Proseminar *Algorithms and Data Structures* Game Tree

Eric Michel Wagner
Supervision: Harold Bruintjes

Winter Semester 2015/16

Contents

1 Introduction 3

2 Game Tree 4

3 Evaluating a game state 5

4 2-player games 7

4.1 Minimax Algorithm 7

4.2 Optimization 9

4.2.1 Alpha-beta pruning 9

4.2.2 Game specific optimizations 12

5 N-player games 12

5.1 Paranoid 13

5.2 Max^n 13

5.3 Paranoid versus Max^n 14

6 Conclusion 14

1 Introduction

In computer science, the search for potent AI opponents is dating back over 100 years to Ernst Zermelo, who introduced the topic of game tree search. Game trees are by themselves pretty simplistic: just build a tree of every possible move and then use a path finding algorithm to derive the best move from it. Although the basic concept of the game tree was long known, it took researches until 1997 before a chess grandmaster, Garry Kasparov, could be beaten by a computer for the first time. The reason for this is that a brute force algorithm is simply not feasible for games with the complexity of chess.

In 1950, Claude Shannon estimated a lower bound for the number of possible chess games at 10^{120} in his paper “Programming a Computer for Playing Chess“ [?]. To put this number into perspective, the number of atoms in the observable universe is estimated to be between $4 * 10^{79}$ and $4 * 10^{81}$.
It becomes clear that building a complete tree and using a simple path finding algorithm will be impossible for most games. The size of the used trees has to be reduced and the algorithms have to be able to work with limited information. So basically we need a program that can decide logically if a move is good or not, without knowing for certain that that move leads to the highest probability to win. The IBM Deep Blue, the computer that was finally able to beat Kasparov in 1997, only looked 6 to 20 moves ahead, depending on the situation, and used that information to determine its next move.

The basic idea of the tree and the underlying algorithm can be show with the help of a game of Tic-Tac-Toe, also known as knots and crosses. The game is really simple and every player who has a little knowledge of the game knows that the starting player can always force a win. In a game of Tic-Tac-Toe the players are taking turns drawing knots and crosses on a three by three grid and whoever gets a line of 3 equal symbols first, wins. For this example we assume the AI plays the crosses and started the game. For the sake of argument we assume to get the board in *Figure 1* after some turns, even though this would not be possible if we use a perfect AI, as the perfect start move would be a cross in the middle of the grid.

Figure 1: A possible board in Tic-Tac-Toe

X	X	O
	O	
	O	X

For humans this is an easy problem to analyse. Humans see that the knots are only missing one move to win. There is only one possible move to deny this. The cross has to occupy that spot. Unfortunately computers can not make the same logical deduction as humans, at least not at a similar scale. The logic of humans can not be implemented one to one into a processor.

In this report we will first introduce game trees as a way to represent a game and its possible moves. Later we will see how a computer can use this information to deduct the best possible move. The necessary algorithms will first be introduced for 2-player games, for which there will be presented some ways for optimization. Finally we will see how the algorithm can be tweaked to work for games which consist of more than two players.

2 Game Tree

Game trees build the fundamental for the path finding algorithm used by AI opponents. In a game tree each node represents a game state. The root of the tree is the current game state from which has to be deviated the next move. Edges represent a move by one of the players. In the example Tic-Tac-Toe game, the move from a uneven to an even depth are always the possible moves of the AI, and the other ones are from the opponent. In a complete game tree the leafs represent the final states where either one of the 2 players has won the game or it ended as a tie.

