# Programming Assignment 2 – The Othello Champion FIREBRAND

Magnus Almgren, Jenny Berglund, Ido Milstein

November 8, 2000

**Abstract**

The following report describes the process of building a good Othello player, using well-known results from the field of artificial intelligence. We describe the features we use to evaluate the board, and also other ways we try to speed up our program such as best-first search to increase the gain from $\alpha\beta$-pruning and more. The final game, FIREBRAND, beats the authors even though it might need some more features to beat a world-class player.

# 1 Documentation of Code

Most of the code for the Othello program was given to us. We added routines for $\alpha\beta$-search, and several feature functions to evaluate the board. We also designed data structures to perform fast node ordering to increase the gain from the pruning (see Section 2).

The code is well-documented, and almost legible in itself. We have divided the structure of the program in the following pieces (not all of which are used in the final version for FIREBRAND):

| | |
|---|---|
| `boardelement` | defines a class for keeping track of new board positions and what value they have if we apply the evaluation function on the board |
| `heap` | implements a heap |
| `priorityQueue` | implements a priority queue through a linked list. |
| `board` | was given to us and implements helper functions concerning the board. |
| `maxmin` | implements two simple functions to find max or min of two numbers. |
| `*.tcl` | supports the GUI for the game, and was provided for us. |
| `evaluate` | contains functions evaluating the board positions. |
| `search` | contains functions dealing with finding the best successor state. |
| `server` | runs the server with the game functionality. |
| `client` | is one of the clients to play in a game. |
| `server4train` | is a modified version of the server, so that we can train FIREBRAND in the best possible fashion. |
| `client4train` | is a modified version to train FIREBRAND on several games and boards in an automatic fashion. |
| `server4train8.2` | is a variant on `server4train`, where the clients always play the same color. |
| `client4train8.2` | is a variant on `client4train`, where the clients always play the same color. |

# 2 Optimizations

The question of optimization covers both the training phase and the actual game-playing. Both will affect the final performance of FIREBRAND.

## 2.1 Training Stage

There are different strategies for training the weights for the game. One is simply to start with a clean board and then play the game to the end. Another is to first train FIREBRAND on final stages of the game and then slowly increase the number of moves necessary to complete the game ([1]). It is also possible to mix the ordering of the games, and play one almost finished game followed by one that is in its middle stages followed by a blank board. The different strategies are shown in Table 1.

By first training the game on the final stages, we gain three advantages: the playing goes fast (few moves to make), the feedback is almost immediate and thus we can calculate a "true" value of the the evaluation function. When the weights are stabilized for this stage, we can then increase the number of necessary moves and train the program again.

This should be compared with starting with a blank board all the time. In the beginning, the weights for the feature functions will be random and it is impossible for the program to

| Order of Games | | |
|---|---|---|
| begin, begin, begin | middle, middle, middle | end, end, end |
| begin, middle, end | begin, middle, end | begin, middle, end |
| random, random, random | random, random, random | random, random, random |

Table 1: *By changing the order we presented the games to the training algorithm, the final resulting game changed.*

evaluate if the moves are good or bad. Even though we do not have an explicit time limit for training our game, we are still subject to the final deadline of the project. For that reason we used the first method of training the game incrementally. This way, we were hoping to reach convergence quicker.

We created ten boards each, with 10, 20, 40, 60, 80 blanks for the game to play.

## 2.2 Playing Stage

During the actual game competition, the player is restricted by an explicit time limit. It is important that a good move is found as quickly as possible. The program needs to run fast, and avoid unnecessary calculations. Note that this is closely linked to the time management policy described in Section 3, but still a separate issue. The latter makes sure FIREBRAND has enough time to make moves all through the game and thus avoid a technical loss while the former makes sure the search for the best move is fast and efficient.

One of the best ways to increase the efficiency of the searching is to implement some sort of $\alpha\beta$-pruning. There are variants on this message, but the variants have not been proven to be much more efficient than the original version. We used the algorithm described in [3].

The pruning can increase the performance with magnitudes, but the result may also be exactly the same as without the pruning. The ordering of the nodes to expand is very important for the gain of performance. We tried to use a *best-first search*, i.e. we tried to expand the node that would give us the best result. In practice, there is no way we can know which node to use so we tried a few different strategies.

