# A New Experience: O-Thell-Us – An AI Project

**Mathias Ganter[1] and Jonas Klink[2]**

[1] Department of Genome Sciences, University of Washington, Seattle, USA
[2] Department of Computer Science and Engineering, University of Washington, Seattle, USA

Contact: mganter@u.washington.edu jklink@u.washington.edu

**Abstract**

Othello is a two player strategy board game established in England in the 1880s. It is also known as Reversi. The motivation to implement this game as an AI project comes from adversial search strategies, applying a minimax algorithm with an alpha-beta pruning not taking branches into account that cannot possibly influence the final decision. By adding various heuristic evaluation functions, the search problem can be solved in a more effective way, thus yielding better moves compared to a simple greedy method.

## Introduction

This Othello implementation covers various aspects of the game. Firstly, we implemented it as a Java application to be more user-friendly and secondly, we focused on a reasonable level of intelligence by taking only valid moves and rules into account.

The final version only allows one player-mode, i.e. human versus computer.

This paper focuses on an overview of the Othello game including some general strategies, its implementation in Java, and finally adversial search strategies with a short outline of the minimax algorithm with an alpha-beta pruning. The real focus lies on the heuristics optimizing the AI. Up to date computer programs of Othello are able to beat humans easily, as Logistello did in 1997 with the then current world champion Takeshi Murakami. That program's considerable playing strength is mainly due to several (by the time) new approaches for the construction of evaluation features, their combination, selective search, and learning from previous games.

## Overview of the Othello Game

Othello is now a popular two player strategy board game in many countries - though it is not as popular as chess or backgammon, there are numerous players around the world (although most are in Asia). In a regular Othello game, on which we are focusing, there is a green 8-by-8-board with 64 squares. The discs are black and white.

Contact: mganter@u.washington.edu
jklink@u.washington.edu

### Rules of the Game

The game starts with fixed board positions (Figure 1), with black taking the first move. When it is the turn of the white player, he can place a disc of his color onto one of the empty squares on the board, provided that this move flips at least one of the opponent's discs.

Flipping is done by evaluating the surrounding squares; vertically, horizontally and diagonally (totally 8 directions, except for in corners and edges, where the board limits the options). If this evaluation finds an unbroken line of the opponent's discs, followed by a disc of the player's own color, then a flip of all intermediate discs is done in this direction. This procedure alternates between the two players. If there is no legal move, the player has to pass.

The game ends if all squares are occupied or if none of the players can take a legal move on the remaining empty board squares. The winner of the game is the one that possesses more discs in such a situation.

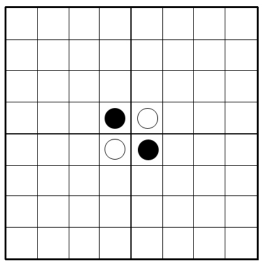


Figure 1 – Starting position of every game.

## General Strategies

There are three basic ideas to accomplish a winning game. But before throwing ourselves at them, let us just start with a surprising fact: having a lot of discs at a certain time does not guarantee to win the game eventually, even it is very close to the end of the game.

First, let us consider the importance of corners. A disc placed in a corner can never be re-flipped, because it faces two adjacent edges.

By gradually placing more discs around the corner, this area cannot be flipped by the opponent, thus creating the second general idea of stable discs. But taking the corners is not the only way to create stable discs. To avoid your opponent taking the corners, you should avoid playing the squares diagonally next to the corner squares and the squares next to corners (although the latter are less dangerous).

Third, there is the concept of mobility, i.e. the number of legal moves of a player in the game at a certain time. Obviously, it is a good idea to maintain high mobility whenever possible. This can be achieved by only having a few discs, that are packed and that are surrounded by the opponent at the same time.

On the other hand, a player should avoid creating empty squares that are not possible to enter anymore and building long thick "walls".

