AI Course Final Project - Checkers

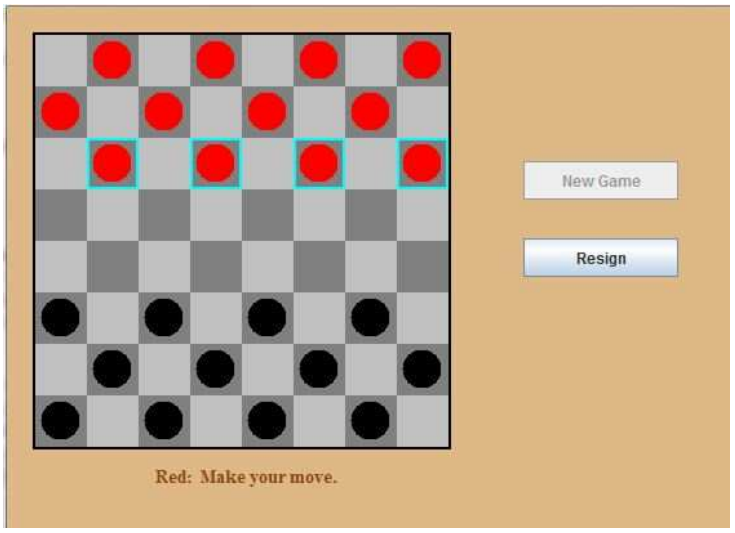
Miki *Alex*

ABSTRACT

Checkers is a very popular game all over the world. The first attempts to build the first English draughts computer program were in the early 1950s. In 2007, it was published that the program “Chinook” was able to “solve” the 8X8 board from all possible positions.
In this paper, we'll present several approaches for implementing a computer agent that plays in high level.

TABLE OF CONTENTS

1) Introduction
2) Game Properties
3) Agents Approaches
    3.1) Random
    3.2) Minimax (depth limit)
    3.3) Alpha-Beta pruning
4) Implementation Issues
5) Evaluation Functions
    5.1) Opening-Middlegame
    5.2) Ending
6) Comparison
7) Conclusion



1. INTRODUCTION

The game of checkers is considered a complicated game with 10^20 possible legal positions in the English draughts version (8*8 board) alone (much more on higher dimensions). Our approach is to create a computer agent based on the Minimax algorithm, together with possible improvements, which is the state of the art in one-on-one games. We start from implementing a basic minimax player with a basic evaluation heuristic, and step by step we improve the pruning by implementing alpha-beta pruning and in addition, we improve our evaluation function to reach better approximation of the value of the position for our computer player.

2. GAME PROPERTIES

Our implementation relies on the rules of English draughts as they're described in Wikipedia [1], and on some additional rules that follow from them. Such as:

a. Regular capture is possible only in forward direction, except if the capture is a part of multiple stages capture on the same move, it's possible to capture also backward.

b. When a pawn reaches the last line it is crowned to King. Kings are able to move both backward and forward and they're also able to capture both backward and forward.

Our implementation is generic and can be easily expanded to even higher dimension boards that preserve the current game properties (The same rules, the first 3 lines of each side hold pieces).

3. AGENTS APPROACHES

This is the main section of our project that deals with the different ways to solve the problem (in our case – building a good AI computer player for the game of checkers). We will try to give the reader an idea of a number of directions that can be taken in this case. There obviously will be more detailed explanation of the algorithms that we did implement for this project.

3.1 Possible approaches

There are many possible approaches for solving similar problems (zero-sum games). In this section, before explaining the algorithms that we've implemented, we will give a couple of examples to such approaches that we didn't use.

An example of a possible approach for such games is reinforcement learning. During the preparation of the project, we thought about adding a sort of feature of reinforcement learning in order to correct possible mistakes the computer made during calculations with more accurate view of the position (after the move was made). Eventually we acknowledged that adding the system in its full properties is impractical due to the large branching factor and the huge amount of positions that may appear in the game (10^20(!) only in the 8*8 version. exponentially growing on higher dimensions). Another feature we thought about is also a learner for the first 5 moves (to make the computer stabilize on the opening moves) but we considered its contribution as insignificant in contrast to the effort it takes.

