Laboratory 3

Introduction to Artificial Intelligence

Game Playing Algorithms

Ernesto Vieira Manzanera. Student id: 251663

1. Abstract

This report describe the implementation of minimax algorithm to the Othello game, and the design and consequent implementation of evaluation functions that take part. Moreover, minimax is extended with alpha-beta pruning algorithm to increase the efficiency of the algorithm.

2. Othello

Othello is strategy board game for two players, played on a 8x8 uncheckered board. There are sixty-four identical game pieces called *disks,* which are white on one side and black on the other. Players take turns placing disks on the board with their assigned colour facing up. During a turn, any disks of the opponent's colour that are in a straight line and bounded by the disk just placed and another disk of the current player's colour are turned over to the current player's colour. The object of the game is to have the majority of disks turned to your colour when the last playable empty square is filled.

Rules:

- The board is set up initially with two black discs (i.e. a disc with black side uppermost) placed on squares e4 and d5 and two white discs on d4 and e5.

```
  A B C D E F G H
1 _ _ _ _ _ _ _ _ 1
2 _ _ _ _ _ _ _ _ 2
3 _ _ _ _ _ _ _ _ 3
4 _ _ _ W B _ _ _ 4
5 _ _ _ B W _ _ _ 5
6 _ _ _ _ _ _ _ _ 6
7 _ _ _ _ _ _ _ _ 7
8 _ _ _ _ _ _ _ _ 8
  A B C D E F G H
```

Fig 1. Initial configuration of the Othello game.

- Black always plays first with players then taking alternate turns.
- At each turn a player must place a disc with their colour face up on one of the empty squares of the board, adjacent to an opponent's disc such that one or more straight lines (horizontal, vertical or diagonal) are formed from the newly placed disc, through one or more of the opponent's discs and up to other discs of their own colour already on the board. All the intervening discs of the opponent's colour are flipped to the colour of the newly laid disc.

- Discs may be flipped from one colour to the other but once played are not moved from one square to another or removed from the board.
- Players may not pass unless there is no valid move available to them in which case they must pass.
- Game continues until neither player is able to move, usually when all 64 squares have been played.

3. Design

Minimax algorithm contributes to game playing solving by performing a search in the consequent moves than can be given up to a certain depth. At each depth, the algorithm must decide if it maximises or minimises the expected utility value for a specific player.

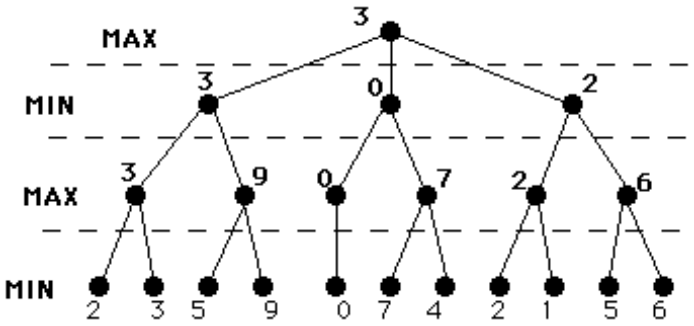


Fig 2. Schema of mini-max searching tree

The values at the leaf nodes result from the heuristic evaluation of the board configuration at such position. The algorithm is implemented as following:

```
1.  function minimax(position, depth, alpha, beta, maximizingPlayer)
2.      if depth == 0 or game over in position
3.          return evaluation of position
4.
5.      if maximizingPlayer
6.          maxEval = -infinity
7.          for each child of position
8.              eval = minimax(child, depth - 1, alpha, beta false)
9.              maxEval = max(maxEval, eval)
10.             alpha = max(alpha, eval)
11.             if beta <= alpha
12.                 break
13.         return maxEval
14.
15.     else
16.         minEval = +infinity
17.         for each child of position
18.             eval = minimax(child, depth - 1, alpha, beta true)
19.             minEval = min(minEval, eval)
20.             beta = min(beta, eval)
21.             if beta <= alpha
22.                 break
23.         return minEval
```

Fig 3. Structure of minimax algorithm with alpha-beta pruning[1]

[1] Sebastian Lague's https://pastebin.com/rZg1Mz9G

The previous schema for mini-max algorithm with alpha-beta pruning makes use of two main function:

- Evaluation of position: a function that takes the current representation of the board at level d of depth, and the player with respect to which the evaluation must be made, and calculates a heuristic evaluation of the board. The outcome of the function is dependant on the heuristics applied, which will be discussed further in the paper.

$$f : \{W, B\} \times \{\delta_1 \dots \delta n\}_d \mapsto \mathbb{R}$$

  Where the set $\{W, B\}$ represents the set of players, and $\{\delta_1 \dots \delta_n\}_d$ represents the set of n possible board settings at depth d.

- Child of position: given a specific board setting, a child of it with respect to player p is one of the outcomes resulting of the player p playing one of its allowed movements. Though for minimax algorithm the function is limited to yielding the board resulting of each movement, alpha-beta pruning requires a function that yields firstly those movements which are expected to produce higher values of the evaluation function. Thus, heuristics for sorting the movements and yielding them in the required order are needed.

