# An Analysis of Heuristics in Othello

**Vaishnavi Sannidhanam and Muthukaruppan Annamalai**

Department of Computer Science and Engineering,
Paul G. Allen Center,
University of Washington,
Seattle, WA-98195

{vaishu, muthu}@cs.washington.edu

**Abstract**

Game playing, as one of the most challenging fields of artificial intelligence has received a lot of attention. Games like Othello, which have proven to fit in well with computer game playing strategies, have spawned a lot of research in this direction.

Though numerous computer Othello players have been designed, and have beaten human world champions, it is not very clear as to how the various Othello heuristics interact. This paper implements and examines various heuristics, in an attempt to make observations about the interplay between the heuristics, and how well each heuristic contributes as a whole. Identifying heuristics that contribute immensely to Othello game-play implies that more processor cycles could be allocated in that direction to enhance the quality of play. Due to the complexity of accurate calculations, most heuristics tend to approximate. Like a typical stability heuristic, that approximates stability, instead of accurately calculating it. By realizing the importance of the stability heuristic, it enables one to decide the amount of time to spend in such a function.

Various experimental data is collected and analyzed in the evaluation section, where an insight is provided as to what the optimal way of using heuristics would be. Though this paper concentrates on Othello and heuristics pertaining to it, such analysis is applicable for other similar games, which descend from Go [16].

## 1. Introduction

Game playing, as one of the well-admired components of artificial intelligence research, has captured tremendous amounts of attention. Computers, looking ahead beyond the next move and judiciously rationing out the next best move, mimic human intelligence, and at times, surpass it. Computers have, to a good extent, captured the essence of game playing along with its intricate complexities. The importance of game playing arises out of the competition that exists between the human race and machines. It is a race to prove superior intelligence.

Othello [14], a board game derived from Go, has been an example where computers have exemplified great game play, beating human world champions [9]. The primary reason being the small branching factor, allowing the computer to look ahead in abundance, thrashing human intuition and reasoning. As processors increase in speed and complexity, the ability of computers to reason beyond the current state increases, while human intelligence maintains a fairly static average performance over generations. This leads to an extending gap in the Othello-playing ability of humans and computers.

The massive success of Othello, apart from the small branching factors involved, can also be attributed to heuristic functions successfully representing the state of the game. Heuristics in Othello, suffer from few pitfalls, when chosen cautiously. The typical calculated heuristic value is a linear function of various different heuristics. We implement various heuristics, along with several optimizations, and determine the contribution of each heuristic to the entire game play. In the process, we also determine the manner in which each heuristic interacts with the others. Understanding the role of each heuristic in game play would enable us to use extra processor cycles in a productive fashion.

We examine related work in Section 2, and offer a description of the rules of Othello for the reader

in Section 3. Section 4 discusses our different search strategies, while Section 5 examines the different heuristics we have implemented. Section 6 presents experimental results and we conclude with future work in Section 7.

## 2. Related Work

The birth of artificial intelligence had spawned research to enable computers to adopt a methodology of thinking that simulates the human brain. But success was brought about through various other techniques such as alpha-beta search, which do not mimic the human thought process. Such techniques, coupled with the massive computing power of a commodity system, have proven to be far superior to the human brain in certain situations.

The use of Othello to demonstrate the game playing capabilities of machines is not new. Othello's rigorous constraints and rules severely constrain the next set of possible moves, hence implying a very low branching factor, making it practically feasible for a computer search. The most common approach to playing Othello and other related games, is to perform searches on game trees using the alpha-beta search [4,11]. [4] provides an overview of different tree search algorithms, while [11] makes a comparison between various modified forms of the minimax tree search algorithm. Computer Othello players, have comprehensively beaten human world champions [9], leading to a greater interest in this game to stay ahead of the human community of Othello players.

Improved game play in Othello can result from (i) improvements in the search strategy used, or (ii) from better heuristic functions, or (iii) by using learning methods to enable the computer to learn for itself from massive amounts of data, or (iv) by adding extra hardware support.

Improvements in the search strategy was done in [7,8]. [7,8] add probability to the minimax search algorithm, adding non-determinism to the approach, making it more effective. [11] provides an analysis of various modifications to the minimax strategy. Extra hardware support has been added in [12] to enhance the game playing process. Coin-flipping and other Othello related tasks are extremely fast on such machines.

