ENGINEERING AND COMPUTER SCIENCE
THIRD YEAR PROJECT 1996/7

# OTHELLO GAME PLAYER

---


DAVID CURRIE
ST JOHN'S COLLEGE
CANDIDATE NUMBER - D1610



PROJECT PARTNER : ALISON BROWN, ST ANNE'S COLLEGE
PROJECT SUPERVISOR : MIKE REED, COMPUTING LABORATORY

## TABLE OF CONTENTS

**INTRODUCTION** ..... **1**

THE GAME OF OTHELLO ..... 1
PROJECT AIMS ..... 1
PROJECT MANAGEMENT ..... 1

**PROGRAM STRUCTURE** ..... **2**

**LOOKAHEAD SEARCH** ..... **4**

GAME TREES ..... 4
MINIMAX LOOKAHEAD ..... 4
ALPHA-BETA PROCEDURE ..... 5
ALPHA-BETA PERFORMANCE ..... 6
IMPLEMENTATION ..... 6

**GENETIC EVOLUTION** ..... **7**

OTHELLO STRATEGIES ..... 7
SURVIVAL OF THE FITTEST ..... 7
GENETIC ALGORITHMS ..... 8
STANDARD METHOD ..... 8
CROSSOVER ..... 9
RANK METHOD ..... 10
DIVERSITY ..... 10
APPLICATION TO OTHELLO ..... 11
RESULTS ..... 12

**CONCLUSION** ..... **13**

**BIBLIOGRAPHY** ..... **15**

BOOKS ..... 15
WORLD WIDE WEB RESOURCES ..... 15

**APPENDIX A - RULES OF OTHELLO** ..... **16**

**APPENDIX B - SAMPLE DOS OUTPUT** ..... **18**

**APPENDIX C - MINIMAX CODE** ..... **19**

**APPENDIX D - SAMPLE GENETIC ALGORITHM OUTPUT** ..... **21**

**APPENDIX E - SCREEN SHOTS** ..... **24**

# INTRODUCTION

## THE GAME OF OTHELLO

Othello is a relatively modern game. Its origins lie in England at the end of the nineteenth century but the game as it is known now was set down in Japan as recently as 1971. It rapidly became popular and the first world championship was organised in 1977. The motto of Othello is "Easy to learn, difficult to master" and as such it is ideal for computer implementation. The brief introduction to the game in Appendix A shows how simple the rules are and therefore why the initial coding of a playable game of Othello should, and did, not take long. However, the tactics and strategies of Othello, as explored by Alison, are very complex. Although computer implementations of Othello do exist we strove to implement and combine these existing strategies in a new way.

## PROJECT AIMS

The main aims of the project were set out as follows, with the majority of the project time to be spent on the second objective.

- Produce a computer program implementing the game of Othello.
- Implement a computer player with a varying skill level based on lookahead, heuristics and an innovative combination of ranking techniques.
- Include a user-friendly graphical interface.

It was decided to program the project using Microsoft Visual C++ 4.0. C++ classes allow for easy division of the project, providing a clear understanding of the required interfaces is had by both parties. Object Oriented methods also meant that it was possible to implement the player in such a way that various computer players could be played against each other, or a human player.

Before the front end was added, the program could be run under both UNIX and DOS. Visual C++ then enabled the relatively simple implementation of a graphical front-end for the Windows 95 environment.

## PROJECT MANAGEMENT

While investigating the possibilities of Visual C++ both Alison and myself wrote simple DOS implementations of Othello. Due to my greater initial knowledge of C++ I was able to exploit its features to a greater extent and it was decided that I should flesh out further my version of the program to provide the basis for the project. This would also provide the basic lookahead strategy.

Meanwhile Alison set about researching the many strategies that have been put forward for Othello. We decided to implement more than one of these. Alison was charged with writing the functions to perform the strategies while I would write a genetic algorithm which could then be used to evolve weightings to combine these strategies into one overall strategy. Although I have seen the use of a genetic algorithm for obtaining square weightings before, I believe that its use for combining strategies is a first. Finally I converted the program to the Windows 95 environment and Alison added the graphical interface.

## PROGRAM STRUCTURE

Object Oriented technology is based around the definition of classes. Instances of a particular class can be constructed and are known as objects. Each class contains methods which can be called to manipulate an object of that class. Objects may also have data attributes associated with them upon which these methods will operate. For example, an object of type CGame has data attributes representing the board and two players. It also has a method called play() which starts the game running.

Another important feature of Object Oriented languages is the idea of inheritance. For example, the black and white players stored in the CGame object have type IPlayer. IPlayer is an abstract class in that it is not possible to actually construct an object of this type. Instead, the classes CHuman and CComp1 are derived from IPlayer. This means they will inherit from IPlayer data attributes such as player colour, and methods such as makeMove(). makeMove() is what is known as a virtual method. IPlayer does not define makeMove(), instead the derived classes CHuman and CComp1 must provide their own implementations. In this way CGame can call the makeMove() method on a player without having to know whether it is a computer or human player, IPlayer acting as an interface between the two.

