Minimax and Monte Carlo Tree Search Implementations for Two-Player Game

Toma Rončević[1], Marina Rodić[2] & Ljiljana Despalatović[3]

1 University Department of Professional Studies at University of Split, Kopilica 5, 21000 Split, Croatia (toma.roncevic@oss.unist.hr)

2 University Department of Professional Studies at University of Split, Kopilica 5, 21000 Split, Croatia (marina.rodic@oss.unist.hr)

3 University Department of Professional Studies at University of Split, Kopilica 5, 21000 Split, Croatia (ljiljana.despalatovic@oss.unist.hr)

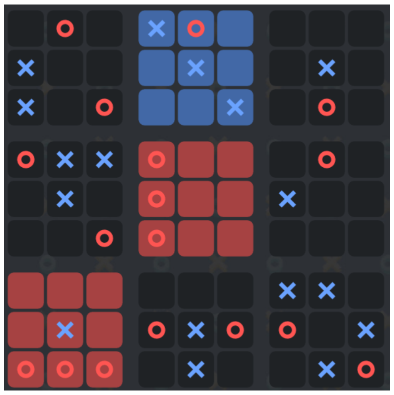
In this paper, we examine practical implementations of two algorithms: minimax and Monte Carlo Tree Search (MCTS). Both algorithms have been applied in the domain of two-player games with different extensions, modifications, and success. In this work, the two algorithms are implemented for a game that is an upgradeof the children's Tic-Tac-Toe game, now played on nine 3x3 boards with additional rules, called Ultimate Tic-Tac-Toe (UTTT). While the original game is trivial, this version is much harder to solve and represents a good use case for more advanced algorithms' implementation. The game was also used in an open competition among several hundred enthusiasts where our minimax approach arrived sixth in the finals. We give an overview of both algorithms, discuss their advantages and drawbacks, and describe their common modifications. We also discuss differences between algorithms, their results, and implementation details, both related and unrelated to our use case.

Keywords: *minimax, mcts, uttt, implementation*

1. Introduction

Games were always an ideal testing ground for different artificial intelligence approaches. Two-player games encapsulate adversary multiagent environments with simple and clear rules while also portraying decision problems of varying difficulty. Ultimate Tic-Tac-Toe (UTTT) is one such game. UTTT is played on a 9x9 grid board that is divided into 9 small 3x3 grid boards, as illustrated in **Figure *1***. In the illustration, a situation in the middle of a game is shown. In this situation, of smaller boards, central and lower-left boards are won by "O", the upper central board is won by "X", while all other boards are still undecided. The overall game is still unresolved since no three small boards in one line are won by either player. Players alternate turns and place their pieces, usually represented by “O” or “X”, in cells of the board. The goal of the game is to win 3 small boards in a single row, column, or diagonal. Each small board can be won by placing 3 pieces in a single row, column, or diagonal, inside the small board. Additional rules require that each player is limited by the opponent’sprevious move. A player can only place their piece on one of thesmall boards if the opponent played inside any small board on the respective position. For example, if the opponent played in the upper left corner of any small board, the other player can only place their piece anywhere in the upper left small board. An exception to this rule is if the small board is full or already won by one of the players. In that case, a player can play on any small board that is not resolved yet. The first player can also play anywhere during the first move of the entire game.

Figure 1 Visualization of Ultimate Tic-Tac-Toe game



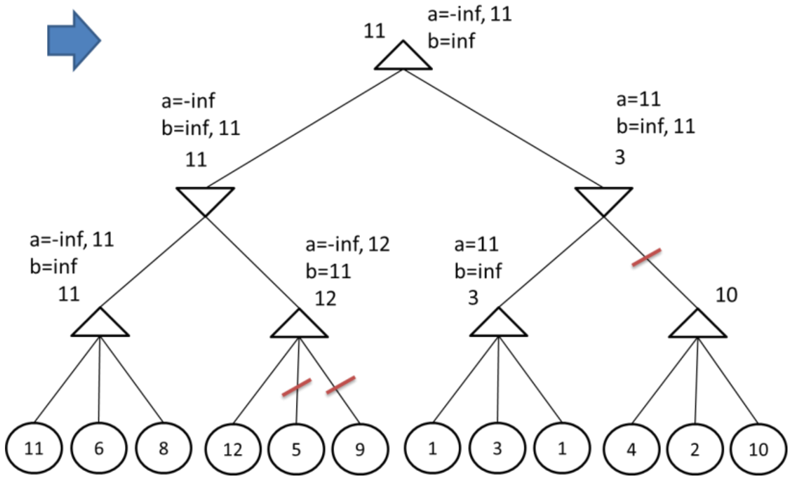
This game represents a deterministic environment with complete information and a known state-space transitional model. This means we can try to solve it by searching formed the state-space. We can describe the difficulty of this game by estimating the number of different states the game can take which need to be considered when choosing the next action. This number (size of state-space) can be approximated noting that the average game lasts until some 2/3 of the board is filled (54 cells) and the average move allows for about 7 different actions. The total number of states is approximately 7^{54}. For comparison, chess is estimated at 35^{100} states, while checkers is estimated at 8^{47} states (Russel and Norvig, 2010, 175). This problem is still too big to be searched exhaustively in a reasonable time, even if our estimations are slightly pessimistic.

