Seminar lecture – elaboration

Topic:

Search strategies for two player games.

Entered by:

Tilman Renneke

Lecturer:

Prof. Dr. Sebastian Iwanowski

Fachhochschule Wedel

Outline

1. Motivation, why study Games?

Games are essentially a continuing succession of decision-making problems. In many cases the rules of a game are defined in way that can be easily implemented on a computer. These two property's make games a very good research opportunity for general decision-making. Since games like chess are very well researched by humans we also have a very good basis to test the quality of our algorithms.

2. Games

As the title of this presentation implies, we are only gonna look at two player games. In games with more than two players we would have two consider strategies of two or more player allying.

In addition to that the games have to be zero sum games, this means that sum of the utility for both players is always a constant value[1] for Example in chess:

Win white:

white = 1, black = 0 → white + black = 1

Win black:

white = 0, black = 1 → white + black = 1

Draw:

white = 0.5, black = 0.5 → white + black = 1

1 Russel and Norvig p. 162

an example for a non zero sum game would be a classic soccer league:

Win home:

home = 3, guest = 0 → home + guest = 3

Win guest:

home = 0, guest = 3 → home + guest = 3

Draw:

home = 1, guest = 1 → home + guest = 2

There are two main reasons why we will only consider zero sum games. The first one is that in a non zero sum game like in the soccer example it could be beneficial to not play perfectly. What is meant by that is for example in soccer if both teams know that they will always draw, than it would be theoretically better for them to win by turns and therefore getting three instead of two points out of two games.

The second reason is that in a zero sum game we can always calculate the utility for the second player with K – P1 = P2, where K is the constant value of the game and P1, P2 are the utility values for both players.

Another criterion for our games is that they have to be turn based, meaning that there is a well defined time during which we have to make our decision. During this time the decision making must not be influenced by the opponent. This means that it might be possible for both players to make their moves at the same time but without knowing what the opponent is doing.

As already mentioned games can be easily described by algorithms. One variant of a general interface for two player games is the following.

GameState S_0 initial game state, this represents the initial positions of each piece on the board and the initial value of all meta information, for example in chess the number of turns since the last pawn move.

Player player(GameState s) Returns wich players turn it is.
Action [] actions(GameState s) Returns every leagal move for a given position.
GameState result(GameState s, Action a) Represents the transition model of how to get from one game state to an other using a specific move. The action a must be a legal move for the game state s.
Boolean terminalTest(GameState s) Returns true if we have reached a game state at wich the game end's. These game states are alsow called terminal nodes.
Float utility(GameState s, Player p) Returns the utlity at a terminal node for a given player.[2]

3. Mini-Max Search

The minimax algorithm is the most basic approach for two player zero sum games.

It utilizes the property of zero sum games that maximizing the utility for one player is equivalent to minimizing the utility for the other player. Therefore the utility function only need's to return the value for one player. This is usually the starting player who is then also called the maximizing player while the other player is the minimizing player.

2 Russel and Norvig p. 162

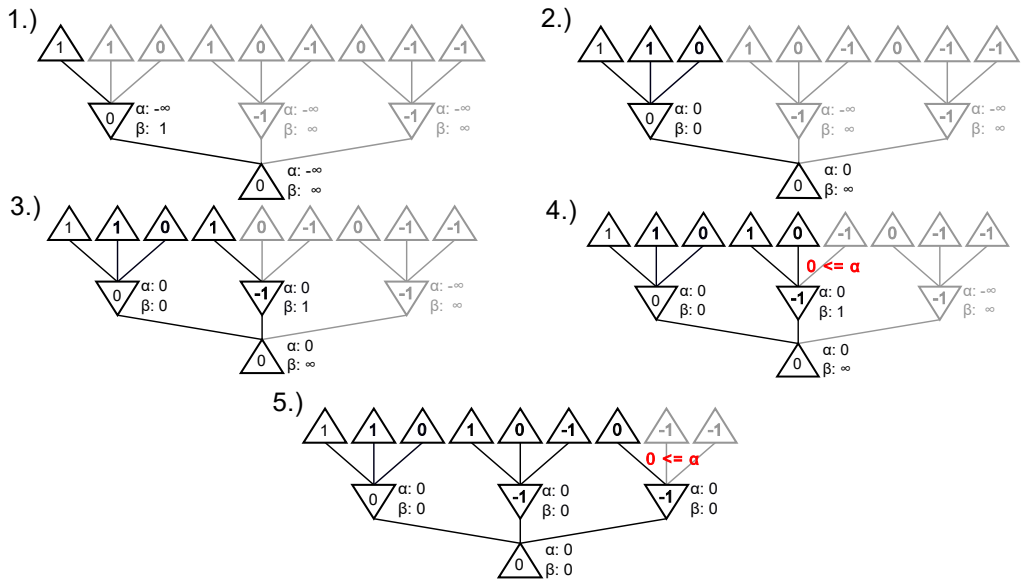


Figure 2: Alpha beta pruning example

α : best value for max.