The Object Interaction Diagram (OID) in Figure 1 illustrates how the program structure operates. The calling program constructs a game, passing in two players, either human or computer, as parameters. This also creates a board which defaults to the initial counter layout. The numbers in the diagram now indicate a typical progression. To set the game in motion the play() method is called on the CGame object. This in turn displays the board by calling its display() method.

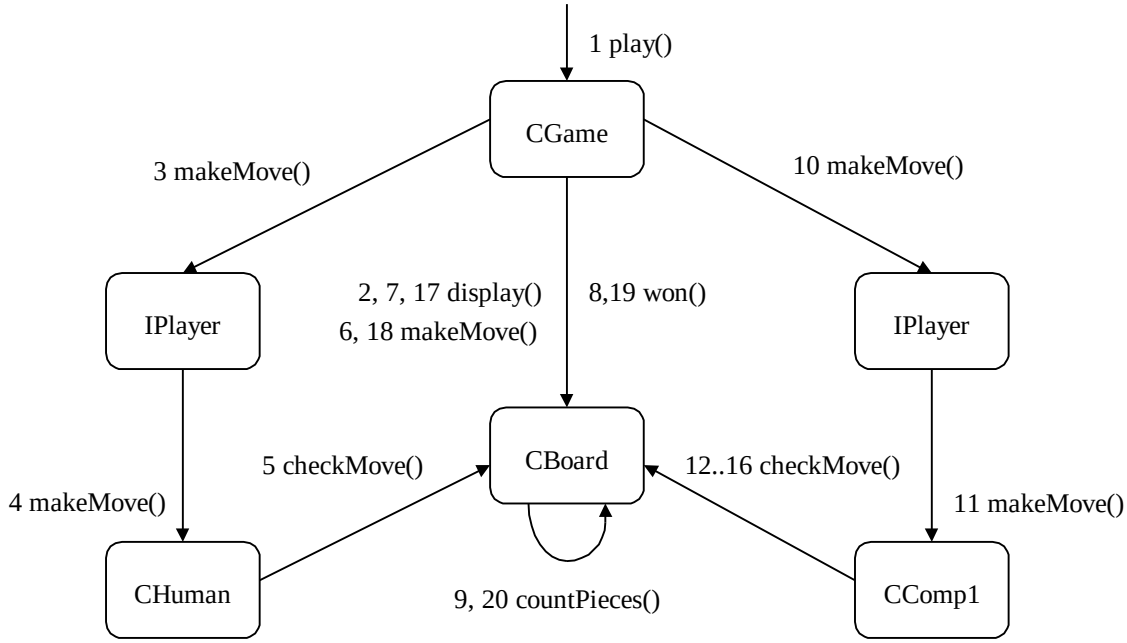



*Figure 1- DOS Version Object Interaction Diagram*

The CGame object now passes the board to the black player as an argument to its makeMove() method. The board is passed as a constant object so that the player is not allowed to change any of its data attributes, i.e. it can only call methods on the board which are also defined as constant. If the player wishes to alter the board they must do so on a copy. CGame also passes an object of type CMove. This simple class consists of two board co-ordinates and a boolean value indicating whether a player wishes to pass. It is now up to the CHuman class to allow the user to select a move. In the example above CHuman checks that the move is valid using checkMove(), a constant method. The CMove is passed by reference so that when the makeMove() method returns it will hold the value of the selected move.

CGame now attempts to make the move on the board by calling its makeMove() method. If the move is legitimate the method alters the pieces on the board and returns TRUE. If FALSE is returned then CGame will call the makeMove() method on the black player again. Assuming the move is legal CGame gets the board to re-display itself, then, providing that the game is not over, it calls makeMove() on the white player. To check that the game has not ended it is necessary to ensure that both players still have some pieces on the board and that the board is not full. Calling won() on the board returns TRUE if the game is over and passes back the colour of the winning player by reference. CBoard calls its own method, countPieces(), to see how many pieces remain on the board. It is also necessary to make sure that the last two moves were not passes, in which case the game would also end.

Here white is played by CComp1, the original dumb computer player. This simply tries each move on the board one by one starting at the top left of the board. For each square it calls checkMove() until it finds a valid move. If no valid move is found then it will pass. Once again CGame plays the move on the board, re-displays it and then checks if the game is over. The process then starts again at step 3 and continues until the game is finally over.

In addition to the methods outlined above CBoard also contains constant methods to check the current status of a board square, to return the move number in the game and to see if pass is a valid move. A sample of the output from the final DOS/UNIX program is given in Appendix B. This uses the final computer player, rather than the simplistic version given above.

A large part of this program structure remained unchanged for the Windows 95 version. The role of the CGame class was taken over by a dialogue box class, COthelloDlg. The human player was however no-longer required. User input was taken directly from the mouse click methods in the dialogue class. A continuously running timer checked periodically whether it was the computers turn to play. This would then read the current skill level settings, construct the computer player object and call its makeMove() method.

## LOOKAHEAD SEARCH

### GAME TREES

The progress of an Othello game can be represented as a tree. The nodes of the game tree represent board situations and the branches how one board configuration is transformed into another i.e. a move. The ply of a game tree, p, is the number of levels of the tree, including the root level. A tree of depth d has p = d + 1.