Learning has proved to be an efficient component in game playing systems. This is primarily due to the fact that massive amounts of data of Othello game proceedings can be taken advantage of. [5] proposes an Othello playing program, Bill, which is a direct descendant of Iago [1]. Bill uses Bayesian learning techniques for the suggested feature combinations in Iago. [2] uses supervised learning to improve minimax searching.

Though there has been a lot of prior work on developing heuristic functions [13], the interaction between heuristics and their contribution to Othello game-play is unclear. Our work concentrates on identifying fruitful heuristic components and analyzing their behavior. To aid us in this process of analysis, various experiments were conducted.

## 3. Othello

Othello belongs to the family of board games that were derived from Go. It consists of an 8 x 8 board with black and white coins. We name each column from left to right with an alphabet starting with 'A'. Each row is numbered from top to bottom starting with 1. This is the notation that shall be followed throughout the rest of the paper.

The initial configuration of the board is as shown in Figure 1. Each player is associated with a coin color, either white or black. A player owns the squares in which his/her color coin is present. White initially owns d4 and e5, while black owns e4 and d5. Black always moves first.

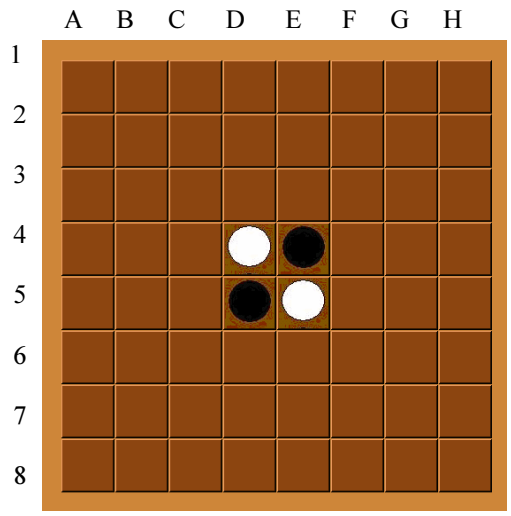


**Figure 1: Board showing the starting game configuration for Othello**

The game progresses as each player makes moves. A move is made by placing a coin in an empty square. When a player does so, all of the opponent's coins *bracketed* between the newly placed coin and another coin of the same color get *flanked* into the color of the newly played coin. The flanking of coins is shown in Figures 2a and 2b.

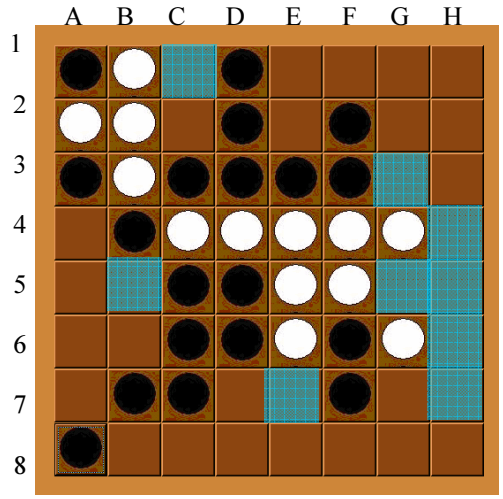


**Figure 2a: Blue highlighted squares show valid moves for black.**

A move is valid only if it flanks at least one of the opponents' coins. When such a move does not exist, the player skips a turn and the opponent makes a move.

The game ends when at least one of the following conditions hold true:

1) The board is full, with no empty spaces
2) The board has coins of only one color
3) When no player has a valid move

## 4. Search Strategies

This section examines one of the two most vital components of a game playing computer, the search mechanism. The second component, heuristics, is examined in the subsequent section. The search technique allows the computer to look ahead and explore different moves through a systematic generation of next possible moves. The heuristic function complements the search process by evaluating the state of the game along the various paths.

