VOL. 3, NO. 4, April 2012 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences
©2009-2012 CIS Journal. All rights reserved.

http://www.cisjournal.org

Search Algorithm in the Five-piece Chess

Tan Shunhua, Chen Miao
School of Information Engineering, Southwest University of Science and Technology
Mianyang, Sichuan, China
Email: {Tanshunhua,chenmiao}@swust.edu.cn

ABSTRACT

Game playing theory is an important research branch of artificial intelligence. In this paper, we first describe how to build a game tree for five-piece chess game. Then, we introduce a maximin search algorithm in game tree to find the optimal playing strategy, and $\alpha-\beta$ pruning strategy to improve the search efficiency. Finally, we evaluate the performance of search algorithm by analysis and an example program based on Visual C++ 6.0.

Keywords: *Game tree; minimax serach; $\alpha-\beta$ pruning; evaluation function*

1. INTRODUCTION

Game playing theory is an important research branch of artificial intelligence, mainly including intricate chess games. Existing resolved chess games [2-3] benefit from the development of game playing theory of machine in the last half a century. The advantage of computer lies in its high speed computing power. Many complex problems can be solved by enumerating all possible and feasible solutions, and then selecting the optimal strategy. However, chess game playing should not depend on the high speed computing power excessively, since this problem is very complex. In fact, the complexities of state spaces of existing chess games have exceed the total number of grains existed in the whole universe. The computing power of existing computer has not been strong enough to enumerate all such a large-scale state space. Fortunately, we can decrease the scale of the problem by considering the constraints of domain knowledge and enhanced knowledge, both of which are derived from good understanding of the problem to solve and good description model. For five-piece chess game example, the characters existed in four directions can help search the optimal solution more effectively. In this paper, we first build a game tree for five-piece chess game, which is used to evaluate and predict the whole situation of the game. Then, we introduce a maximin search algorithm based on the newly built game tree to find feasible playing strategies. Next, we also point out how to enhance the search efficiency by applying α-β pruning theory. Finally, we analyze the efficiency of the proposed algorithm and evaluate its practical performance by comparing our example playing program with Renju Solver, which is the most famous five-piece chess playing program.

2. SITUATION EVALUATION

a. Game Tree

When the chess game is going on, the game situation is steered jointly by the actions of both players. At each step, the whole situation of the chessboard is described by a node. When one player adds a new stone, the current situation is transferred into another. For the current situation, the current player has many choices, which will transfer the current situation into different situations. In fact, each blank position in the chessboard is a feasible choice for the current player. A playing process reflects a path which originates from the root node, which represents the situation that the whole chessboard is blank, to the leaf node, which represents the final situation of the game that one player wins the game or there is a draw. If the whole tree is known by one player in advance, he can always find the optimal strategy to deal with the opponent and win the game finally. However, the game tree is so large in scale that existing computers neither build the whole picture within reasonable period, nor save the tree in their storage systems.

b. Situation Score

The situation evaluation is the basis of the whole game playing system. Whether the situation of the chess game is evaluated precisely or not determines the final performance of the game playing program.

The player needs to evaluate the situation if the next stone is placed at a blank position, including its own stone and the opponent’s stone. Thus, all remainder blank positions in the chessboard should be analyzed. For a generic blank position, the player should analyze the situation in four directions, including horizontal, vertical, diagonal, and back-diagonal. The situation is actually the progress that the player or the opponent wins the game. We assign two scores to each blank position to describe the situation. One is for the player self and another is for the opponent. If the new stone is placed at this position, the player checks whether local stones generate any pattern in Table 1. If yes, the position is assigned with the corresponding score. The final score of a position is the sum of four directions. The higher the score is, it is more probably that the player wins the game once the new stone is placed here. On the contrary, the player also estimates the score for this position if a stone of the opponent is placed here. The value in Table 1 is derived from game rules and playing experiences. The scores of the player and the opponent are denoted by $G_1(i, j)$ and $G_2(i, j)$, respectively, where i and j indicate the position. For example, if a stone of the player is placed at position (i, j) and local stones of the player match Sleep-Three, Connect-Two, Sleep-One, Sleep-One, in four directions, the score of the player is given by

$$G_1(i, j) = 600+300+15+15 = 930 \qquad (1)$$

On the other side, if a stone of the opponent is placed at position (i, j) and local stones of the opponent match Sleep-Three, Connect-One, Big-Connect-Two, Connect-Two, in four directions, the score of the opponent is given by,

$$G_2(i, j) = 480+40+200+80 = 600 \qquad (2)$$

c. Evaluation Function

Table 1: Score Table for the situations of gobang

Situation	The one who is playing with stone	The rival
Sleep1	15	10
Connect1	50	40
Sleep2	100	80
(big)connect2	230	200
Connect2	300	240
Sleep3	600	480
(big)Three	1200	1000
Three	1800	1200
Four	2100	1800
Straight four	5000	3000
Five in a row	100000	10000