. This tree can then be searched to provide the most promising move. It is, however, impossible to use an exhaustive search of the game tree. An effective branching factor for Othello was found to be of the order of 7. The effective tree depth is 60, assuming that the majority of games end when the board is full. The number of branches is then $7^{60} = 5\times10^{50}$, far too many to evaluate. If there was an infallible way to rank the members of a set of board situations then it would be a simple matter to select the move which lead to the best situation without using any search. Unfortunately no such ranking procedure is available in Othello.

Another strategy is to analyse the situation after a given number of moves and countermoves. This cannot be pursued too far otherwise the tree size becomes excessive once again. The leaf-node situations can then be compared working on the assumption that the merit of a move clarifies as the move is pursued.

### MINIMAX LOOKAHEAD

We assume that we posses a function (the static evaluator) that will evaluate a board situation into an overall quality number (the static evaluation score). A positive number indicates an advantage to one player (the maximising player), a negative, an advantage to the other (the minimising player). The degree of advantage increases with the absolute value of the number. The maximiser looks for a move that leads to the largest positive number and assumes that the minimiser will try to force play towards situations with negative static evaluations.

The decisions of the maximiser take cognisance of the choices available to the minimiser at the next level down and vice-versa. Eventually the limit of the tree is reached when static evaluation is used to select between alternatives. This scoring information passes up the game tree using a procedure called MINIMAX.

*To perform a MINIMAX search,*

- *If the limit of the search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.*
- *Otherwise, if the level is a minimising level, use MINIMAX on the children of the current position. Report the minimum of these results.*
- *Otherwise, the level is a maximising level. Use MINIMAX on the children of the current position. Report the maximum of the results.*

The whole idea of MINIMAX rests on the translation of the board quality into a single number, the static value. Unfortunately this is a poor summary. MINIMAX can also be expensive as either the generation of paths, or static evaluation, can require a lot of computation.

## ALPHA-BETA PROCEDURE

Fortunately the static evaluator does not have to be used on every leaf node at the bottom of the search tree. The Alpha-Beta procedure can be used to reduce both the number of tree branches that must be generated and the number of static evaluations that must be done.

> *The Alpha-Beta principle:*
>
> *If an idea is definitely bad, do not waste time seeing just how awful it is.*

The Alpha-Beta procedure is so named because it uses two parameters, alpha and beta, to keep track of expectations. Whenever you discover a fact about a given node you check what you know about its ancestor nodes. It may be the case that no further work is required below the parent node. Also, the best you can hope for at the parent node may have to be revised. The Alpha-Beta procedure is started on the root node with alpha set at - ∞ and beta at +∞. The procedure is then called recursively with a narrowing range between the alpha and beta values.

*Alpha-Beta procedure,*

- *If the level is the top level let alpha be -¥ and let beta be +¥.*
- *If the limit of the search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.*
- *If the level is a minimising level,*
  - *Until all the children are examined with Alpha-Beta or until the alpha is equal to or greater than beta,*
    - *Use the Alpha-Beta procedure, with the current alpha and beta value, on a child; note the value reported.*
    - *Compare the value reported with the beta value; if the reported value is smaller, reset beta to the new value.*
  - *Report beta.*
- *Otherwise, the level is a maximising level:*
  - *Until all the children are examined with Alpha-Beta or until the alpha is equal to or greater than beta,*
    - *Use the Alpha-Beta procedure, with the current alpha and beta value, on a child; note the value reported.*
    - *Compare the value reported with the alpha value; if the reported value is larger, reset alpha to the new value.*
  - *Report alpha.*

## ALPHA-BETA PERFORMANCE

In the worst case the branches may be ordered such that the Alpha-Beta procedure does nothing and all $b^d$ nodes must be statically evaluated. If, as earlier suggested, we take $b=7$ and $d=60$ then this would take $5\times10^{50}$ evaluations. If the tree is arranged such that the best move is the leftmost alternative at each node then the moving player has to check each possible move but only the left-most of the opposing player's moves. The number of static evaluations, s, in an optimally arranged tree, is therefore given by:

$$s = b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$$

Using the values above this gives $s = 4.5\times10^{25}$. This is a lower bound on the number of static evaluations required. Unfortunately, the Alpha-Beta procedure does not prevent exponential growth. Also, if there was a way to arrange the tree so that the best move was always on the left then there would be no need to perform the search, we would just pick the leftmost option.

## IMPLEMENTATION

The source code for the method weighting() in CComp, which implements the static evaluation, MINIMAX lookahead and Alpha-Beta pruning, can be found in Appendix C. It can be seen how the board is passed as a constant reference and therefore a copy needs to be made to play the moves on. bestMove is also passed by reference so that the move with the highest static score can be passed back to the calling method at the top level of the search tree. The static evaluation uses the weightings described in the next section along with the methods designed by Alison.

To simplify the code a distinction is not made between minimising and maximising levels. Instead, when weighting() is called on the next level down in the search tree the values of alpha and beta are swapped and negated. This means that on the next level down the opponent is now trying to maximise their score, and therefore when it is returned back to the current level it is negated.