β : best value for min.

1. α and β are initialized with ± ∞. In the first min position β can be set to 1.

2. Since α is still -∞ where can't be a cutoff in this min node.

3. Since we have completely evaluated the first min node the β value at the max node at the root is set to 0 since when we would pick the left move that is the best value that max is guaranteed.

4. Because the α value is 0 an α cutoff happens. If we would order the moves differently the cutoff could have happend earlier.

5. A second α cutoff occurs, this time it could not have happend earlier.

Since the minimax algorithm searches through each possible game state it's complexity $O(n^m)$. Therefor the minimax algorithm is essentialy a brute force solution.

3.1. Alpha beta pruning

The alpha beta pruning is based on the minimax algorithm. It introduces two new parameter's alpha and beta. They represent the best value that the maximizing and minimizing player can reach. If a terminal node is reached where the utility is worse or equal to alpha / beta the search is cutoff. This can be done because we know that the other player will never make the move leading to this game state because he has already found a better move. This pruning strategy reduces the complexity of the search to $O(n^{m/2})$ with optimal ordered moves. As figure 2 shows the effectiveness of this approach is dependent on the order in witch the child nodes are evaluated. With randomly ordered moves this approach has a complexity of $O(n^{m3/4})$. To optimize the move ordering a heuristic can be used to order the moves. This can't never achieve a perfect sorting though because otherwise this function would be already returning the perfect move for each game state.[3]

3.2. Imperfect decisions in Games

So far we have discussed algorithms for games which solve the games and therefore will always result in perfect play. Since most games are to complex to be solved in a reasonable time an other approach is needed.

One solution is to replace the utility function with a heuristic and the terminal test with a cutoff test.

Heuristic

The heuristic is a function that tries to estimate the utility of a game for a given state

3 Russel and Norvig c. 5.3,1

which is achieved with optimal play. The heuristic needs to order terminal nodes the same way the utility function does. The heuristic needs to have a strong correlation with the actual chances for the outcome of the game. Since the heuristic is the part of the algorithm that will be executed the most it has to be as fast as possible. Heuristics usually evaluate certain features of a game state like the value of each piece on the board or a good pawn structure.

One idea to estimate the outcome out of the features is to sort game states into classes where each of these features has the same value. For each of this feature classes the expected outcome is than calculated as the average outcome for this feature class based on experience.

The main problem with this approach is that for most games there are to many feature classes so that there is not enough experience for each individual class.

A different approach are so called weighted linear functions. For each feature $\mathbf{f_n}$ in a weighted linear function there is a weight $\mathbf{w_n}$. This results in the following heuristic function:

$H(s) = w_1 * f_1 + w_2 * f_2 + \ldots + w_n * f_n.$

A problem with this approach is that the weight's are always constant, but it might be useful to change their values over time. In chess for example usefulness of rook increases over time when lesser pieces are on the board therefore it seems to be useful to reflect this in the weight of the rook. These type of heuristics are also called non-linear weighted functions.[4]

4 Russel and Norvig c. 5.4.1

Cutoff function

As we have established it is impractical to evaluate the complete tree, therfore we have to decide at which point during the search we are gonna stop and evaluate the heuristic. The simplest solution for this is to define a fixed depth and use depth >= fixedDepth as a cut of test. This approach has multiple problems.

For once, because we are constantly switching our target(from min to max utility), there is high chance that with the next move the value of the heuristic is changing a lot. To avoid returning a heuristic value at a point there it's volatile a quiescence search is done. For example in chess if there is a favorable option to take a piece a position isn't quiescent.

Another Problem is the horizon effect. This occurs if for example an inevidable loss of a piece is pushed beyond the cutoff point of the search by the use of delaying tactics.