The evaluation function is used to estimate the situation of the whole chessboard and reflect the situation advantage of the player. In terms of five-piece chess game, the advantage of the player usually depends on the situation of the most important position, namely the position with the maximal score, which is defined as

$$MaxG_1 = \max\{G_1(i, j) \mid 0 \le i < L, 0 \le j < W\} \qquad (3)$$

and the most important position for the opponent is defined as

$$MaxG_2 = \max\{G_2(i, j) \mid 0 \le i < L, 0 \le j < W\}, \qquad (4)$$

where L and W are the length and width of chessboard, respectively.
The evaluation function is defined as the difference between $MaxG_1$ and $MaxG_2$, namely,

$$F_{state} = MaxG_1 - MaxG_2 \qquad (5)$$

F_{state} is lager, the situation is more advantageous for the player. On the other side, the situation is more advantageous for the opponent.

3. MAXIMIN SEARCH

During the game playing process, the player faces so many blank positions and has to decide which position to put the stone at each step. The player must select the most urgent position. The decision depends on both the current situation, and the next potential actions of the opponent and future situations. All potential situations can be reflected by the game tree. Thus, the player can predict subsequent situations by searching the game tree. Obviously, the player searches deeper, the future situations are clearer but it also incurs more search cost. After evaluating all situations, the player can make the local optimal decision to increase the probability of winning the chess game. To predict the actions of the opponent, we design a minimal-score playing strategy for the opponent, namely, the opponent always chooses the strategy that will minimize the situation score of the player among all possible strategies. The future game playing process can be predicted by alternate maximizing and minimizing searching actions. Then, the player can predict the situation several steps later and make the optimal decision. The flow of the maximin searching algorithm is illustrated by Fig.1.

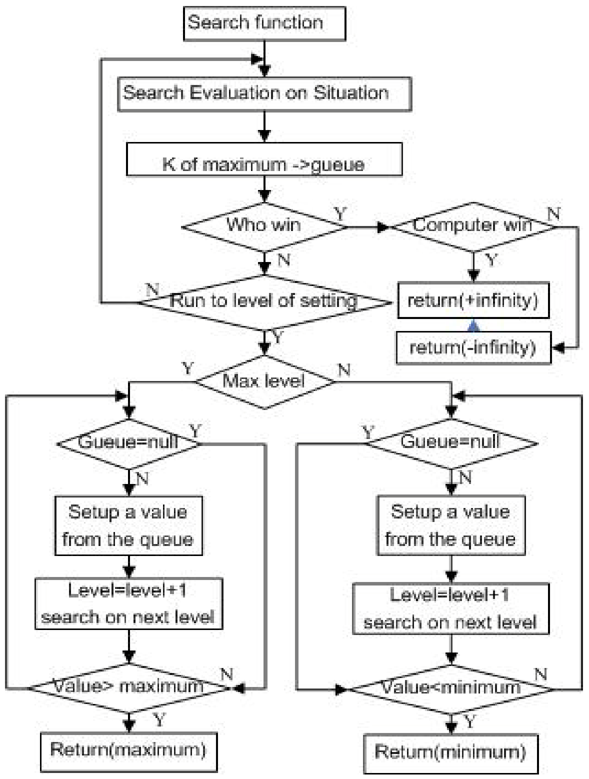

Fig 1: Flowsheet of Minimax Search Algorithm

As shown in Fig. 1, the algorithm first enumerates all blank positions and assigns a score for each position as described above. Then, the K-most important positions are put into a queue, which are also the searching objects. If the player or opponent wins the game, namely that a position which generates the pattern Five-in-Row is found, the search returns this position and game is over. If the search has arrived at the deepest layer, the algorithm returns the final score of the current position. Otherwise, the algorithm continues the search at the next layer. Before the next layer search, the algorithm checks the current layer is to maximize or minimize the situation score.