The Alpha-Beta pruning has been supplemented by also exiting when a high score, corresponding to a winning move, is returned. If there are no playable moves then the method checks if the previous move was a pass. If it was then this represents a leaf node on the game tree and an end to the game. A large bias is created towards the winner of the game with an additional weighting based on the difference in the number of pieces that each player has on the board. If it is not the end of the game, the computer simply moves down to the next level having passed.

## GENETIC EVOLUTION

### OTHELLO STRATEGIES

In the previous section we assumed that we had a static evaluator which would provide a quality score for a given board. This was the hardest part of the project and is provided by the game heuristics. Alison decided that the project should implement six types of Othello strategy. These were edge avoidance, maximising frontier pieces, minimising long sequences, maximising pieces, mobility and special squares. The details of these strategies are given in Alison's report and are not important to what follows here. Alison wrote a method for each strategy which, given a board and the colour of the player for whom it is to be evaluated, returns a value in the range 0 to 1. These methods also use the move number to alter the importance of a given strategy during the game.

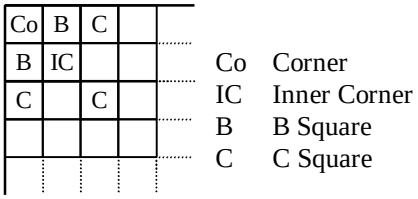



*Figure 2- Special squares*

Here we shall look more deeply at only one of these strategies, implemented in the specialSquare() method. The corner square is of great importance in Othello as once won it cannot be taken by your opponent. However, there are three other types of square, shown in Figure 2, which are commonly recognised as having special significance. For example, if you place a counter on a B square it may enable your opponent to take the valuable corner square.

The initial problem was to obtain weightings for these squares relative to their importance in the game e.g. the corner square should have the highest weighting, but how high? Once these had been obtained we could generate weightings to combine all six strategies into one overall static score. With ten different weightings, even assuming a limited range of values, there are an enormous number of combinations to try. The solution to this problem is given by nature through the process of genetic evolution.

### SURVIVAL OF THE FITTEST

In higher plants and animals each cell contains a single nucleus which in turn contains chromosomes. These chromosomes carry the trait-determining factors known as genes that are passed on when cells divide and offspring are parented. Chromosomes are usually paired, with each parent contributing one chromosome. The pairs are homologous as for each gene in one chromosome there is a corresponding gene in the other chromosome with the same purpose. Cells containing paired, homologous chromosomes are called diploid cells.

In preparation for mating, homologous chromosomes are duplicated and bundled together. The bundling causes cleavage and reconnection, scrambling the genes in a process known as crossover. There are now two complete sets of scrambled chromosome pairs, one ending up in each half when the nucleus, and then the cell, divides.

The cells then divide again but without duplication. One chromosome from each pair ends up in each of the two new cells. These haploid cells are either eggs or sperm. Mating then produces a fertilised diploid egg. Subsequent cell division does not involve crossover.

Occasionally the chromosome copying process goes wrong and a gene which is different from the corresponding gene in the parent occurs. This process is known as mutation. The mutation may produce a more, or less, successful individual, or may make no difference. A mutation in one chromosome of a diploid cell need not be fatal as the other chromosome is usually normal. However, inbreeding can lead to the same defective gene being present in both chromosomes.

In lower plants and animals, chromosomes are only paired briefly during reproduction. Usually there is only one parent whose chromosomes are copied. The original set goes into one of the two new cells and the copy into the other. Occasionally two parents are involved and each contributes a set of chromosomes.

In the *Origin of Species*, published in 1859, Charles Darwin put forward the principle of evolution through natural selection.

- Each individual tends to pass on its traits to its offspring.
- Nevertheless, nature produces offspring with differing traits.
- The fittest individuals - those with the most favourable traits - tend to have more offspring than those with unfavourable traits, driving the population towards those favourable traits.
- Over long periods, variations accumulate producing entirely new species whose traits make them especially suited to particular ecological niches.

Natural selection is enabled by the variation that follows from crossover and mutation. Crossover assembles existing genes into new combinations. Mutation produces new genes, hitherto unseen.

## GENETIC ALGORITHMS

The computer equivalent of natural selection, a genetic algorithms, is surprisingly easy to implement especially using an Object Oriented approach. The population, individuals and chromosomes are the obvious choices for objects. My initial version of the genetic algorithm was completely divorced from the game of Othello. It was simply designed to traverse two dimensions of a three-dimensional space. These two dimensions were the gene values and the third was the corresponding quality score. The shape of the space was chosen as a mountain with a moat around it, designed to test all the features of the algorithm.

