CS1006 – Project 4:
Othello

Students: Simone Ivan Conte, Ivars Zubkans
Tutor: Ian Miguel

Context

Literature Review:

Othello, or Reversi, is a board game played by two players on a 8x8 board. Each players has a set of disks of opposite colours (black and white). The game starts with four pieces, two per player, in the middle of the board and the black starts first. Each player can place his piece in position so that there is at least one opponent's piece between any of his pieces and the new places piece. The opponent's piece that is in between the player's pieces will be flipped (turn of the opposite colour).
The main goal of the game is to have more pieces on the board at the end of the game (no more legal moves or empty squares on the board).
The game's names origins from the fact that one player reverses (i.e. Reversi) the opponent's pieces. At the beginning the pieces were red and green[1], but in 1972 Goro Hasegawa re-designed the game using black and white pieces. Since this date the game has also been known as Othello. The name Othello derives from the Shakespeare's play: The Tragedy of Othello, the Moor of Venice, where the author continuously underlines the contrast between the Moor Othello (Black) and Iago (White)[2].
However, the origin of the game itself has not been confirmed yet. It is claimed, in fact, that two Englishmen, Lewis Waterman and John W. Mollett, have invented independently the game in 1883[3].
Like many board games, Othello also became a computer game in 1976. The 22nd November of that year the american newspaper "TIME" wrote about an Othello program called IAGO and developed at the University of Caltech, California[4]. Other programs were then developed, but the first sellable computer version of the game was "published and developed by Nintendo" only in 1978[5].
Always in the 70s computer scientists started challenging each other by developing Artificial Intelligence (AI) implementations of Othello. In France, the journal "Personal Computer", organized the first Othello competition for machines. There were only 6 machines competing and the winner was Philip Keller and its SWPTC 6800[6].
In the April 1980 the international chess master David Levy published an article on the journal "Personal Computer" on games with reflection, such as Othello. That article was followed by many others for the two years after. Levy's work was an important contribute for developing better AIs. Meanwhile the competitions were held at regular intervals. On the 19th April 1980 Jean Maingour and its TRS 80 won the competition, with an average score of 51 to 13, thanks to the use of the Alpha-Beta pruning algorithm[7]. Americans were also working hard to create the best Othello program and in September 1980 the first international Othello competition for machines was held in France.
In the year 1989 the first Computer Olympiad took place in London, UK. There were various games, including Othello. There were 15 participants and the winner was a British machine called Polygon[8].
On the 8th March 1992, in Paris, it finally took place the first competition man-machine. There were 7 best French machines against 7 French grand-master players. At the end of the competition the machines' team won 29.5 points to 19.5[9]. This was an important result, since it showed to the entire world that machines can beat grand-master players.
In the year 1997 the Othello program Logistello, written by the programmer and Othello grand-master Michael Buro, beat the human world champion Takeshi Murakami six games to none[10]. Logistello made use of disc patterns and statistical calculations when evaluating a certain position on the board.
Today programs can look even more than 25 moves ahead and their evaluation functions are so

1 http://members.chello.at/theodor.lauppert/games/othello.htm visited on 9th May 2011 at 16.20
2 http://www.shakespeare-online.com/plays/othello/othelloessay3.html visited on 9th May 2011 at 16.00
3 http://www.beppi.it/public/OthelloMuseum/pages/-history-/history-of-reversi-through-documentation.php visited on 9th May 2011 at 16.15
4 http://brunodlb.pagesperso-orange.fr/ot_story.htm visited on 9th May 2011 at 16.50
5 http://nintendo.wikia.com/wiki/Computer_Othello visited on 9th May 2011 at 16.30
6 http://brunodlb.pagesperso-orange.fr/ot_story.htm visited on 9th May 2011 at 17.17
7 http://brunodlb.pagesperso-orange.fr/ot_story.htm visited on 9th May 2011 at 20.20
8 http://www.grappa.univ-lille3.fr/icga/tournament.php?id=130 visited on 9th May 2011 at 20.31
9 http://brunodlb.pagesperso-orange.fr/ot_story.htm visited on 9th May 2011 at 20.38
10 http://skatgame.net/mburo/event.html visited on 9th May 2011 at 20.44

