A study on search techniques in the game-tree

Shaojia Zhang

Hengshui High School of Hebei, Hengshui, 053000, China

jlovei85072@student.napavalley.edu

Abstract. Artificial intelligence has developed a lot in the game field and search techniques on game-trees are essential for AI-game-playing. As many techniques for searching the game-trees have been published, the time consumption of search has decreased and the accuracy of it has increased. This paper would provide a comprehensive review of existing algorithms and improvements, including their mechanisms and performances. These techniques are first divided into two sections, techniques based on Alpha-Beta pruning and techniques based on Monte-Carlo tree search. Furthermore, techniques based on Alpha-Beta pruning are further subdivided into narrow window properties and information from previous searches based on the specific aspects they focus on. Finally, this paper concludes by summarizing the performance of these techniques and identifying their suitable application scenarios, as well as suggesting potential directions for future research.

Keywords: game-tree search, Minimax, Alpha-Beta, improvement, windows, Monte-Carlo tree search.

1. Introduction

For a long time, people are keen to build AI programs to compete against humans in games. While humans were always winners in the early time, with the invention and development of many game-tree algorithms, computers are now in some sense able to defeat humans in many games, such as Tic-Tac-Toe, Checkers, Chess, and even Go. Studying search techniques in game-tree and further improving or enhancing them can make AI in games smarter and reduce the cost of such machine thinking. All techniques can be divided into two sections, techniques based on Alpha-Beta pruning and techniques based on Monte-Carlo tree search.

Most game-tree search algorithms are defined in the context of a two-person and zero-sum game-tree of perfect information. In such a game-tree, the root represents the initial game state; each node represents a position in the game; a node's ply, introduced by Arthur Samuel in [1], represents the depth of that node or the number of moves to reach that position; all branches of one node represent all legal moves from that position; leaves, which have no successors, are terminal positions, from which the result of the game can be determined—win, lose, or draw. In the most ideal case expected to happen, the entire game-tree has been generated and searched before the computer knows every strategy for every position of the game. However, this case only happens in simple games, such as Tic-Tac-Toe, but it is unachievable for more complex games, such as Chess, because the time complexity and space complexity increase exponentially in a game-tree. Therefore, evaluation systems and pruning algorithms are needed to predict results without reaching leaves and reduce useless nodes that do not affect final results, respectively. This paper reviews several sorts of evaluation mechanisms and search algorithms to make the development and category of search techniques on game-trees clear.

This paper would first introduce some basic algorithms to help readers understand other advanced algorithms. Furthermore, it focuses mainly on improvements and algorithms based on the Alpha-Beta algorithm, the classic algorithm used for searching game-trees, and also discusses Monte-Carlo tree search, which is the most popular search technique recently. It also summarizes characteristics, advantages and disadvantages, and suitable circumstances for these algorithms.

2. Background

This section introduces the basic principles and basic algorithms of game-tree search.

2.1. Minimax and Negamax algorithm

In the Minimax algorithm, two players are commonly called Max and Min. Max moves at odd ply and tries to maximize the score while Min moves at even ply and tries to minimize the score [2]. All leaves have scores evaluated according to the positions they represent. For any other node, if it is at an odd ply, its score is the maximum of scores of all its successors; similarly, if it is at an even ply, its score is the minimum of scores of all its successors. All nodes get their values through one of these two processes recursively. Max would try to move to the node that has the maximal value among the successors while Min would try to move to the node that has the minimal value among the successors.

The Negamax algorithm is a variant of the Minimax algorithm [2]. It is simpler and more convenient than the Minimax algorithm because it does not need to be checked who the next mover is. It is defined that the score of a position for one player is the negation of that for the other one. Except for leaves, every node's value is the maximum of the negation of scores of all its successors, shown as $\mathrm{V} = \max(-\mathrm{V}_1, -\mathrm{V}_2 \ldots - \mathrm{V}_\mathrm{b})$. Both two players would try to move to the node that has the maximal value among the successors.

The time complexities of both the Minimax algorithm and Negamax algorithm are $\mathrm{O}(\mathrm{b}^{\mathrm{ply}})$, in which b is the branching factor (the average number of successors each node has) and ply is the depth of the entire game-tree.

2.2. Alpha-beta algorithm

Either the Minimax algorithm or the Negamax algorithm is considered a brute-force algorithm, so pruning or optimization is needed to reduce the time consumption. The Alpha-Beta algorithm defines alpha and beta as the lower bound and upper bound of the score, which means that any value less than alpha or greater than beta is unimportant for final results [3]. For the example shown in Figure 1, no matter what value X is, the score of node-1 is certainly 7. In this case, the subtree of node-5 is not necessary to be searched because it would not have any influence on the final result, so it can be cut off.