### 2.2.1 Node Ordering through a Priority Queue

It is likely that the successor deemed the best from our evaluation function will actually be the best node. That is the reason we have trained the evaluation function. For this reason, we decided to go through all the possible successor states, and calculate the value of these boards. By sorting this list, and then expanding the nodes in decreasing order we hoped that the actual best node would be among the first few expanded. This method was applied recursivley through every level of nodes of the search tree.

As we had hoped, the method could cut the number of nodes expanded by half. It worked very well when choosing the best node to expand to gain as much as possible from the $\alpha\beta$-pruning. However, the disadvantage we found was that the actual bookkeeping and sorting took longer than it would have taken to visit the nodes avoided by this method.

The first version used a simple priority queue, built from a linked list. We improved the code using a heap and an array, and also by trying the STL library for priority queues. The price of using this method was still higher than the actual gain we made by not visiting all the nodes.

As a final try we allocated all heaps in the beginning of the program and then reused them during the play. This way we avoided the time penalty of allocating memory for each node search. It was still too expensive.

### 2.2.2 Avoiding Search through a Hash Table

Another approach that we tried is the one described in the article about the Othello program BILL [2]. Here, a hash table is used to store for each encountered position the move that is currently believed to be the best one. If the same position is encountered again, that move is expanded first, which hopefully leads to more nodes being pruned than if no node ordering is used. We implemented this strategy in the function `ABHashSearch`, but again it proved to be slower than just using an $\alpha\beta$-search without any clever node ordering.

### 2.2.3 Node Ordering Conclusions

Through the experiments, we saw the gain to be made from node ordering. However, the price we paid for the book-keeping of the new data structures was always more expensive than the time gained from not visiting all the nodes.

We tried to profile our code, but for some unknown reason the profiler did not accept the code. We were thus unable to find out exactly where the program spent the time. We have looked into the problem of solving this problem, but by the time of the deadline of this project we had no success.

Our final program uses a simple recursive search algorithm with $\alpha\beta$-pruning but no node ordering.

## 2.3 Avoiding Search by Using Opening Tables

As described in [2], we can avoid some expensive searches by storing previously found values. We were allotted to use 50 MB memory for the tournament, but only 10 MB of hard disk space. This means we cannot store a lot of opening moves (which are the most difficult to calculate). We decided not to use this option, in view of the limited amount of opening moves. The objective is also to experiment with learning and not use predefined tables.

# 3 Time Management

The program runs under the constraints of time, so one of the key components of a successful player is good time management. It is very important to have enough time towards the end of the game to make good moves. However, the earlier stages still play a major role because mistakes during these phases impact the final moves.

Our program uses the following strategy for time management: First, it allocates a certain amount of time for the move. This is done so that moves earlier in the game get less time than moves towards the end. It then performs a search to depth 3, which it times. The quotient between the time for the depth 3 search and the allocated time is then used to figure out the appropriate depth of the search. We also use an alarm that goes off when there is only five seconds left. In that case, the search is immediately interrupted and the result from the depth 3 search is used. If the alarm goes off during the depth 3 search or if the time left is less than one second, a search to depth 1 is tried instead.

# 4 Features for the Game

## 4.1 Evaluation Function

### 4.1.1 Normal Features

Picking the "right" features is many times where AI comes into play. As none of us had prior knowledge of Othello playing, we used some common heuristics, together with some features we read about, and finally added some innovative features of our own. The list of features in FIREBRAND is:

1. Counting corners held
2. Counting pieces 1 step from corner
3. Counting pieces on the edges
4. Counting pieces on concentric inner squares (we had 1 feature for each square – total of 4 features
5. Counting pieces held by player
6. Counting how many moves are possible

7a Counting how many pieces are "locked" within opponents pieces (like in White-Red-White)

7b Counting pieces on the edges that had 2 empty spaces on both sides.

8 Counting corners and parts of edges that are adjacent to a corner and that have the same color as the that corner (The corners themselves counted as 4 squares here).

All the features are symmetric making this a zero sum game. The symmetry was obtained by actually using the difference between the values each player had for the features. For example, feature 1 is actually the *difference* in number of corners that each player has.