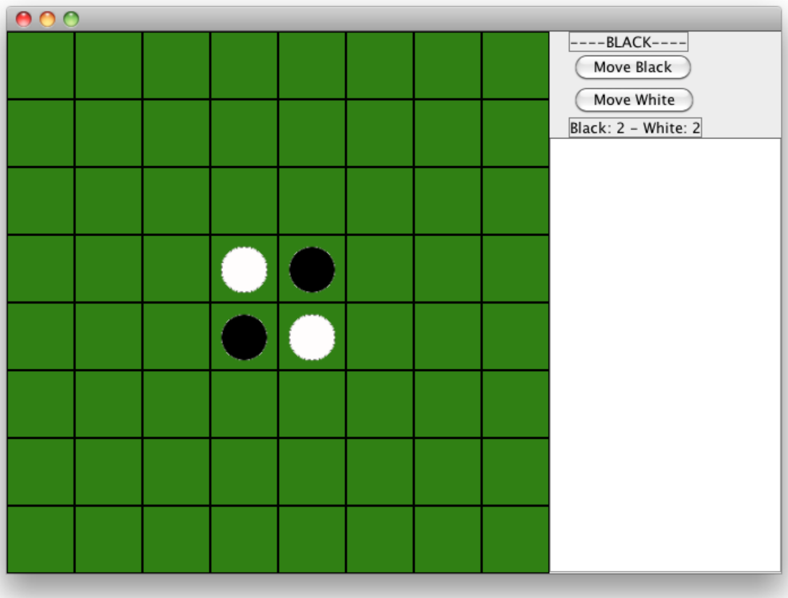
complex that to beat a good Othello program implementation has become extremely challenging.

Design:

For this fourth project we were already given some packages so that we could focus more on the Game itself and the AI, other than trying to figure out how to create a proper RMI (Remote Method Interface) system.

The main goals of this project were to build a GUI (Graphical User Interface), a Board (with legal moves checking) and an AI.

The GUI



The GUI is created dynamically in the GUIFactory that is a class already given for this project. The GUIFactory instantiate OthelloJFrame, where all the graphical objects are created and put together. The OthelloJFrame extends JFrame and contains a main JPanel. The JPanel contains 64 othelloSquares and the layout is GridLayout so that the othelloSquares can be places systematically in a grid. OthelloSquare is a class that extends JButton and defines the main properties for the buttons to be used in the board (i.e. Color). Whenever one of these buttons is pressed, the Game is notified and plays the move if this is valid.
In addition the JFrame contains two JLabels, showing whose turn it is to play and the actual score, and two JButtons, used for making the AI to move, on the right side. Each of these buttons deals with the black or white player. When one of the buttons is pressed the AI calculates the best move for the player and the method Play from the Game is called to make that specific move.
On the right-side there is also a JEditorPane used to display the time taken by the AI for making each move.
The OthelloJFrame also implements Observer and in its constructor it is defined as observer of game. Whenever the Play method in the Game is called and the move is successfully made, the Game notifies its observers (i.e. OthelloJFrame). The OthelloJFrame implements the method Update from Observer by looping over the Board and updating the GUI depending on the content of the Board.

The Board

The Board is instantiated dynamically in the BoardFactory, which was already given for this project. In the constructor of the Board the four initial pieces are placed first. Than the actual content of the board is converted to bitBoard through the constructor of the BitBoard class. The bitBoard then returns two primitive long numbers, one for white pieces and one for black pieces, which are used to find the moves to be stored in the two ArrayLists: whitePossibleMoves and blackPossibleMoves.
The method Play in the class Board checks first if the move proposed is valid, then if this is true it updates the bitBoard and also the list of possible moves. The stage of the game is also increased by one. This is an important variable, since its value is used by the AI in its evaluation function.