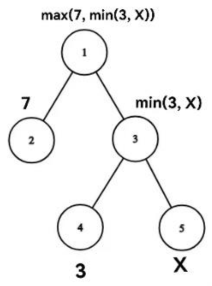

Figure 1. A sample tree that can cause a cutoff.

This algorithm can be unstable, for the time complexity is $\mathrm{O}\left(b^{\frac{ply}{2}}\right)$ in average cases but $\mathrm{O}(\mathrm{b}^{\mathrm{ply}})$ in worst cases. Although the time consumption is much less than Minimax and Negamax algorithms, it is still very large for complex games.

3. Improvements based on the Alpha-Beta algorithm

Since the pure Alpha-Beta algorithm still has problems with time complexity and stability, lots of improvements based on the Alpha-Beta algorithm have been proposed. In this paper, they are divided into two sections according to which aspect they optimize.

3.1. Algorithms based on the property of narrow windows

The efficiency of the Alpha-Beta algorithm significantly depends on the values of alpha and beta. The greater alpha is and the less beta is, the more cutoffs can be caused. Since alpha should be less than or equal to beta, it can also be written as the narrower the window is, the better the performance of the Alpha-Beta algorithm is. There are two techniques based on this property, aspiration window search and minimal window search.

3.1.1. Aspiration window search. Aspiration window search stipulates the initial window is $(\mathrm{V}-\mathrm{e},\mathrm{V}+\mathrm{e})$, where V is the estimated value of the position and e is the expected error limit, rather than $(-\infty,+\infty)$ used in the original Alpha-Beta algorithm. If the actual score of a position lies within the window of $(V-e,V+e)$, which is expected, it will not only return the correct result but also prune lots of subtrees since the initial window is much narrower than that of Alpha-Beta algorithm so that there is a good chance to find some subtrees useless. In contrast, if the actual score does not lie within the window of $(V-e,V+e)$, this subtree needs to be re-searched. In this case, the initial alpha is reset to $-\infty$ when the actual score is less than $\mathrm{V}-\mathrm{e}$ (failing low) while the initial beta is reset to ∞ when the actual score is greater than $\mathrm{V}+\mathrm{e}$ (failing high).

Such re-searching increases time consumption a lot. The ordering of moves is quite important to aspiration window search and it performs nearly perfectly in the case of searching strongly ordered trees. It is better to use the aspiration window search together with the iterative deepening and transposition tables. Alpha-Beta with aspiration windows is more effective than Alpha-Beta in most cases [4-5].

3.1.2. Minimal window search. The core property of minimal window search [6] is that to prove a subtree inferior is faster than to confirm its exact score. It is based on the supposition that the move that is going to be searched is the best move and other moves are inferior. The case that all other moves prove to be inferior is highly expected, but if it does not happen, a new supposition continues until the real best move is found. This approach performs significantly well, especially when used together with reordering mechanisms. There are two typical algorithms using minimal windows: principal variation search and memory-enhanced test driver.

Principal variation search (PVS), equivalent to Negascout search, is an algorithm based on minimal windows. It searches the first subtree with a full window (alpha, beta) and gets the exact score V of the first subtree. Then it searches other subtrees with null windows (V, V+1) to find whether there are moves that fail high, which means they are better than the first move. If a move is found to be better, re-searching for the subtree of this move is needed to get the exact score of this move and continue similar verifications. According to reference [7], PVS runs faster than aspiration window search. One of the properties of PVS is that the tree searched by it is asymmetrical because PVS always tries the best moves first.

MTD(f) is short for Memory-enhanced Test Drive [8]. Unlike PVS performs a full window search, MTD(f) completely uses null windows for searching, so it in many cases can outperform PVS. It is based on the idea to guess the Minimax value iteratively. The score V for a position starts with a value guessed to be the best or closest to the actual score. Then the algorithm uses a null window (V, V+1) to search the tree and get the information about how to adjust V due to whether the result fails high or low. Furthermore, since it does lots of re-searches, the algorithm has to use a transposition table to retrieve information of the same position. Although the algorithm re-searches the same nodes quite many times, it is still pretty much cheaper to do a null window search than do a full window search.

3.2. Enhancements based on results from previous searching

In a complex and huge game-tree, there is a high probability that there are several nodes having the same position, so it is not uncommon to encounter a position that is identical or similar to a position searched previously. Therefore, for better performance, it is attractive to use previous information that has already been got without searching again. Several most popular techniques based on this idea and discussed in this paper are iterative deepening, transposition table and refutation table, and killer heuristic and history heuristic [6].