After a short period of training it was clear that the `7a` feature was unstable at final moves, causing all the weights to grow to infinity. We therefore restricted its use to up to the 75:th move. After that point it was no longer used. However, even though this solved the problem of the weights diverging this weight quickly converged to 0. Because it did not seem to matter at the earlier stages of the game, and was unusable for the later stages we replaced it with feature `7b`. As further tests showed, also this feature was not very good.

We observed 2 interesting phenomena. First, in almost all the runs we made, feature 8 got a very high value compared to the rest (about 0.7 to 1.0 while the others got $-0.2$ to 0.2). Its dominance erased feature 1 completely, because feature 8 has feature 1 as a component.

Second, we noticed that the weights that were attached to the features converged to different values when trained on different stages of the game. This is further described in Section 5.2.

### 4.1.2 Time Weights

In order to reflect the different stages of the game, we added a new set of weights (see Section 5.2). This involved a change to the evaluation function from

$$\sum \omega_i * F_i$$

to

$$\sum \omega_i * F_i * (2 * Time + \Omega_i * (1 - 2 * Time)), \quad Time = (\#pieces - 4)/96$$

This is a linear function of time, that equals $\Omega_i$ when the game starts, and equals 1 at the middle of the game.

By changing the learning rules to:

$$\omega_i \leftarrow \alpha_1 * F_i * (2 * Time + \Omega_i * (1 - 2 * Time)) * Err$$

$$\Omega_i \leftarrow \alpha_2 * \omega_i * F_i * (1 - 2 * Time) * Err$$

the program could learn these new *Time Weights* through reinforcement learning.

Notice that giving a *Time Weight* a value of 1 actually means that the value of the feature does not change with time. Our training scheme was therefore:

1. train normally until the weights seem to stabilize
2. add the *Time Weights* all with value 1
3. go on training

From some of our observations (described in Section 5.2) we expected the *Time Weight* of the 8th feature to grow, and so make this feature important at the beginning of the game, and not much at the end of it. This is also the case, as seen in Figure 1. As a nice additional result some of the other values we trained got weights further from zero, and appropriate *Time Weights* to attach these new values to either the beginning or the end of the game (also seen in the same picture).

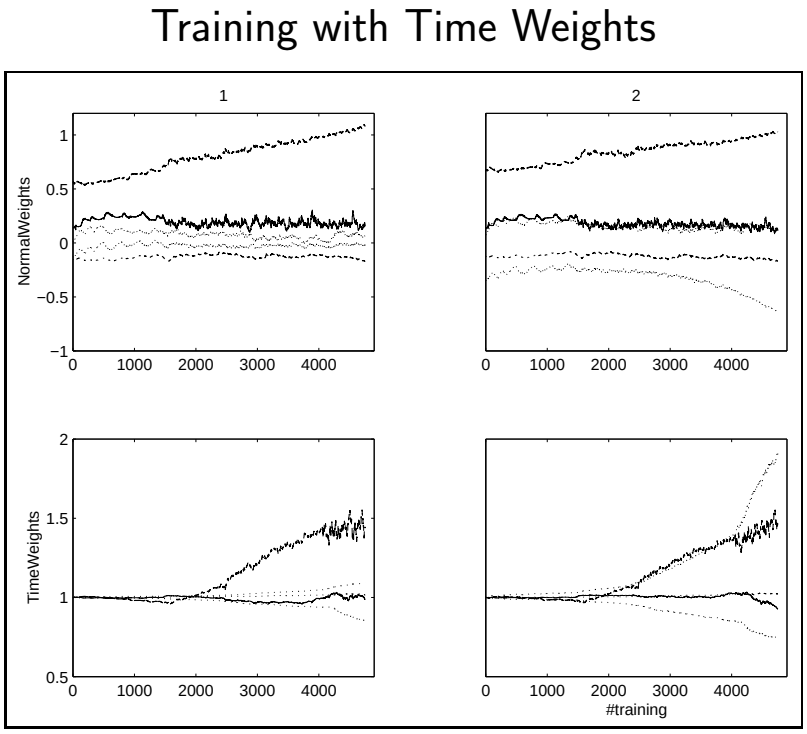


Figure 1: *By using* time weights *to represent what stage we are in the game, we get more stable weights, and the right feature is used at the right time. The growing feature is feature 8 (Section 4)*

By using polynomial functions instead of linear functions for the *Time Weight* functions, it is possible to have features whose importance will change at different stage of the game (in a

non-linear fashion). This might lead to two drawbacks: First, more weights is needed. Second, the training rules become more complicated and more costly to compute.