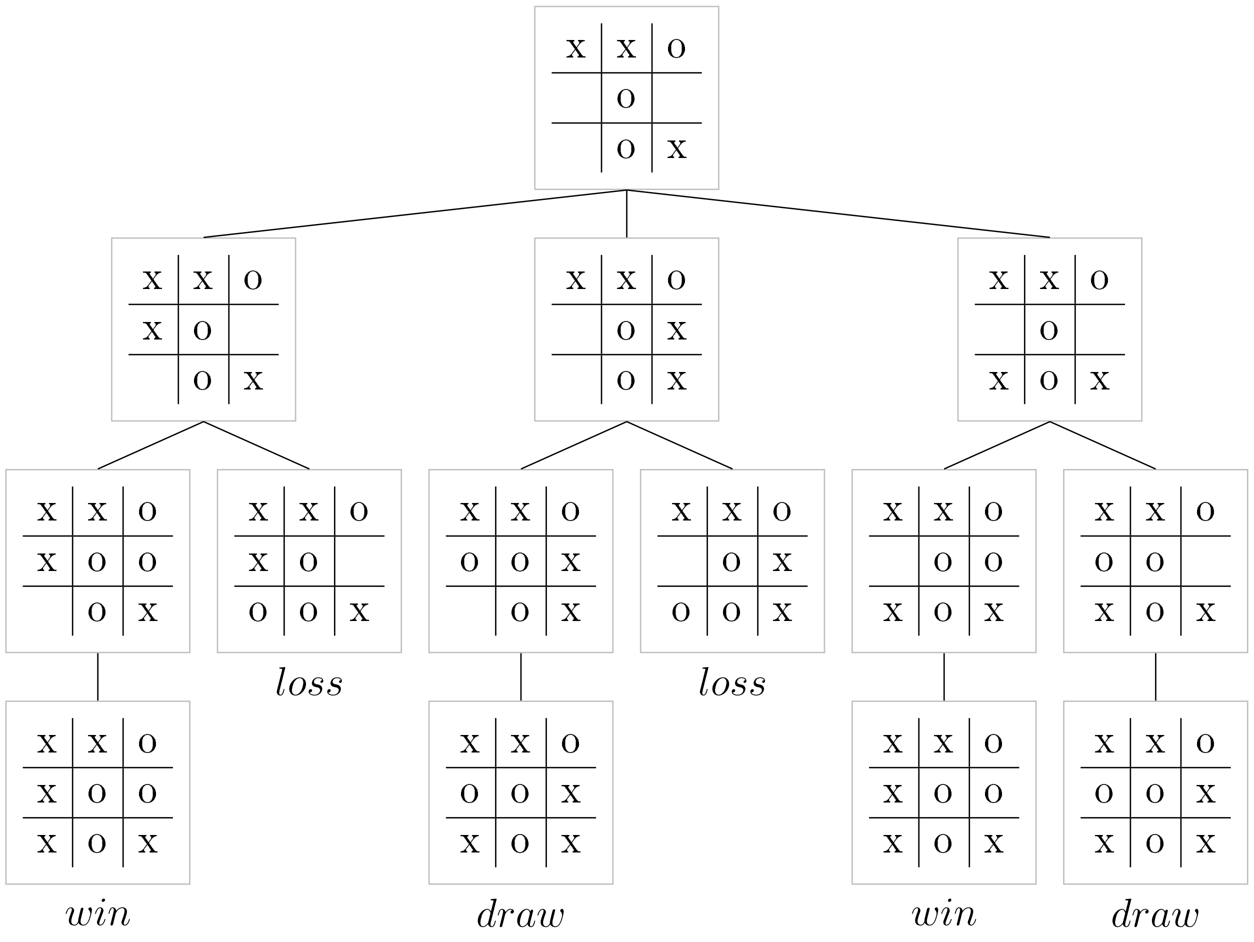
Tic-Tac-Toe has the advantage that the three by three grid is filled after a maximum of nine moves. This means that the height of its game tree can never exceed nine. For Tic-Tac-Toe we can calculate the number of possible game states by taking every single combination of crosses, knots and blanks(3^9) and subtracting the number impossible combinations(impossible count of knots relative to the number of crosses, double winners) and the number of boards which are just rotations of another board that is already counted. This will result in a total of 5,478 different game states.A game tree of that size can still be handled easily by modern computers. Because of the nature of the game, the number of children shrinks the further into the game we are. The tree in *Figure 2* arises from the example from the introduction.