The AI

Overview

Artificial Intelligence (AI) is a branch of Computer Science that focus on creating systems capable of re-generating human behaviour. The Turing's test is a test used to evaluate a machine's creativity and human behaviour. So far no machine has ever passed this test. In board game, especially, it is much harder for a machine to use creativity to chose what move is the best as a grand-master would generally do. Thus, machines use brute-force (or semi-brute force) search algorithms to look moves ahead in the game and then evaluate rationally which move is the best.

It is quite evident that the better the evaluation function is more likely the AI will win the game. In most of the international competitions computers have a set limited time of thinking, therefore calculation's speed also becomes an extremely important factor in the game. A full brute-force search is also much quite harder to accomplish, thus most of the AI implement Alpha-Beta algorithms (we used NegaScout). Alpha-Beta searches , in fact, are much faster than the usual MiniMax since the former excludes nodes (or branches) that are not worthy to evaluate.

The bitBoard

The main purpose of the AI is to look as many moves ahead as possible into a certain given time. To accomplish this task efficiently we decided to use bitboards. The class BitBoard stores two long variables, whitePieces and blackPieces, that contains the informations about the pieces' position for each player. If a bit in the long is 1 then there is a piece in that position. The leftmost bit represents top-left corner, and the direction is from left to right and top to bottom.

We used the primitive type long because the game Othello is played on a 64 squares board and long is 64 bits. In addition nowadays machines (i.e. the machine in the laboratory) use 64bits processors, so working on long variables won't slow down the system.

In addition the use of long variables allow us to make calculations faster. In fact, by applying shift and/or boolean operations most of the methods are speed up enormously.

Below we show the pseudo-code and its java implementation we came up to find the possible moves for the player.

Pseudo-code

```
Create bitBoards for the player's pieces, opponent's pieces, for the empty
squares (1-empty square, 0-occupied by one of the two players) and
possibleMoves.

Create a temporary bitBoard, shiftedPieces.

for index from 0 to DIRECTIONS {
    shiftedPieces = shift PlayersPieces by one step to the direction index;

    shiftedPieces = shiftedPieces & opponentsPieces;

    while(shifterPieces != 0) {
        shiftedPieces = shift PlayersPieces by one step to the direction
index;

        possibleMoves = possibleMoves or (shiftedPieces and emptyCells);

        shiftedPieces = shiftedPieces & opponentsPieces;
    }
}

return possibleMoves;
```

Java - Code

```java
// Sets the colours right for players and opponents pieces.
long playersPieces;
long opponentsPieces;
if (isWhiteMoves) {
          playersPieces = whitePieces;
          opponentsPieces = blackPieces;
          }
          else {
          playersPieces = blackPieces;
          opponentsPieces = whitePieces;
          }

// Bitboards representing the empty cells and possible moves.
long emptyCells = ~(playersPieces | opponentsPieces);
long possibleMoves = 0L;

// Loops over all 8 directions and shifts players pieces in that
direction to check for possible moves.
long shiftedPieces;
for (int i = 0; i < DIRECTIONS; i++) {

// Shifts the players pieces in a specific direction.
shiftedPieces = shiftPieces(playersPieces, i);

// Removes players pieces that didn't overlap with opponents
pieces after the shift.
shiftedPieces = shiftedPieces & opponentsPieces;

// Check if there are actually opponent's adjacent pieces.
while(shiftedPieces != 0) {

          // Keeps shifting the players pieces in the specific
direction.
          shiftedPieces = shiftPieces(shiftedPieces, i);

          // If after shifting and overlapping with opponent's a
player's piece goes on empty cell, it's possible move, so adds
it.
          possibleMoves = possibleMoves | (shiftedPieces &
emptyCells);

          // Removes players pieces that didn't overlap with
opponents pieces after the shift.
          shiftedPieces = shiftedPieces & opponentsPieces;
          }
}

return possibleMoves;
```