### 4.2 Reinforcement Learning

We implemented method 2 described in the assignment handout([1]). Thus, we use the actual value of the new board and not the conjectured value where we assume we play against an opponent who reason like us.

The bulk of the training was done against different versions of FIREBRAND, and then we tried it against the best programs of last year's tournament.

Each version of FIREBRAND started with random weights, and then competed against itself. During the course of the training we had functionality to explore the space a little and not always choose the most optimal move.

The learning will be turned off during the tournament for a variety of reasons. We will never meet the same opponent twice, and a few games will not help to improve our weights much compared to the thousands of iterations we have done previously. We also want to avoid writing new information to the weight files during a critical tournament (Murphy's law). The final reason is that we hope to save some time by excluding the training steps.

## 5 Experiments

We ran several experiments to test how different factors influenced the final result, i.e. how good the final game was. We tested the following:

- learning rate
- stages of the game
- different depth of search
- evaluation features of the board
- train a red and a white player separately
- does weights converge to same values
- analyzing strategy and manually adjusting weights

Some of the graphs shown in subsequent figures have different number of training iterations. There was no time to complete the same number of iterations due to the complexity of the search (very deep search etc). The trend can still be seen in the figures.

### 5.1 Different Learning Rates

*How will the magnitude of the learning rate affect the convergence of the weights?*

We wanted to see how the learning rate influenced the training process. A large learning rate should lead to faster convergence but if the rate is too large, the weights will never converge.

We ran the learning process for the six different values of $\alpha = 0.001, 0.01, 0.05, 0.08, 0.1, 0.5, 1$

In Figure 2 we see the result. With a small learning rate the weights never fluctuate but many iterations are required before they converge. Considering that each game is quite expensive to run, in terms of CPU time this is not optimal. A somewhat larger learning rate

will converge faster, but towards the end the weights will start to fluctuate. The optimal strategy seems to have a large learning rate that will slowly decrease as a function of the number of training examples. This way we can combine the faster convergence and avoid the fluctuations in the final stages[1].

As can be seen in Figure 2, the weights fluctuate all the time if the learning rate is too large.

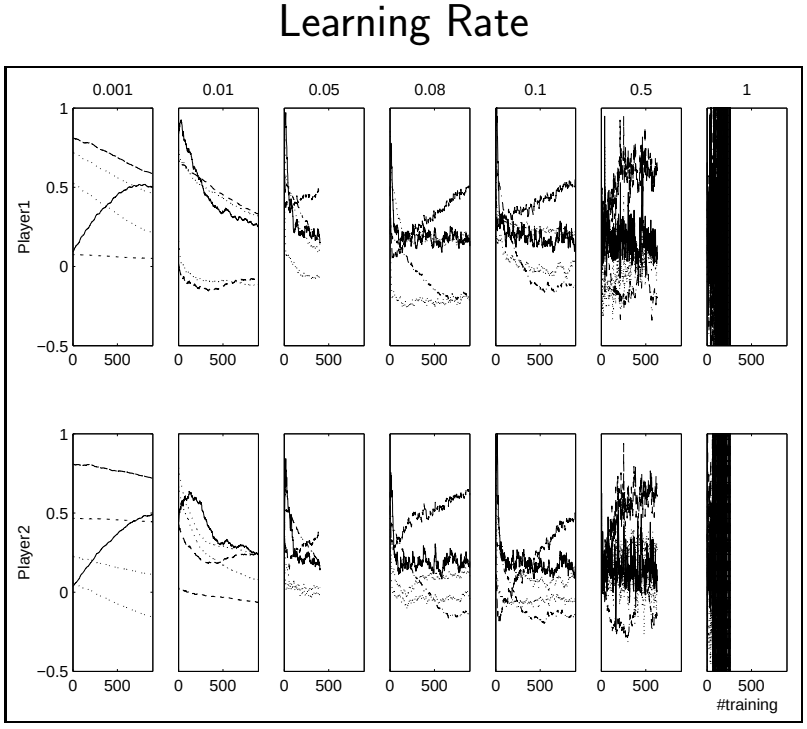


Figure 2: *Different learning rates affect the convergence. As can be seen, a small learning rate requires many iterations while a larger one reaches converges faster. The larger rate will then start to fluctuate.*

## 5.2 Stages of the Game

*Are there different stages in the game of Othello, as in other board games and how will that affect our training?*

Most complex board games have different stages where the advanced player should use different strategies. The same could go for Othello, and this would in turn mean that the weights might converge to different values at different stages in the game.