We see that the number of descendants of a node is variable. It is actually equal to the number of possible legal moves. In Tic-Tac-Toe this value starts at nine and is decreasing from there. For more complex games these numbers get a lot higher.
In chess, we have a much more complex game. The number of possible opening moves for each player is already twenty (16 pawn moves and four knight moves). And chess has the particularity that the pieces are packed together at the start. So over the first turn the number of moves increases drastically to a potential of over 100 possible moves for a single turn. And as it turns out games of chess also last a lot longer than TicTacToe games. In 1950, Claude Shannon approximated the number of possible reasonable states at 10^{43} [?] with the assumption that an average game lasts 80 turns.
This means that for games like chess it is not feasible to draw the entire game tree. For games like chess the tree is only generated up to a certain depth. The best move is then calculated from that information.

Another way of seeing game trees is to look at them as AND/OR trees [?]. This name reflects the speciality of the tree a bit better. Since even though every edge represents one players move, only every second move is controllable by the computer. One can neither expect the opponent to play a perfect game, nor that he does mistakes.
The nodes of an uneven depth are therefore considered as OR nodes, while the other ones are

AND nodes. An OR node represents the choice of selecting one move or another one. On the other hand the computer has no influence on the reaction to this move by his opponent. Therefore, in AND nodes, the program has to expect the opponent to do every single possible legal move.

Figure 2: Game Tree for Tic-Tac-Toe



3 Evaluating a game state

The evaluation of a given game state is done by a so-called evaluation function. An evaluation function f takes a game state S and a player P and returns an value representing how good a state is for that player. Higher numbers represent better situations while lower numbers represent worse situations. The result of f for a state S where P has won is defined as $+\infty$. A state where P has lost is given by $-\infty$ and a tie is represented by the number 0.
A general form for an evaluation function is given by:

$$f\colon (S, P) \to \mathbb{R} \cup \{-\infty, +\infty\}$$

In a game like Tic-Tac-Toe, where the game tree can always be explored entirely, a simplified version can be used. The leafs will always be states where the outcome of the game has been decided. In this case the evaluation function just need to inform that such a state is not yet reached, without representing how promising the board is. One could give such a function as:

$$\begin{aligned} f\colon \quad (S,P) \quad &\to \{-\infty, -1, 0, +\infty\} \\ (s,p) \quad &\mapsto \begin{cases} +\infty & \text{if player p has won} \\ -\infty & \text{if player p has lost} \\ 0 & \text{if the game ended in a tie} \\ \text{-(number of moves player 2} & \text{else} \\ \text{is away from winning)} & \end{cases} \end{aligned}$$

Unfortunately, for most other games this method is not possible, because, as we can not draw the entire tree, we need to have a better representation of the game state. In these cases the programmer of the algorithm has to put a lot of work into the development of something that is a real reflection of the state of the board.

To get a better understanding of how such an heuristic could be implemented we can look at how an evaluation function for chess could be implemented. An evaluation function f might take into account the following factors:

Material advantage

The material advantage takes into account which pieces are remaining for the two players. As some pieces are more powerful then others, we need to include a weighting factor α_{figure} for each figure. α_{king} has the value of $+\infty$ as the game is over once one player has lost it. For all the other figures these values have to be adjusted through research or experimentation.
If F is the set of all chess figures and p_1 and p_2 are the player, we can calculate the material advantage with the following formula:

$$material\ advantage = \sum_{figure \in F} (\alpha_{figure} \cdot (|figure|_{p_1} - |figure|_{p_2}))^*$$

$^*(+\infty \cdot 0)$ has to defined as 0 for formula to work

This term is by far the most important one. Even if one gains a lot of momentum by sacrificing a figure, in the long term the material deficit becomes more noticeable.

Pawn formation

Chess literature mentions several positive and negative formations for pawns. Some of the more important are:

- Double Pawns: Two or more pawns in one row are usually bad.
- Pawn Rams: Two opposing pawn blocking each others movement.

- Passed Pawns: Pawns who have advanced far enough and threaten to archive promotion.
- Isolated Pawns: Pawns with no friendly pawns on either side are vulnerable to attacks.
- Eight Pawns: Too many pawns restrict mobility, especially for rook movement.

Mobility
A player is more likely to find a good move if he has more options. Another reason why the mobility is important, is that one characteristic of checkmate is that the loser has no more legal moves remaining. So the number of legal moves is counted here and taken into consideration.