The efficiency of the search technique determines the extent to which the game tree is explored. The greater the efficiency of the search technique, the greater the number of nodes that can be searched, hence the farther the look ahead,

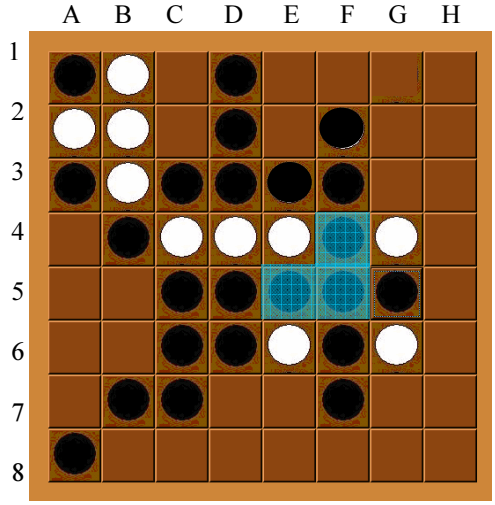


**Figure 2b : Blue highlighted squares show the flanked coins once the black has been placed in (G, 5)**

and the greater the insight obtained. An ideal search mechanism should be efficient and prune irrelevant paths without loss of any important information. We utilized three different search strategies: (i) Minimax search (ii) Alpha-Beta

search and (iii) Alpha-Beta search with iterative deepening. The following few subsections examine each one in greater detail.

The game tree to be searched over is the tree of all possible game states with the current state of the game at the root. The search cannot explore the entire game tree for a predominant portion of the game, since that would require tremendous amounts of time. Hence, the search is typically done to a particular depth. If the search depth is six, then that implies exploring all possible states which are at most six moves away from the current state. Towards the end of the game, the search can be carried all the way to the end.

We have a database of predetermined moves, which we refer to when possible. This avoids any time wasted in searching when the next move to make is an obvious one. For example, if a corner could be captured, then that should be the next move executed, hence searching would be redundant in such cases.

### 4.1 Minimax Search

The minimax search [4,11] does a depth-first exploration of the entire game tree. The heuristic functions are utilized at the leaves to provide utility values. The utility values are then backed up all the way to the root. The manner in which the values are backed up to a node depends on whether the node is a min node or a max node. The max player has to make a move while at a max node, while the min player has to do so at the min node. The max player is the player who has to make the actual next move in the game, and has to maximize his/her utility value, while the min player does the reverse. A typical minimax tree is such that the min player and max player alternate. This need not necessarily hold throughout, since turns may be skipped by players in certain games, such as Othello.

### 4.2 Alpha-Beta Search

Alpha-beta search [4] is similar to minimax, except that efficient pruning is done when a branch is rendered useless. Such pruning tends to be rather effective and the search can proceed to great depths, allowing the computer to implement a relatively more powerful look ahead. Pruning is done when it becomes evident that exploring a branch any further will not have an impact on its ancestors.

### 4.3 Alpha-Beta Search With Iterative Deepening

It becomes unclear as to the optimal depth to search up to, if the depth is to be defined statically. Hence, we implemented alpha-beta with iterative deepening. The algorithm sets depth to a reasonable initial value of three. Then the depth is increased and the search is conducted again. This is done till the timing constraints are not violated. Before searching with an increased depth, a naïve check is made to ensure that the search about to be spawned will not violate timing constraints. Using the time taken for the previous depth, we approximately calculate a new depth at which the next iteration can take place. This new depth would be such that it finishes, according to the approximation, before the allotted time. If it doesn't, the process is not preempted, and the execution exceeds by a few milliseconds.

## 5. Heuristics

The heuristic functions control the ability of the computer to correctly determine how good a particular state is for a player. A number of factors determine whether a given state of the game is good for a player. For Othello, factors such as mobility, stability, corners and coin parity determine how favorable a particular position is for a player. The most intuitive way to calculate a heuristic value is to create a linear combination

of the quantitative representation of the various important factors.

We have two major functions that return the utility value of a state. The first utility function is a linear combination of several heuristic components that are critical to the evaluation of the state. The second utility function uses statically assigned weights to squares on the board to calculate the utility value. Both these functions are discussed in greater detail in the following subsections, and either one can be used to return the utility value of a state.

### 5.1 Component-wise Heuristic Function

This methodology of calculating the utility value uses various different heuristics and assigns different weights to those heuristics. The state of an Othello game is evaluated after determining the mobility, coin parity, stability and corners-captured aspect of the configuration. We have a heuristic function to determine each one of these. Each heuristic scales its return value from -100 to 100. We weigh these values appropriately to play an optimal game.