Figure 3 shows the weights when training on different stages of the game. The first 250 iterations were made with boards having only ten blank squares. The next 250 iterations were made with 20 blank squares. As can be seen, the weights seem to start to converge to a value at each stage, but when we switch state the weights changes too. The features seem to play different roles in the different stages of the program.

[1]This combination was never tried in practice for FIREBRAND.

To elucidate the point further, Figure 4[2] shows the development of the weights for one of the features of FIREBRAND. As can be seen, it has little importance in the end stages of the game (beginning of the graph) but in the initial stages it is very important (end of graph).

The stage of the game should somehow be reflected in the game. There are several ways to do this:

- Change to different stages based on the number of empty squares left. This option seems too simple, and it does not reflect the true stage of the game. If the corners are still free but more than 25% of the game has been played, some of the beginning strategy might still be valid.
- Change to different stages based on features of the board, such as all corners taken etc. This is much more complex than the first option, but it has two main disadvantages: it is discrete and a lot of expertise is needed to set up the features when to jump.
- Use reinforcement learning to automatically learn about the stages. Why not use reinforcement learning to find out about the stages, as we use it to find out how to play the whole game of Othello.

We decided to use the last option, and for that reason we added the additional *Time Weights* described in Section 4.

To contrast training with and without *Time Weights* (see Section 4.1.2 we first trained the program without these weights until it reached an initial convergence. At that point we let one program train with the *Time Weights*, and one without. On the left in Figure 5 (column 1), we see the normal weights trained *with no Time Weights*. In the middle (column 2) we see the normal weights trained *with Time Weights*. On the right (column 3) we see the values of the *Time Weights*. We can see that while not much training goes on in either the graphs in column 1 or column 2, training commences on the time weights.

## 5.3 Search Depth and Time Constraints

*How does the search depth affect the convergence of the weights?*

There is no question that the deeper a search the better program. By being able to look further ahead in the game the program can make better educated guesses on the best current move. However, it is not clear how the search depth affects the convergence of the weights. In Figure 6 we have plotted the weights for two different programs using the search depth of 2, 4, and 8.

As can be seen, the normal weights on the feature develop similarly independent of the search depth. The weights regulating what stage the program is in (see Section 5.2) are more sensitive to the depth. By having a deeper search, the effects of the earlier boards will probably not matter as much. As an extreme example, consider a board with 80 blank squares and a search depth of 10 compared to a board with 90 blank squares and a single search depth. These games will be more or less comparable, because the same final value will be obtained in both cases.

The normal weights for the features are not as affected, because the *time weights* make sure that these are independent of stage (see Section 5.2).

[2]The program crashed during this experiment and was restarted. That affected the training order: 10 blanks, 20 blanks, 10 blanks, 20 blanks, 40 blanks etc.