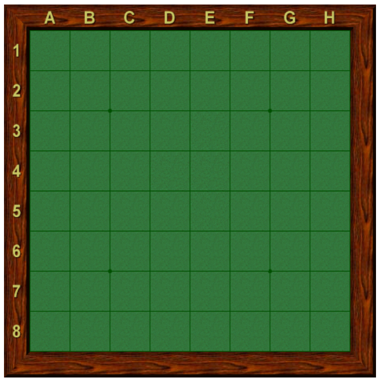


Figure 2 – The board

## The Othello Implementation

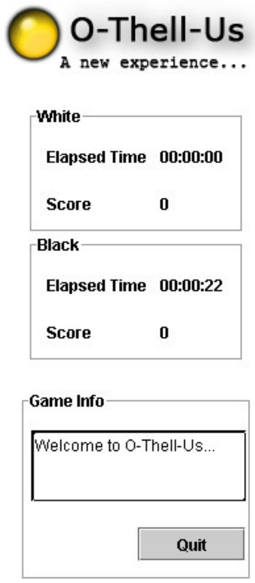
Our O-thell-us implementation is written in Java. Our aim was to implement a user-friendly, i.e. readable, code consisting of various classes. Their names refer to their purpose. Worth mentioning, we started to work from scratch on our program and built it all on that.

In the following, we will give the reader a short introduction and explanation to our user interface and the important parts of the source code. For more details that are not covered in this outline, please take a closer look in Appendix B.

### The User Interface

As mentioned above, our GUI was designed and created for simplicity and to be able to transmit vital information to the user in a pleasant and non-disruptive way.

The interface consists of two main parts: the board (represented by a self-designed picture, as viewed in Figure 2 and the user information and control panel (presented in Figure 3) The board (placed to the left in our GUI) is combined with an overlying mouseListener that catches clicks within the picture. Only legal moves are executed, and the turn switch back and forth after taken move, flipping the appropriate disc between moves.



00:00:00
0
00:00:22
0
Welcome to O-Thell-Us...

Figure 3 – The user information and control panel

For the user information and control panel, this is further subdivided into a statistics box for white, another one for black, a message printing window and a Quit-button.

The statistics boxes hold elapsed time for each user (using our very own StopWatch class) and each player's current score (how many discs he has on the board).

For the message box, this part is used to print information and status messages, and it can also be effectively used for debugging.

Of course there are third party applications such as JBuilder from Borland and Visual Café from Symantec, which provide GUI builders. They provide a palette of elements in a manner made popular by Visual Basic, and allow us to draw our widgets directly on the screen. We felt that it would be good experience to hand code the interface however, and writing it yourself allows greater control and flexibility.

## Classes and functions

*Othellus* This is our main class. It extends a JFrame (providing the container for the GUI) and also extends an ActionListener. It handles all initializations, painting, window handling, message printing and the actual game process. The latter is controlled by a method called *boardPictureMouseClicked(),* which is the starting routine for method *getPosition().* This method takes care of the current "click" and does all the work regarding flipping, clock-handling, score-counting and switching turns.

*Othellus* furthermore also calls *miniMaxAB()* for the computer player(s), which is the "engine" behind the AI. The method compares the available moves, and by using minimax search with alpha-beta pruning together with some nice heuristics, it will choose a move to make, which maximizes it current underlying utility function (more on this algorithm in the Adversarial Search Strategies section below). We also consider the fact when a player has to pass, i.e. the number of possible moves is zero and if the game has reached an end-state.

*Generator* The class Generator consists of both the utility function and the heuristics. This class is accessed from the outside by using the *getUtility()* method with parameters describing the current world (board) status; the active player, the underlying board matrix, the moves available currently, the position of a suggested move and the current turn number. These parameters are used in different ways in the used heuristic, to calculate a utility score to return to the minimax search.

Depending on the choice made by the user at initialization time, one heuristic (or, more often, a combination of several) is used to give an estimation of just how good a suggested move is. The heuristics are described at some length in the Adversarial Search Strategies section below.

*Matrix()* This class initializes the board matrix (placing the two black and the two white discs in the starting position) and controls its access and storage. It also includes a routine for counting and updating the white and black scores, by simply going through the underlying board matrix and counting the occupied squares. The *printMatrix()* method included here is used for debugging activities.

*Moves()* The *possibleMoves()* method is used to calculate and store all the possible moves that the current player can take, and also returning their number. It is used extensively, also by controlling the access to the matrix of possible moves. The calculating routine consists of 8 sub-checks for vertical, horizontal and diagonal testing, while imposing restrictions to edges. All sub-checks follow a certain (simple) algorithm:

1. Check if the current position in the board matrix is empty

2. If so, then check for adjacent fields that belong to the opponent

3. If there are some, then check the fields next to the opponent's square, on a line through both the opponent's one and the one being checked as a possible move.