1 http://en.wikipedia.org/wiki/English_draughts

If we would have wanted to integrate learning in our agents, we would have likely trained them on kings endings with maximum 5-6 kings (in order to make it possible to model the endings easily) and then combine the learning method with our alpha beta player in the phase of the game where 5-6 kings are reached.

After all things considered, we decided to examine the Minimax approach learned in class as in our eyes it is the most logical and it serves our purposes well for an one-on-one zero-sum game.

3.2 Our approaches

In this section we will explain the direction that we chose for this project, which is mainly, relies on the famous Minimax approach with a couple of optimization tricks.

3.2.1 Random

Our baseline approach was to build a randomized player that picks each move from a set of possible moves. As the moves are picked randomly, that causes an irrational player which doesn't play by any strategy. A property of checkers is that one mistake is enough to assure the other player a sure win (even if it takes few dozens of moves).

3.2.2 Minimax (with depth limit)

Minimax (sometimes **minmax**) is a decision rule used in decision theory, game theory, statistics and philosophy for *mini*mizing the possible loss while *max*imizing the potential gain. Alternatively, it can be thought of as maximizing the minimum gain (**maximin**). Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves. It has also been extended to more complex games and to general decision making in the presence of uncertainty.

The Min-Max algorithm is applied in two player games, such as tic-tac-toe, checkers, go, chess, and so on. All these games have at least one thing in common, they are logic games. This means that they can be described by a set of rules and premises. With them, it is possible to know from a given point in the game, what are the next available moves. So they also share other characteristic, they are ‘full information games’. Each player knows everything about the possible moves of the adversary.

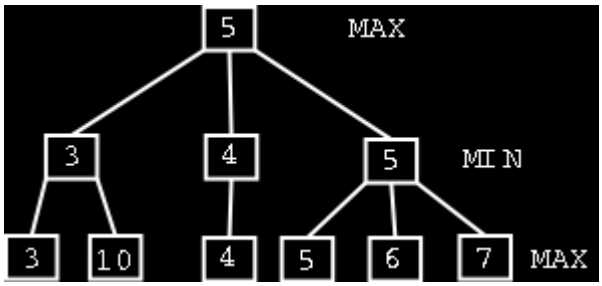


Figure 1: A representation of a search tree for a logic game.

There are two players involved, MAX and MIN. A search tree is generated, depth-first, starting with the current game position up to the end game position. Then, the final game position is evaluated from MAX's point of view, as shown in Figure 1. Afterwards, the inner node values of the tree are filled bottom-up with the evaluated values. The nodes that belong to the MAX player receive the maximum value of its children. The nodes for the MIN player will select the minimum value of its children.

So what is happening here? The values represent how good a game move is. So the MAX player will try to select the move with highest value in the end. But the MIN player also has something to say about it and he will try to select the moves that are better to him, thus minimizing MAX's outcome.

Optimization:

However, only very simple games can have their entire search tree generated in a short time. For most games this isn't possible, the universe would probably vanish first. So there are a few optimizations to add to the algorithm.

First a word of caution, optimization comes with a price. When optimizing we are trading the full information about the game's events with probabilities and shortcuts. Instead of knowing the full path that leads to victory, the decisions are made with the path that might lead to victory. If the optimization isn't well chosen, or it is badly applied, then we could end with a dumb AI. And it would have been better to use random moves.

One basic optimization is to limit the depth of the search tree. Why does this help? Generating the full tree could take ages. If a game has a branching factor of 3, which means that each node has three children, the tree will have the following number of nodes per depth:

Depth	Node Count
0	1
1	3
2	9
3	27
…	..
N	3^n

For many games, like chess that have a very big branching factor, this means that the tree might not fit into memory. Even if it did, it would take too long to generate.

The second optimization is to use a function that evaluates the current game position from the point of view of some player. It does this by giving a value to the current state of the game, like counting the number of pieces in the board, for example. Or the number of moves left to the end of the game, or anything else that we might use to give a value to the game position.

Instead of evaluating the current game position, the function might calculate how the current game position might help ending the game. Or in another words, how probable is that given the current game position we might win the game. In this case the function is known as an estimation function.

This function will have to take into account some heuristics. Heuristics are knowledge that we have about the game, and it can help generate better evaluation functions. For example, in checkers, pieces at corners and sideways positions can't be eaten. So we can create an evaluation function that gives higher values to pieces that lie on those board positions thus giving higher outcomes for game moves that place pieces in those positions.