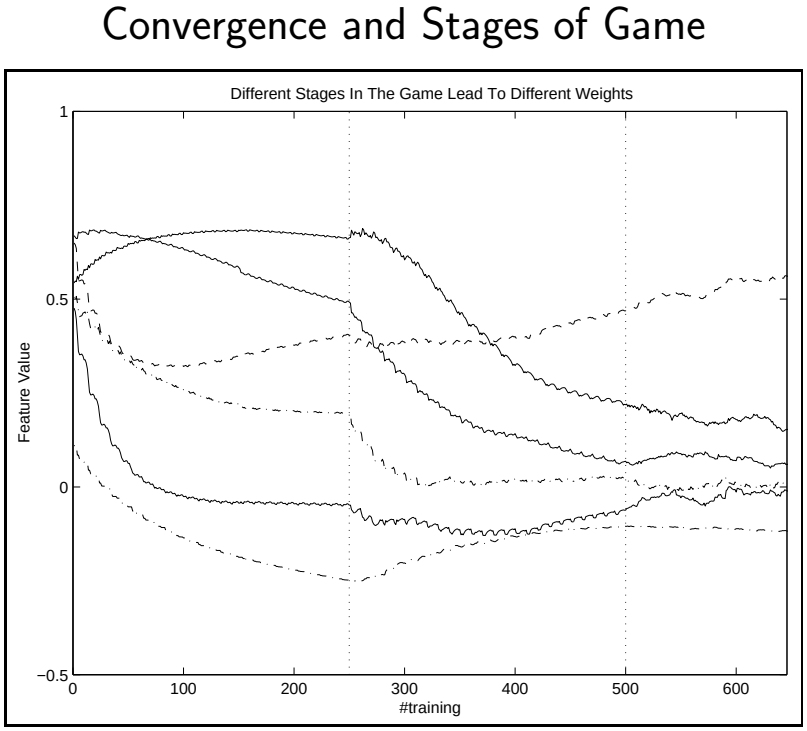


Figure 3: *We let* FIREBRAND *train on played games as described in Section 2.1. First it trained on games with ten blank squares, followed by 20 blank squares after 250 iterations. As can be seen, the weights of the features seem to converge to a value in the first stage, and then in the second stage they try to converge to a different value.*

This raises the question how the program will work when the depth is not constant. As described in Section 3, we use different search depths during the actual game play depending on how much time we have left. These effects will be explored for the next version of FIREBRAND.

However, the difference in the graph may also only point out that a smaller search depth leads to slower convergence (the "true" values will propagate back slower because the search depth is smaller). As can be seen in Figure 6, the *form* of the graphs are similar. We only had time to run a few experiments, and it is necessary to have more data to make a good interpretation.

## 5.4 Separate the Red from the White Player

*Is there a difference in playing White versus Red?*

We wanted to examine the difference between playing white or red. Our training algorithm lets the programs alternate between the colors for each game. However, there could be a different optimal strategy for white and red. Thus, we changed the way we trained the games to get to games that only had played a single color. We then compared their weights for the features with a normal game which has played both sides.

As can be seen in Figure 7, the weights still converge on the same value. The four programs in the figure were all started with the same set of initial weights. The first two graphs depict

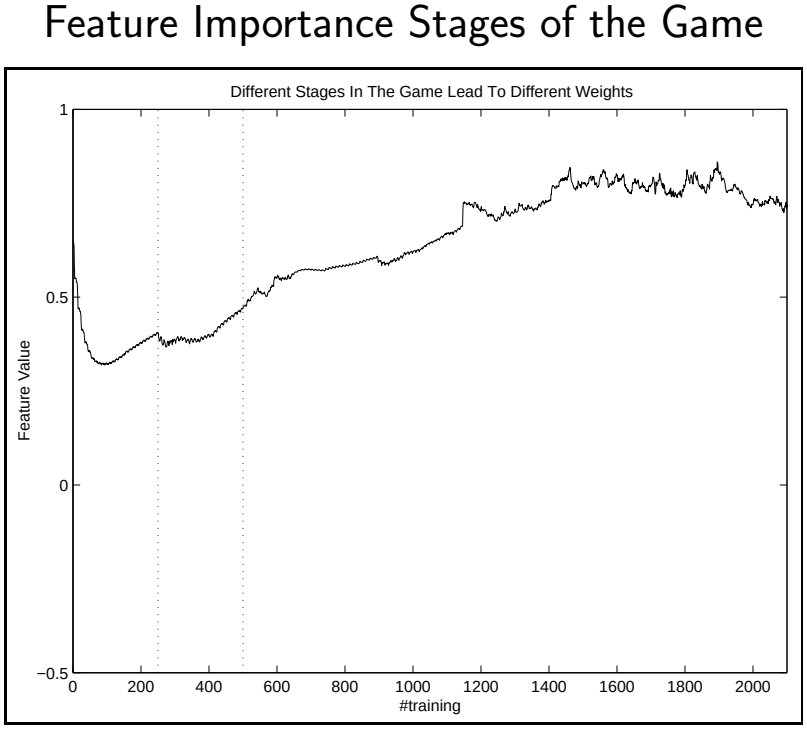


Figure 4: *In different stages of the game, the same feature may be more or less important. This is the feature 8 (see Section 4), which is not very important in the end of the game (beginning of graph) but more important in the beginning of the game. During this experiment, the program crashed. We restarted the program, with the effect that we trained on 10 blanks, 20 blanks, 40 blank and then repeating the whole cycle.*

the games that played both colors while the last two graphs shows games which either only played red or white.