At each move, the agent has to decide the best action to play. If state-space was tractable, the agent could find the best actions for both players (so-called Nash equilibrium) and follow them. Since this is virtually impossible under reasonable constraints of time, the agent has to rely on partial search and heuristics. Two algorithms that can deal with this kind of environment are minimax and Monte Carlo Tree Search. Both of these algorithms were applied to similar games and were used for almost all top-ranking agents in the mentioned competition.

2. Minimax

This algorithm was invented within the game theory field and is one of the most researched algorithms for chess and similar games. It was used by countless commercial and amateur chess engines, culminating in the victory of Deep Blue over world chess champion Kasparov in 1997 (Goodman and Keen 1997). Until recently with Alpha Zero in 2018., it was, and some would argue still is, the best algorithm for chess. In essence, the algorithm executes a recursive depth-first tree visit, aimed to determine the value of some game state (current game position). The tree consists of child states from each game state with the current state at the root. Child states are resulting states of each legal action in a given state. Appropriate updates of child state values are executed and the true value of the root state is determined at the end of the tree visit. In two-player games, we consider one player a “max” player and the other a “min” player and updates depend on whose turn it is. If “max” is about to play in any state, he will pick a child state (or action that leads to it) with the highest value, while the “min” player will pick a child with the lowest value. The value of each child state is determined recursively by examining its child nodes. This recursion would stop once the terminal state (like checkmate position) is reached where appropriate values are assigned. We would assign a large positive value for terminal states where “max” has won and a large negative value for “min” player. This approach would determine the value of the root state as Nash equilibrium, a value that neither “min” nor “max” can improve upon by choosing different actions. This basic algorithm is hopeless in the face of a large state-space and several standard improvements are almost always added to it. The most common is alpha-beta pruning which mathematically achieves the same result as the basic minimax algorithm but at a largely reduced number of nodes visited. This upgrade is independent of the use case and is very simple to implement. An illustration of this algorithm is shown in **Figure *2***. In this case, leaves are visited from left to right and each leaf is evaluated by a score heuristic that gives a numeric value to a state and will be described later. Internal nodes are min (the triangle pointing down) and max nodes (the triangle pointing up). "a" and "b" represent alpha and beta values that start from infinite values and are replaced by better values during the search. During the search, some branches of the tree will be cut-off if alpha reaches or passes the beta value. This cut-offguarantees that in that branch there can be no possible value that would be acceptable to both “min” and “max” players. So one of the players would certainly want to avoid that line and it can be safely cutoff.

Figure 2 Illustration of minimax depth-first visit to state-space search tree with alpha-beta pruning