#### 5.1.1 Coin Parity

This component of the utility function captures the difference in coins between the max player and min player. The return value is determined as follows:

**Coin Parity Heuristic Value =**
**100* (Max Player Coins –Min Player Coins)/**
**(Max Player Coins + Min Player Coins)**

The most natural strategy that many primitive computer Othello players employed was to base their move on a greedy strategy that tried to maximize the number of coins of a player at any point. Such strategies failed miserably, and obviously so. A single move can flank at most 18 coins, which implies that games can swing from the control of one player to another very rapidly. Since a complete exploration of the game tree would not be possible till the very end stages of the game, such a strategy does not incorporate the drastically dynamic nature of the game. Neither does it account for the instability of coins. A couple of stable coins might be better than ten unstable ones.

#### 5.1.2 Mobility

An interesting tactic to employ is to restrict your opponent's mobility and to mobilize yourself. This ensures that the number of potential moves that your opponent has would drastically decrease, and your opponent would not get the opportunity to place coins that might allow him/her to gain control. Mobilizing yourself would imply a vast number of moves to choose from, hence indicating that you can exercise power and control the proceeding of the game.

Mobility comes in two flavors [1], (i) actual mobility and (ii) potential mobility. Actual mobility is the number of next moves a player has, given the current state of the game. Potential mobility is the number of possible moves the player might have over the next few moves. Note that moves that are currently not legal, but might become legal in the near future are accounted for in the calculation of potential mobility. Hence, potential mobility captures the mobility of the player in the long term, while actual mobility captures the immediate mobility of the player. Potential mobility looks ahead on its own without the help of searching strategies, hence it can compensate for a small depth, when quantifying the mobility aspect of the game.

Actual mobility is calculated by examining the board and counting the number of legal moves for the player. Potential mobility is calculated by counting the number of empty spaces next to at least one of the opponent's coin. Note that potential mobility is a rather crude measure but it proved to be rather effective. There exists a tradeoff between the complexity of the calculation of potential mobility and its effectiveness. The more effective it is required to be, the more complex it would become because looking ahead for mobility is a difficult task. If

the routine becomes too complex, then it would take up a significant amount of processor time, which could have been spent searching the game tree. The actual mobility heuristic value is calculated as follows and the potential mobility heuristic value is calculated in an identical fashion.

```
if((Max Player Actual Mobility Value + Min Player Actual Mobility Value) !=0)
    Actual Mobility Heuristic Value =
        100* (Max Player Actual Mobility Value –Min Player Actual Mobility Value)/
            (Max Player Actual Mobility Value + Min Player Actual Mobility Value)
else
    Actual Mobility Heuristic Value = 0
```

### 5.1.3 Corners Captured

Corners are the four squares a1, a8, h1, and h8. The specialty of these squares is that once captured, they cannot be flanked by the opponent. They also allow a player to build coins around them and provide stability to the player's coins in the environment. Capturing these corners would ensure stability in the region, and stability is what determines the final outcome to quite a large extent. There is a high correlation between the number of corners captured by a player and the player winning the game. Of course, it is not true that capturing a majority of the corners would lead to victory, since that clearly need not hold. But capturing a majority of the corners, allows for greater stability to be built.

We assigned weights to corners captured, potential corners, and unlikely corners. A player's potential corner is one which could be caught in the next move, while an unlikely corner is poised such that it cannot be captured in the near future. These factors weighed together give rise to a player's corner heuristic value. The return value is calculated as follows:

```
if((Max Player Corner Value + Min Player Corner Value) !=0)
    Corner Heuristic Value =
        100* (Max Player Corner Heurisitc Value –Min Player Corner Heuristic Value)/
            (Max Player Corner Heuristic Value + Min Player Corner Heurisitc Value)
else
    Corner Heuristic Value = 0
```

### 5.1.4 Stability

Stability of coins is a key factor in Othello. The stability measure of a coin is a quantitative representation of how vulnerable it is to being flanked. We classify coins as belonging to one of three categories: (i) stable, (ii) semi-stable and (iii) unstable. Stable coins are coins which cannot be flanked at any point of time in the game from the given state. Unstable coins are those that could be flanked in the very next move. Semi-stable coins are those that could potentially be flanked at some point in the future, but they do not face the danger of being flanked immediately in the next move. Corners are always stable in nature, and as you build upon corners, more coins become stable in the region.