One of the reasons that the evaluation function must be able to evaluate game positions for both players is that you don't know to which player the limit depth belongs.

However having two functions can be avoided if the game is symmetric. This means that the loss of a player equals the gains of the other. Such games are also known as ZERO-SUM games. For these games one evaluation function is enough, one of the players just have to negate the return of the function.

Even so the algorithm has a few flaws. Some of them can be fixed while other can only be solved by choosing another algorithm.

One of flaws is that if the game is too complex the answer will always take too long even with a depth limit. One solution it limit the time for search. If the time runs out choose the best move found until the moment.

A big flaw is the limited horizon problem. A game position that appears to be very good might turn out very bad. This happens because the algorithm wasn't able to see that a few game moves ahead the adversary will be able to make a move that will bring him a great outcome. The algorithm missed that fatal move because it was blinded by the depth limit.

3.2.3 Alpha-Beta pruning

As mentioned previously, the minimax algorithm can still be inefficient and may use further optimization. In this section we will describe an algorithm based on minimax with depth limit but with additional optimization.

There are a few things can still be done to reduce the search time. Take a look at figure 2. The value for node A is 3, and the first found value for the subtree starting at node B is 2. So since the B node is at a MIN level, we know that the selected value for the B node must be less or equal than 2. But we also know that the A node has the value 3, and both A and B nodes share the same parent at a MAX level. This means that the game path starting at the B node wouldn't be selected because 3 is better than 2 for the MAX node. So it isn't worth to pursue the search for children of the B node, and we can safely ignore all the remaining children.

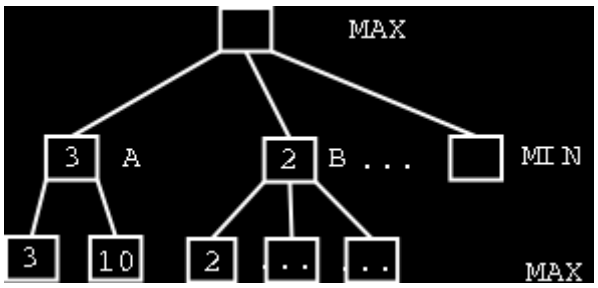


Figure 2: Minimax search showing branches that can be cuttoff.

This all means that sometimes the search can be aborted because we find out that the search subtree won't lead us to any viable answer.

This optimization is known as alpha-beta cuttoffs (or pruning) and the algorithm is as follows:

1. Have two values passed around the tree nodes:
    - the alpha value which holds the best MAX value found;
    - the beta value which holds the best MIN value found.
2. At MAX level, before evaluating each child path, compare the returned value with of the previous path with the beta value. If the value is greater than it abort the search for the current node;
3. At MIN level, before evaluating each child path, compare the returned value with of the previous path with the alpha value. If the value is lesser than it abort the search for the current node.

How better does a MinMax with alpha-beta cuttoffs behave when compared with a normal MinMax? It depends on the order the search is searched. If the way the game positions are generated doesn't create situations where the algorithm can take advantage of alpha-beta cutoffs then the improvements won't be noticeable. However, if the evaluation function and the generation of game positions lead to alpha-beta cuttoffs then the improvements might be great.

With all this talk about search speed many of you might be wondering what this is all about. Well, the search speed is very important in AI because if an algorithm takes too long to give a good answer the algorithm may not be suitable.

For example, a good MinMax algorithm implementation with an evaluation function capable to give very good estimative might be able to search 1000 positions a second. In tournament chess each player has around 150 seconds to make a move. So it would probably be able to analyze 150 000 positions during that period. But in chess each move has around 35 possible branches! In the end the program would only be able to analyze around 3, to 4 moves ahead in the game. Even humans with very little practice in chess can do better than this.

But if we use MinMax with alpha-beta cutoffs, again a decent implementation with a good evaluation function, the result behavior might be much better. In this case, the program might be able to double the number of analyzed positions and thus becoming a much tougher adversary.

4. IMPLEMENTATION ISSUES

In this part of the project we will explain our baselines in the implementation and the thoughts behind them.