4. If we hit a player's square on our way along the line, directly after an opponent's one (without passing an empty square), we are done. We can then return the current position as a possible move. Otherwise, the current square we are checking cannot be considered as a possible move. The figure 4 below shows a possible move in filled grey.

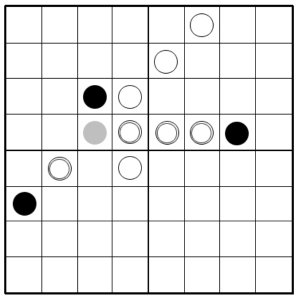


Figure 4 – Possible move

The other important method in this class is *flip().* Flip works similar to *possibleMoves(),* but instead of returning possible moves, it flips the discs between a taken move and the one(s) that are already on the board (according to the rules of the game). Therefore, all adjacent fields have to be checked. Since this is done quite similarly to the possible moves, we thought about combining these to methods into a general one. However, we dropped this for the sake of small time savings available in a less general method.

*Player* For managing attributes like player color and which player is active, we maintain a small class called Player. Grouping this information together enables much easier use, debugging and a more readable code.

*Position* This is just another grouping together-class, this time to keep track of the current x and y position as well as the current board value of a move under evaluation by the minimax algorithm.

*Stopwatch* and *TargetsTimer* These classes work closely together to do an exact measuring of the elapsed time for each player. The *TargetsTimer* will correct itself by using a method called *fireActionPerformed().* This method overrides the very inaccurate one in javax.swing.Timer. It does that by basically keeping track of the previous time and then determining how inaccurate the update was. *Stopwatch* uses an instance of the *TargetsTimer*, and adds accessor functions and formatting options on top of the accurate time-keeping.

*Tuple* This small class is just a grouping of two values, e.g. used for sending calculated scores together in a packet.

*Evaluator* For doing the important job of evaluating all of our different combinations of minimax and heuristics, we have this special class to do the trick. It evaluates the success of a move (and also a game and the average move) according to our own, modified version of four of the suggested parameters by William A. Greene in [Greene 91]. These parameters are used in the context of an Othello game using learning algorithms, but works after our changes equally well on evaluating O-thell-us also. More to come on this special chosen type of evaluation, and how we implemented it, in the Evaluation section.

## Adversarial Search Strategies

To decide on the right move can be quite exhaustive and/or misleading if the wrong search strategy is used. This is partly due to the search depth used, but also to other circumstances, e.g. the horizon effect.

To make the computer a challenging opponent, while still upholding an even pace in the game, we need to consider the above mentioned parameters and also include some interesting heuristics.

First, however, let us get a brief reminder of what game theory and adversial search strategies actually is.

### Game Theory and Adversarial Search

This project is concerned with a specific area of AI, namely that of game theory. Game theory is one of the most useful branches of modern mathematics. It was actually anticipated by French mathematician Emile Boel in the early 1920's, but it was John von Neumann who published his proof of the for us highly useful minimax theorem in 1926. It was further developed in the 1940's by von Neumann, with the help of Oskar Morgenstern, in his work *Theory of Games and Economic Behaviour*. [Walker 95]

In this context, the word 'game' does not simply apply to board or video games. A game is "any set of interactions governed by a set of rules specifying the moves that each participant may make and a set of outcomes for each possible set of moves." [Bullock 99]

In this way it can apply to many areas and is used in such diverse fields as economics, political science, marketing and even warfare. Anything that involves conflict of some kind is a possible area for game theory.

The idea is to make intelligent moves based on the current game state, a set of predefined rules, potential moves the opposition will make and the game objectives.

Here, game theory will be applied to a set of rules we defined earlier - that of the board game Othello.

Adversarial search is used in problems such as games, where one player's attempts to maximize their fitness (win) is opposed by another player.

The search tree used in adversarial games such as Othello consist of alternating levels where the moving (MAX) player tries to maximize fitness and then the opposing (MIN) player tries to minimize it. To find the best move the system first generates all possible legal moves, and applies them to the current board. Depending on how many levels of lookahead an adversarial search uses, the amount of legal moves (nodes) can be huge.