Weights are associated to each of the three categories, and we sum the weights up to give rise to a final stability value for the player. Typical weights could be 1 for stable coins, -1 for unstable coins and 0 for semi-stable coins.

```
if((Max Player Stability Value+ Min Player Stability Value) !=0)
    Stability Heuristic Value =
        100* (Max Player Stability Value–Min Player Stability Value)/
            (Max Player Stability Value+ Min Player Stability Value)
else
    Stability Heuristic Value = 0
```

### 5.2 Static Weights Heuristic Function

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | -3 | 2 | 2 | 2 | 2 | -3 | 4 |
| -3 | -4 | -1 | -1 | -1 | -1 | -4 | -3 |
| 2 | -1 | 1 | 0 | 0 | 1 | -1 | 2 |
| 2 | -1 | 0 | 1 | 1 | 0 | -1 | 2 |
| 2 | -1 | 0 | 1 | 1 | 0 | -1 | 2 |
| 2 | -1 | 1 | 0 | 0 | 1 | -1 | 2 |
| -3 | -4 | -1 | -1 | -1 | -1 | -4 | -3 |
| 4 | -3 | 2 | 2 | 2 | 2 | -3 | 4 |

**Figure 3: Shows the static weights assigned to each individual position in the board**

An alternative to using the utility function discussed in Section 5.1 is to have a static board of weights associated to each coin position as shown in Figure 3 [17]. The heuristic value for a player is calculated by adding together the

weights of the squares in which the player's coins are present.

The static board implicitly captures the importance of each square on the board, and encourages the game play to tend towards capturing corners. Dynamically changing these weights would mean that we would have to use heuristics to calculate the weight of a position based on its stability, offer of mobility and etc. This would imply that the calculation of the utility value would be similar to the one discussed in Section 5.1.

**Utility Value =**
**Max Player Utility Value– Min Player Utility Value**

## 6. Evaluation

We implemented the Othello game along with the various heuristics and search strategies in the Visual Studio .NET C++ framework. The entire code base consisted of around 4000 lines of code. The search strategies implemented were the ones discussed in Section 4, while the heuristics implemented were the ones explained in Section 5. We ran all tests on a system with 2 GB of RAM, and 4 Intel Xeon processors, each clocked at 2.8 GHz.

| | Coins | Corners | Stability | Mobility |
|---|---|---|---|---|
| Coins | N/A | 27-37 Corners | 26-38 Stability | 14-50 Mobility |
| Corners | 53-11 Corners | N/A | 39-25 Corners | 39-25 Corners |
| Stability | 58-0 Stability | 13-51 Corners | N/A | 23-41 Mobility |
| Mobility | 59-5 Mobility | 29-35 Corners | 0-42 Stability | N/A |

**Table 1a: Shows how each heuristic performed against the other heuristics. Heuristics in column 1 played first and each cell specifies the score and the heuristic that won. A cell x,y represents the results of the game between heuristic x and heuristic y.**

This section illustrates the relative importance of the various heuristics. A series of experiments were conducted with our Othello game, in which we enabled different heuristics of varying weights and activated computer versus computer auto play. Though we implemented the three different search strategies mentioned in Section 5, we use the alpha-beta search strategy with a depth of 5, unless otherwise mentioned.

### 6.1 One-on-One Heuristic Comparison

Table 1(a) shows the results of the games that were conducted. Table 1(b) represents the results in Table 1(a) as the total number of coins won and lost by each heuristic. We played each heuristic against every other heuristic, and two such games were played for each pair, one with a different heuristic starting both times. The games were arranged such that the heuristics listed down column 1 played first.

It is evident from the tables that the corner heuristic is the most powerful stand-alone heuristic. The corner heuristic beats all heuristics. Mobility and stability have a close competition for second place, with both matching the other almost equally.

The corner heuristic guides the game in a direction that enhances the chances of capturing corners. The greater the number of corners captured, the more the control a player can exercise over the middle portions of the board, thus flanking a significant portion of the opponent's coins. Thus, irrespective of what the other heuristics play, this heuristic ensures that corners are not easily given up. This enables the corner heuristic to nullify to a certain extent the advantage of the other heuristics when playing against them.

| | Coins Won | Coins Lost |
|---|---|---|
| Corners | 254 | 130 |
| Mobility | 204 | 158 |
| Stability | 199 | 157 |
| Coins | 83 | 295 |

**Table 1b: Shows the number of coins won and lost by each of the heuristics on the whole, when it played against other heuristics.**

Stability, with its classification of stable, semi-stable and unstable moves is able to guide the

game in a good direction. It ensures that as many coins as possible are captured and promoted higher in the stability order. This strategy minimizes the opportunity for the opponent to flank coins and take over the game. Games