## STANDARD METHOD

The first possible definition of fitness uses the standard method. The fitness of an individual is the probability that it survives to the next generation. The fitness of the ith individual, $f_i$, can be related to the quality of that individual, $q_i$:

$$f_i = \frac{q_i}{\sum_j q_j}$$

*Natural selection is now mimicked as follows,*

- *Create an initial population containing one individual.*
- *Mutate one or more genes in one or more of the current chromosomes producing one new offspring for each chromosome mutated.*
- *Mate one or more pairs of chromosomes.*
- *Add the mutated and offspring chromosomes to the current population.*
- *Create a new generation by keeping the best individual of the current population along with other individuals selected randomly from the current population. The random selection is biased according to the assessed fitness.*

It was decided to allow a maximum of four individuals to survive from one generation to the next. If the number is too low, all the individuals have the same traits but if the number is too high, computation time is excessive. Each of these survivors is a candidate for survival to the next generation along with any newcomers.

Following the pattern of lower plants and animals each individual only has one chromosome. To mimic mutation, one of the genes is selected at random and then 1 is randomly either subtracted or added, taking care to stay within the range of acceptable values. If the mutation rate was made smaller, new traits appeared too slowly. If the rate was much larger each generation was unrelated to the previous. If the mutant differed from all the candidates accumulated so far then it was added to the population.

The three-dimensional space was used to look up the quality score for each individual. The chromosome with the highest score survives to the next generation. The remaining survivors are selected at random according to the standard method. The initial implementation did not have crossover but without this the zero quality moat in the search space could not be traversed. Therefore, if the initial individual was the wrong side of the moat, the peak of the mountain was not reached.

## CROSSOVER

In addition to dealing with moats, crossover can also unite individuals that are doing well in different genes. This assumes that the global maximum can be searched for by finding the maximum in each dimension independently. Crossover reduces the dimensionality of the search space.

Crossover was implemented by considering the individuals that survived from the previous generation. For each individual a mate was selected at random from the other survivors using the standard method. The offspring was then created by randomly selecting each gene from one or other of the parents. If the offspring was different from any candidates accumulated so far it was added to the candidates.

This still performs poorly as individuals in the moat which may contribute favourably to crossover are not chosen as they have zero probability using the standard method.

### RANK METHOD

The standard method offers no way to influence the selection procedure. The rank method allows a control on the bias towards the best chromosome and avoids implicit biases introduced by a given measurement scale.

> *To select a candidate by the rank method,*
>
> - *Sort the n individuals by quality.*
> - *Let the probability of selecting the ith candidate, given that i-1 candidates have been selected, be p, except for the final candidate, which is selected if no previous candidate has been selected.*
> - *Select a candidate using the computed probabilities.*

The rank method was therefore used to replace the standard method. On the basis of trial and error, a value of two-thirds was chosen for p. It was now possible to tunnel through the zero quality moat as the lowest fitness is determined by quality rank rather than by quality score.

### DIVERSITY

Diversity is the degree to which common individuals exhibit different genes. As the algorithm stood individuals tended to get wiped out if they scored just a bit lower than a chromosome close to the best current chromosome. This resulted in uniformity throughout even a large population. In nature unfit-looking individuals survive well in ecological niches outside the view of other, relatively fit-looking individuals.

> *The diversity principle:*
>
> *It can be as good to be different as it is to be fit.*

One way to measure the diversity that would be contributed by an individual, if selected, is to sum the inverse squared distances between that individual and the individuals already selected.

$$\text{Diversity} = \sum_i \frac{1}{d_i^{\,2}}$$

Now the rank method is used on the sum of the quality and diversity rank giving the rank space method. Individuals which are both high quality, and diverse, score highly. Diversity also overcomes the problem of local maxima. Some of the individuals will populate a maximum driving off other individuals which will then find other local maxima, one of which will also be the global maximum.

## APPLICATION TO OTHELLO

In the case of Othello each individual represents a different computer player. The original problem was divided into two. Firstly, only the specialSquare() method was used and the genes represented the weightings of the squares. Once these had been evaluated for each of the lookahead levels they were fixed and the weightings for the strategies were evolved. The hardest problem was deriving a quality score for each computer player. This is best explained through an example.

The OID in Figure 3 shows the process of evolution from one generation to the next in a population where, for simplification, the number of survivors is limited to two. We will assume that we already have two individuals in the population but in reality the first is entered by the user and the second mutated from it. To set the process in motion the application calls the evolve() method on CPopulation. The initial population is then displayed.