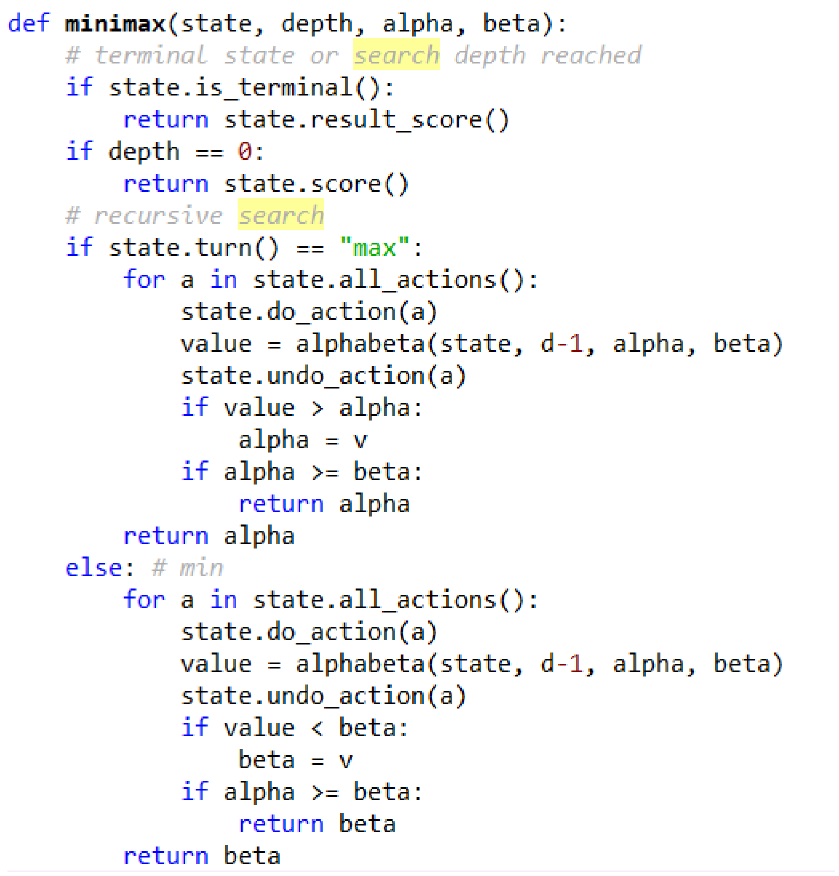


Another important part of a typical minimax algorithm is the score heuristic. Since the search tree is usually far too large to be searched exhaustively, the search is usually limited at a certain depth. When the leaf is reached, and if it is not a terminal state, the state is evaluated by some heuristic that estimates how likely is a victory for the min or max player. This estimate consists of a single number within victory values for each player and is then used by minimax in the same way as it was a result of subtree visit. This enables to reduce exponential growth of subtree visit to constant time O(1) calculation but at the cost of precision and mathematical correctness.

Besides this heuristic, minimax algorithms usually rely on other heuristics where the most important one is heuristic for move ordering. Alpha-beta pruning can prune a large number of nodes if move ordering prioritizes good moves. This results in a large reduction of branching factor, i.e. average number of child nodes examined at each node. In the case of chess, best engines, with many clever improvements, were able to reduce the branching factor down to around 1.5 – 2, so less than 2 child nodes were examined on average and all but the first move were cut-off.

Listing (**Figure *3***) shows the pseudocode of the entire recursive procedure with alpha-beta pruning, score heuristic, and move ordering.

Figure 3 Code for a typical minimax search function.



```
def minimax(state, depth, alpha, beta):
    # terminal state or search depth reached
    if state.is_terminal():
        return state.result_score()
    if depth == 0:
        return state.score()
    # recursive search
    if state.turn() == "max":
        for a in state.all_actions():
            state.do_action(a)
            value = alphabeta(state, d-1, alpha, beta)
            state.undo_action(a)
            if value > alpha:
                alpha = v
            if alpha >= beta:
                return alpha
        return alpha
    else: # min
        for a in state.all_actions():
            state.do_action(a)
            value = alphabeta(state, d-1, alpha, beta)
            state.undo_action(a)
            if value < beta:
                beta = v
            if alpha >= beta:
                return beta
        return beta
```

Another practical consideration for the minimax algorithm is the time constraint. Typically, the engine or player will have limited time to determine and play their move. This means that the time budget for executing minimax is limited, so the search depth will have to be limited. It is hard to anticipate how long would minimax execute at some depth and minimax can't guarantee the correct results if the visit of the entire tree didn't complete. This can be solved with iterative deepening. This modification runs a minimax algorithm at a successively increasing depth. It allows minimax to stop the search at almost any time and use results from previously completed searches at a lower depth. The drawback is that the search is repeated each time from scratch. This can be alleviated by memorizing values and best moves for most states and retrying it as the first move at repeated search. Hopefully some values could be reused in different tree branches and the cut-off would occur earlier to compensate for time lost in the repeated search. In practice, this works well and is a standard part of chess engines.

To memorize the best moves for various states, as well as their value, a large hash table is usually kept in memory. This is a common way to have O(1) access to previously searched states. Hash keys are calculated using Zobrist hashing keys (Zobrist, 1970), usually alongside game mechanics for executing actions. States in this table are updated or replaced at collision since those states are usually from earlier stages of the game.