### The Othellus Minimax search with αβ-pruning

Now let us turn our attention for a while to the algorithm that does all the actual "dirty work" of searching through this often very large search space, and how we implemented it in our game. To first get a feel what the algorithm does, let us have a look on the pseudo code:

```
Alpha-Beta-Search(state) {
  if (depth = 0) {return board's estimated
score;}
  successors = valid moves from state;
  if (successors is empty) {return board's
estimated score;}
  if (is a Minimising node) {
    for each successor in successors {
      set Beta to min( Beta,
              miniMax(successor, Depth - 1,
           Alpha, Beta));
        if (Alpha >= Beta) {return Alpha;}
    }
    return Beta;
  } else {
    for each successor in successors {
      set Alpha to max( Alpha,
              miniMax(successor, Depth -1, Alpha,
           Beta ));
        if (Alpha >= Beta) {return Beta;}
  }
  return Alpha;
  }
}
```

Figure 5 – Our alpha-beta search pseudo code

The minimax algorithm with alpha-beta pruning is, as can clearly be seen, a fairly simple recursive algorithm. Yet it is very powerful. By reducing the search space by pruning away the nodes that cannot possibly be chosen by either MIN or MAX, we are able to easily treat games in Othello with a lookahead of as much as 9 plys. After this it still works, but slows down to in some cases as much as 2 minutes of evaluation time for the computer. Given that the branching factor of Othello is between 5 and 15, a 10 ply lookahead gives us a search space (before pruning) of between about $10^7$ and $10^{12}$ nodes. For our search algorithm with pruning and heuristics to be able to handle this as well as it does, is well over our expectations at the beginning of this project.

## The Othellus Heuristics

Finally we arrive to the part we all have been waiting for: the heuristics. What is then a heuristic and how do we use it? A heuristic involves or serves "*as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods*"[1]. In this section, we focus on the (in our game) implemented heuristics. Intuitively, it is likely that not all features have a consistent importance throughout the game. For example, mobility is very important in the middle of the game while it is less significantly at the beginning and the end. But more on this and other small details later.

We have implemented and tested a fairly big number of different heuristics. Most of them are complements to each other and to our own two base ones, which uses position and mobility together with a (in the basic version) static board weight matrix to calculate the current utility.

*Random move* Before turning our attention to the more advanced heuristics, we would like to start with a very simple method that almost can't even be considered as a real heuristic – taking a *random* move without doing any evaluation. The move is in the most basic version chosen among all possible legal moves available to a player; each one is equally weighted with the same probabilistic value, i.e. every move is a likely to be taken as the next. This can be considered as the play of a very (!) inexperienced human player or as an AI without an implemented intelligence, and our testing of this purely random player gave us an terribly bad computer opponent.

Why do we then even spend time thinking about this seemingly useless heuristic? Well, a random element is useful in as we found at least one situation: to avoid having the same games played over and over again. Playing as human vs. computer (or for testing purposes, computer vs. computer) often leads to an repetitive game with a clear pattern, or a game that ends in a sort of playing deadlock (both computer players doing the same series of moves over and over again). To be able to test our other heuristics in many different situations, a random element therefore needs to be introduced in the test games, and therefore we chose to play human vs. computer for testing, while measuring the heuristics proficiency with an independent statistics evaluator (described in the Evaluation section below).

*Board Weights* Next step towards building a more complex heuristic, is our simple strategy idea of assigning weights to the board squares according to their position and choose on each turn to play on a square of best weight.

The basic underlying idea is a board matrix containing the weights. But how does one come up with a correctly balanced set of weights for the board weight matrix? Well, combining together some basic game playing strategies we could come up with a good starting position for the weights and then adjusting them as we went along with playing. Ideally, to come up with a near-optimal distribution of board weights, one would like to train the matrix successively by using machine learning. In this case, there just was not time enough for that.