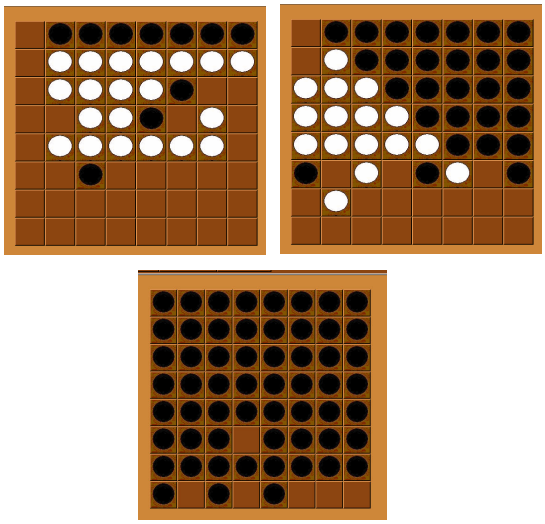

**Figure 4: Board configurations in the match between Coin parity (white) and Stability (black) heuristics**

played with stability as the primary concern, tend to play moves to capture corners and edges as fast as possible, and build upon these regions. This introduces stability in the region, which pervades through the board subsequently. Figure 4 shows how the stability heuristic captures corners and edges when it plays against the coin parity heuristic. Though the greedy coin parity heuristic starts well, by grabbing unstable coins, the stability heuristic prevails as it grabs stable coins and flanks the opponent's coins. Stability loses to the corner heuristic primarily because the corner heuristic is more hungry for corners, hence preventing the stability heuristic from capturing them. This means that stability cannot be built by the stability heuristic, because of the lack of its ability to grab the most stable regions of the board, the corners, against the corner heuristic.

Mobility, on the other hand, is effective mainly because it minimizes the number of moves the opponent has, while maximizing the other player's moves. This implies that the opponent is never able to take complete control over the game due to the lack of available moves. Mobility forces the opponent to choose from a constrained set of moves. The player using the mobility heuristic, on the other hand, would enjoy a wide variety of moves.

As expected, the greedy strategy of maximizing the number of coins does not perform too well. In Othello, it is easy to gain a lot of unstable coins in one move, but losing them is equally easy. The player with the most stable coins in the final stages of the game controls the board. Hence, though the ultimate goal is to maximize the number of coins, using that as a heuristic fails.

### 6.2 Heuristic Contributions

| | E-Coins | E-Corners | E-Stability | E-Mobility |
|---|---|---|---|---|
| Everything | 60-4 Everything | | | |
| Everything | | 39-25 Everything | | |
| Everything | | | 47-17 Everything | |
| Everything | | | | 58-6 Everything |

**Table 2a**

| | Everything | Everything | Everything | Everything |
|---|---|---|---|---|
| E-Coins | 53-11 E-Coins | | | |
| E-Corners | | 14-50 Everything | | |
| E-Stability | | | 4-60 Everything | |
| E-Mobility | | | | 28-35 Everything |

**Table 2a & 2b: These tables show the importance of a heuristic X by playing all heuristics against everything with heuristic X omitted. Heuristics in column 1 played first and each cell specifies the score and the heuristic that won. A cell x,y represents the results of the game between heuristic x and heuristic y.**

Tables 2a and 2b depict the results for the games with a player with all heuristics enabled versus a player with all but one heuristic enabled. Such an experiment would depict the importance of the omitted heuristic, and would give an approximate idea of their impact on the game play, with optimal weight settings. The weights for the heuristics were as follows: the corner heuristic had a weight of 30, the mobility heuristic had a weight of 5, while the stability

heuristic had a weight of 25 and the coin parity heuristic also had a weight of 25. These weights were chosen after a lot of experimentation against online computer Othello players, such as [15]. After varying them extensively, these were found to provide maximum benefit. These are the weights used for other experiments as well, unless otherwise stated.