Another known problem is the so-called effect of the horizon where search at a fixed depth can miss obvious strong moves only a few depths further. This can be mitigated by using a quiescence search instead of a direct cut-off with a scoring function. This search is usually done as a simplified version of the full minimax search and considers only "strong" moves. In chess, strong moves are usually all captures and sometimes check moves and promotions. This enables a varied search depth that will be deeper in tactically interesting positions.

Since move ordering is crucial for alpha-beta pruning, other approaches are also often used, like killer moves or history moves. Both of these approaches try to keep track of the quality of moves in other parts of the search tree and over different game stages. A move is generally considered good and moved up in the move order, if it caused a cut-off somewhere else in the search tree.

The minimax algorithm has the advantage of being superior to MCTS for deterministic games with a moderate number of actions, demonstrated on numerous chess engines. It is also widely researched through chess with countless improvements and variations. The principal drawback is that, for reasonably large problems, it relies on a score heuristic. This heuristic has to be designed by experts and its bonuses have to be balanced to cover a large number of game situations. Usually, there are also many special situations that also have to be included. Its correct design is crucial for the decision quality of the entire approach, but it is not always clear how to correctly design it and balance it. Even with a carefully designed heuristic, it is likely that there will be situations where its estimate can be incorrect and one can only hopethat the search part of the minimax algorithm will compensate for this error.

3. Monte Carlo Tree Search

This algorithm relies on randomness to evaluate states and guide the state-space search in a sort of "best first" manner. For two-player games, it slowly builds the search tree in its memory while simulating game playouts. It is appropriate for stochastic environments, i.e. games with uncertainty, by design, but it can also be applied to deterministic environments. Successful applications, usually against a minimax algorithm, were made in games of Go and Backgammon. The former game has a large number of actions at each move (>100), while the latter game has an inherent randomness from using game dices.

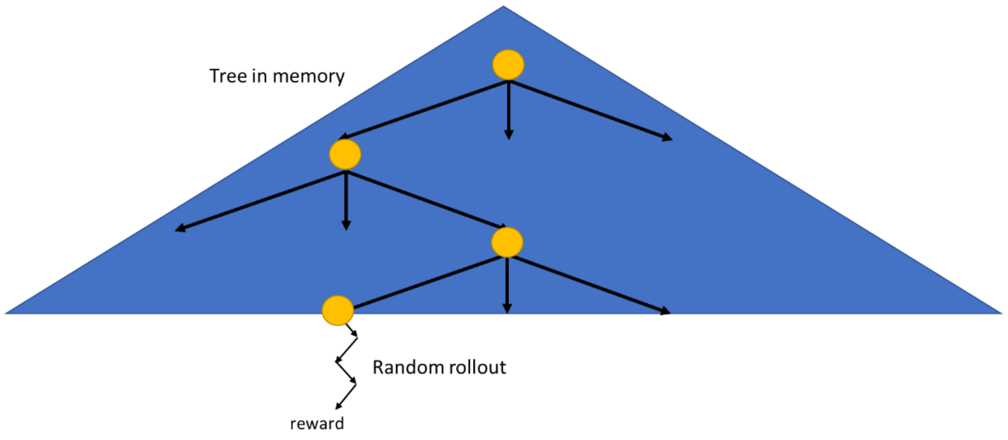
In essence, the algorithm is iteratively building a tree in the memory node by node, guided by results of random rollouts outside of the tree built up until that moment. At each iteration, the algorithm starts from the current state at the root of the tree and descends until it reaches a leaf.

Then it adds another child node to the leaf and evaluates it by the random rollout. The entire process is illustrated in ***Figure 4***. In the illustration, the blue area is a tree built up to this iteration. Select phase descends through the tree in a memory built on previous iterations, then it expands a single new node, evaluates it with a random rollout, and backpropagates reward up to the root in update phase. The core algorithm consists of four successive phases:

- Select
- Expand
- Random rollout
- Update

At each iteration of the algorithm, all phases are executed in sequence.

Figure 4 Illustration of MCTS algorithm.



Select phase descends from the root to a leaf. At each node, a single child has to be selected and followed. Strategy for selecting the child can be anything, but the current standard is using Upper Confidence Bound (UCB1) formula, as first used in (Kocsis and Szepesvári, 2006), and usually referred to as the UCT version of MCTS:

$$ucb_a = c\sqrt{\frac{ln\, n}{n_a}}$$

This formula assures a balance between so-called exploration/exploitation dilemma, i.e. trying inferior actions that might become superior after enough iterations. In the formula, n is the number of times this state was traversed during different iterations, n_a is the number of times action a was attempted, and constant c determines how often will exploration be chosen instead of exploiting the current best actions. A large constant will disperse selected actions and make the algorithm more like a breadth-first search, while a small constant will concentrate on the best actions (exploitation) and make the algorithm closer to the best-first search. Value ucb_a is simply added to the q value of the node when choosing the next action during the select phase. Action with the highest value is usually selected, but in this case, with two players, the one with the lowest value is chosen for the opponent's actions.