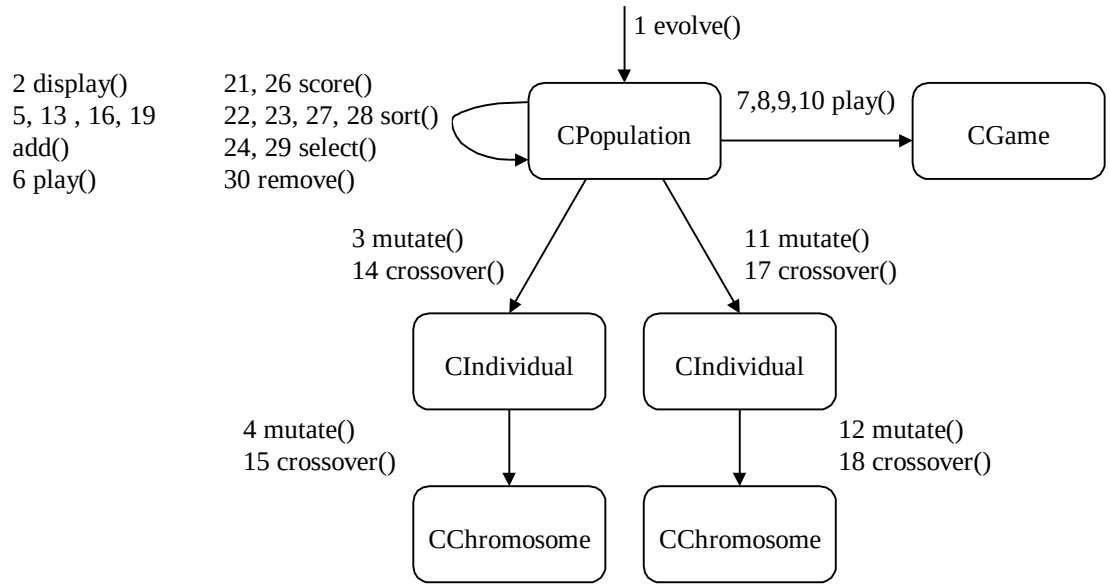



*Figure 3- Genetic Evolution Object Interaction Diagram*

The first process to take place is that of mutation. The mutate() method is called on each individual in the population one by one. They in turn call the mutate() method on their chromosomes. This has the effect of duplicating the chromosome but altering one of the genes by ± 1. This is then passed backed to CPopulation which creates a new individual using that chromosome. The add() method is then called with this new individual as a parameter. If the individual is different from those already in the population then it is added to it.

The new individual is then played against the individuals currently in the group with the results being stored in a table. Each individual gets to play as both black and white as, in general, black has an advantage by starting first. Each gene of the chromosome is used as one of the weightings for the strategies. One of the weightings is kept constant and the others vary in proportion to it.

This continues until all the original individuals have been mutated. In the example above, the first mutant was new to the population and was played against each of the other two individuals twice. The second mutant was already present in the population and was therefore discarded.

The next process to take place is crossover. Each individual from the initial population is passed one other individual from the initial population as a parameter for the method crossover(). This second individual is chosen at random but biased by the rank fitness of the individuals (this information is carried in from the previous evolution). In the case above, as there were only two individuals they must be crossed with each other. Once again this is then added to the population, unless, as in this example, it already exists.

Now the selection process can take place, started by calling the rank() method. Using the table of results each individual is given a quality score on the basis of 3 for each win and 1 for each draw. The individuals are then sorted on this quality score to give the quality rank. At this stage the diversity is set to zero as no individuals have been selected. The total rank is therefore the same as the quality rank. The rank fitness can now be calculated.

The first to be picked is the fittest individual. Calling the select() method marks this individual as selected so it is excluded from further selection and ranking. The rankings must now be re-calculated. This time the diversity can be calculated as one individual has already been chosen. The next individual is then selected randomly biased by the rank fitness. This continues until all the survivors into the next generation have been chosen. The remove() method is then called to remove from the population those individuals which did not survive. This process re-adjusts the table of game results so that results of games between the survivors are carried forward into the next round of evolution and do not have to be replayed.

## RESULTS

Appendix D contains a sample of the actual output from the program along with comments to highlight further the points made above. The tables below show the final weightings that were evolved for each of the special squares and then for each strategy. The algorithm was run on the departmental workstations. The size of the mutations were slowly decreased over a period of a week to home in on the optimal weightings given below.

*Table 1- Weightings for special squares*

| | Number of Lookaheads | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| Corner | 63 | 58 | 57 | 58 | 60 |
| Inner Corner | 11 | 15 | 16 | 15 | 16 |
| B Square | 2 | 2 | 2 | 2 | 2 |
| C Square | 10 | 11 | 12 | 10 | 12 |

*Table 2 - Weightings for strategies*

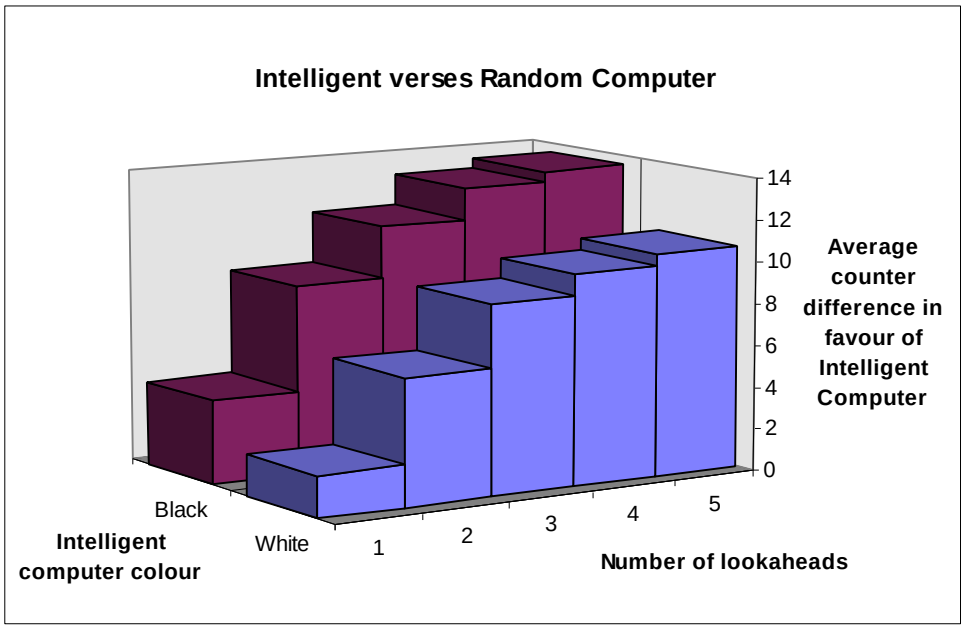
| | Number of Lookaheads | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| Edge Avoidance | 1 | 1 | 1 | 1 | 1 |
| Frontier | 3 | 3 | 3 | 3 | 3 |
| Long Sequence | 10 | 10 | 10 | 10 | 10 |
| Mobility | 18 | 16 | 18 | 19 | 19 |
| Special Square | 1 | 1 | 1 | 1 | 1 |
| Maximum Disc | 10 | 10 | 11 | 11 | 11 |