The shifting into a certain direction is done by shifting the bits by a certain number and by applying the right BitMask. Below is the code we used to perform this computation. Running some tests we found out that this approach was really efficient, in fact we were able to run it 10 billion times in less than 6.5 seconds.

```java
/**
 * Shifts all the pieces in the passed bitBoard in the specified direction.
 * @param direction
 * 0 is East.
 * 1 is NE
 * 2 is N
 * 3 is NW
 * 4 is W
 * 5 is SW
 * 6 is S
 * 7 is SE
 * @return long representing shifted Bitboard.
 */
protected static long shiftPieces(long bitBoard, int direction) {
          switch (direction) {
          case 0:
                    return (bitBoard >>> 1) & REMOVE_FIRST_COLUMN_BITMASK;
          case 1:
                    return (bitBoard << 7) & REMOVE_LAST_ROW_AND_FIRST_COLUMN_BITMASK;
          case 2:
                    return (bitBoard << 8) & REMOVE_LAST_ROW_BITMASK;
          case 3:
                    return (bitBoard << 9) & REMOVE_LAST_ROW_AND_LAST_COLUMN_BITMASK;
```

```
			case 4:
				return (bitBoard << 1) & REMOVE_LAST_COLUMN_BITMASK;
			case 5:
				return (bitBoard >>> 7) & REMOVE_FIRST_ROW_AND_LAST_COLUMN_BITMASK;
			case 6:
				return (bitBoard >>> 8) & REMOVE_FIRST_ROW_BITMASK;
			case 7:
				return (bitBoard >>> 9) & REMOVE_FIRST_ROW_AND_FIRST_COLUMN_BITMASK;
			default:
				System.out.println("Wrong direction passed to the shiftPieces method.");
				break;
			}

			return bitBoard;
		}
```

The BitMasks are primitive long variables used to remove certain parts of the board. To accomplish this we have written a small method to convert a bitBoard of 1s and 0s to long.
This was done by applying the following formula:

$$\ldots + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 + b_{-1} \cdot 2^{-1} + b_{-2} \cdot 2^{-2} + \ldots$$[11]

Below it is the code we used for creating one of the bitMasks used in our implementation of Othello.

```
int[][] tMat2 = { {0, 0, 0, 1, 1, 0, 0, 0},
		{0, 0, 0, 0, 0, 0, 0, 0},
		{0, 0, 0, 0, 0, 0, 0, 0},
		{1, 0, 0, 0, 0, 0, 0, 1},
		{1, 0, 0, 0, 0, 0, 0, 1},
		{0, 0, 0, 0, 0, 0, 0, 0},
		{0, 0, 0, 0, 0, 0, 0, 0},
		{0, 0, 0, 1, 1, 0, 0, 0}};

		long bitMask = 0;
		power2 = 1;
		for (int i = tMat2.length - 1; i >= 0; i--) {
			for (int j = tMat2[0].length - 1; j >= 0; j--) {
				bitMask += power2 * tMat2[i][j];
				power2 *= 2;
			}
		}
```

The NegaScout (Principal Variation Search) algorithm:

In a game like Othello even looking a move ahead can make the difference since pieces flip really easily. To search as much moves ahead as possible we decided to implement the Principal Variation Search (PVS). The PVS or NegaScout[12] is an Alpha-Beta pruning improved search algorithm, introduced first by Alexander Reinefeld in 1983. This algorithm is a negamax (simlar to minimax, but relies on the zero-sum property of a two-player game[13]) that can be even 10% faster than Alpha-Beta. This is achieved because the PVS will never examine a node that the Alpha-Beta may prune. However this gets really efficient only if the moves are ordered correctly.
In this algorithm, move ordering is used so to find the best line of play when evaluating a node of the tree. At the beginning of the search the PVS uses a normal alpha-beta window, however further searches are made using null-windows (0 < beta – alpha < Small value). Null-windows speed up the search because they restrict radically the range of search, including only reasonable moves.

11 http://mathworld.wolfram.com/Binary.html visited on 10th May 2011 at 19.07
12 "Search-wise PVS and Negascout are identical, they are just formulated differently" [**Yngvi Björnsson**, January 05, 2000]
13 http://en.wikipedia.org/wiki/Negamax visited on 10th May 2011 at 19.21

Whenever a beta-cutoff occurs the algorithm performs a re-search with an increased window size.