4. Heuristic Evaluation

The heuristic evaluation of a given board with respect to a player $p \in \{W, B\}$ is carried out by the implemented function

```
CPUConfig.evaluate(player: {W, B}, position: Board)
```

This function is responsible for calling the chosen heuristic evaluation function specified in `CPUConfig.evFunction` for the current player. The implemented functions are `absoluteEvauation(player, board)` and `relativeEvaluation(player, board)`. These two evaluation function make use of the following features of the game in order to provide a evaluation of the board, which is the result of performing a linear combinations of these parameters in accordance with a weights vector $\overrightarrow{w}$:

$$f = \langle \overrightarrow{w}, (p_1, p_2, p_3) \rangle$$

A. Mobility

The concept of mobility is of high importance is this game. It is defined as the number of movements that a player can perform in its turn, and turns out to be a great indicator of success in the game. Thus, the Mobility parameter has a great weight in the evaluation of the board.

It is calculated as the length of the *legal_ moves* array of the given player.

B. Positional Strategy

The positional strategy has its foundations in the fact that having specific cells under our control provides us with certain advantage over our opponent, and this is reflected as a bonus for owning such cells. Similarly, controlling some cells might result in an advantage for the opponent, and thus the bonus obtained from having them must be negative.

Due to the symmetry of the game-board, the bonus for the cells is given only for one quarter:

	A	B	C	D
1	99	-8	8	6
2	-8	-24	-4	-3
3	8	-4	7	4
4	6	-3	4	0

Fig 4: Schema of bonuses per cell.[2]

Such schema represents only the increments of bonus for each cell with respect to the cell (D4) - and its symmetrics. The actual value of the bonus is given by the product of the cell value by the weight assigned to this parameter.

Although many literature assign positive values to the cells surrounding the corner, the fact that the only way of accessing the so precious corner is by jumping over a opponent tile that is set in those cells, makes us avoid placing any tile in those positions and thus setting the bonus value as negative.

C. Number of disks

The number of disks is yet another parameter that must be carefully studied to obtain a heuristic out of it. Although the goal of the game is to end up with the highest number of owned tiles, a strategy based continuously on maximising the number of tiles per round has proven to not be effective, as the high volatility of the tiles can drastically change the winner in the last moves:

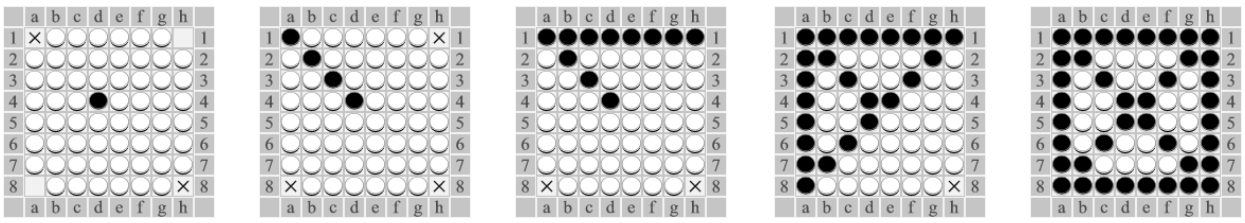


Fig 5: Example of how an seemingly advantageous position can lead to losing the game

This phenomenon is consequence of the Evaporation Strategy of "Less is more" strategy, which defends that fewer owned tiles on the board provides higher chances

[2] Schema obtained from http://www.samsoft.org.uk/reversi/strategy.htm#rules

of success. This is due to the fact that in order to increase the relative mobility with respect to the opponent, it is easier done having fewer tiles.

The evaporation strategy can be implemented assigning a negative weight to the parameter `number_of_disks`. However, a strategy based purely on this fact arises the problem of ending up with very few tiles when the match finishes. This issue can be solved if we use a varying weight that starts at a negative value but get positive when the end of the game gets closer.

The weights vector $\overrightarrow{w}$ is initially fixed. However, we can obtain more dynamism and better performance if we make it vary throughout the match as a function of `round`:

$$\overrightarrow{w} = \mathbf{f}(r),\ f : \mathbb{R} \mapsto \mathbb{R}^3$$

For the weights vector for $(legal\_moves, disk\_count, positional\_bonus)$, we can establish different values depending on the stage of the game following the previous idea. The chosen values are:

`if round < 30:` `  self.coefficients = (50, -10, 5)`	**Initial stage**: the player must aim to achieve high mobility rates and prevent the opponent from getting so. The number of allowed movements is essential, while the number of disks is needed to be minimal.
`elif round < 40:` `  self.coefficients = (10, 10, 5)`	**Consolidation stage**: the player aims to keep their mobility while increasing their number of disks under their control.
`else:` `  self.coefficients = (5, 30, 3)`	**Final stages**: the player must posses as many disks under their control as possible as the game gets closer to the end.

