Adversarial search

How to avoid loosing

Seminar report by

Sebastian Einsiedler

Artifical intelligence for games

Prof. Dr. Köthe

June 23, 2019

This report is an extended summary of the fifth chapter of "Artificial intelligence: a modern approach" [1]

Contents

1 Introduction

Adversarial search is a solution algorithm for turn based games, with perfect information assuming perfect play for the opponent. The game is modeled as a decision tree. We take the path that gives the deciding player the maximal reward, which in case of a zero-sum game is the minimal loss, hence we try to avoid loosing.

2 Min-Max algorithm in a practical example

Let's start with a simple example. A zero-sum game with one move and two player resulting in two plies. We have to cups and four Cookies. The first cup contains three cookies the second one cookie. You start and can move one cookie from one cup to the other, the I choose a cup. What would you do?

Assuming we both have a interest in winning the cookies, you would likely transfer a cookie from the first into the second cup, since you assume I would take the cup with more cookies in it. So you predict my move to plan your move. This is the center of adversarial search.

Let's start to formalize things a little bit by turning this game into a decision tree as shown in Figure 1.

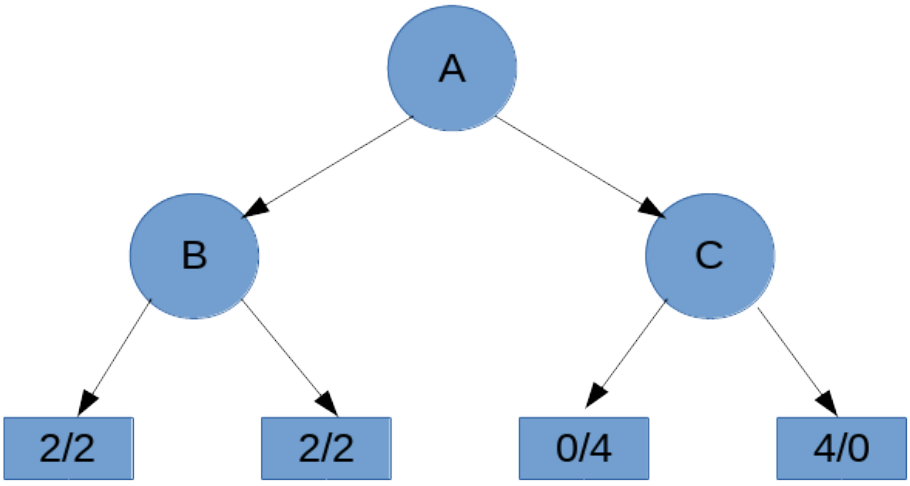

Figure 1: Cookie Cup game, with non terminal states displayed as circles and terminal states as boxes with payout for each player.

Here we have a starting state "A", after which you transfer the cookies between the cups, resulting in two intermediate states "B" and "C". Leading to two terminal states each, depending on my choice. The left number in the terminal state gives your win, the right one mine.

To get a more algorithmic perspective we normalize the payout to one scalar variable between 1 and 0. This is possible, because it is a zero-sum game. So four cookies correspond to the maximal possible reward. So if you get four cookies the payout is 1. If I get four Cookies the payout is 0. So you want to maximize the payout and are therefore called player "MAX", while I try to minimize your payout and are therefore called player "MIN". The new tree is shown in Figure 2.

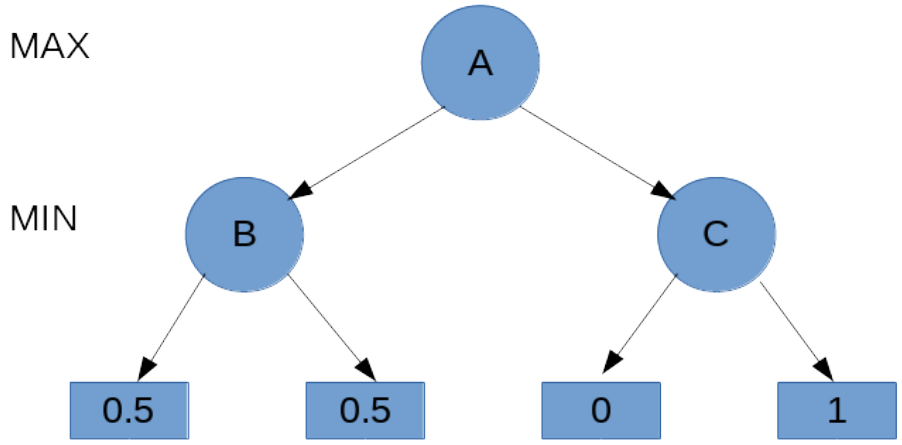

Figure 2: Cookie Cup game, with non terminal states displayed as circles and terminal states as boxes with normalized payout.