As soon realized by any player (even at the lowest level), having the corners is always good. These discs cannot be re-flipped, as argued earlier. By the same argument, the fields directly adjacent to the corners are usually bad to be the first to take, since this gives your opponent a good chance of taking the corner, while shrinking your options of taking it yourself. Furthermore, the squares even one step further out from the corner are good to take, to gain a good striking position for the corner. These ideas are summarized in Figure 6.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| + | − | + | | | + | − | + |
| − | − | | | | | − | − |
| + | | + | | | + | | + |
| | | | | | | | |
| | | | | | | | |
| + | | + | | | + | | + |
| − | − | | | | | − | − |
| + | − | + | | | + | − | + |

Figure 6 – The board estimation

As can be seen from the above figure, this reasoning still leaves us with 28 fields without any relative value, and the classified ones are only just that; relative to each other without any more precise measure.

So, how to proceed? As the next general advantage rule, we turn our attention to the edges. After the corners (which has protection from flipping in all directions), the edges are second in value to take, since the can only be flipped from two separate directions. Also, they provide a great basis for future moves of flipping discs towards the interior of the board. Therefore, they should be given a higher score than the interior of the field (the center 16 squares). For fields affected both by the corner rule and the edge rule (lying on the edge between those two areas), the scores are higher than for fields just belonging to one rule.

Finally, the interior of the board can be seen as a miniature board of itself. The argument supporting this declaration is given to the reader in the following section, on our Several Stages principle. Of course, being nowhere close to the value of having the "real" corners and edges, this interior "board" has lower scores relative to the more permanently advantageous positions of the field.

Our final board weights are presented in Figure 7. The exact values for each square are a combination of the above reasoning, and evaluation of the extensive testing we have done.

[1] http://www.m-w.com/cgi-bin/dictionary?book=Dictionary&va=heuristic

```
private int boardWeight[][] = {
    { 100, -10, 11,  6,  6, 11, -10, 100 },
    { -10, -20,  1,  2,  2,  1, -20, -10 },
    {  10,   1,  5,  4,  4,  5,   1,  10 },
    {   6,   2,  4,  2,  2,  4,   2,   6 },
    {   6,   2,  4,  2,  2,  4,   2,   6 },
    {  10,   1,  5,  4,  4,  5,   1,  10 },
    { -10, -20,  1,  2,  2,  1, -20, -10 },
    { 100, -10, 11,  6,  6, 11, -10, 100 }
};
```

Figure 7 – Board weights matrix

*Several Stages* The next idea divides the whole game into several stages; three to be more precise. The parts identified on the subject are the beginning, the middle and the end phase. The three phases are characterized by interior stability, mobility and greediness, respectively.

This is done by various mobility weights that simulate a Gaussian distribution. (Figure 8 – simulation of the Gaussian distribution). The advantage of taking mobility into account is to force the opponent to very few possible moves, thus enforcing bad choices on the opponent, while keeping as many possible next moves for oneself. This is done by giving an extra score for every square controlled by our agent and a negative score for every square controlled by the opponent. In the first phase, mobility can be disregarded because the major aim is to gain control of the underlying 4 x 4 sub-board in the middle. This because it gives you a strong base for later expansion, and the beginning phase is identified empirically by us to last for about the first 12 moves. Concluding the discussion on the first phase, this tells us that only small values or even negative values are taken into account as a mobility weight as a beginning.

The second phase considers the whole board, implying high mobility weights so that the player can spread as far as possible over the board, thus gaining as many corners and favorable edge squares as possible. Mobility score for this part is therefore relatively high.

The third and last phase aims for the goal of gaining as many squares as possible thus simulating a greedy behavior (or cruelly, going in for the kill). This is implemented by decreased mobility weights, giving more power to the constantly underlying position gaining heuristic. This whole procedure is implemented in a heuristic called *mobPos()* that is an extension of *position()*.

*position()* uses the board weight to calculate the gain of a possible move compared to another. The sum of the weights is used as the utility value for a possible move, as requested by the minimax algorithm.

*mostFlipped* This heuristic aims to take as many pieces of the opponent as possible and is therefore purely greedy for all plys of lookahead. The standalone version does not take any other values like the current game situation or the importance of various squares into account (cp. the board matrix). But on the other hand, it can be easily combined with functions like *positions()* or *mobPos()* forcing them to be more greedy in the second phase without changing any

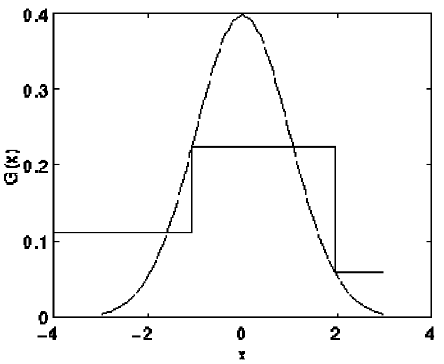


Figure 8 - simulation of the Gaussian distribution

parameter settings. This method works similar to the *flip()* function, but of course instead of flipping the values, it returns an utility value as measured in the number of flipped pieces.