Board Control
Another factor to consider is the position of certain pieces. Not every board with a high mobility has a lot of viable options. For example if the bishops are blocked, they, as one of the most powerful pieces, have no imminent influence on the game.

Development
Another important aspect to look at, especially at the beginning, is that bishops and knights(viewed as minor pieces) should jump into the battle quickly, while major pieces(rock, queen) should remain quiet and the king should castle early. The development value represents in how far these principles are fulfilled.

King safety
Last but not least we have to look at king safety. As the game is over as soon as the king has no more moves to escape an opponents piece. This is already represented by the development value, but as the game continues these two have to be handled separately, the importance of the king safety increases while the development value becomes insignificant.

If these values are weighted correctly, sometimes with factors that vary with the duration of the game, we get a basic evaluation function. Of course there is still room for improvements.

An evaluation of the board for player 1 and 2 might both result in a positive value. To get an relative value of the strength of player 1's board, the value of player 2 has to be subtracted. If the result is greater then 0, player 1 has an advantage over player 2.

4 2-player games

In the following part we will see how the results can be used to develop an AI for games for 2 players.

4.1 Minimax Algorithm

Minimax(*Figure 3*) will be the first path finding algorithm for game trees we take a look at. It is the brute force way of doing things. The algorithm starts at the bottom of the game tree

and evaluates the board for every leaf. Then it climb up level by level by choosing alternately the maximum value, if the AI has to make a move on that level, and the minimum value otherwise.
The root represents the value of the best possible move at that point and the algorithm returns the move that leads to one descendant with the same value in its node.

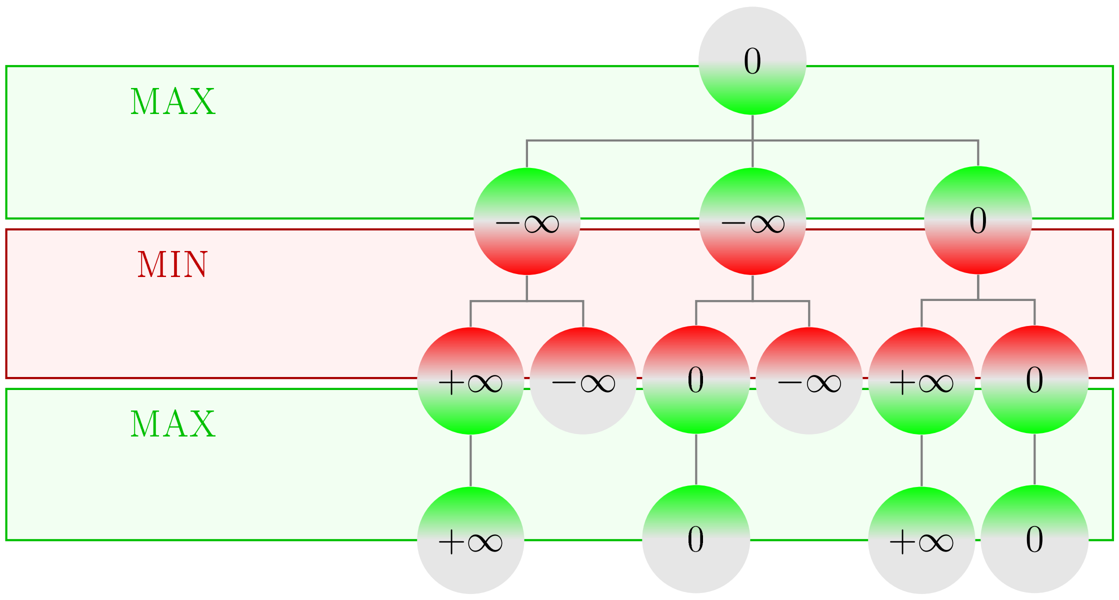
Figure 3: Minimax algorithm

```
Player original;

public Move bestMove(State s, Player player, Player opponent){
    original=player;
    (move,score)= minimax(s,maxdepth,player,opponent);
    return move;
}

private (Move,int) minimax(State s, int depth, Player player,
   Player opponent){
    int maxvalue= null;
    if((depth<0) || (s.validmoves(player).isEmpty())){
        score= evaluate(s,original);
        return(null,score);
    }

    best=(null,null);
    foreach(Move move:s.validmoves(player)){
        s.execute(move);
        (move,score)=minimax(s,depth-1,opponent,player);
        s.reverse(move);
        if(player==original){        /* looking for best move*/
            if(score>best.score)
                best=(move,score);
        } else {     /* expecting the best move from the opponent */
            if(score<best.score)
                best=(move,score);
        }
    }
    return best;
}
```