There are a few differences in the graphs but the weights seem to converge to the same value in all graphs. We still used the same features for both red and white, which influenced the result.

## 5.5 Forcing the Strategy by Manual Adjustment of Weights

By looking at the final weights for the features we tried to analyze the strategy of the game. It was very surprising for us to find out that in one of our best functions the weight of feature 5 (difference in number of pieces) was negative, and the corresponding *Time weight* was large. This means that this feature is trying to minimize the number of pieces at the beginning of the game. This is seen in the Figure 1.

Doubting the strategy of minimizing pieces, we manually tried to raise the importance of this weight. By doing this, we hoped the program would avoid a local minimum. At the end of the game we increased the normal weight value with 0.4 and the time weight value with 0.4. The result was that the evaluation function "adopted" the change and even made it more

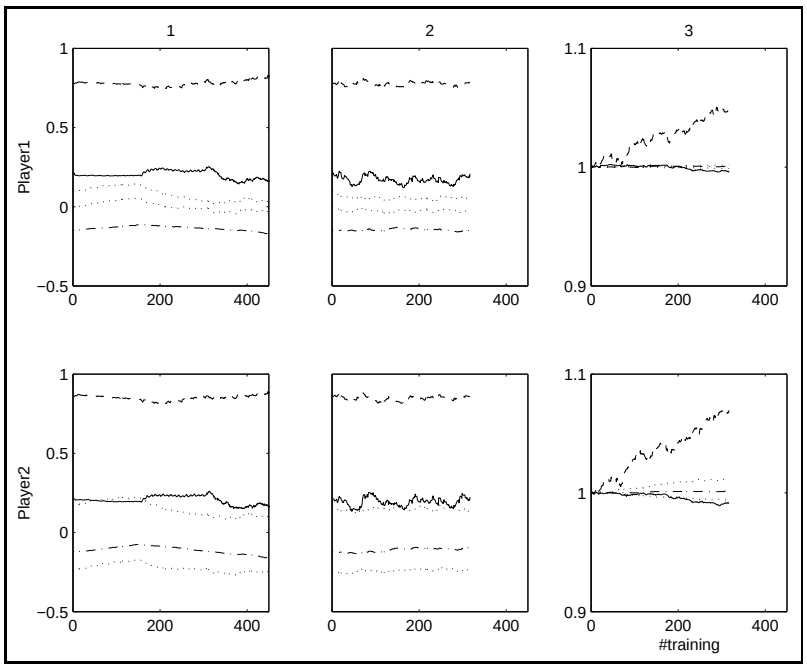


Figure 5: *When normal training hardly changes the weights, time weights can still give reward from training*

radical. To evaluate the resulting games, we let them compete (against each other and others). They performed equally well.