3.2.1. Iterative deepening. This method searches game-trees by iteration with a depth limit D which starts with 1 ply and at the beginning of each iteration, the limit is extended by 1 ply. The main idea of iterative deepening is to do better moves in D-ply search based on (D-1)-ply search. Since there is a high probability that the principal variation of (D-1)-ply search is a prefix of the principal variation of D-ply search and even further searches, first trying or examining the best moves of the last iteration is a good strategy. Moreover, the results of the last iteration can be used to set up alpha and beta in this iteration. Even though, the most superior point of iterative deepening is that it can give final results with high accuracy at any time. This is very useful in time-limited tests. It is usually used with transposition tables or refutation tables.

3.2.2. Transposition table and refutation table. The transposition table uses the idea of hash tables to store information as a large direct access table. When a node is reached, if the previous search of the position of this node reached the desired depth or height, the previously stored score of it can be directly used and further search of this subtree can be stopped. Otherwise, not the score but the best move of it can be used for subsequent faster searching. This optimization is quite significant for searching, especially for iterative deepening, in terms of time.

However, the memory consumption of the transposition table is huge since it records every position searched. An alternative to a transposition table is a refutation table, whose space complexity is merely $O(d \cdot b)$ or so. Often used with iterative deepening, it records the principal variation or continuation of each iteration and direct moves of the next iteration. According to [7], using refutation tables improve iterative deepening by roughly 30% in terms of time.

3.2.3. Killer heuristic and history heuristic. Killer heuristic is a method used for dynamically re-ordering moves in a search. At each ply, the best moves, which can cause a cutoff or get an excellent score, are recorded. Based on the thought that a move that is the best move for a position may also be the best move for another position that is similar to the previous one at the same ply, this approach would first search the best moves recorded previously in the hope of making cutoffs. Although it seems to be pretty good, it does not perform very well in practical implementation due to its critical uncertainty.

History heuristic is a generalization of killer heuristic. The judgements of best moves for both techniques are similar. Killer heuristic records only one or two best moves of each ply independently while history heuristic records all best moves of the entire search tree, no matter which ply the move is at. A killer move for killer heuristic has a high probability to be one of the best moves for history heuristic. Also, the history heuristic proved to have a better performance than the killer heuristic.

4. Monte-Carlo tree search

Since 2006 when Monte-Carlo tree search (MCTS) got unprecedented attention for searching game-trees, it has become a powerful technique used for complex AI games, such as Go and chess, and made a big breakthrough in the field of game-trees due to the huge difference between it and traditional Alpha-Beta algorithm [9]. With the use of MCTS and other AI implements, AlphaGo beat professional human Go players, which was a milestone for the whole field. Today MCTS is still under research for further improvements and other inspirations.

4.1. Mechanisms and characteristics

MCTS uses best-first search with a random sampling of the search space. It looks for the best move iteratively. There are four phases that are performed repeatedly in an iteration (Figure 2):

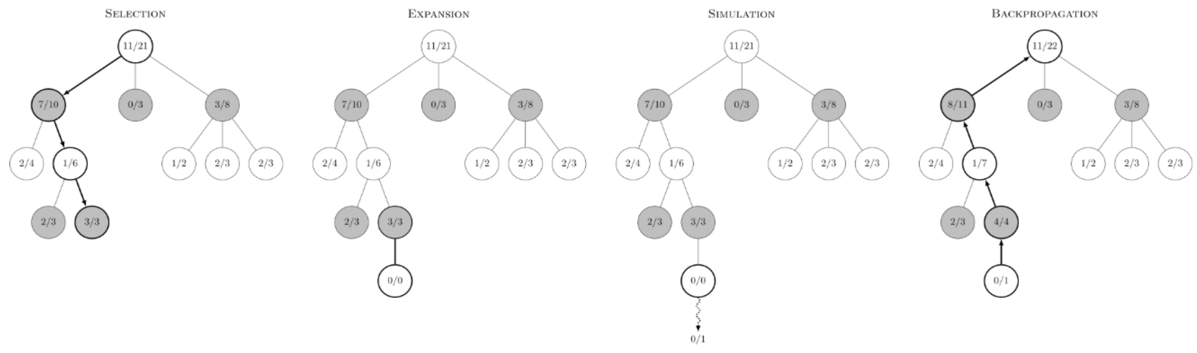

Figure 2. Four steps of monte-carlo tree search [10].

A) Selection: Start from the current root node and search the tree with a bias of visiting more promising moves until it reaches a leaf node, which is the terminal node at the current state but still has potential successors that have not been visited.

B) Expansion: Add one or more successors of the leaf node found in the selection phase to the search tree.

C) Simulation: Perform a playout (also called roll-out), which refers to a process of constantly choosing random moves to reach the very end of the game, from the new node to produce a sample outcome.

D) Backpropagation: Update back father nodes recursively with the simulation result.