The previous parameters are calculated in order to yield an evaluation value. However, the way we combine these parameters, let alone the associated weights, allows us to define two different evaluation functions:

ABSOLUTE_EVALUATION: calculates the parameters with respect to the given player without taking into consideration the position of the opponent. In such way:
- Mobility is calculated as the raw number of legal movements available.
- Positional Strategy is the sum of the bonuses of each cell controlled by the player.
- Number of disks of the player's colour.

This implementation is easy to calculate and very straightforward, but does not take into account the relative position respect to the opponent. This prevent us from establish which movements allow us to have advantage over the other player. In order to solve this issue, we can define a second evaluation function:

RELATIVE_EVALUATION: the calculations are made as a comparison with the other opponent, obtaining an index of how distributed is the parameter between both players.

Each relative parameters $\hat{p}_i$ is calculated from the absolute parameter p_i:

$$\hat{p}_i = \frac{p_i(player)}{p_i(player) + p_i(opponent)}$$

This implementation supposes calculating the three parameters for both players, which duplicates the calculation time. Operations like finding the number of disks are not costly, but finding the legal movements for a given player is, for example.

5. Movement Yielding

Minimax on its own gets to perform certainly good for small ranges of movements, or shallow searches. However, when the game develops and the number of the player's legal moves is high enough, minimax starts losing efficiency and takes longer times to provide an optimal movement. In such situation where we cannot cut off searching depth either, it is when alpha-beta pruning comes useful.

Alpha-beta pruning works by avoiding searching those branches that are known to provide worse results than the one already got. In order for it to work properly, we need to provide the algorithm a way of start searching the movements that are expected to yield the best results.

The function `CPU.yieldMovements(possibe_moves: list, round: int)` applies certain heuristics to the possible_moves list in order to return those movements which are expected to originate good results. The function implementation is as follows:

```
def yieldMovements(self, possible_moves, round: int):
    '''Receives a list of moves, and yields them in the order specified by a
heuristic'''
    moves = []
    seen_moves = 0
    for movement in possible_moves:
        #Assumes every movement is a 1..* list of movements to the same cell
        cell_bonus = self.config.getCellBonus(target_cell)
        p = ProbabilityThereIsACellWithHigherBonusThan(cell_bonus, round)*(total-
seen_moves)/total
        if p > 0.3:
           moves.sortedInsert(movement)
        else: yield movement

    while len(moves) > 0:
        yield moves.pop(:-1)
```

Fig 6: pseudocode of the movements generation function

The function is implemented in python as a *generator,* as it allows us to deliver the results on the fly, and not iterating the whole array of movements each time. In order to deliver a movement m_i, it is first calculated the **probability that a better movement m_{i+k} is**

contained later if the array. For sake of simplicity, the probability is calculated as follows:

$$P = P(B(m_{i+k}) > B(m_i)) * \frac{Number\_of\_moves - seen\_moves}{Number\_of\_moves}$$

where $B(m)$ returns the bonus index of the end cell of the movement m.

Furthermore, the probability $P(Bonus(m_{i+k}) > Bonus(m_i))$ is the **probability that at a given round, a cell of bonus higher than $Bonus(m_i)$ is achievable.** This function is implemented by

```
getProbabilityThereIsACellWithHigherBonusThan(x: float, round: int)
```

and it is defined as:

$$P(B(m_{i+k}) > B(m_i)) = \frac{|b|}{64} * \frac{1}{F(round)}$$

$$\text{where } b = \{x : B(x) > b_0\}$$

and $F(round)$ is a function of round that must be an increasing function and represents how the cells can be obtained by the players during the development of the game. For simplicity, it has been defined to be a linear function $F(round) = k * round$.

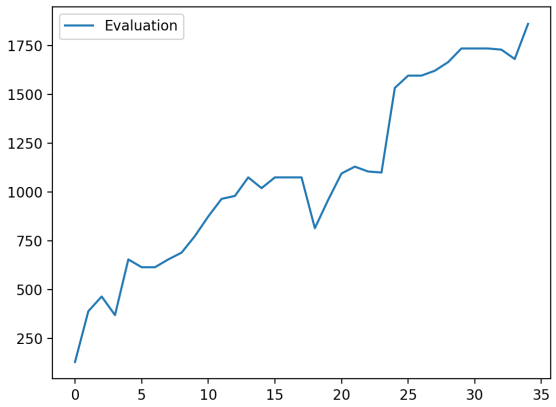
The movement selection is decided based on such probability. If the probability of finding a better movement is higher that a certain threshold, the movement is not returned and it is inserted in a sorted list based on a simple evaluation. When the whole movements list is iterated, then the movements that have not been delivered are yielded in descendant order or evaluation.