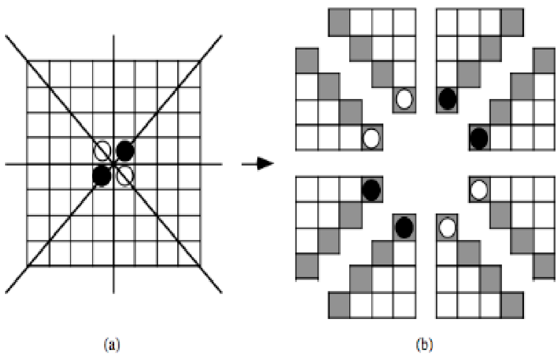
Negascout Pseudo-Code

```
b := ß                                              (* initial window is (-ß, -ɑ) *)
foreach child of node
       a := -negascout (child, depth - 1, -b, -ɑ)
       if ɑ < a < ß and child is not first child      (* check if null-window failed high *)
          ɑ := -negascout(child, depth - 1, -ß, -ɑ)  (* full re-search *)
          ɑ := max(ɑ, a)
       if ɑ ≥ ß
          return ɑ                                   (* Beta cut-off *)
       b := ɑ + 1                                     (* set new null window *)
return ɑ;[14]
```

The evaluation function

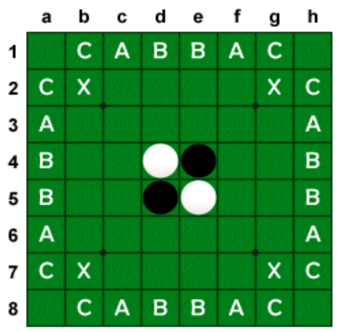
Disc Square

The board in the Othello game is symmetric horizontally, vertically and along the two diagonals as shown in the picture below.



This is quite a good property because it allows us to use only few bitMasks and speed up the calculations.

We identified four different positions on the board plus the corners as shown in the picture below.



14 http://en.wikipedia.org/wiki/Negascout visited on 10th May 2011 at 21.21

The difference between the player values and the opponent values is taken into consideration:

discSquare = playerDiscSquares - opponentDiscSquares

Mobility

Mobility is the measure of legal moves a player has. This is quite an important property in the game Othello, where one of the aim is to play moves so to make the opponent to pass, or to force him to play a bad move, e.g. giving away a corner.
In calculating the mobility value for a certain state of the board we apply the following arithmetic formula:

mobility = playerPossibleMoves - opponentPossibleMoves

Pieces differential

This is merely the difference of pieces between the player and the opponent. This feature is not that much important at the beginning of the game, but it becomes fundamental toward the end of the game when the pieces differential determines who is going to be the winner.

Parity

Since in a game without passes the white makes the last move, he gets a slight advantage as his last pieces cannot be flipped. So there is bonus for the player who moves last and if there is pass it changes.

Stability

Stability is the measure of the pieces which cannot be flipped by the opponent. This gives a better measure of the situation of the game. Implementing this function is however quite complex and it would have slowed down the entire AI by a lot, so we decided to write just a simple stability function. Our stability function looks for pieces along the edges next to each other starting from the four corners.

Stability = numberOfPlayerStablePieces - numberOfOpponentStablePieces

Frontier differential

The frontier or potential mobility is the measure of the number of pieces adjacent to an empty square. The higher the frontier values is the less likely the player will increase its mobility.
In our implementation the frontier differential may be represented by the following formula:

frontier differential = opponentFrontier – playerFrontier

Each player frontier is calculated by shifting by one step its bitBoard in all eight directions. The logical AND operator is applied with the bitBoard of empty squares to find out which pieces are actually adjacent to an empty square.
Below is the code we wrote:

```
whiteEmptySquares = blackEmptySquares = ~(bitBoard.getWhitePieces() | bitBoard.getBlackPieces());

long temp = 0;

for(int i = 0; i < BitBoard.DIRECTIONS; i++) {
        temp = BitBoard.shiftPieces(bitBoard.getWhitePieces(), i) & whiteEmptySquares;
        whiteEmptySquares = whiteEmptySquares - temp;

        frontierDifferentialEvaluation -= Long.bitCount(temp);

        temp = BitBoard.shiftPieces(bitBoard.getBlackPieces(), i) & blackEmptySquares;
        blackEmptySquares = blackEmptySquares - temp;

        frontierDifferentialEvaluation += Long.bitCount(temp);
}

return frontierDifferentialEvaluation;
```

End of the game

If there are no legal moves and the game is over, then the evaluation function just determines which player has more pieces and returns a huge number (bigger than any other possible heuristics value) for that player.