Expand phase creates a new node and adds it to the tree in memory. Usually, some initialization based on the game state is also done here, such as generating all possible actions. Here we also initialize values for *q* (sum of child values) and *n* (number of times a node was traversed) that are later used in the UCB1 formula.

The random rollout starts from a new node and plays out the entire game using random or semi-random actions for both players. This phase produces numerical rewards depending on the game result. The result is often a simple 1 for a won game, 0 for a lost game, and 0.5 for a draw.

The update phase adds a reward to all nodes in the tree that were visited in the select phase. Counters *n* for the UCB1 formula are also increased by one, while *q* values are increased by the numerical reward. To be noted here is that the *q* value in the UCB1 formula is actually the expected value obtained by dividing the sum of child value *q* with the number of traversing that node *n*.

The entire procedure can be described by pseudocode in listing (**Figure 5**).

Figure 5 Code for typical MCTS recursive function

```
def mcts(state, node, tree):
    if state.terminal():
        return state.result()
    elif node is None: # out of tree (rollout phase)
        action = random_choice(state.actions())
        state.do(action)
        return mcts(state, None, tree)
    elif node in tree: # in tree (select phase)
        action = node.choose_action_by_UCB1()
        state.do(action)
        score = mcts(state, tree[state], tree)
        node.update(score, action) # update phase
        return score
    else: # expand phase
        tree[state] = Node(state)
        return mcts(state, None, tree)
```

Same as for the minimax, different improvements have been proposed. Some of them try to improve the select phase by artificially initializing values for children based on a certain heuristic, instead of using usual zero values. This method is referred to as “warming up” or “nodes pre-warming” in literature. Other improvements use results of the search in other lines of the game to affect values for children in the select phase, so-called “history” values. Similar improvements can be used in the random rollout phase where the randomness of an action can be affected by heuristics or success of actions in other lines of the search tree. Close to game endings, the algorithm can limit the action selection only for deterministically winning actions by using bounds similar to the bounds for alpha-beta pruning.

The main advantage of the MCTS approach over the minimax approach is that it requires no scoring function. This function is basically replaced by random rollouts that estimate leaf values. Also, at any time, the search tree at the root node has an estimation for the best move. The algorithm can be stopped practically at any time and the move played. Another smaller practical advantage is that no inverse mechanic (for taking back moves) needs to be implemented since each rollout only requires a single copy of the current state and never takes back moves. The main disadvantage is that the actual search tree has to be built in memory so, in addition to the time limit, there is usually a memory limit for algorithm execution. On the other hand, this allows to reuse a large part of the built tree for the next moves, i.e. simply continue the search from one of the root's child nodes, while the rest of the tree can be removed from memory. Randomness tends to require large numbers of rollouts to have a good estimate of the best action. In practice, this can be comparable to minimax's need to search a large number of game states.

4. Implementations and results

Both algorithms were first implemented in Python programming language and then translated and further developed in C programming language. As expected, both implementations in C were several times faster than Python implementations. State representation was done using so-called bitboards, an approach often used in chess engines. In chess, the entire board can be represented in parts by many integers encoded on 64 bits, where each bit represents one square on the board and each integer usually represents the position of one or more pieces of the same type. This approach allows for the speedup of some parts of chess mechanics by using bitwise operators and avoiding loops. The same approach was used in our implementation where each smaller board was represented by two 32-bit integers, one for each player. Another two integers were used for the big board in the same manner. Since boards have only 9 cells, only the first 9 bits were used.Full integers were still used since changing integer type didn't bring any improvement. This also allowed for pre-calculating all possible board positions. By keeping these pre-calculated positions in a lookup table, mechanics for checking wins and similar were done in single access inside a table.