The board – we chose to implement the board as a matrix (dimension X dimension). We think that this is the most logical way to do it and it will also make the GUI making process a bit simpler.

Possible moves – for each game stage (or position if you will) there is an array of moves that the player can make. This is also quite useful for the GUI.

Attack moves – a Boolean to tell us if some of the possible moves are attack moves. This is needed for the game to play properly.

Move – a class representing a move in the game. Contains more than just the Src and the Dest, but also all the data we need.

Pieces – for each game stage there is an array of pieces for both players. This is for knowing when the game ends, if the move if legal and other various issues.

Draw counter – in the ending of each game (see the end-heuristics section) if there was no attack move for 80 (can be changed) moves we declare a draw.

We've also kept a certain level of abstraction and polymorphism (mainly in the heuristics and players) so that the code will more easy to read and understand and it will be simple to add new evaluation functions and players.

We have both GUI and non-GUI versions of the game. The non-GUI version was very helpful at the beginning of our way for both developing and debugging. Even now, with fully functional GUI version, it can prove useful for diagnostic and analysis as the GUI version is only for human-vs.-computer games (but the computer player can be any of the algorithms that we have implemented, with any evaluation function, at any depth). The non-GUI version has a sort of “virtual” GUI accomplished via printing the board after each move so it is quite usable and self explanatory.

5. EVALUATION FUNCTIONS

As mentioned above (in the agents section) the algorithms use different heuristics to form various evaluation functions. Here we will describe our attempts at creating several fine heuristics.

5.1 Opening-Middlegame evaluation functions

All our evaluation functions can be divided into 2 parts – the main in-game part (opening-middlegame) and the ending part which will be described later. In the first part we try to reach some optimal stage (not necessarily the end of the game). Some possible ways for doing so will be discussed in this section.

5.1.1 Eval I – Piece to value

Our most basic function counts for the player who builds the tree the value of his pieces and subtracts from it the value of opponent's pieces. Since Kings are considered more powerful than regular pawn, we double their value in comparison to them.

Specifically:
Pawn's value = 1
King's value = 2

5.1.2 Eval II – Piece & Board part to value

This function is built over the Piece to Value function (in a sense of again evaluating pieces and subtracting) and attempts to take into account some more properties of the game. It's easy to understand that advanced pawns are more threatening than pawns that are on the back of the board. Therefore, since advanced pawns are much closer to become Kings, we give them extra value in our evaluation. Of course, we still evaluate kings more than any pawn.

Specifically:
We split the board into halves.

Pawn in the opponent's half of the board value = 7
Pawn in the player's half of the board value = 5
King's value = 10

5.1.3 Eval III – Piece & Row to value

This function is a small modification to the previous function in a sense that this function gives specific value of row to heuristic.

Pawn's value: 5 + row number
King's value = 5 + # of rows + 2

5.1.4 Eval IV – Piece & Board part to value (modified)

This function is a normalized version of the second heuristic – in this case the function prefers to minimize the number of pieces on the board. The calculations are still the same, just this time we also divide by the total number of pieces on the board.

5.2 Ending eval functions

This is the second part of the evaluation functions – the ending. When we arrive at a certain optimal stage (both sides have only kings at their disposal) we try to find some optimal strategy to continue. Such strategies will be described in this section.

5.2.1 Sum of distances

For each piece (king) of the player we sum all the distances between it and all the opponent's pieces. If the player has more kings than the opponent he will prefer a game position that minimizes this sum (he wants to attack), otherwise he will prefer this sum to be as big as possible (run away).

5.2.2 Farthest piece

This is for diagnostic purposes only. It cannot be chosen from the game (both GUI and non-GUI) and requires a minor change in the code.

This function deals with the maximum distance of all possible distances mentioned in the previous function. Again, if you're the "winning" player (more pieces) then you will prefer this distance to be as small as possible and as big as possible otherwise.