To determine his move player "Max" looks at the value of the resulting states or the children of the initial state "A". In our cases "B" and "C". Their values are determined by the move of player "MIN". We assume player "MIN" will choose the most favorable outcome for him or herself. To determine this outcome he looks at the children of the state he or she is currently in. This is a recursive process, that continues until a terminal state is reached. For us this is the Case now. So we can walk the recursion back up. Player "MIN" will, according to our assumption of perfect play, choose a 0.5 terminal state for "B" and a 0 terminal state for "C". So the value of A is 0.5 and the value of B is 0.

Player "MAX" will now choose the state or node with the highest value "A". So the payout of the game is 0.5, as shown in Figure 3.

Please note that we did not only figure out the perfect strategy for player "MAX", but also for player "MIN". Our solution actually relies on player "MIN" choosing the perfect strategy.

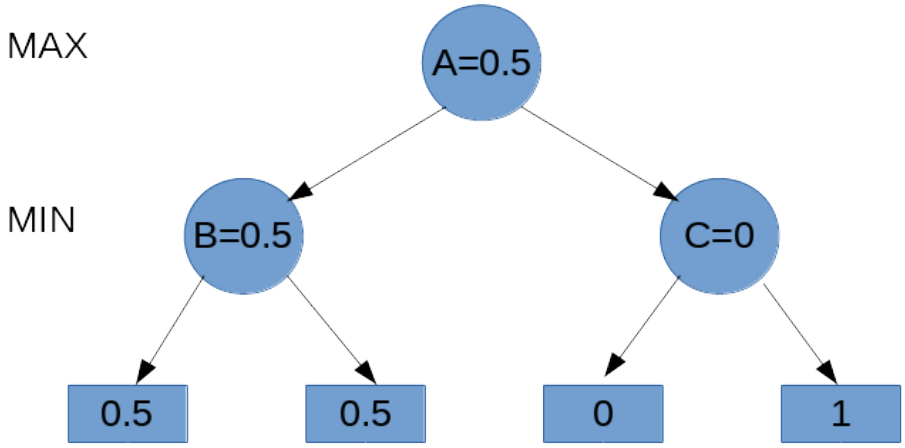

Figure 3: Cookie Cup game, with non terminal states displayed as circles and terminal states as boxes with normalized payout.

3 Alpha-beta pruning

Currently our game tree has three nodes and 4 leaves. This is of course not a game we would like to solve with a computer. We would probably attempt to solve a game, that takes a few dozen moves, and has options in the double digits for each ply. This gives us a rather large game tree. So we do not want to go true the whole tree recursively. One way would be to use "Alpha-Beta-Pruning", which was also described by Fabian Jaeger in this seminar [2, p. 6].

The general concept is to abandon a part of the tree, if it will not be reached by optimal play. To do this we abandon a part of the tree, if the player doing the ply already has a better option. If we take Figure 3 as an example. We would not need to explore the leaf "1", because we know that the minimal possible outcome for this node is 0. So Player 1 would not choose it.

If the two leaves would have been ordered differently, we would had to explore the whole tree. So ordering is important. Assuming perfect ordering we only need to examine the square root of all nodes [1, p. 169] This means we need either half the time or can search twice the depth [1, p. 169]

4 More than two players and chance

Many games have more than two players. To adapt to this reality we have to abandon our scalar payout and return to a tuple payout. Each entry in the tuple is the payout or utility for a player. The tree is resolved similar to the method in section 2. Every player selects in their ply the action leading to the