If the current layer is to maximize the situation score, whether the queue is empty or not is checked first. If yes, the search terminates. Otherwise, it picks nodes orderly and continues the next layer search. When the next layer search returns, the chessboard is recovered. If the returned value is lager than the given threshold, it begins pruning and terminates the search at the current layer. The flow is similar with that when the current layer search is to minimize the situation score.

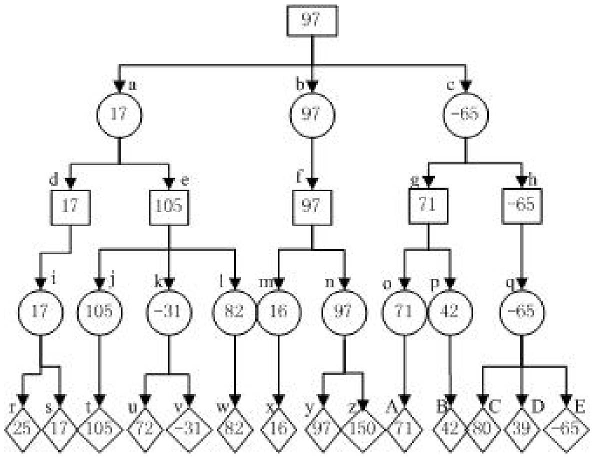

Fig 2: Flowsheet of Minimax Search Algorithm

Fig.2 illustrates a searching example. The square box represents the maximizing action of the player and the circle box represents the minimizing action. The value in the square or circle box is the score of the current situation and the value in the diamond box is the score of the situation located in the maximal search depth. At the beginning, the player faces three most important positions, namely, *a*, *b*, and *c*. If position *a* is chosen, the opponent has two most important positions, namely, *d* and *e*. Of course, the opponent will choose *d* to minimize the situation score of the player. When the search arrives the maximal depth, the situation score of the player is 17 if it chooses position *a* at first. Likewise, the situation scores are 97 and -65, if position *b* and *c* are chosen respectively. Therefore, the player can obtain a good situation by the expected searching path *b-f-n-y*.

4. IMPROVEMENT OF PRUNING STRATEGY

Further improvement can be achieved without sacrificing accuracy, by using ordering heuristics to search parts of the tree that are likely to force alpha-beta cutoffs early. Recently, many improvement version ofα-βpruning have been proposed and proved to be very useful to improve the searching efficiency in practical areas. In this section, we propose a new improvement pruning strategy, which is based on tabbing order, iterative deepening, minimal window search, and quiescence search.

a. Tabbing Order

It is well known that the pruning efficiency depends on tabbing and searching order greatly. Different orders result in different pruning. To make use of the cutting-off effect adequately, all nodes are examined first and sorted in descending order of situation scores.

b. Iterative Deepening

Iterative deepening is first proposed as a time controlling mechanism of game tree search. It refers to how to determine the search depth according to the search time. Iterative deepening is prone to prolong the search at some layer, which seems a waste of time and resource. However, the iteration at a lower layer consumes less resource. Besides time control, Iterative deepening is more effective than the common direct search. The reason is that the subsequent search order can be adjusted according to the results of the previous search, which improves the search efficiency greatly and also incurs few additional costs. Iterative deepening means repeatedly calling a fixed depth search routine with increasing depth until a time limit is exceeded or maximum search depth has been reached. The advantage of doing this is that you do not have to choose a search depth in advance; you can always use the result of the last completed search.

c. Minimal Window Search

In the pruning process, the pruning probability increases as the window decreases. The window, where $\alpha = \beta - 1$, is the minimal window. Thus, the search efficiency is maximized when using the minimal window. Both NegaScout and MTD is the search algorithm based on the minimal window. Their advantages are more obvious in large-scale game trees.

d. Quiescence Search

A depth-fixed search algorithm enumerates all branches to the same depth. Instead of calling Evaluate when depth=0 it is customary to call a quiescence search routine. Its purpose is to prevent horizon effects, where a bad move hides an even worse threat because the threat is pushed beyond the search horizon. This is done by making sure that evaluations are done at stable positions, i.e. positions where there are no direct threats. A quiescence search does not take all possible moves into account, but restricts itself e.g. to captures, checks, check evasions, and promotion threats. The art is to restrict the quiescence search in such a way that it does not add too much to the search time. Major debates are possible about whether it is better to have one more level in the full width search tree at the risk of overlooking deeper threats in the quiescence search.