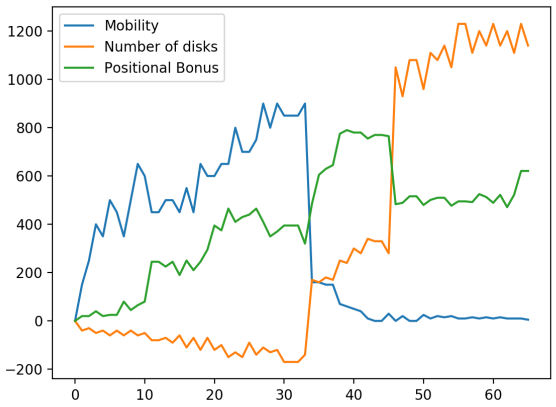
6. Evaluation Function Tests

ABSOLUTE FUNCTION: Computer vs Human

Black Tiles: 42 – White Tiles: 22

```
   A B C D E F G H
1 W W W W W W W B 1
2 B B B B B B B B 2
3 B B W B B W W B 3
4 B B B B B W B B 4
5 W W B B W B B B 5
6 W W B B B B B B 6
7 W W W W B B B W 7
8 B B B B B B B W 8
   A B C D E F G H
```

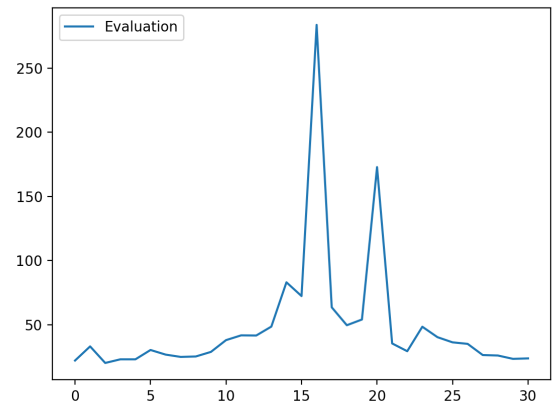


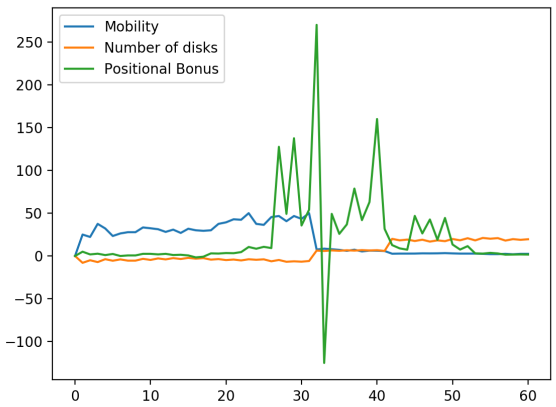


RELATIVE FUNCTION: Computer vs Human

Black Tiles: 38 – White Tiles: 26

```
   A B C D E F G H
1 B B B B B B W W 1
2 B B B B B W W W 2
3 B B B W W W W W 3
4 B W B B W B W W 4
5 B W B W B B W W 5
6 B B W B B B W W 6
7 B B B B W W W W 7
8 W B B B B B B B 8
   A B C D E F G H
```





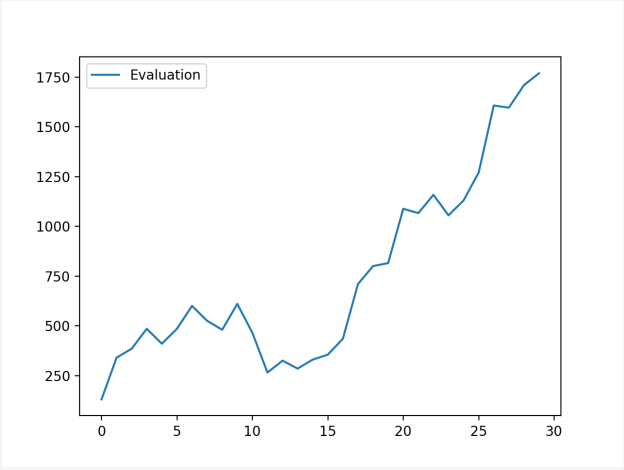
The tests above were made with a maximum depth of 4. We can observe that the absolute implementation yields better results when it comes to final scores. The stages of the game specified by the weight vector are clearly distinguishable.

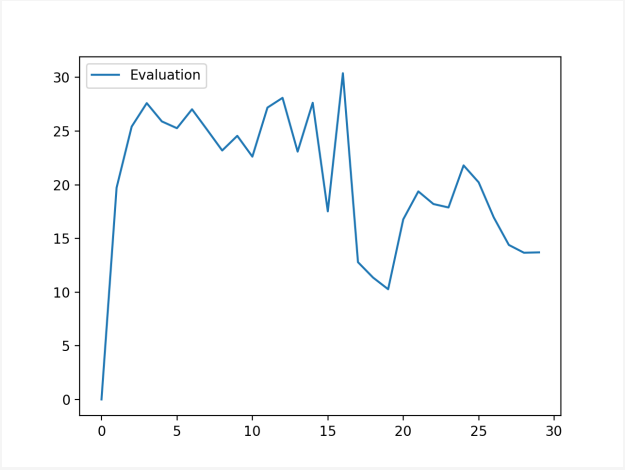
ABSOLUTE vs RELATIVE: Computer vs Computer