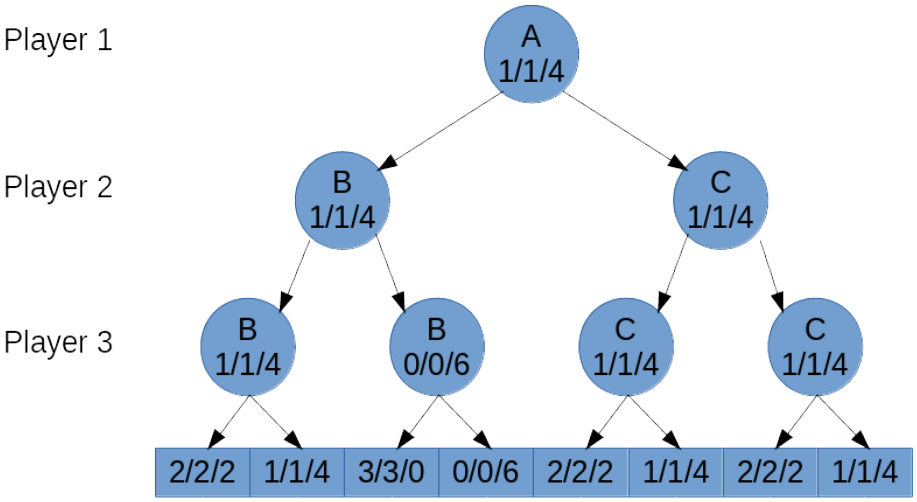

Figure 4: 3 player cookie cup game, evaluated assuming perfect play.

highest payout for them, assuming the other players will do the same.

Let us expand our example from section 2 for three players. We again have two one with two cookies, one with four cookies. Now Player 1 and 2 move one cookie around after each other. The Player 3 chooses one cup and gets the cookies in it. The cookies in the other cup are split equally between Player 1 and 2. We now get a different tree shown in Figure 4.

This game seems to strongly favor Player 3, if everybody plays perfectly. But what if Player 3 doesn't? Unfortunately we can not really simulate a non perfect player with this algorithm. But we can try to approximate non perfect play by assuming the player will choose a action at random. So the actions of Player 3, are assigned a probability as shown in Figure 5. The value of each node is now no longer the best choice for Player 3, but the expected value (sum over all values multiplied by their probability). Now the game seems to favor Player 1 and Player 2.

This would also be the model for a game of chance for 2 Players, if we would adjust the utility vector. Player 3 would then be the random factor, like a dice roll or a card drawn. So we can also use adversarial search for games of chance.

5 Eval-Funktion

The beginnings of evaluation functions, were in this seminar described by Jacqueline Wagner [3].

A evaluation function does in general assign a utility to a game state and determines inf the game is finished. If the game is not finished the utility value

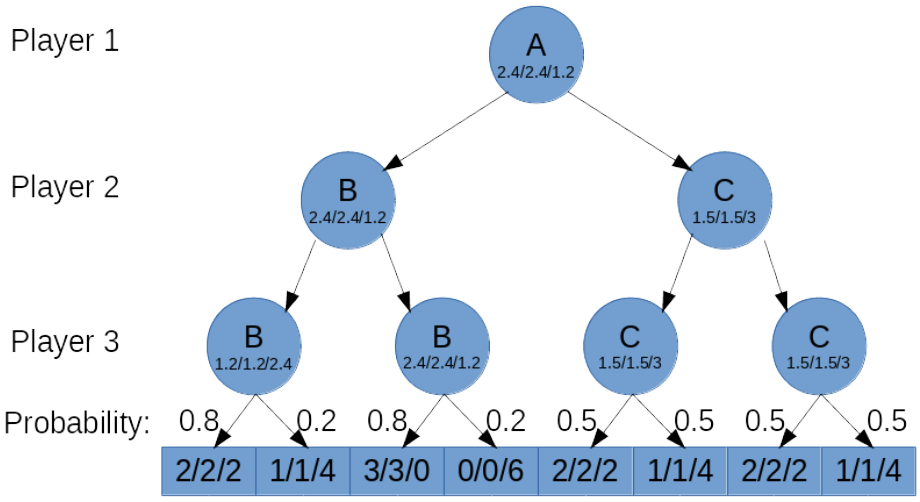

Figure 5: 3 player cookie cup game, evaluated assuming Player 2 chooses actions at random.

is an estimation, and could be based on certain game features, like number of pieces on the board in chess.

Evaluation functions can be very complex, but should full fill this criteria [1, pp. 171-172]:

- The final states should be ordered by the evaluation function, in the same way, as the utility function orders them. If this criteria is not met, a win could look trough the evaluation function like a loss, which would lead to suboptimal results.
- It should be fast. Otherwise we could just keep searching to the end of the tree.
- The evaluation function should correlate with the chance of winning. Otherwise it will guide us into a loosing position.