## CONCLUSION

In conclusion I would judge the project a success. All three main aims of the project were met. The choice of Visual C++ as a programming environment proved very workable and the Object Oriented aspects of C++ were ideal for both division of the project and creating the overall structure of the program. This meant that the initial program implementing the Othello rules could be built upon without significant difficulty and remained largely unchanged throughout the project development.

The computer player itself also appears very successful. It was tested on people ranging from complete beginners to those who have played the game for some years. Each could find a skill level at which they could beat the computer and another at which the computer would win the majority of games.

It is, however, very difficult to quantify the degree of success. One method I tried was to play the intelligent computer player against another computer player which selected moves at random. The results of these games are shown in the graph below. Three things stand out. Firstly, the large advantage enjoyed by black in starting the game. Secondly, that the margin by which the intelligent player won is not that wide and, indeed, some of the games it lost. Thirdly, that this margin does not keep increasing with the number of lookaheads but appears to level off. These latter two characteristic highlight a flaw in the intelligent computer player, namely that it assumes that its opposition thinks like it does. The MINIMAX procedure works on the basis that the other player will always choose the best move available to them. This means that searching to additional depth doesn t yield the benefits it should when the opposition is playing randomly.




The user interface was highly successful. The power of Visual C++ allowing us to create a front-end conforming to the Windows 95 conventions including a useful help feature. One important change that needs to be made is to make the computer player multi-threaded as currently the user cannot change any options, nor will the window redraw, while the computer is thinking.

The idea that the players should not be trusted to make only one move on the board and therefore that all moves should be made by CGame was overly complicated. In hindsight it should be possible to rely on the classes implementing the players to stick to the rules. For example, the CHuman class should check that the user input is valid, as indeed it already does. The initial program was written using C++ conventions I learnt during an IT internship and unfortunately these were not compatible with those used by Visual C++, thereby creating additional work.

With particular reference to the lookahead procedure there are a number of improvements I would like to make. If a time limit is to be added to the game (as would be used in tournaments) then the procedure needs to implement progressive deepening. Instead of searching to the bottom of one branch and then the next, all possibilities should be searched to depth 1, then to depth 2 etc. until time runs out.

The lookahead is also prone to the horizon effect as it always searches to a constant depth in the game tree. It is possible that vital information may lie just beyond that horizon. One way round this problem is to use singular heuristic extension whereby if one move's static value stands out from all the others, indicating the rise before a fall, then the search should continue deeper.

The search could also be limited by using heuristic pruning whereby the branching factor is varied with depth of penetration. A tapered search can be used to direct effort towards more promising moves. This method ranks the children of a node in order of quality. The number of branches searched from a child node is then the number of branches its parent had minus its ranking. However, any limit on the search that prevents seemingly bad moves will not find those which later turn out to be spectacular.

The genetic evolution procedure had a number of minor problems. The time taken to home in on the correct weightings was prohibitive especially when the lookahead increased above two. This was mainly due to the time consumed obtaining the quality score. What is really required is a more objective approach to quality which does not involve playing every individual against every other. One possibility is to only play games against survivors from the last generation. This reduces the maximum number of games that may need to be played in one generation from 112 to 64. However, this still doesn't solve the problem that one individual may appear better than another at playing a third but when the third individual is changed then the second individual may triumph.

It would also have been useful if the algorithm automatically altered the size of the mutations. Initially the mutations should have been large to quickly get into the region of the optimal solution. Then as the same individual continues to survive from one generation to the next the size of the mutations should be decreased to home in on a more accurate solution. Finally, the genetic algorithm was not aided by the random opening game facility that was added by Alison.

Finally, in the future I would like to convert the project to Java. This has all the object oriented benefits of C++ and the project management facilities of Visual C++ can be obtained by using Visual J++. In addition, techniques such as multi-threading are easy to achieve. Most importantly Java is platform independent and therefore the final version, complete with graphic interface, could be run on any platform. On top of this the project can be written as both an application and applet in one. The latter would enable the game to appear on the World Wide Web and if combined with a form would allow direct feedback from players.