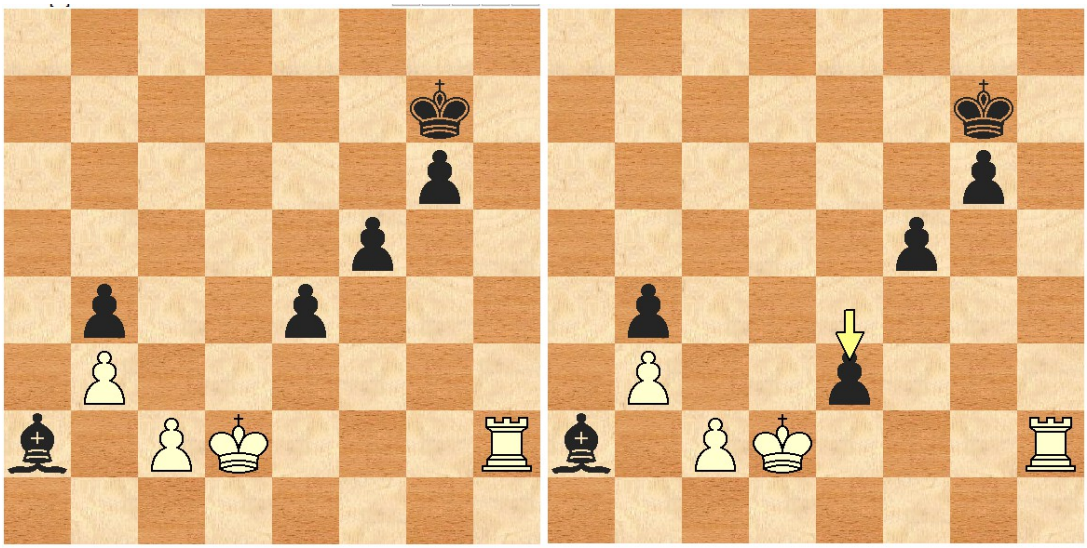


Figure 3: Horizon Effect Example

In this example The loss of the black rook is inevidable but by consecutevly setting the white king in check using the pawns the loss of the rook can be pushed over the horizon.

To avoid this a singular extension is done. This means that every move that is much better than all it's siblings is marked. For all moves that are marked this way a singular extension is done. This meens that the search is coninued to check for the horizon effect.[5]

3.3. Conclusion

As we have seen using the alpha beta pruning algorithm it's possible to create a very simple and fast algorithm for solving two player games. With the use of heurstics and a cutoff function it can easely be adabted to make good imperfect decisions in

5 Russel and Norvig c. 5.4.2

games that are to complex to be solved completely.

In games where the branching factor isn't to big like chess alpha beta has prooven to be a very sufficient solution. For games like GO with high branching factors it's usefulness is limmited.

One practical problem with the algorithm is that it can't be easily interupted at any time and therefore has problems in games with fixed time constrains.

Though it's very deacent in many two player games it's adabtability to problems other than two player games is very limmited.

4. Monte Carlo Tree Search

The monte carlo tree search originated in 1993 out of the lack of a good heuristic for games like GO which have a high branching factor and little good knowledge to develop a heuristic upon. The base idea is to use random playouts and use the average outcome to determine the best move. Though this approach has proven promising, especial to make good strategic decisions, it lacked usefulness because after only a small amount of playouts it converged. 

Later in 2005 this approach was revisited, the idea was to focus more on promising moves. To achieve this, results of the research of the multi arm Bandit problem where applied.

In the multi armed Bandit problem the player is faced with multiple one armed bandit's and has to decide on which he should play. Each bandit has a different average outcome. To decide on which bandit the player should play the following algorithm has been developed.