As minimax always analyses every branch, it best-, average- and worst-case time complexity are all the same. The time complexity of minimax for a tree of depth d and an average branching factor of b is $O(\sum_{i=0}^{d} b^i) = O(b^0 + b^1 + ... + b^d) = O(b^d)$.

For the Tic-Tac-Toe game from the introduction, together with it evaluation function, we get a tree with nodes filled with either 0 or $+\infty$ or $-\infty$.
Then the tree gets filled by alternately choosing the minimum and maximum from its kids. For this example we get the tree in *Figure 4*.

Figure 4: A example tree for minimax



After the nodes of the tree have been filled, the parent node has the value of the best move. The stated algorithm chooses always the first appearance of that value to do its move, but one could choose any and would get the same end result (if the opponent plays perfectly).

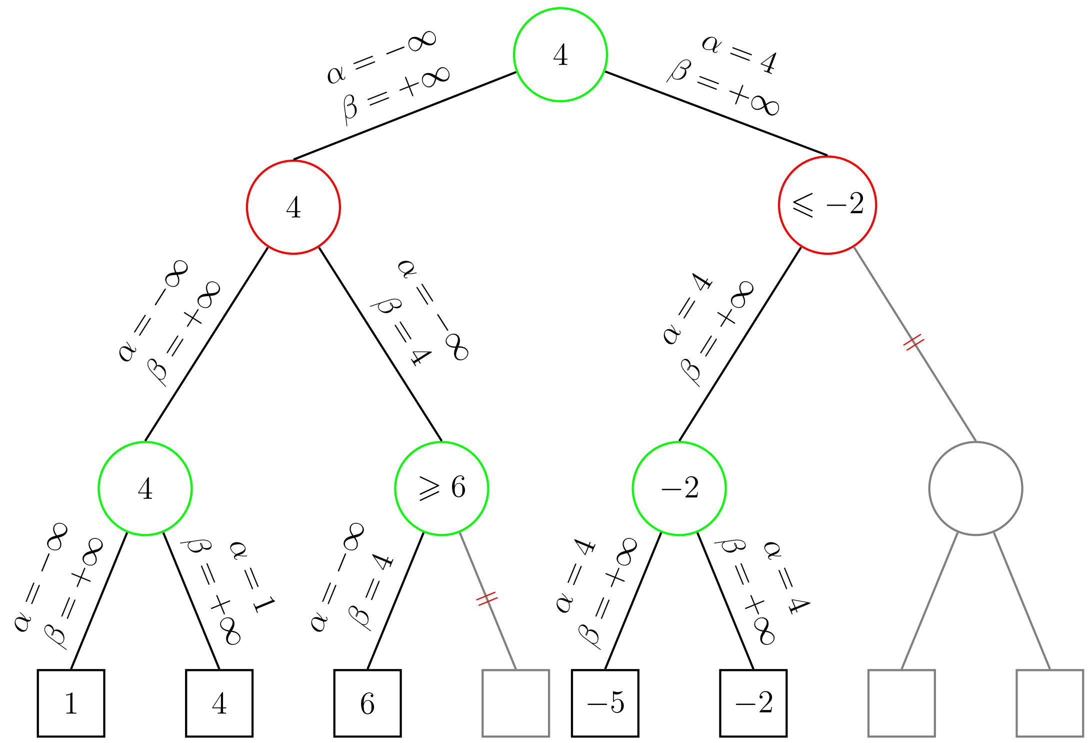
4.2 Optimization

As already stated, minimax on its own is not a very efficient approach. Although Deep Blue used minimax as its base algorithm, it used a number of optimization strategies to make more efficient use of its computing power.
A first improvement is the α-β-pruning algorithm, which is a more efficient way of implementing minimax.
After that we will take a look at iterative deepening, opening books, end-game databases and quiescence [?]. These are more game specific optimizations, but they can be altered to fit a variety of other games.