Stages of the game

The strategy changes drastically as game progress. At the start of the game one wants to reduce his number of pieces as it increases his mobility and forces opponent to make bad moves, while at end of the game the aim is to have as many pieces as possible.

For this reason our heuristics function determines in which stage it is, depending on the number of moves. Then depending on the stage it has different weights for the evaluation features.

The Killer Table

Since NegaScout relies on moves being ordered we are using a killer table. For each move it orders all other possible moves by how frequently they were the best response of our search algorithm to this move in previously played games. For this reason we have historyKillerTable which stores when a move was a response to certain move. Then from this table the killer table is made.
To populate it we played many games between our program where one version played semi-randomly – it sometimes did not pick the best possible move, but maybe the 2nd or the 3rd one. We also played games against applets on internet and other programs like WZebra[15] or Cassio[16].
The implementation of killer tables allowed our search algorithm to go from depth 8 to depth 9.

The Transposition Tables

In one of our versions we also implemented transposition tables. By testing the program numerous times against other versions of itself we found out that the version with the transposition tables was slower than without. The idea behind transposition tables was that in Othello the same position on the board can be reached by different paths, hence the searching algorithm would evaluate the same positions more than once. So by storing the positions value, it could be retrieved without research.

So transposition table stores a position on the board and its value until certain depth, so if the required depth by the search algorithm is smaller, then it just takes that value from the table, it also checks the alpha, beta bounds, as they affect the result of the search.

The transposition tables were implemented as LinkedHashMap, as it allows to delete least recently used entries to limit the size of the table, as it is impossible to store all positions. Hash-table also

15 It is possible to download Wzebra at this link: http://radagast.se/othello/ visited on 10th May 2011 at 23.00
16 It is possible to download Cassio at this link: http://cassio.free.fr/ visited on 10th May 2011 at 23.11

allows to access its elements in constant time.

A possible reason for transposition tables being slower is that, if the table is large it takes a lot of time access and update it, but if it is small then it is not used often enough to increase the speed.

The Time-Management

The AI has also a time-management system, so that after a certain amount of time it stops searching and chooses the best found move so far. This is a requirement we had to implement to participate to competitions.

When writing a time-management system we took in consideration the fact that at different stages of the game the branching factor[17] is different, which means the AI is faster at some points while it gets slower at other stages.

Every time we run the NegaScout we count the number of nodes searched and the number of children evaluated. Thus we can get easily the branching factor by applying the following formula:

$$\text{Branching factor} = \text{numberChildren} / \text{numberOfNodesSearched}$$

After evaluating the best move we also use the time spent using the NegaScout. To calculate the average time spent per node we apply this other formula:

$$\text{Average time spent per node} = \text{Time} / \text{numberOfNodesSearched}$$

Since the number of nodes increases exponentially in the tree so does the time:

$$\text{TIME} = (\text{branching factor})^{\text{depth}} \times \text{Average time spent per node}$$

By fixing the TIME to a fixed value (i.e. 5 seconds) we can predict what depth we can use for the next search by simply using the following equation:

$$\text{depth} = \log(\text{TIME} / \text{Average time spent per node}) / \log(\text{branching factor})$$

However this approximation is always a bit lower than what we want to use since using NegaScout we cutoff Beta and lose branches of the tree. Thus every time we just add a constant factor (DEPTH_GAIN_FROM_NEGASCOUT) to adjust the depth to use.