*lowSurroundings* This method designed by us returns the square with the lowest sum of board weights of surrounding squares as the optimal move. By using this strategy, the agent reduces the opponent's chance to gain a very good square because it only chooses squares that are surrounded by less attractive ones. Since the rule for the opponent is that it must make a move on a square adjacent to one of the other player's discs, this strategy works extraordinarily well. Success is especially apparent when we are in phases of the game which only has few possible moves (like in the beginning or the end).

*Corners* As already mentioned before, corners are very important in the game because they face two edges and therefore prohibiting a changing occupancy. We classify such a disc as stable (unflippable, if you want). Expanding this area leads to more stable discs and therefore to a favorable situation in the current game. This process is simulated in the heuristic *enlargeCornerArea()* that modifies the current board weight matrix dynamically, before one of the two heuristics *positions()* or *mobPos()* is called. Changes are only applied to edges and with a constant, non-optimized changing value, due to a lack of time for this project. Further explored, this heuristic allows a more dynamic and updated view of the value of taking different board fields that from the beginning was classified as bad. As an example of this, consider the squares adjacent to a corner. Before you own the corner, these are really bad to take, but following the discussion in the *lowSurroundings* heuristic, growing the corner area can be a good strategy.

*Edges* Another of our own heuristics is called *checkNeighbours.* It basically tries to increase the number of the agent's pieces on an edge, thus running danger of loosing all of them, but on the other hand increasing the opportunity to catch more discs towards the interior of the board. It can be considered as a preceding method or complement to *positions()* or *mobPos();* increasing their playing ability.

*Opening strategy* While playing the game, we figured out that it is always desirable to play a perpendicular or a diagonal opening. This simple heuristic reduces the number of pieces that could me flipped in the next move by the opponent, i.e. from at most two to at most one. (Figure 9 – The opening, where the upper discs are the new possible ones)

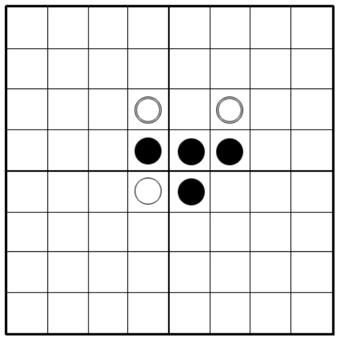


Figure 9 – The opening

## Evaluation

Since we were testing the game by playing it human vs. computer (we had human vs. human enabled at an early stage, and also computer vs. computer for testing), we needed a independent way of measuring the performance of the different combinations or standalone heuristics applied to our minimax search with αβ-pruning.

For this purpose, the paper *Machine Learning of Othello Heuristics* by William A. Greene (1991), presents some interesting parameters for measure (even though we in fact do not use machine learning).

The theory behind Green's suggested evaluation is the use of six different parameters; (1) corner strength and potential, (2) corner stability, (3) edge stability, (4) interior stability, (5) mobility, and (6) square advantage. Let us have a short overview of what these parameters mean, before we go into why they are important to us, and how we implemented them.

The corner strength and potential is intended to be helpful for steering play early in a game, when the corner regions are thinly filled. The heuristic gives an assessment of the degree to which a corner region is already advantageous, offers positive opportunities or poses risks to the player at hand. This is implemented in our program by checking if a corner was lost or won during the evaluated move, and if the player or his opponent holds new, potentially advantageous squares for taking the corners.

The idea for the corner strength is basically to give a measurement on how big a corner-based right triangle is owned by the given player. We chose not to implement this measuring parameter, since it is a bit complex, requires much calculation and therefore slows down the fluency of moves generated and evaluated by the computer. The same goes for interior stability, where the idea is to calculate how many stable (unlikely to be taken by the opponent), unstable (likely to be taken by the opponent) and semi-stable (too close to call) squares there are in the centre of the board.