4.2.1 Alpha-beta pruning

α-β-pruning(*Figure 6*) uses the knowledge it gets from building previous branches of the tree to stop developing unnecessary branches. For maximizer nodes it keeps track of the maximum value that it can receive at any moment time. Similar, the minimum value is stored for every minimizer node. There values are stored as α and β respectively. These are now used to decide which branches do not need to be evaluated.

Figure 5: A example tree for α-β-pruning



If, for example, a maximizer M develops his outer most child and receives a number x as score. We know that its value will be at least x. This information will be passed down to the next child of M as α value. This child can now evaluate its left child an get a value of y. As a child of a maximizer is a minimizer(in most cases), its value will always be smaller or equal to that y. If y is smaller than α, respectively x, we know that M will always choose another node over the one looked at, at the moment and therefore there is no need to develop other branches of that node, because its value will never get higher then y. If y would be greater than x, the next child has to be developed to see if any of them result in a value smaller than α. If every branch is developed and the node still has a higher value than α, it becomes the current best choice. α will now have a new value for the calculations of the other, still undeveloped, nodes.
Similar, the maximizer nodes get passed down a β value from its parents. If we realize that they are able to achieve a value higher than that β, the development of that branch is stopped.

α-β-pruning results in the same move as minimax, but with a big performance difference. While minimax has an worst-, average- and best-case time complexity of $O(b^d)$, the average case for α-β-pruning reduces this to $O(b^{3d/4})$. And its best-case time complexity is $O(b^{d/2})$ as the algorithm can skip half of the nodes [**?**].

In the example of *Figure 5* we see how the algorithm works. It build the tree from left to right and always passes down the α and β value from the path to the root.

When it climbs back up it changes the values for α and β and compares if $\alpha \geqslant \beta$ and climbs back up further if there is no reason to keep looking.

Figure 6: α-β-pruning algorithm

```
Player original;

public Move bestMove(State s, Player player, Player opponent){
    (move,score) result= alphabeta(s,maxdepth,player,opponent,
  Integer.MIN,Integer.MAX,true);
    return result.move;
}

private (Move,int) alphabeta(State s, int depth, Player player,
  Player opponent,int alpha, int beta, boolean isMaximizer){

    if((depth<=0) || (s.validmoves(player).isEmpty())){
        score = evaluate(s,original);
        return(null,score);
    }

    (Move, int) best=(alpha,null);
    foreach(Move move:s.validmoves(player)){
        s.execute(move);
        (Move,int) current=alphabeta(s,depth-1,opponent,player,
  alpha,beta,!isMaximizer);
        s.reverse(move);

        if(isMaximizer && current.score>alpha){
            alpha=current.score;
            best=(move,alpha)
        } else if (current.score<beta) {
            beta=current.score;
            best=(move,beta)
        }
        if(alpha>=beta){     /*cut off branch*/
            return best;
        }
    }

    return best;
}
```

4.2.2 Game specific optimizations

α-β-pruning can be applied whenever the minimax algorithm is used. For most games you can also use some of the following methods to further reduce the size of the tree.

Iterative deepening

In general the exact growth rate of a game tree is not predictable. To still be able to have an answer after a given time, without giving up to much computation time, the search is broken down over multiple iterations over the same tree.
Because for a lot of games, game trees grow exponentially, the cost of the last iteration outweighs the previous one. This means that time lost on repetition would not result in far deeper calculations.
The calculations with iterative deepening can even be faster than the immediate search for a set depth, as we can use the previous calculations to reorder the nodes, so that the α-β-pruning algorithm approaches its best-case time complexity of $O(b^{d/2})$. [?]

Opening Books

Especially in chess there are a set of standard openings and responses that have been developed and analysed over decades. Computers can use this knowledge and look up the most effective moves, instead of calculating a suboptimal one.

End-game databases

Similar to opening books, end-game databases use pre calculated information. The database contains the evaluated values of all board configurations with a small amount of pieces remaining.