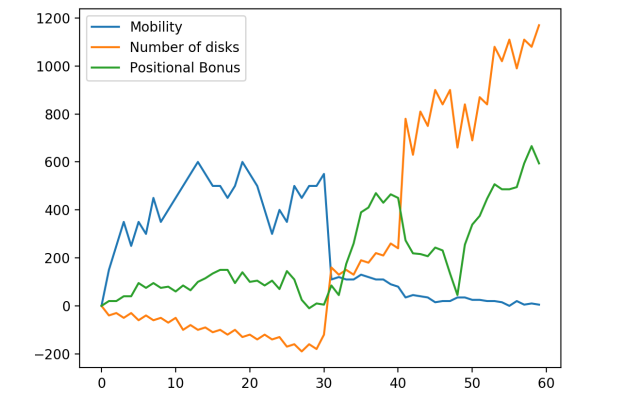
D = 3 ABSOLUTE	D = 3 RELATIVE

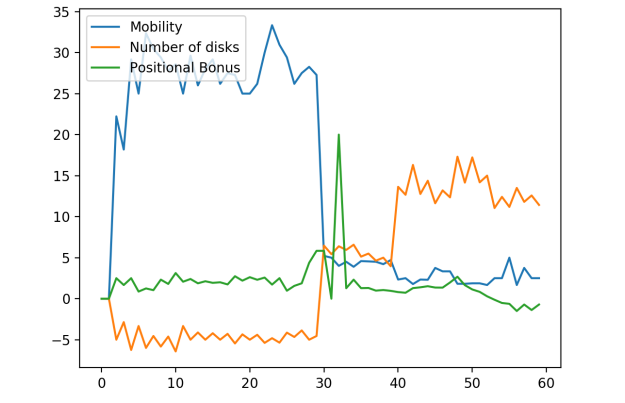
Black Tiles: 33 – White Tiles: 31

```
  A B C D E F G H
1 W B B B B B B B 1
2 W W W W B W W B 2
3 W W W B W W W B 3
4 B B B B W W W B 4
5 B B B W B W W B 5
6 B W W W W W W B 6
7 B W W W W W B B 7
8 B W B B B B B B 8
  A B C D E F G H
```







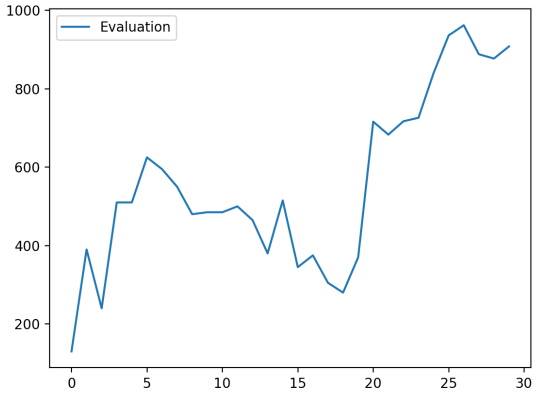


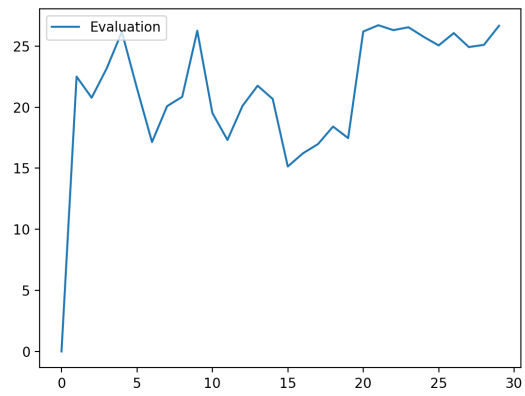
The function with the absolute evaluation of the parameters is proven to perform better than the Relative version. However, at maximum depth 3, both functions seem to compete against each other with almost same result, as we can see in the final number of disks owned by each one. We can observe that the evaluation value for the ABSOLUTE CPU player has maintained its always growing tendency, except from the change in weights that is given from beyond round 30 (Turn no. 15).