Such repetition terminates when the computational limitation, maybe time or memory, is reached and an action would be applied before the next iteration.

In order to achieve the best performance, it is essential to balance exploitation (focusing on superior moves) and exploration (focusing on inferior moves), because there is a big chance that some moves are inferior now but superior when the search goes deeper, which are called "trap states". Many techniques can be used for this purpose, such as bandit-based methods including upper confidence bounds (UCB), which is used to develop the upper confidence bounds for trees (UCT) algorithm, the most popular variant of MCTS.

The first characteristic of MCTS is that MCTS can return results at any time, just like iterative deepening, since MCTS also uses iteration and all information keeps updated. Moreover, the other one is that the tree generated by MCTS is asymmetric because it prefers better moves like principal variation.

4.2. Comparison with Alpha-Beta algorithm and advantages and disadvantages

Unlike the Alpha-Beta algorithm, which needs to determine exact position scores for all nodes, MCTS uses randomization to predict which is better. Alpha-Beta algorithm is based on depth-first search while MCTS is based on best-first search. According to several experiments, MCTS reaches better performance than the Alpha-Beta algorithm and other similar algorithms.

The biggest advantage of MCTS is that it can be applied to search trees without any domain-specific knowledge (human knowledge). That is also the reason why it can make a big success in Go, because the evaluation function in the Alpha-Beta algorithm needs domain-specific knowledge and a position can be evaluated correctly in chess but is very difficult to be evaluated in Go. Furthermore, it turns out that MCTS can also perform well in chess games. And this advantage has been magnified due to the use of convolutional neural networks (CNN) [11].

One obvious disadvantage is that even though there are methods to deal with exploitation-exploration dilemmas, it is still possible that MCTS misses better moves due to "trap states" and this might be the reason for its failures in competing against human players. Another disadvantage is that it is possible that sometimes playouts consume pretty much time because random moves may lead to deadlocks.

5. Conclusion

Different kinds of popular techniques for searching game-trees have been discussed in terms of their mechanisms, advantages, shortcomings, and practical performances which are influenced by time complexities, space complexities, and whether the trees are strongly ordered or random. And they are divided into different categories based on their essences. Brute-force algorithms, pure Minimax and Negamax algorithms, are only used in very simple games. For other complex games, the Alpha-Beta algorithm must be used, usually together with the iterative deepening and transposition tables or refutation tables, sometimes also with the history heuristic. MTD(f) outperforms PVS and PVS outperforms aspiration search. MCTS with AI technology is now widely used and is more used in Go than in any other game.

Currently, some algorithms of machine learning, such as regression and the neural network, have been combined with game-tree search algorithms. Despite they are used with improvement methods, some algorithms are still sometimes unstable and can be enticed by human players to go into losing moves. Future research about game-trees can try to make these algorithms with as much certainty as possible and can define some special human-strategies and transform them into a form that computers can understand.

References

[1] Samuel, A. L. (1959). Some Studies in Machine Learning Using the Game of Checkers. IBM Journal of Research and Development, 3(3), 210–229.

[2] Heineman G T, Pollice G, Selkow S. Algorithms in a nutshell: A practical guide. " O'Reilly Media, Inc.", 2016.

[3] Russell, Stuart J.; Norvig, Peter. (2021). Artificial Intelligence: A Modern Approach (4th ed.). Hoboken: Pearson. pp. 149–150.

[4] Kaindl H, Shams R, Horacek H. Algorithms with and without Aspiration Windows. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1991, 13(12).

[5] Shams R, Kaindl H, Horacek H. Using Aspiration Windows for Minimax Algorithms, (International Joint Conference On Artificial Intelligence. 1991: 192-197.

[6] Elnaggar A A, Gadallah M, Aziem M A, et al. A comparative study of game tree searching methods. International Journal of Advanced Computer Science and Applications, 2014, 5(5): 68-77.

[7] Marsland T A, Campbell M. Parallel search of strongly ordered game trees. ACM Computing Surveys, 1982, 14(4): 533-551.

[8] Tommy L, Hardjianto M, Agani N. The analysis of alpha beta pruning and MTD (f) algorithm to determine the best algorithm to be implemented at connect four prototype. IOP Conference Series: Materials Science and Engineering. IOP Publishing, 2017, 190(1): 012044.

[9] Świechowski M, Godlewski K, Sawicki B, et al. Monte Carlo tree search: A review of recent modifications and applications. Artificial Intelligence Review, 2023, 56(3): 2497-2562.

[10] https://upload.wikimedia.org/wikipedia/commons/thumb/2/21/MCTS-steps.svg/1200px-MCTS-steps.svg.png.

[11] Mastering the game of Go with deep neural networks and tree search. Nature, 2016, 529(7587): 484-489.