1. Play each bandit one's 
2. Play the bandit that maximizes the following function:

$$\frac{w_i}{n_i} + 2 * \sqrt{\frac{\log(N)}{n_i}}$$

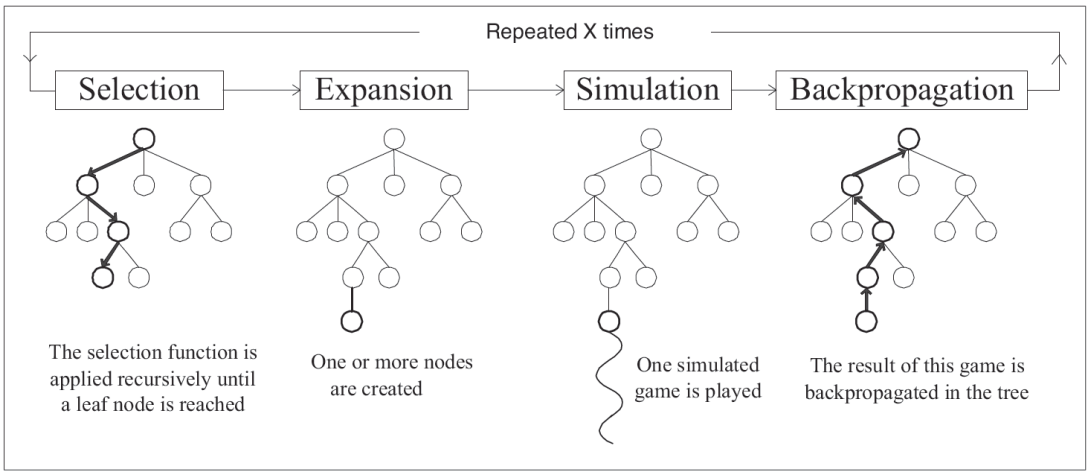
w_i : earnings on bandit i.

n_i : number of plays on bandit i.

N : number of plays on all bandits

This formula is also call upper confidence bounce or UCB1.[6]

To understand how this can be applied to a tree search, each node in the tree has to be seen as a multi armed bandit problem where each of the child's represents one arm of the bandit.



The algorithm to search for best possible move works the following:

1. Selection:

   Starting from root of the tree select the child that maximizes the function in-

6 Kozelek c. 3.2

troduced from the multi armed bandit problem until we reach a leaf.

2. Expansion:
   Create one or more nodes that can be reached from the current node.
3. Simulation:
   Play one randomized game until a terminal node is reached.
4. Backpropagation:
   update the n and w of all ancestor nodes.

This is the basic variant of the MCTS as its used today. There exist multiple variants on how this algorithm can be improved.

One approach is to insert domain specific knowledge into the algorithm. There are two options to there this can be done. The first option is to add it to the simulation phase. In this case instead of playing completely random the chances for each node to be taken will be determined by a heuristic. Such playouts are than called heavy playouts.

The second option is to apply the knowledge during the expansion. This can be done by replacing w / n term in the equation with a value h given by a heuristic. This value can than either be replaced after a certain amount of playouts per node or be graduatly faded out. Another variant is to replace the playouts completely with the heuristical value.

4.1. Conclusion

Compared to alpha beta the mcts algorithm is far more complex. Due to the selection which always starts at the root of the tree the overhead of mcts for each

evaluated node is fairly high especialy as the tree grows larger.

Althoug the mcts algorithm converges to the minimax strategie with the ammount of runs increasing in a situation there a game can be solved completely it is not very efficent.

One practical improvement of mcts is that it can be easily interuped after each run.

The biggest bennefit of the mcts alorithm though is it's adabtability. It can not only be easily adabted to games with more than two players or games with imperfect information it can alsow be usefull in non game related problems.

5. General Optimisations

5.1. Transpositions

A transposition is a position that can be reached on different way's and therefore occurs mutliple times in the game tree. To avoid evaluating transpositions multiple times during a tree search transposition tables are used. In these tables positions that might be transpositions are entered. Bevore a position is further evaluated it is first checked if it is a transposition. If a match for a position is found in the transposition table the result for the entry in the transposition table can be returned.

5.2. Opening Libraries

In Most games the number of possible starting positions is fairly limited, often only being one starting position. Because of that the number of different move's in the beginning of the game is limmited. Now instead of reevalutating these starting positions every game we can evaluate them once offline with high search depth and

store the results in a library. By focosing on not obviously one sided positions and eficent packing of the data it is possible to store the best stratgies up to a reasonable depth. In Chess for example up to 10 moves.[7]

5.3. Endgame Libraries

In games like chess the number of pieces and therefore the number of possible positions decreases over time. For example the number of possible position for KBNK (King, Bishop, Knight, King) endgame is 462. Because of that it is possible to generate every possible position with a certain amount of pieces left. To get the utility for all of these positions we first get the positions that are terminal nodes and assign them their utility. Starting from the terminal nodes we then do unmoves instead of moves to go back and assign the moves prior to terminal moves their utility until we have assigned a utility to all moves. We can than store the utility values efficently in a library and look them up if needed.[8]

7 Russell and Norvig c. 5.4.4

8 Russell and Norvig c. 5.4.4

6. Sources

1. Tomáš Kozelek (2009). Methods of MCTS and the game Arimaa (PDF). Master's thesis, Charles University in Prague.

2. Stuart J. Russell and Peter Norvig (2016). Artificial Intelligence A Modern Approach Third Edition. Pearson