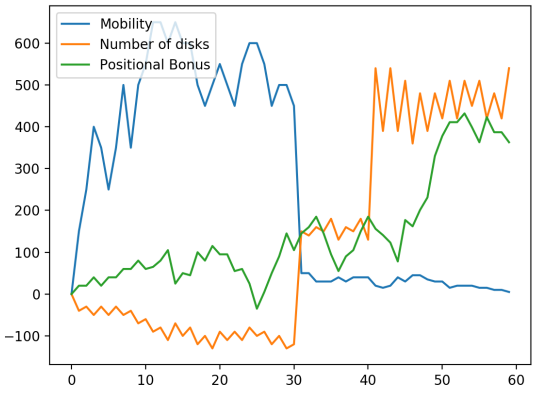
D = 4 ABSOLUTE	D = 4 RELATIVE

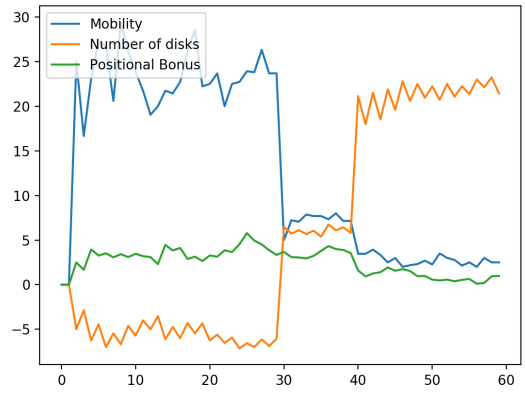
Black Tiles: 16 – **White Tiles: 48**

```
  A B C D E F G H
1 B B W W W W W W 1
2 W W B W W W W W 2
3 W W B B W B W W 3
4 W W W B W B W W 4
5 W W B W B B W W 5
6 W W W B W B W W 6
7 W W W W W W W W 7
8 W B B W W W W B 8
  A B C D E F G H
```









However, when we let the algorithm deepen another level of search, the relative implementation of the evaluation function comes out to be really aggressive. The phases of the game specified in the weights function are diligently followed by the CPU Player, and the IA quickly maximises the values for each parameters according to the weights vector.

7. Performance Testing

When it comes to performance, minimax is not efficient at all from depths equal or greater to 4. The amount of board positions that would not yield a successful result yet are still analysed is too high and gives time results that are too elevated for the game to be playable. Alpha-beta pruning comes handy in this situation reducing the execution times for each minimax instance. The following plots represent the times taken by the computer player at each turn:

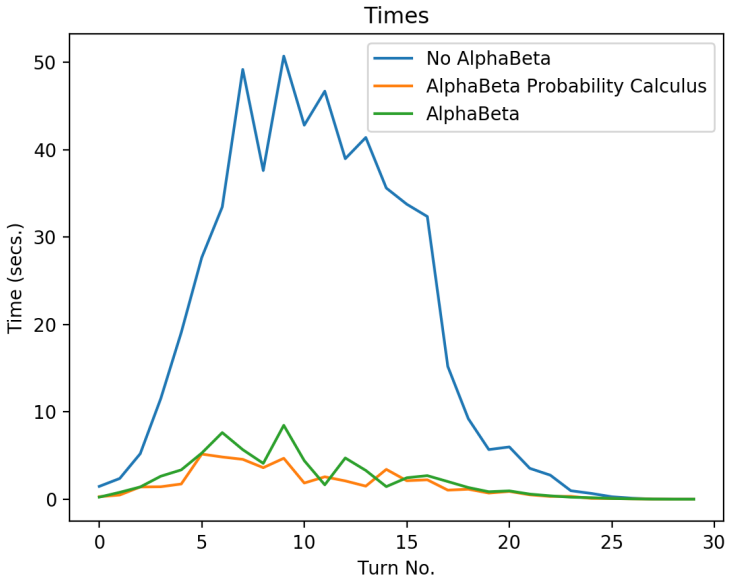


Fig 7: Time plots over turns of different versions of minimax.

We can observe a massive decrease of the execution times when alpha-beta is included. Two versions of alpha-beta have been included: the first one (green line) represents a alpha-beta pruning on a movements array that has not been sorted. The orange line plots the execution times for alpha-beta pruning applying a movement ordering heuristic. On average, each turn computation time is less for the implemented heuristic than for the raw movement vector.

Effect of the threshold probability

As it was described before, the implementation of alpha-beta pruning works by setting a probability threshold of finding a better movement during the movement yielding. The value of this threshold must be set empirically at least, and for different values we obtain the following values:

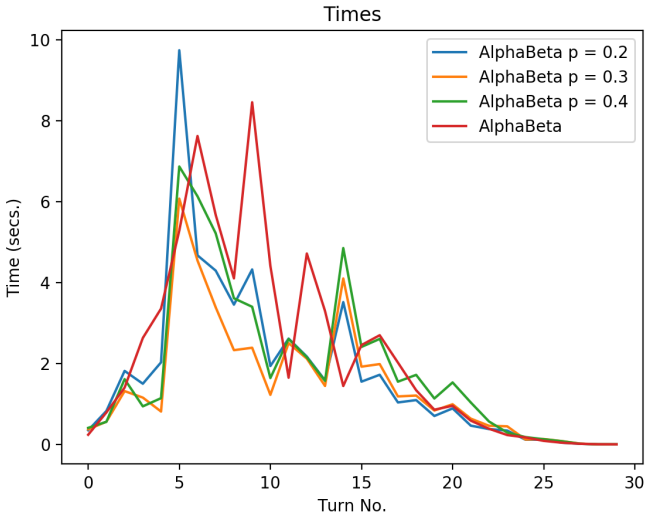


Fig 8: Execution times over turns for different values of threshold probability.

The AlphaBeta plot represents the performance of an unordered array of movements. We can observe that for a probability threshold of 0.3, the algorithm performs the most efficiently.

Effective branching factor

Another way of measuring the efficiency achieved by alpha-beta is by calculating the effective branching factor of the implementations. The branching factor is the number of successors generated by a tree node.