6. COMPARISON

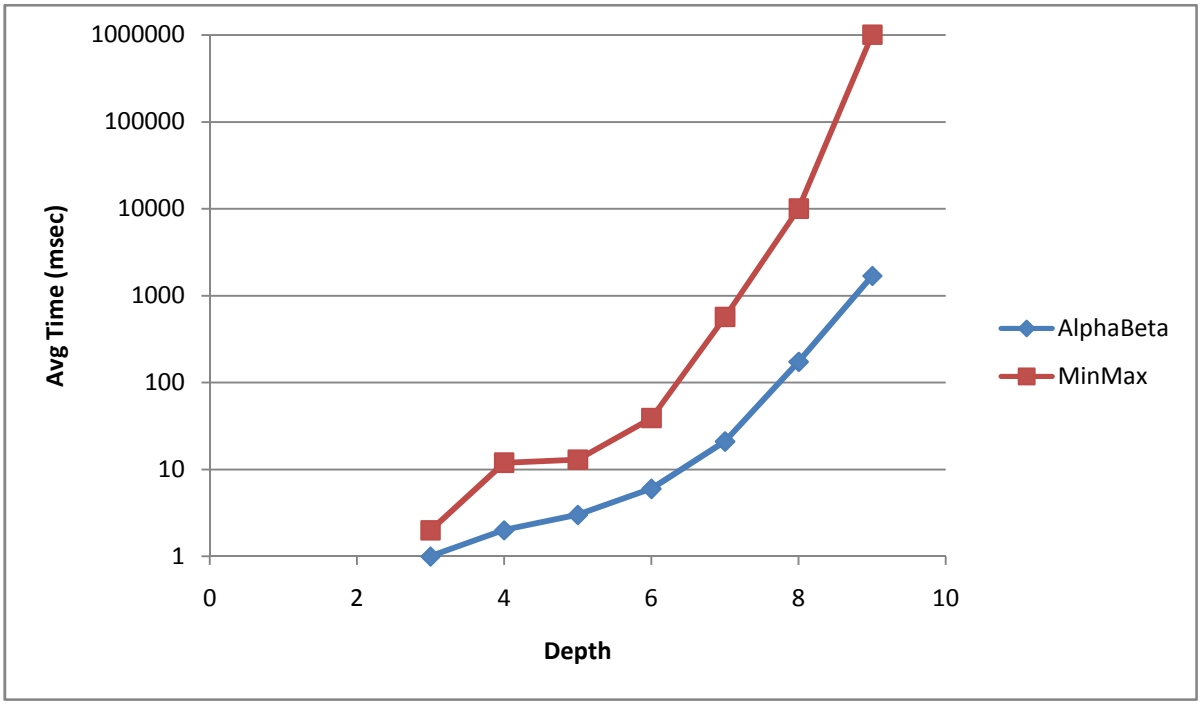
In this section of our project we will discuss various results from using different algorithms with different evaluation functions on a number of possible game settings.

6.1 Time per move comparison

We will calculate the average time per move (in milliseconds) of both Alpha-Beta and MinMax. It will show us that Alpha-Beta is far more time efficient than MinMax given the same situation.