We use the term *Everything* to represent the function that contains all the heuristic components discussed in Section 5.1. *E-x* is used to represent the function that calculates the heuristic value using all the heuristics in *Everything* except x. It is interesting to note the results of the games between the *Everything* heuristic Vs *E-Coins* heuristic. In one game, *Everything* wins, and in the other *E-Coins* wins. This fickle behaviour can be attributed to the greedy nature of the coin parity heuristic. While using the coin parity heuristic, it is very easy to be caught in a local maxima trap, which is what happened in the game in which *E-Coins* won. Since the coin parity heuristic drove the *Everything* heuristic to a local maxima, the *E-Coins* heuristic was able to beat it easily. In the other case, where *Everything* won, either no such local maxima was encountered, or the rest of the heuristics managed to maneuver the computation safely away.

In Section 6.1, it was noted that the most powerful standalone heuristic was the corners heuristic. The results in tables 2a and 2b suggest that when in a group, stability makes the greatest impact. The main reason for this can be attributed to the corners-stability heuristic combination. The corners heuristic wins corners and hence stable positions. The stability heuristic builds upon these stable positions to provide a strong position for the player. When *E-Stability* is played with *Everything, Everything* is defeated drastically both the times. This is because without the help of stability, *E-Stability* is not able to work up a strong hold on the game, while *Everything* uses its stability heuristic to take control.

To confirm the effect of the corner-stability heuristic combination, we conducted a game between the stability heuristic against the corner and stability heuristics put together. The corner and stability heuristics won both the times with scores of 36-28 and 20-44. This proves that the stability heuristic fares well in the presence of the corners heuristic. In such a group setting of heuristics, stability seems to have the most impact.

Corners and mobility, as expected, cause a degradation in game play quality when they are removed from the *Everything* heuristic. This is very obvious, since the corners heuristic ensures that corners, as starting stable positions are caught, and that the stability heuristic can build upon this. Corners appear to be a less powerful heuristic than stability when it is not a standalone heuristic, primarily because stability is better complemented by the mobility heuristic. Capturing corners, from our experience, increases mobility, hence the mobility heuristic aids the stability heuristic to capture corners and build stability. The mobility heuristic, however, does not aid the corner heuristic to build stability once the corners have been captured.

The mobility heuristic ensures that the opponent does not have too many moves to choose from, hence restricting the opponents control over the board. Though mobility has a low weight, it has such a great impact on the game play, as demonstrated by the results. This suggests that increasing the weight of the mobility heuristic might enhance play quality, but that was not true during our experimentation, because high mobility downplayed other heuristics leading to bad moves.

### 6.3 Component-wise Heuristic Vs Static Board Heuristic

The component-wise heuristic is basically a mix of the four heuristics mentioned in Section 5.1. The static board heuristic is the one that was discussed in Section 5.2, where board positions are assigned certain static weights. Two games

were played between the two. The game with the static board heuristic starting first had a result of 26-38 with the component-wise heuristic winning. While the other game, with the component-wise heuristic playing first, had a score of 60-4 with the component-wise heuristic winning again.

The main reason for the defeat of the static board heuristic is due to its lack of ability to dynamically change weights to represent the current state of the game. The component-wise heuristic captures that state of the game and suitably modifies the weight in order to guide the game in the right direction. For example, the static board heuristic does not take into account the stability of the current state of the game before making the next move. The static board heuristic takes a rather narrow-minded view of the game by claiming that certain squares are always good positions to play coins at, irrespective of the current state of the game.

### 6.4 Component-wise Heuristic: Static Weights Versus Dynamic Weights

The weights of the component-wise heuristic, if made static, do not fulfill the need for different strategies at different points in the game. Hence, we changed the weights dynamically. During the first few moves, stability and mobility were given high weights. This was to ensure that we try to curb the opponent's ability to make moves early on and we also try to build stability. As the game progresses, corners and stability obtain a high weight to ensure that they both play hand-in-hand to build a stable position. When we are able to search the tree to the end of the game, we give all the weight to the coin parity heuristic. This would be fine because there would be no local maxima anymore since we can search till the end and avoid any pitfalls. Two games were played between the component-wise static weights and component-wise dynamic weights. The game with the dynamic weights starting first had a score of 46-18 with the dynamic weights winning. The other game with the static weights starting first had a score of 16-48 with dynamic weights winning. This proves the superiority of the dynamically varying the weights.