Quiescence

This approach is also known as selective deepening. After a game tree is build to a certain, it's most promising results are evaluated further to see if they still showcase a good move some level deeper into the game tree. [?]

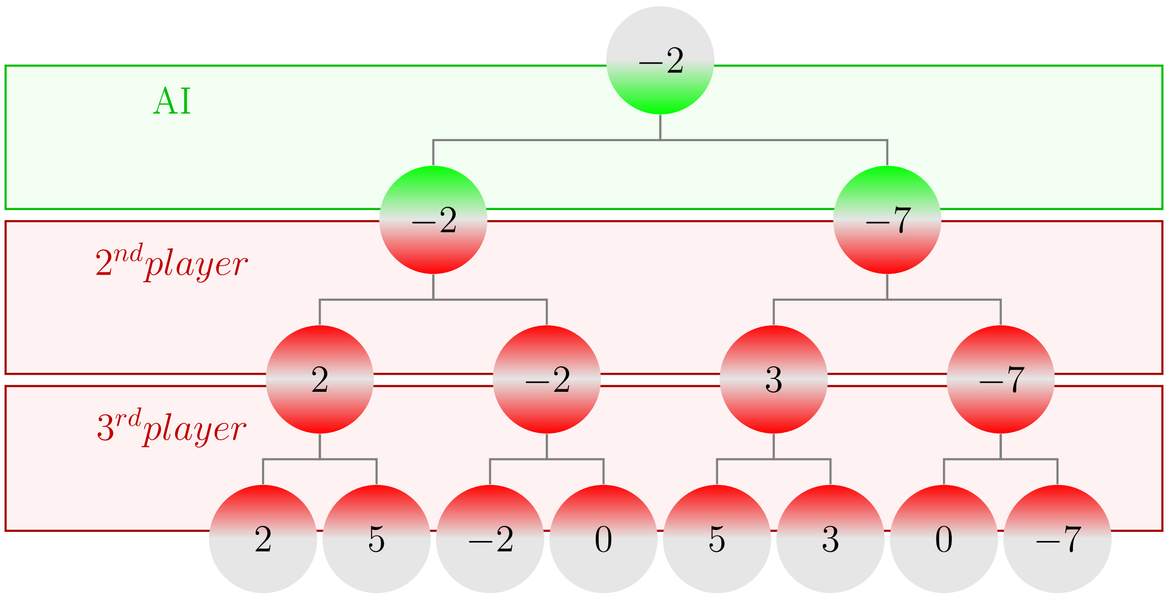
5 N-player games

N-player games work a bit differently than 2-player games, because there exist situations where it is not clear which move is better. Earlier we saw that for 2 player games it makes no difference which move we pick if they both lead to the same score. In n-player games it would still make no difference for the player doing the move, but each move would affect the other players in a different way. This and the fact that each player has a choice of different strategies to pick from, for example playing extremely defensively, makes it impossible to come up with a perfect move. In 2-player games we assumed that the opponent plays perfectly and any other move would lead to an even better outcome for the AI. This perfect play is not defined any more, because if all players play perfectly, there are still moves that make the situations for the AI worst. We will now see two commonly used strategies [?] to approach this problem and what are the differences between them.

5.1 Paranoid

The paranoid algorithm(*Figure 7*) is the easiest way to evaluate a n-player game tree. It tries to reduce a n-player game to a two player game, by assuming that $n-1$ players build a coalition against the AI. [?] This would mean that the other players do not want to win individually, but just want one of them to win. Under this assumption, the AI does not need to know the perfect move of every player, but only the least favourable move for itself.
This algorithm functions very similar to minimax. The main difference is that for every maximizer node there are $n-1$ minimizer node, which try to reduce the score of the AI.