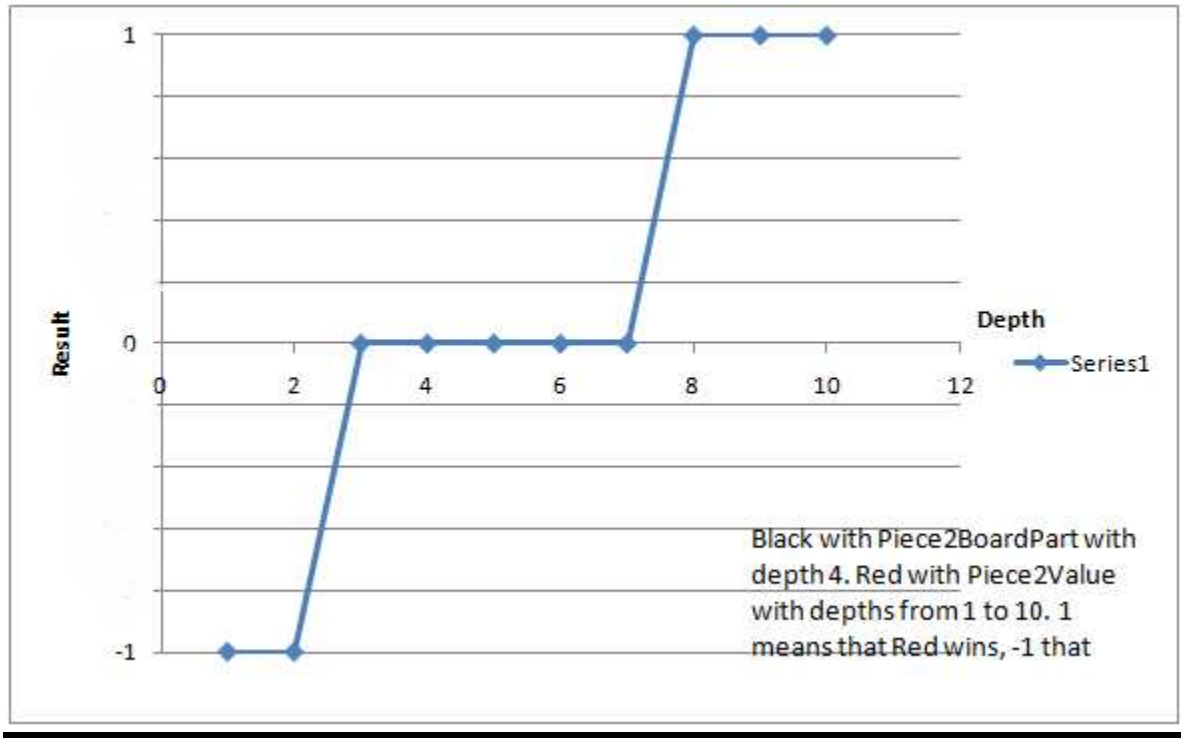
Player	Opponent	Avg Time per Move
AlphaBeta, ev=Piece2Val, depth=3	AlphaBeta, ev=Piece2Val, depth=3	1
AlphaBeta, ev=Piece2Val, depth=4	AlphaBeta, ev=Piece2Val, depth=4	2
AlphaBeta, ev=Piece2Val, depth=5	AlphaBeta, ev=Piece2Val, depth=5	3
AlphaBeta, ev=Piece2Val, depth=6	AlphaBeta, ev=Piece2Val, depth=6	6
AlphaBeta, ev=Piece2Val, depth=7	AlphaBeta, ev=Piece2Val, depth=7	21
AlphaBeta, ev=Piece2Val, depth=8	AlphaBeta, ev=Piece2Val, depth=8	173
AlphaBeta, ev=Piece2Val, depth=9	AlphaBeta, ev=Piece2Val, depth=9	1680

Player	Opponent	Avg Time per Move
MinMax, ev=Piece2Val, depth=3	AlphaBeta, ev=Piece2Val, depth=3	2
MinMax, ev=Piece2Val, depth=4	AlphaBeta, ev=Piece2Val, depth=4	12
MinMax, ev=Piece2Val, depth=5	AlphaBeta, ev=Piece2Val, depth=5	13
MinMax, ev=Piece2Val, depth=6	AlphaBeta, ev=Piece2Val, depth=6	39
MinMax, ev=Piece2Val, depth=7	AlphaBeta, ev=Piece2Val, depth=7	568
MinMax, ev=Piece2Val, depth=8	AlphaBeta, ev=Piece2Val, depth=8	Inf (too long)
MinMax, ev=Piece2Val, depth=9	AlphaBeta, ev=Piece2Val, depth=9	



This grapgh is on a logaritmic scale. We can clearly see that AlphaBeta if far more time efficient to use.