### 6.5 Alpha-beta Search with Iterative Deepening

We conducted experiments to determine the effectiveness of the alpha-beta search with iterative deepening. We ran a few experiments with a time limit set to three seconds. The experiments were run with the optimal weights. The average depth searched was 5.6, and the maximum time taken for any search was 2.75 seconds. The time taken for an average move was 2.69 seconds. This shows that our approximation was well within the bounds. We used an average branching factor of 10 to compute the expected computation time for a particular game tree for various depths. Changing the branching factor for estimation purposes led to frequent timing violations, hence we maintained it at 10. A feature like this is required to enable a computer Othello player to enter a tournament. It would also ensure less frustration from the user who would not be required to wait for tremendous amounts of time.

### Summary

The results suggest that stability is a very powerful heuristic when used in conjunction with the corner, coin parity and mobility heuristics. Currently, the implementation of a typical stability heuristic provides an approximation to the actual stability value. This is due to the complexity involved. But allocating more processor power for this task would definitely prove to be profitable.

## 7. Conclusion and Future Work

Game playing has always been one of the most attractive fields of artificial intelligence research, and it will continue to be so. It is one of the parts of artificial intelligence that the common man observes and interacts with.

We evaluated the importance of a few heuristics in enhancing Othello game play. This paper tried

to explain the interaction amongst the various heuristics that are utilized to evaluate the state of an Othello game. We also analyzed their importance. We found that increasing the accuracy of the stability heuristic would enhance game play greatly. It was also interesting to note that though corners was the most powerful standalone heuristic, stability played a major role in the component-wise heuristic function. This study would allow one to identify the most important aspects, and enable more processor power to be thrown into it in order to increase the accuracy of the heuristic.

Future work would include the incorporation of learning strategies into the system. Such strategies tend to be very powerful, since they could make use of vast amounts of already existing data and can avoid pitfalls that deterministic algorithms can suffer from. We can also build more upon our framework to give rise to an extremely good Othello player, that can then go on to participate in tournaments. That would require careful weight modification, optimized lookup tables, and powerful learning strategies.

## References


[1] Rosenbloom, P., *A world-championship-level Othello program,* Artificial Intelligence*, 19*, pp 279-320, 1982.

[2] Buro, M., *Improving heurisitic minimax search by supervised learning,* Artificial Intelligence, 134, pp 85-99, 2002.

[3] Utgoff, E. P. *Feature Construction for Game Playing*, Technical Report, University of Massachutes, Amherst, MA.

[4] Russell S., Norvig P., *AI: A Modern Approach*, (Prentice Hall) 2nd edition, 2003.

[5] Lee, K.; Mahajan, S.: *The Development of a World Class Othello Program*, Artificial Intelligence, 43, pp. 21 – 36, 1990.

[6] M. Buro, How Machines have Learned to Play Othello, IEEE Intelligent Systems J. 14(6) 1999, 12-14

[7] M. Buro, *Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello*, Workshop on game-tree search, NECI, 1997.

[8] M. Buro, *ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm*, ICCA Journal 18(2), pp 71-76, 1995.

[9] M. Buro, *The Othello Match of the Year: Takeshi Murakami vs. Logistello* , ICCA Journal 20(3), pp 189-193, 1997.

[10] M. Buro, *The Evolution of Strong Othello Programs*, in: Entertainment Computing - Technology and Applications, R. Nakatsu and J. Hoshino (ed.), Kluwer, pp. 81-88, 2003.

[11] Campbell, Murray S. and Marsland T.A. *A Comparison of Minimax Tree Search Algorithms* Artificial Intelligence 20, pp 347-367, 1983.

[12] Hewlett C., *Hardware Help in an Othello Endgame Analyzer,* Heuristic Programming in Artificial Intelligence, the first computer Olympiad, pp. 219-224, 1989.

[13] Abramson B., and Korf R.E., *A model of two-player evaluation functions,* Proceedings of the Sixth National Conference on Artificial Intelligence, pp 90-94, 1987.

[14] *History and Basic rules of Othello:* http://www.othello.org.hk/tutorials/eng-tu1.html,

[15]http://home.tiscalinet.ch/t_wolf/tw/misc/reversi/html/index.html

[16] http://www.game-club.com/gohis/go.htm

[17] http://www.cs.rochester.edu/~anustup/othello.html


## Acknowledgements


We would like to thank Dan Weld for his help in guiding the project in the right direction. We would also like to thank Parag, for his initial advice on an appropriate AI project, and what that means.


## Appendix

All the code was written by us, and we did not download any code from anywhere. We both worked on everything together, and did not split the tasks, as we felt the given time was sufficient, and that two brains at a task is much better than one. Please refer to the user guide to play the Othello game.