We can use this to modify Alpha-Beta-Pruning, described in section 3. Now we explore the tree to a certain depth first before we call the evaluation function. This could allow us to prune away node "C" in Figure 3, if the evaluation function gives us a correct estimation.

We could also use the evaluation function for forward pruning. Forward pruning is a greedy algorithm that only considers the best option(s). So we apply the evaluation function to all children of a node and explore the n best ones. If we choose $n = 1$ we call it beam search, because the exploration path is a "beam" down from the start node down to a terminal state.

Of course it is not guaranteed that the estimation is correct or even stable which might lead to the Horizon effect described in subsection 5.1.

5.1 Horizon-effect

Using a evaluation function and exploring the tree only partially leads to the Horizon effect, where a negative consequence is removed by a ply by moving it out of the explored part of the tree. A good example is a situation, in which a piece will certainly be captured in 6 moves, but our program only searches the tree to a depth of 3 moves. So it will waste tree moves attempting to save the piece, until it realizes the futility of this attempt.

To avoid this we can look explore more ply, if a single move is better than all other moves. Hoping the negative consequence is discovered after some additional plies. Since we do this only for a small number of nodes the tree does not grow significantly.

6 Lookup

During a game common situations may occur. Many games have typical opening or ending situations. For example a chess match always starts in the same situation, with all pieces arranged in the same way. We could now explore the hole game tree to find a good opening move, or we just play what experienced chess-players advice as a first move. This saves us a lot of computation time.

The same can be true for the last plies in a game. Assuming the game looses complexity during its progression, an assumption true for chess, because pieces are removed, we will reach a states or nodes, with relative small subtrees. This can be solved in advance and stored for later use.

Russel and Norvig use a chess endgame situation as example [1, p. 176]: In this situation one player has a king, a bishop and a knight left, the other player only a king. They conclude that there are 3,494,568 different possible positions for this pieces on the board. Now the goal is to find a perfect solution for all this positions. So first we sort out all terminal positions and mark them as won, draw or loose. (In our case the player with only one king can not win, so there are only two options.) After looked at all entries, we use a inverted minmax-search. Going upwards from the solved positions to the unsolved and marking, them accordingly. Now we can shorten the game tree, as soon as we discover one of this situations, by assigning the node the value specified in the table.

Now solving such endgame situations becomes a memory and no longer a computational problem. More memory means we can have more and more

complex situations already solved at hand. It is a good idea during the reverse minmax search, to save the origin of the discovering node, allowing to use the table instead of tree search to finish the game, further conserving resources.

7 Incomplete information

Many modern games feature incomplete information or a fog of war. To account for this we have to keep track what could have happened, during the all the game. So every time our opponent makes a move we add expand the number of possible scenarios. We eliminate this scenarios once they are shown to be false.

A perfect strategy should lead to a win not only for all possible moves of our opponent, but also in all still possible scenarios. So we would have to force us trough a larger tree. During this we should also consider that we might get lucky and achieve an accidental checkmate. If we make assumptions about the way our opponent moves like in section 4, we can calculate the chances to get such an probabilistic checkmate, and play towards the highest chance.

It should also be noted that information is important here. Knowing the positions of pieces of the opponent, simplifies the game greatly. An easy way to reveal the position of the pieces, is to play in a predictable manner, like perfect. A player choosing the "perfect strategy" will reveal the positions of his pieces to his opponent by making predictable moves. So it is important to play non-perfect moves from time to time, because the opponent will not anticipate them and be caught of guard.

8 Conclusion

Adversarial search is a method well suited for turn based games, with perfect information and no randomness involved. It assumes perfect play form the opponent, leading it to ignore low risk high gain opportunities, if one outcome is unfavorable as it assumes this will be the outcome chosen by the opponent. It will also encounter problems if information is unknown or random factors start to dominate the game. It is in general a brute force method to solve games.

References

[1] S. Russle and P. Norvig. *Artifical intelligence: a modern approach*. Pearson, 2009.

[2] F. Jaeger. *Checkers Solved*. Seminar: Artifical Inelligence for Games 2019. University of Heidelberg.

[3] J. Wagner. *Programming a Computer to play Chess in the 1950s*. Seminar: Artifical Inelligence for Games 2019. University of Heidelberg.