For edge stability, we evaluate the same parameters as with the interior stability; stable, unstable and semi-stable squares. Implementation of this is done by successively checking all four edges for empty squares. If we find one of those, we check all the adjacent board squares to see if they are dominated by our own discs, the opponent's discs or if it is too close to call. Using this measure, we keep a score of how stable the edges are, by punishing the player's utility for empty squares surrounded by his own discs and vice versa. This process is easily extended to also include interior stability.

The last two parameters, mobility and square advantage, are the easiest ones to implement. The current mobility is just a measure of how many opportunities a player has to place a disc right now, and the future mobility says the same about currently unplayable ones (but at least adjacent to one of the opponent's discs, and therefore potentially good later on). This is calculated by weighting the mobility calculated for our *mobPos* heuristic by a factor of 2 (suggested by Graves), and simply adding to this the number of empty squares that borders to an opponent's square.

Finally, the square advantage is simply calculated as the difference between the number of gained squares for the player and the number of gained squares by the opponent.

The four parameters we chose to implement of these six are implemented in our *Evaluator* class, together with a simple printout method for statistics such as the four separate scores, the total score and the average score per made move.

## Discussion

Certainly, a lot of improvements can be implemented due to the pure facts that, we are not using any knowledge base or endgame databases and no machine learning, to further improve performance. These facts taken together with the fact that we lacked the sufficient time to optimize our variables for heuristics, such as *checkNeighbours,* leaves us with a wish to continue this development process even further.

One idea to alter our interface is to highlight all possible moves a player or an agent can take (as some kind of hinting system). But this increases the complexity of the board so that it is difficult for the player to negotiate its way, although it is easily implemented as a fourth graphical square option. As we implemented a classical board game without this feature, it is also largely irrelevant for us.

Another nice feature would be to add an undo option for the last move. But this can also be seen as a gadget and is therefore omitted.

In the beginning of this project, we aimed in implementing more advanced user options, such as being able to select a 12 times 12 or a 16 times 16 board, instead of the conventional 8 times 8. The code for this is still largely present in our source code, and we also designed the two bigger versions of our board picture (see Appendix B for screenshots). The reason for leaving this option out is mostly practical ones; all the different paddings for the different board sizes made our code hard to get an overview of.

We could also have included the option in the interface to choose between both human vs. human and computer vs. computer (which were available at a testing phase of our interface and heuristics), but since we did not want the interface to be too complex, we left these out as an option for the ordinary user.

As a last point, we want to mention a few facts that would improve our AI. First, adding an opening and an endgame book that are continuously updated. Furthermore, instead of having a fixed deepening factor, one could replace it by an automatic iterative deepening that adapts to game situations and progress. Another idea might be the implementation of responses to certain moves done by the opponent.

## Test results and conclusion

After evaluating our results from the evaluation class, we now want to present the major results including the major drawbacks.

First, our program's playing level corresponds to the one of an average human player. This can be revealed by the fact that the human player wins 3 out of 5 games with an average winning margin of 22 points while the agent wins by averagely 18 points. These results take all lookaheads and all heuristics into account. But as soon as one only considers the individual lookaheads more differences become obvious. An increasing lookahead implies an increasing computing time but also a better play - low lookaheads like 3 or 4 usually lead to a loss, but as soon as they take values of more or equal to 6 the agent increases its chances to win enormously.

Second, there is a strong connection between a high mobility, square advantage, high heuristic and a high average scoring move leading to a win with a fairly high winning margin. This should be totally clear because they simulate the ideal play.

Third, a positive number in corner stability and potential usually leads to a win by the agent as a cause of a stable disc.

Fourth, our fastest and best method is low surroundings yielding the highest winning margins for the agent.

On the other hand, very low corner stability often leads to a low average score per move and to a loss because it is more difficult to build stable discs. But this cannot be generalized because it can happen that corners are gained in the end game which is connected to a low score. The same is true for edge stability.

A problem we are still facing is that our agent sometimes takes good moves while sometimes not recognizing them at all. This can be traced back to the fact of required adjustment of values that can be set like the ones for *mobility*, *mostflipped*, *enlargeCornerArea* and *checkNeighbours*. It is extremely difficult to adjust values simulating the Gaussian distribution that is used in the mobPos method simulating the importance of various game stages.