VOL. 3, NO. 4, April 2012 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences

http://www.cisjournal.org

e. Analysis of Pruning Efficiency

In this subsection, we analyze the efficiency of the pruning search algorithm. The search must arrive at an end node which is located at the deepest layer of a part of game tree, since the values of αandβdepend on the situation score at the deepest layer. Thus, α-βpruning search usually adopts the depth-first strategy. In addition, the number of cut branches is dependent on the similarity degree between the initial values of αandβand the score of the final end node. Extremely, if the final end node appears in the first round of depth-first search, the most braches is cut off and the number of nodes that needs to be visited is minimized.

We first analyze the most ideal progress, where the minimizing search expand branches in ascending order of situation scores and the maximizing search expand branches in descending order of situation scores. Assume the maximal depth is *D* and the branch factor is *B*. Then, the number of end nodes generated in the search tree is $N = BD$ whenα-βpruning is not applied. However, when applying pruning strategies, the number of generated end nodes can be reduced as

$N = BD/2-1$, D is even (6)

$N = B(D+1)/2 + B(D-2)/2 - 1$, D is odd (7)

Thus, pruning search only visits half nodes when compared with the direct search, which improves the search efficiency greatly.

5. PERORMANCE EVALUATION

Renju Solver is the chess playing software currently with the greatest ability, with which we can fully test evaluation the performance of the proposed algorithm. Our testing hardware is DELL PRECISION 530 workstation, whose configurations is listed as following: two Xeon (TM) 1.7 G CPU; 2GB memory; 80G hard disk; WINDOWS XP operating system. The testing method is that two programs running in the same computer play five-piece chess with each other. They begin the first stone alternately. Renju program is set to the highest difficulty, level 9, where the comprehensive search depth is set to level 3, and the testing time is set to extremely slow. This program is set to the most difficult level, or the top level. Testing results of ten rounds competitions are the program that begins the first stone will win the game.

As shown in figure 3, there are few differences among the final situations of all rounds of games and none of two parties can defeat the opponent and win the game quickly. Thus, we conclude that those two programs have the similar performance. Renju Solver spends about 2-4 seconds at each step, with comparation that this procedure cost 40 to 55 seconds at each step. Thus, the search speed of the proposed search algorithm needs to be improved further.

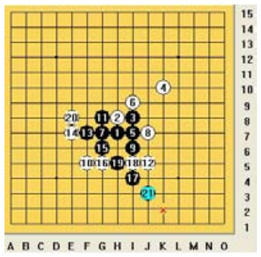

Fig 3: Renju Solver lost on the defensive position

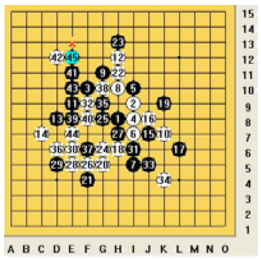

Fig 4: Renju Solver won on the offensive

6. CONCLUSION

This issue studied the minimax search algorithm of game-tree and talked about the basic principle of α-β pruning technology and how to improve it. After analyzing the efficiency of the algorithm, we find out that our software can equally match the ability of Renju Solver when just searching to layer3. But the algorithm and executing efficiency of our software are still far away from Renju Solver. In future, we will go on enhancing the efficiency of our algorithm and perfecting the game rules.

REFERENCES

[1] Zoro Tang, Wang Bo An Algorithm of the Gobang based on NN and α-β pruning, Microcomputer Application Technology, 2009-02-007

[2] WANG Chang-fei; CAI Qiang; LI Hai-sheng Design and Implementation of Intelligent Gobang Playgame. Journal of System Simulation. 2009-04-033

[3] ZHANG Cong-pin; LIU Chun-hong; XU Jiu-cheng Research on alpha-beta pruning of heuristic search in game-playing tree. Computer Engineering and Applications. 2008-16-017

[4] HAO Wei Analysis and Application of Algirism of Five Chess Based on a Optimism Model. Computer Knowledge and Technology. 2009-14-087

[5] XU Nan-shan CONG Lei SUN Feng-ping Parallel Implementation of Gobang AI with Self-Learning Ability. Computer Engineering and Applications. 2006-30-012