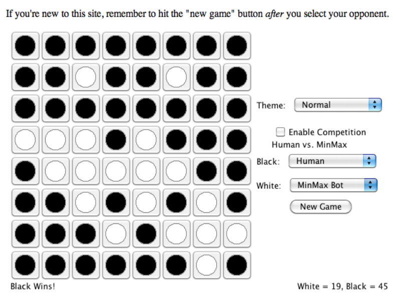
This estimation appears to be good, especially when using move ordering, but sometimes the search lasts more than the time limit. To solve this problem every time the NegaScout is entered it is checked if the time limits has been reached. If this is the case we break the search and return the best move found so far.

17 “The average number of branches (successors) from a (typical) node in a tree. It indicates the bushiness and hence the complexity of a tree.” [Encyclopedia.com] (http://www.encyclopedia.com/doc/1O11-branchingfactor.html visited on 10th May 2011 at 23.18)

Testing:

Describe how you tested your program

– Against the applet: http://www.site-constructor.com/othello/



If you're new to this site, remember to hit the "new game" button *after* you select your opponent.

Theme: Normal

Enable Competition
Human vs. MinMax

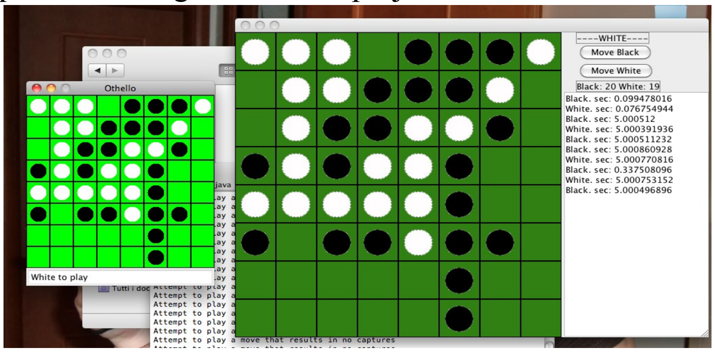
Black: Human

White: MinMax Bot

Black Wins!

White = 19, Black = 45

– Against the implementation given for this project

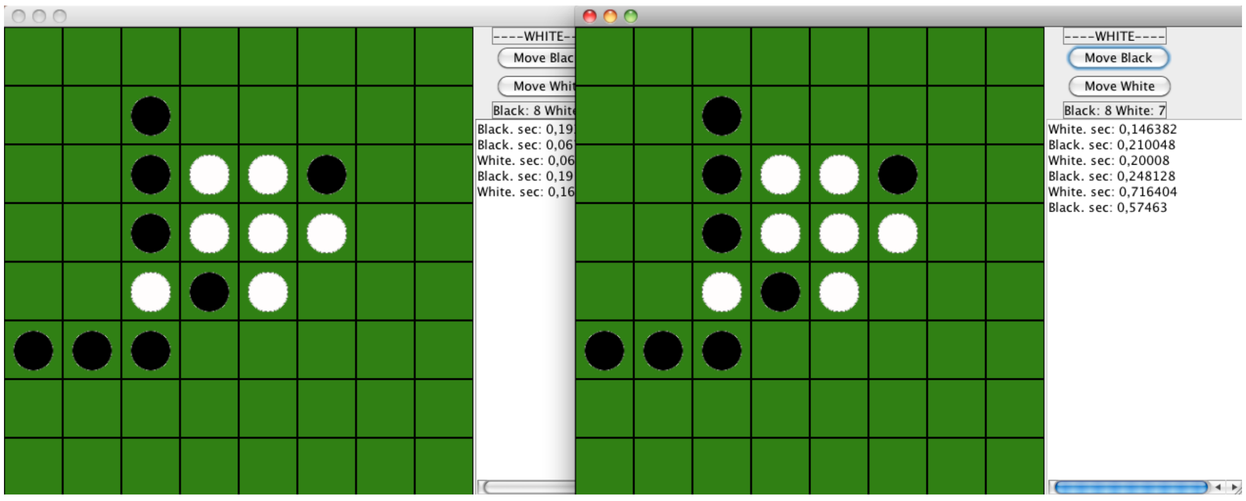


```
Black: 20 White: 19
Black. sec: 0.099478016
White. sec: 0.076754944
Black. sec: 5.000512
White. sec: 5.000391936
Black. sec: 5.000511232
Black. sec: 5.000860928
White. sec: 5.000770816
Black. sec: 0.337508096
White. sec: 5.000753152
Black. sec: 5.000496896
```