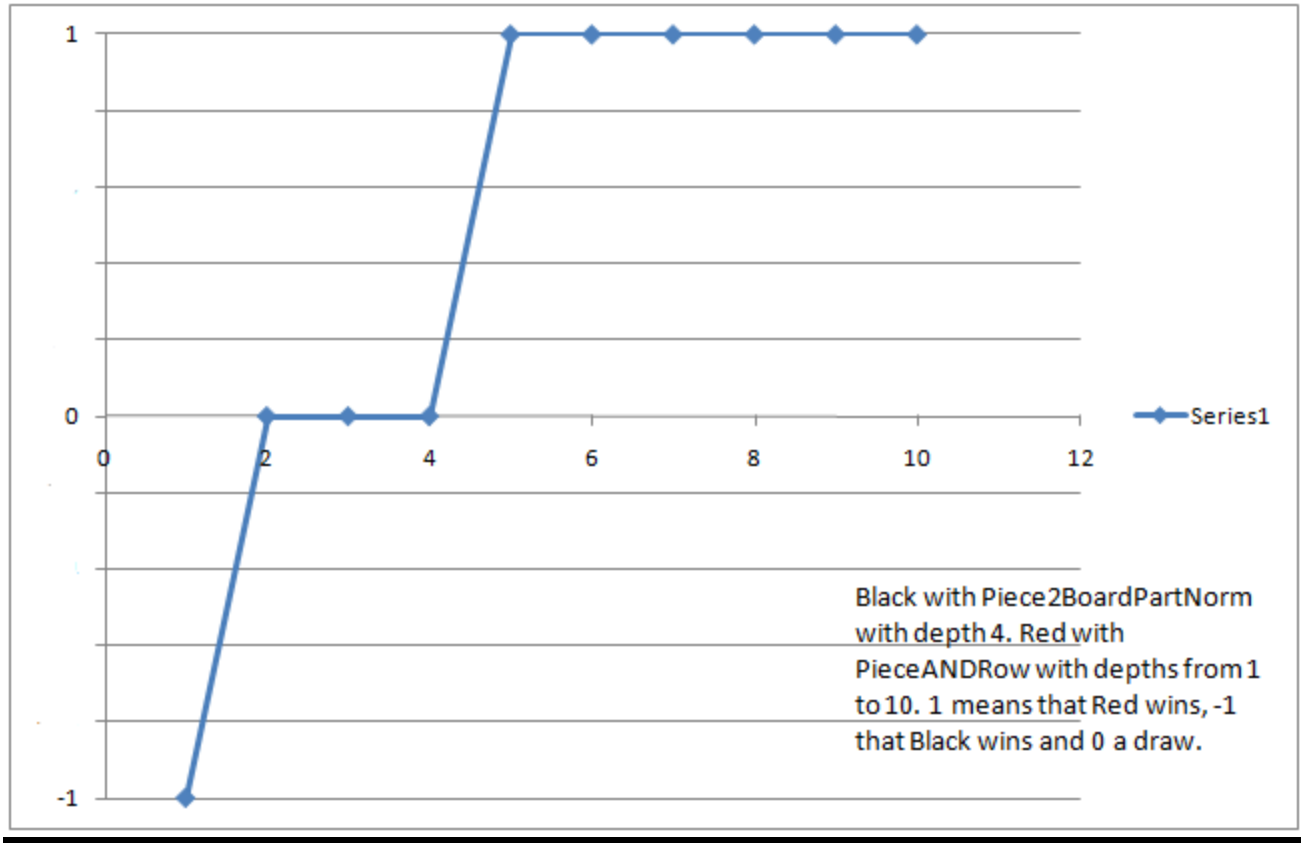
6.2 Piece2Value vs. PieceANDBoard



In this graph we can see that the PieceANDBoardPart is a better heuristic function than the piece2Value.

The black player is AlphaBeta with PieceANDBoardPart and depth 4. The red player is AlphaBeta with Piece2Value and depths from 1 to 10. We can see that although we get a draw at depth 3 (which is good because the opponent is at depth 4) we stay in this condition till the red player reaches depth 7 and only then he starts to win.

6.3 PieceANDBoardNormalized vs. PieceANDRow



In this graph we can see that the PieceANDRow is a better (or at least the same in some situations) heuristic function than the normalized version of PieceANDBoard heuristic.

The black player is AlphaBeta with PieceANDBoardPartNorm and depth 4. The red player is AlphaBeta with PieceANDRow and depths from 1 to 10. We can see that we get a draw at the early depth of 2 (which is good because the opponent is at depth 4) and start to win from the same depth as our opponent, which is 4.

7. CONCLUSIONS

- There are different approaches for heuristics.
- Heuristics can be drastically improved by adding specific features.
- The depth of the game tree has significant influence on the quality of the computer player.
- There's a tradeoff between calculation time and quality of game.
- It is not efficient to use Minimax without optimizations while with them it can be a good solution.
- Alpha-Beta pruning is exponentially improving in comparison to Minimax as the depth grows.
- Certain heuristics are clearly better than others but some of the “bad” ones still work well in some cases.
- Simple algorithms as the random player don’t stand a chance against Alpha-Beta at depth greater than 1.
- There are many other ways to approach zero-sum games but Minimax seems like a good one.