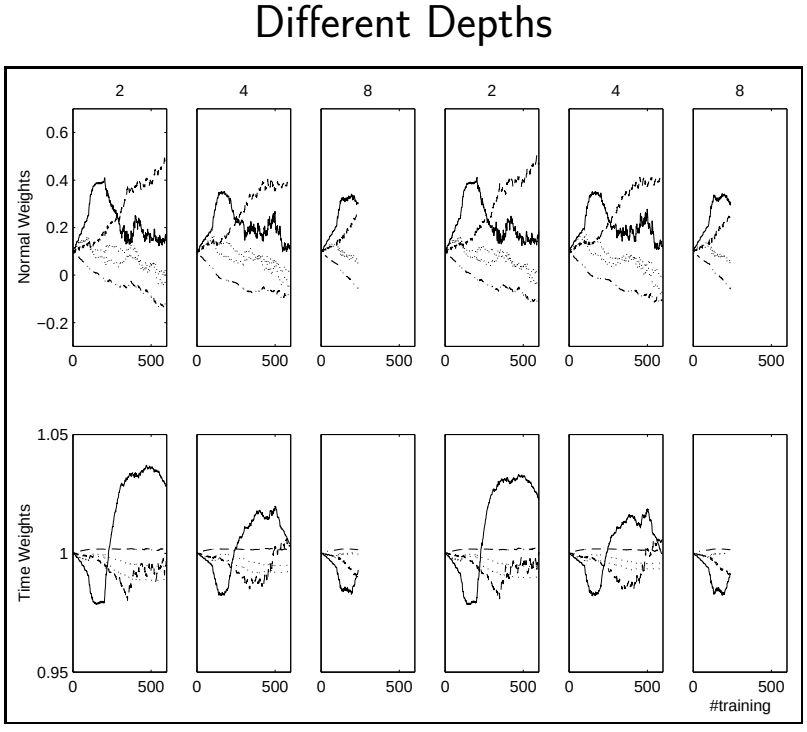


Figure 6: *Weight values for different search depths. As can be seen, the normal weights look similar while the magnitude for the time weights change even though the form stays the same.*

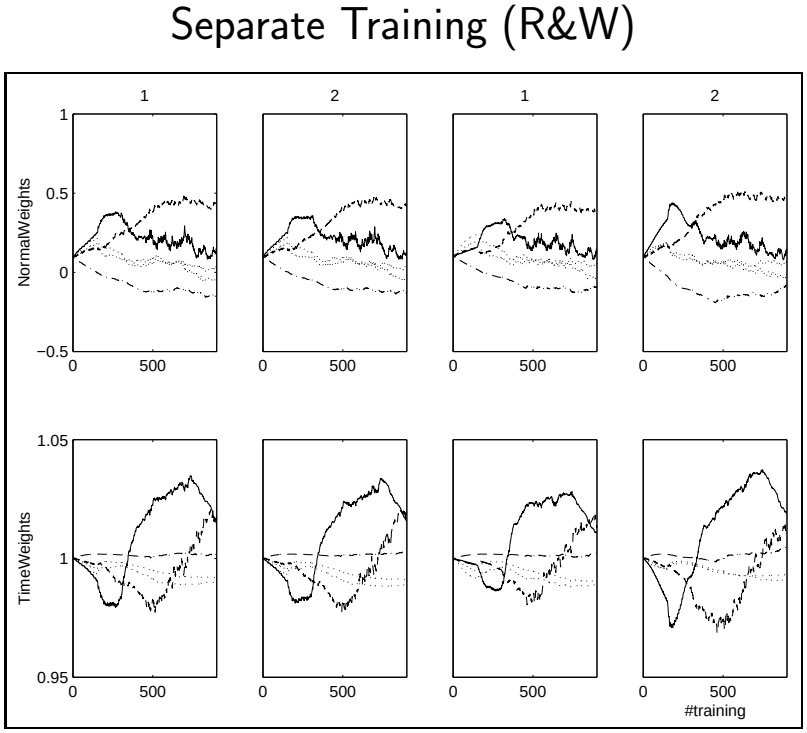


Figure 7: *The first two graphs shows the weights for two programs that alternatively played white or red. The last two graphs shows programs that always played the same color. All four started with the same initial weights. As can be seen, the color does not matter in our case for the importance of the features.*

# 6 Conclusions

As we have shown in this report, it is quite easy to build a moderately good Othello player. FIREBRAND was built by using rudimentary techniques in reinforcement learning. Very little time and expertise was spent in designing the evaluation features. However, after training the program it managed to reach a good level, and could beat its authors. The gain through $\alpha\beta$-pruning has been shown, and also the value of node ordering even though FIREBRAND has no such feature in the current version. We also show an effective time management policy so that FIREBRAND most likely has enough time to complete the whole game.

By analysing the features of the best version of FIREBRAND we find some amazing results. The program tries to keep down its own pieces as much as possible. A reason for this might be that it restrict the moves of the other player.

# References


[1] Programming assignment no. 2. CS221, Stanford, US, (1999)

[2] Lee, Kai-Fu and Mahajan, Sanjoy. The Development of a World Class Othello Program *Artificial Intelligence 43, 1990*, 21–36.

[3] Russel, Stuart and Norvig, Peter. Artificial Intelligence: A Modern Approach. *Prentice Hall 1995*

[4] Many thoughts and ideas were also given by Professor Daphne Koller and her TA:s in the class cs221 taught at Stanford (fall 99/00).