In the plot below, the branching factor of a pure mini-max execution drawn. This value is the average number of movements seen by minimax at each recursion step, and without alpha-beta it equals the number average of available movements for each player at the level of recursion. As a good implementation must tend to the square root of the branching factor of the non-alpha-beta version, such values are plotted in green.

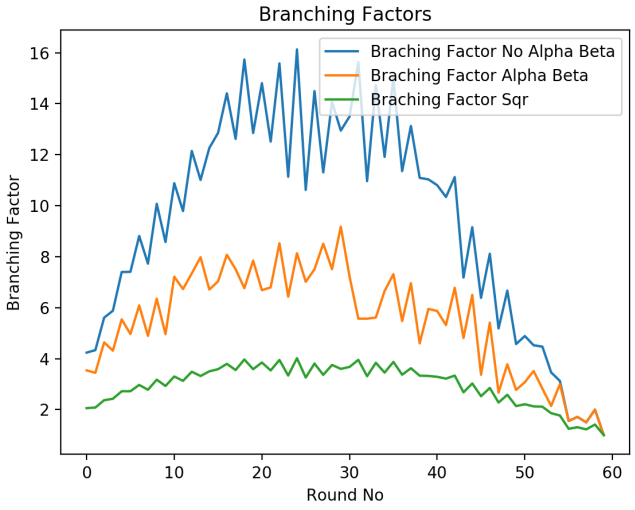


Fig 9: Evolution of the branching factor over the match for both opponents.

Effect of the branching factor in the execution time

We have defined the branching factor as the number of successors generated by each node in the tree. Thus, if b is the branching factor, the worst-time complexity of the algorithm is $O(b^d)$ as each level of the tree generates d successors. For depths 2, 3, and 4, we plot the times over the EBF:

d = 2 **d = 3** **d = 4**

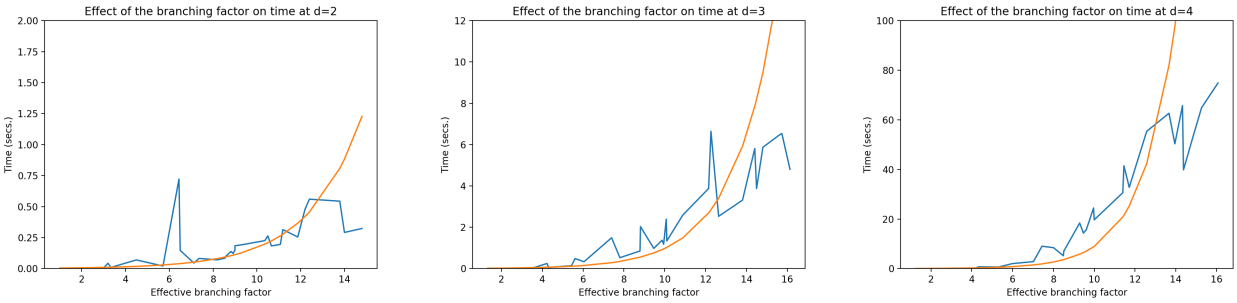


Fig 10: Execution times over branching factors, and exponential fitting

Except from certain peaks, the complexity of the implementation follows the exponential fitting curve. In the last rounds this behaviour starts fading due to the end coming to an end and thus, less legal movements are available on average.

Effect of the probability as a function of round

Recalling the definition of the `yieldingMovements` function, the condition met by a movement being yielded is that the probability of finding a better movement in the list is less than a certain threshold (p = 0.3). This probability is calculated based on the probability of finding a better cell to move to at a certain moment in the game:

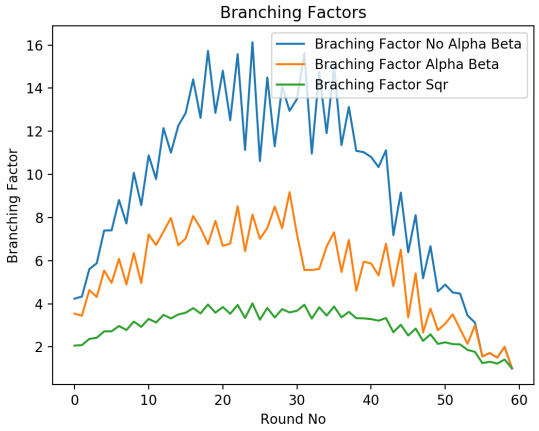
$$P(B(m_{i+k}) > B(m_i)) = \frac{|b|}{64} * \frac{1}{F(round)}$$

$$\text{where } b = \{x : B(x) > b_0\}$$

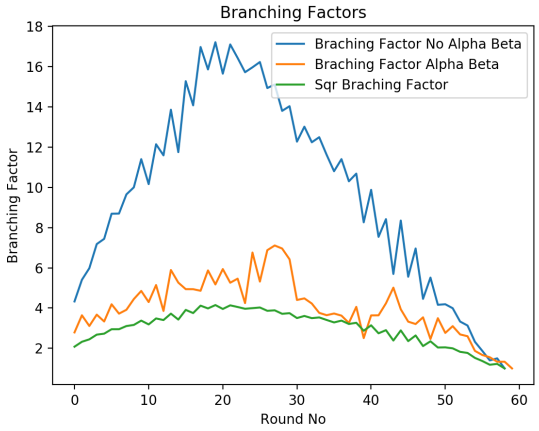
Due to the fact that as we go on in the game the probability of achieving certain places get lower, the function F(round) simulates such probability. For sake of simplicity, the implementation of such function has been decided to be linear of the form $F(round) = k * round$, although better modelings of the game should perform better. The constant k determines how fast probabilities decrease with each round, and must be determined empirically, if not with a training method. A better modelling of the game should perform better in terms of lower branching factors. The next plots represent the effect of such constant in the effective branching factors from the perspective of a CPUPlayer with depth 4:

Fig 11: Branching factors over turns for different values of k

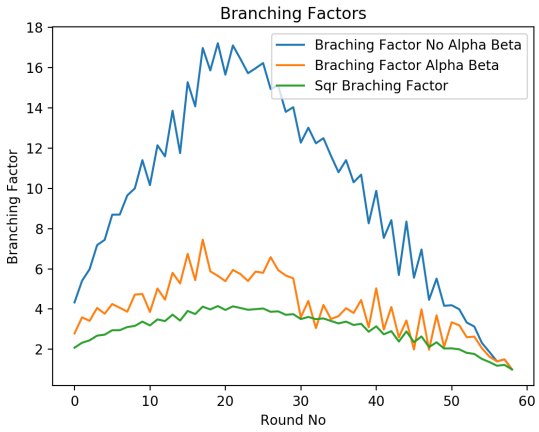
k = 0.01



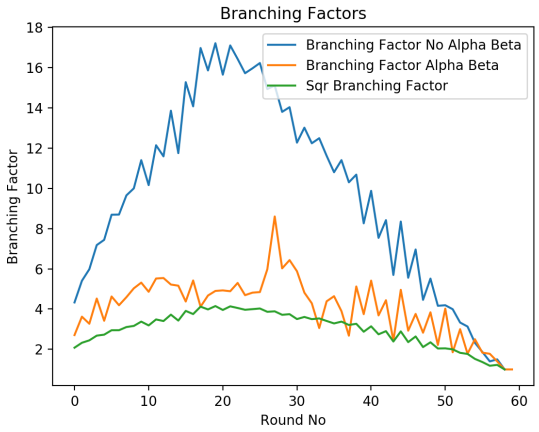
k = 0.1



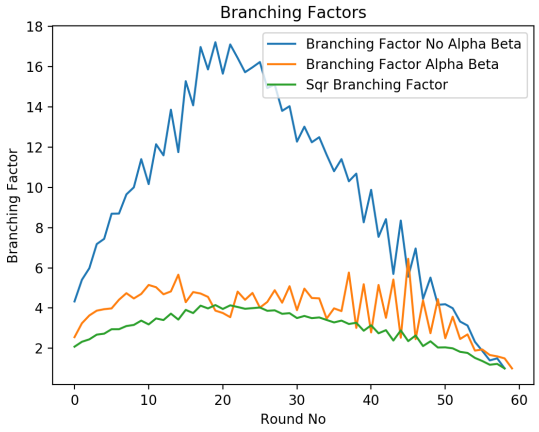
k = 0.5



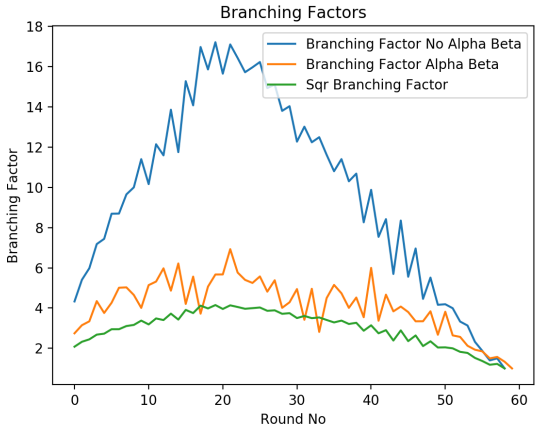
k = 0.6



k = 0.7



k = 1



As it can be observed, a constant of k = 0.7 tends to yield better results for middle stages of the game, but gets worse towards the end of the match. This is definitely due to the definition of the F(round) function, which does not get to reflect the nature of the game it

those later round. The inclusion of more complex functions can solve this problem to some extent.

8. Posible improvements

Though the implementation achieves certain level of expertise in the game, there is no doubt that changes can be made to increase its effectivity. One of the possible changes can be the reformulation of the evaluation parameters to be **Disc Stability, Mobility, and Parity**[3]. Furthermore, although alpha-beta pruning has shown itself to be effective enough to drastically reduce the computation times, a better formulation of the heuristic movement sorting function can be achieved. Furthermore, the weights of the parameters are chosen arbitrary according to some heuristics, but can be optimised to represent the adequate necessities of the players at each movement of the match.

[3] The Evolution of Strong Othello Programs, Michael Buro, NEC Research Institute.