In minimax implementations, several extensions typical for chess engines were implemented but only some were kept in the end since not all showed clear improvement in our tests. Here we mention only those that made it into the final implementation. Iterative deepening was used in combination with a hash table for memorizing the best moves from the previous search depth. A quiescence search was used only where moves that immediately win a smaller board were considered. This allowed the algorithm to discover certain forced lines, mitigating the horizon effect. History moves approach was used where 9x9 array counted how often playing ina certain cell produced cut-off during the search. These values were aged during the game, i.e. their value was halved at each move. In the end, move ordering was: hash table move, small board winning moves, and then history moves in order. Static evaluation of the game state was done along the line of a single board evolution. Value for any small board was proportional to how many pieces were lined up. Each board evolved from a “clear” state, over a “threat” state with 2 pieces lined up, up to a victory for one player or a “blocked” state where neither player can win anymore. Exact values for these states were determined with lengthy testing. Even genetic algorithm-like selection was attempted but without much success.

In our MCTS implementation, only two smaller modifications were made to the original algorithm, while most of the work was invested in speeding up random rollouts. Different attempts with pre-warmed values and fiddling with the UCB1 constant did not yield any improvement so the final version didn't use any pre-warming and the constant was fixed at 0.6. One small modification was to expand the phase. In our version, a new node was not created at each iteration but only at every third iteration. This achieved a higher number of rollouts per second while keeping nearly the same quality of q estimates. Another modification was in the random rollout phase where, if a winning move was available, it was played immediately. This achieved that in the random rollout phase, no clear victory was missed.

The comparison was done over thousands of games for many different versions of both implementations. In the end, final versions were compared on one thousand games and the minimax algorithm was found to be superior over the MCTS algorithm by 56% versus 44% of victories (draws counted as half point for each).

5. Conclusion

The largest problem in developing these implementations was testing. There is little guidance in the literature on how to compare two engines, besides directly confronting two different versions. This has a serious drawback, the engine can become overly specialized in winning against some other particular version, while general improvement is not guaranteed. The problem can be and was mitigated by testing new versions against several older versions. This, of course, incurs extra time cost so it was done seldomly, usually only for larger modifications.

For this use case minimax algorithm had the advantage because it is superior for deterministic games with a fairly low number of actions per state. On the other hand, UTTT often has long semi-forced lines, especially in later game stages. This should give an advantage to MCTS that relies on rollouts till the end of the game and has no problem with the horizon effect. However, we believe that our quiescence implementation mitigated this flaw in our minimax implementation and overcame the horizon effect.

In the end, the comparison showed that our minimax implementation was clearly superior to our MCTS implementation. This result was validated by the fact that almost all other top engines that disclosed their approach were also minimax engines. It should be noted however that authors had much more experience with the minimax algorithm which may have influenced the results. On the other hand, both implementations could be improved. Simpler features for minimax score heuristic might have achieved even better results, while prewarming of q values in MCTS implementation should be able to take some advantage of game particularities.

B I B L I O G R A P H Y

1. Goodman D.; Keen, R. (1997) *Man versus Machine: Kasparov versus Deep Blue*, H3 Publications
2. *Github repository*, https://github.com/toma78/uttt_minimax and https://github.com/toma78/uttt_mcts
3. Kocsis, L.; Szepesvári, C. (2006) *Bandit based Monte-Carlo Planning*, ECML-06, LNCS/LNAI 4212
4. *Repository on Chess Programming*, https://www.chessprogramming.org
5. Russel, S.; Norvig, P. (2010) *Artificial Intelligence: A Modern Approach*, Prentice Hall
6. Zobrist, A. (1970) *A New Hashing Method with Application for Game Playing*. Technical Report #88, Computer Science Department, The University of Wisconsin, Madison, WI, USA. Reprinted (1990) in ICCA Journal, Vol. 13, No. 2

Summary

Minimax and Monte Carlo Tree Search Implementations for Two-player Game

In this paper we examine practical implementations of two algorithms: minimax and Monte Carlo Tree Search (MCTS). Both algorithms have been applied in the domain of two-player games with different extensions, modifications, and success. In this work, the two algorithms are implemented for a game that is an upgrade of thechildren's Tic-Tac-Toe game, now played on nine 3x3 boards with additional rules, called Ultimate Tic-Tac-Toe (UTTT). While original game is trivial, this version is much harder to solve and represents a good use case for more advanced algorithms' implementation. The game was also used in an open competition among several hundred enthusiasts where our minimax approach arrived sixth in the finals. We give an overview of both algorithms, discuss their advantages and drawbacks, and describe their common modifications. We also discuss differences between algorithms, their results and implementation details, both related and unrelated to our use case.

Keywords: *minimax, mcts, uttt, implementation*