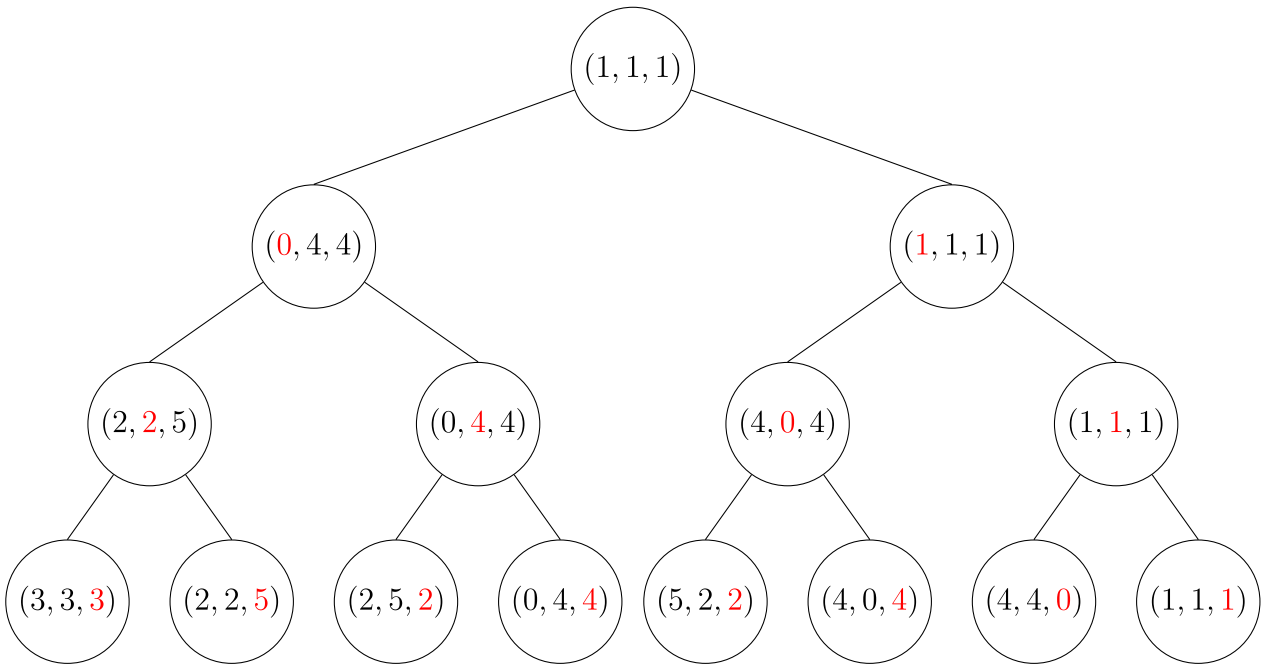
Figure 7: A example tree for paranoid



5.2 Max^n

The approach of expecting all the other player to play against one person is not always very effective for the AI, as it leads to a game where the AI tries to not lose, instead of trying to win. In most situations, the opponents also want to win personally. One can use the max^n algorithm to evaluate a game when we assume that the purpose of every player is to make the move which gives him the highest chance of winning.
Max^n(*Figure 8*) represents every node in a game tree with a vector which consists of n entries, one for every player. The nth number represents the evaluation of the board from the perspective of the nth player. Each node tries to maximizes the ith entry, where i is the player who's turn is represented at that depth [?].

Figure 8: An example tree for Max^n [?]



5.3 Paranoid versus Max^n

For n-player games there is no overall best algorithm to choose. It differs from game to game, which algorithm leads to a better result. The paranoid algorithm is generally a good choice if the game does not offer the players a lot of options for collaboration [?]. The AI can expect the players to play together so it will no get caught off guard, while still having a accurate representation of the game.
On the other hand, in most card games it is for example very easy to play as a team with other players. Here the AI could be in the most favourable position but still be defensive, because the other players as a whole are more powerful. In these example the max^n algorithm yields much better results as it expects every player to pursuit the win for themselves [?].

6 Conclusion

The report shows that the difficulty of programming a game AI comes with the limited resources available. At least for 2 player games, if a computer with unlimited computational power existed, there would be no need for any optimization or evaluation function. The best move would be very easy to determine for most games. The main work of the programmer lies in the development of an evaluation function which represents the power of a board accurately.

Multi-player game are very different as they force the programmer to think about the strategy of the opponents. The decision which algorithm should be used can already take a lot of work. And yet the AI can not really adapt to a strategy change by its opponents, so it will only be able to play flawlessly from its own perspective as it has not enough information over the opponent.