White to play

```
Attempt to play a move that results in no captures
```

– Against ourselves

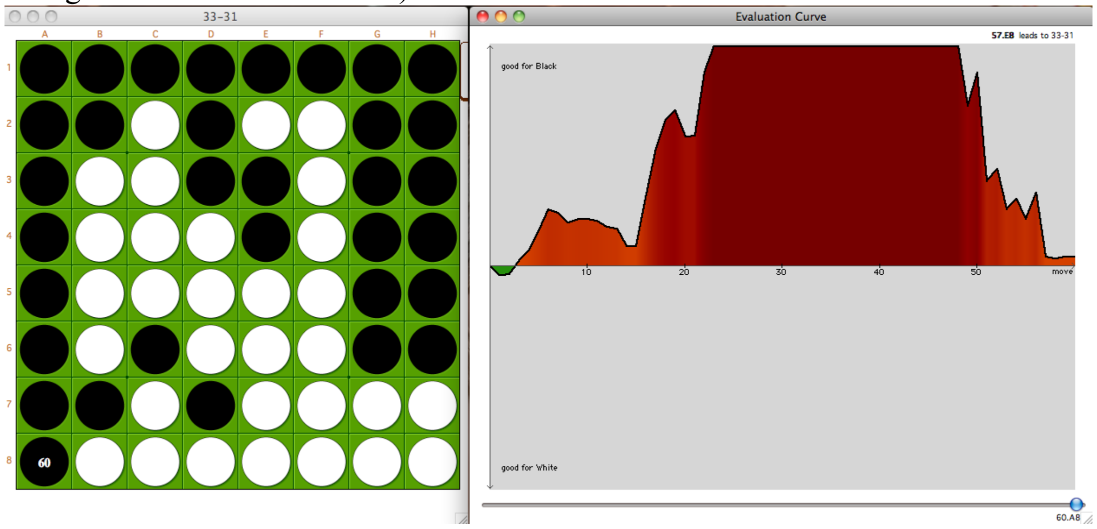


```
Black: 8 Whit
Black. sec: 0,19
Black. sec: 0,06
White. sec: 0,06
Black. sec: 0,19
White. sec: 0,16
```

```
Black: 8 White: 7
White. sec: 0,146382
Black. sec: 0,210048
White. sec: 0,20008
Black. sec: 0,248128
White. sec: 0,716404
Black. sec: 0,57463
```

– Against other students' implementation (August, David, Alex and Neil)

- Against Cassio (in the Graph below we played black and the red curve in the graph shows how good the black has moved).



  From the graph above it appears like our AI has played really good in the middle of the game and then it decided it could lose some pieces since at a certain point it could see the end of the game.
- Against WZebra

Conclusion and Evaluation:

Our project satisfies all the advanced requirements (GUI and Smart AI).
Our main achievement were about the use of bitBoards and the implementation of a good AI. At the beginning we found difficult to work with bitBoards, especially when we were trying to rotate or shift them.
The heuristics of the AI could also be improved by playing more and more games or by implementing a genetic algorithm. In addition we could have improved the stability function and other functions (i.e. patterns, statistical linear regression function) if more time was available to us
Other improvements could be achieved by using an opening book and also searching moves during the opponent's time so to get better transposition tables (if we were to use them). These additional features could also speed up the system.

Bibliography:

- ICGA Tournaments http://www.grappa.univ-lille3.fr/
- Takeshi vs Logistello http://skatgame.net/
- Members Othello http://members.chello.at/
- Shakespeare online http://www.shakespeare-online.com/
- The Othello Museum http://www.beppi.it/
- ANTHOLOGIE DES PROGRAMMES D'OTHELLO http://brunodlb.pagesperso-orange.fr/
- Nintendo Wikia http://nintendo.wikia.com/
- Mathoworld – Wolfram http://mathworld.wolfram.com/
- Wikipedia (en) http://en.wikipedia.org/wiki/
- Encyclopedia.com http://www.encyclopedia.com/