## References

**Wolf, T.** (published Aug '02). *The Anatomy of a Game Program.* Retrieved 11/01/04, from:
http://home.tiscalinet.ch/t_wolf/tw/misc/reversi/html/index.html

**java.sun.com** (published Mar '04). *The Java Tutorial.* Retrieved 10/25/04, from:
http://java.sun.com/docs/books/tutorial/index.html

**Walker, P.** (published Apr '95). *History of Game Theory.* Retrieved 11/01/04, from:
http://william-king.www.drexel.edu/top/class/histf.html

**Greene, W. A.** (1991) Parameters for machine learning in Othello. *Machine Learning of Othello Heuristics.*

**Abramson, B.** (1989) Computer games and control strategies. *Control Strategies for Two-Player Games.*

**Epstein, S.** (1998) Heuristics. *Learning Game-Specific Spatially-Oriented Heuristics*
.
**Bullock, A. & Trombley S.** (1999) Philosophical definitions, amongst other. *The New Fontana Dictionary of Modern Thought* 3rd Edition, Fontana

**Russell, S. & Norvig, P.** (2003) Course book in CSE 573.. *Artificial Intelligence: A Modern Approach* 2nd Edition, Prentice Hall

## Appendix A – Who did what?

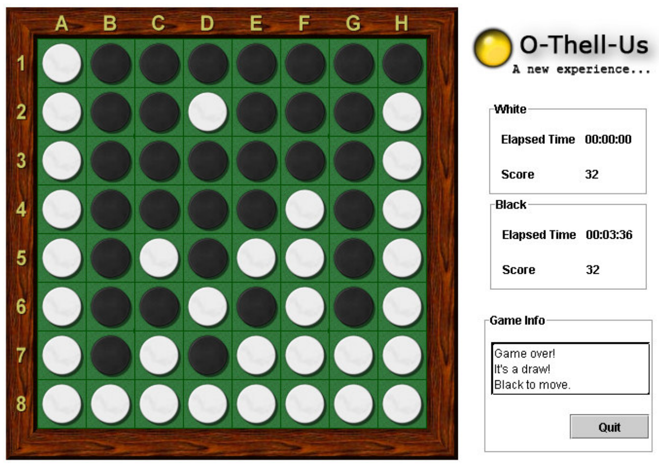
Basically, the whole project was done by the both of us in equally parts thus spending many days and nights together. But one can clearly state that the main class Othellus() with the alpha-beta, minimax, evaluation and fine tuning of the graphical interface was basically implemented by Jonas.

The Generator() class and the Moves() class was basically implemented by Mathias, although the ideas came for different heuristics from different persons in the group.

Worth mentioning, debugging and implementing discussions were usually done by the two group members together.

## Appendix B – Screenshots

A draw game.

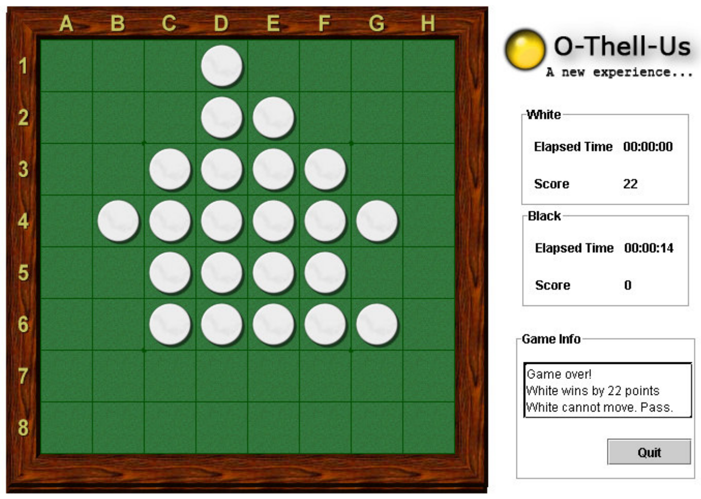


Elapsed Time 00:00:00
Score 32

Elapsed Time 00:03:36
Score 32

Game over!
It's a draw!
Black to move.

A big loss for black.



Elapsed Time 00:00:00
Score 22

Elapsed Time 00:00:14
Score 0

Game over!
White wins by 22 points
White cannot move. Pass.