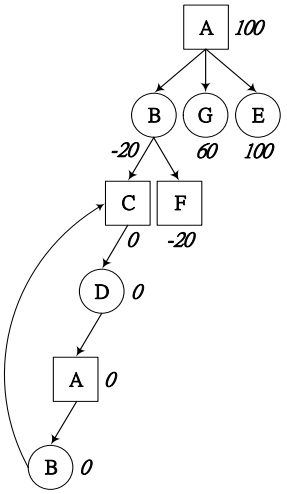

Fig. 2: Solution of loop problem in the example in Fig. 1.

In order to make the most of limited time, or achieve a draw when in an inferior situation, a player adopts repetitive plays to repeat among several positions, for example through circulated checking and freeing. Such games will have loops that are not in the end of a game. To allow all loops to only exist in the lowest level of the database, we adopt the following steps: Before adding a game to the database, we split the game containing loops into multiple games, including a game without the loops, and games whose positions are from the beginning to the loop position and let the loop to be the last moves of the game. After this procedure, these derivative games are added to the database.

5. Implementation

A new game can be added to the database as follows. First, we flip the first move according to the symmetry of the board. Second, we transform the following moves according to the move order relation. Then, we add the game into the hash area of the database if the specific slot is available, or add it to the overflow area if the slot is occupied. Because each node records the relation between the parent nodes and child nodes, the tree structure is maintained. Finally, we update the score of each node with the mini-max method, and gather statistical information on winning ratio, usage rate, and the difference among the frequencies of wins, draws and losses [2].

If the game being added contains loops, the loops must be stored in the lowest level of the tree structure. First, the loops inside a game must be removed, and the game must be added into the database, as mentioned in section 4.2. Next, the combination of loop nodes must be processed with the nodes existing in the database. This section adopts the method of extending the loop nodes, and then duplicating all the non-loop nodes of the lower level to the upper level. This approach avoids the situation in which the program must choose between loop moves and non-loop moves. Incidentally, if loop nodes are ignored, the database itself is a complete tree structure, making it easy to update scores using the mini-max method. Figure 3 shows three situations about the combination of loop nodes and non-loop nodes.

1. A loop exists in the database, and we want to add a game containing no loops, as shown in Figure 3(a).
2. No loop exist in the database, and we want to add a game containing no loops, as shown in Figure 3(b).
3. A loop exists in the database, and we want to add a game containing a loop, as shown in Figure 3(c).

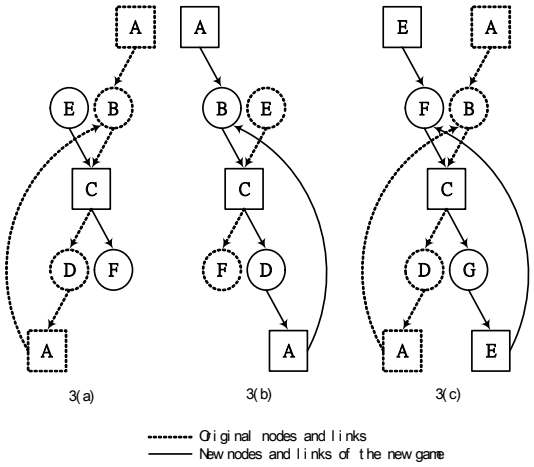

Fig. 3: Three situations about the combination of loop nodes and non-loop nodes

To maintain the architecture that no loops are within the tree structure, when nodes of a new game intersect with existing loop nodes in the database, the loop nodes must be extended into an independent branch that contains only one child, and the branch must be duplicated to the upper node with the same position as the loop nodes.

Let *LOOP* represent the set of nodes whose move results in the same board appearing repeatedly. Let *loop_head* represent the last node in *LOOP* whose board is the same as that of leaf in the game. If the game is a draw due to repetitive boards, and if the loop nodes exist in the database, we take the following steps. First, nodes in database downward whose positions are the same as those of loop nodes recursively are found, and these nodes then are joined to the *LOOP* set, with the last one being set as a new leaf. Then, the intersect node upwards from which an independent loop must be extended is identified. Finally, all nodes between *loop_head* and the intersect node are duplicated to those below the leaf node. After an extension is done, if the extended nodes still collide with another node in the database, the above procedure is repeated. Ultimately, only one branch of the loop exists at the bottom of the game tree.

After an independent branch of the game is extended, the situation in which the game nodes collide with the loop nodes in the database must be extended. Besides, let the upper loop nodes reserve child nodes and statistic information more completely, making it necessary to duplicate all descendent nodes of a lower node to the upper node with the same position. As this point, the nodes are scanned from leaf to root, and loop nodes are encountered before non-loop nodes, and each node is processed depending on whether it is a loop node or not, as shown below.

Loop nodes are processed in the first step. First, because adding a new game result in the loop nodes (say *LOOP_A*) which contained in the database no longer form a complete and independent branch, *LOOP_A* must be extended. Loop nodes (say *LOOP_B*) of the new game are then extended to the child nodes of the corresponding upper loop node in *LOOP_A*. Finally, all child nodes of the last extended node in *LOOP_A* are duplicated to the child nodes of the corresponding upper loop node in *LOOP_B*. Non-loop nodes are encountered in the second step. Because only non-loop nodes of the new game collide with the loop nodes in the database, all that have to do is to duplicate the nodes of the new game to the child nodes of the uppermost loop node after extending the spoiled structure of the loop nodes.

6. Conclusion

This study offers a method of managing the loop problem created by playing repetitive moves so as to cause the same board to appear reiteratively in Chinese chess opening database. This study reaches three key findings, as following.

1. If loop nodes are ignored, the database is a complete tree structure, and is compatible with the previous system we developed. [3]
2. While updating statistical information using the mini-max method, this method does not trap into an infinite loop because no loop nodes exist inside the game tree.
3. Nodes with the same board in a lower level are moved to an upper level, so complete information is stored in upper level nodes, and the mid-game program can rapidly retrieve the optimum move.

The opening database stores loops in the leaf of the database such that repetitive boards could be completely contained and correct scores can be obtained using the mini-max method. The program does not result in draws that should be wins, or in losses that should be draws.

7. References

[1] Chen, J.R., “Chinese Chess Opengame Database Design,” M.Sc. Thesis, Department of Computer Science and Information Engineering, National Taiwan University, Taiwan. (in Chinese)

[2] Huang, S.L., “Strategies of Chinese Chess Opening,” World Culture Inc. Press. 1991.

[3] Hsu, S.C. and Tsao, K.M., “Design and Implementation of an Opening Game Knowledge-Base System for Computer Chinese Chess,” Bulletin of the College of Engineering, N.T.U., No. 53, pp. 75-86. (in Chinese)

[4] Thomas R. Lincke, “Strategies for the Automatic Construction of Opening Books,” CG2000, LNCS 2063, pp. 74-86, 2001.

[5] Steven Walczak, “Improving Opening Book Performance Through Modeling of Chess Opponents,” ACM Conference on Computer Science, pp. 53-57, 1996.

[6] C.E. Shannon, “Programming a Computer for Playing Chess,” Philosophical Magazine, Vol. 41, 1950, pp.256-257.

[7] Yen, S.J., Chen J.C., Yang T.N., Hsu S.C., “Computer Chinese Chess,” ICGA Journal, Vol. 27, No.1, March 2004, pp. 3-18, ISSN 1389-6911.