## BIBLIOGRAPHY

### BOOKS

The following list is only small selection of the books on the subjects involved in this project, but are those which proved to be the most useful.


Gupton, G M - "Genetic Learning Algorithm, Applied to the Game of Othello", Heuristic Programming in Artificial Intelligence, the first computer Olympiad, 1989, pp. 241-254

Holzgang, D A - "Teach yourself... Visual C++", MIS: Press, 1996

Knuth, D E & Moore, R W - "An Analysis of Alpha-Beta Pruning", Artificial Intelligence 6(4), North Holland Publishing Company, 1975, pp. 293 - 326

Lee, K & Mahajan, S - "The Development of a World Class Othello Program", Artificial Intelligence 43 (1990), pp. 21 - 36

Marsland, T A - "Relative Efficiency of Alpha-Beta Implementations", Proceedings of the 8th International Joint Conference on Artificial Intelligence, Vol. 2, Karlsruhe, Germany, 1983, pp. 763 - 766

Moriarty, D E & Miikkulainen, R - "Evolving Neural Networks to Focus Minimax Search" Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94) Powley, Curtis Nelson

Plaat, A, Schaeffer, J, Pijls, W & de Bruin, A - "New Paradigm for Minimax Search" Technical Report TR 94-18, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada, Dec. 1994

Rosenbloom, P S - "A World-Championship-Level Othello Program", in Levy, D, Computer Games, Vol. II, Springer 1988, pp. 365 - 405.

Winston, P H - "Artificial Intelligence", Addison-Wesley, 1992


### WORLD WIDE WEB RESOURCES

The World Wide Web hosts a vast pool of information on the subjects of Othello and computer gaming. The following select list of URLs provide an introduction to the game, its strategies and some of the algorithms currently in use.


Guide To The Game Of Othello - http://www.armory.com/~iioa/othguide/resources/index.html

Beginner's Guide to Othello Strategy - http://www.maths.nott.ac.uk/othello/beginners.html

Logistello - http://www.neci.nj.nec.com/homepages/mic/log.html

Desdemona - http://www.math.hmc.edu/~dmazzoni/cgi-bin/desmain.cgi

Inside Othello - http://lglwww.epfl.ch/~wolf/java/html/othello_desc.html

MAGOO - http://www.science.ubc.ca/departments/sci1/homepages/peter/magoo.html

Iago - http://www.cs.virginia.edu/~dae4e/java/jothello/writeup.html

Various theses on the subject of Othello - ftp://ftp.uni-paderborn.de/unix/othello/ps-files/

Simple Genetic Algorithms - http://aif.wu-wien.ac.at/~geyers/archive/ga/sga/sga/sga.html

## APPENDIX A - RULES OF OTHELLO

The game of Othello is played on an eight by eight board using sixty-four double sided pieces, white on one side and black on the other.

1. The players decide who is to be white and who is to be black. The same colours are kept throughout the game.

2. The initial board configuration is as shown in Figure 4.

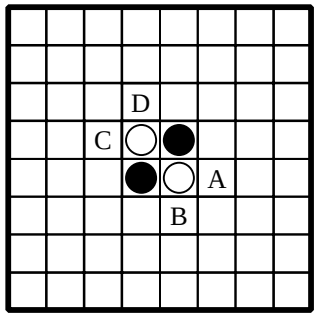



*Figure 4*

3. Black makes the first move.

4. On every move a disc must be placed next to an opponent's disc, either sideways, lengthways or diagonally. The disc placed **must trap** an opponent's disc (or discs) between the one placed and one already on the board in any direction.

In Figure 1, A, B, C and D, represent the moves available to black.

If black decides to place his disc in position A, he has captured the white disc E shown in Figure 5 and turns it over so that it is black side up.

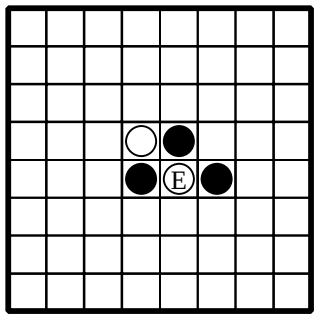



*Figure 5*

5. Now it is white's turn to move. Figure 6 shows the moves now available.

6. Players now continue to take turns.

- The number of discs trapped is not limited to one.
- A disc or discs can only be captured as a direct result of a move.
- If it is impossible to capture one of your opponent's pieces when it is your move then you miss that turn.

7. For example, black placing a piece at position A in Figure 7 results in pieces being captured in all 8 directions, however, the piece at position B is not captured.

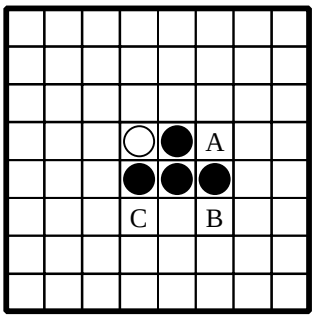



*Figure 6*

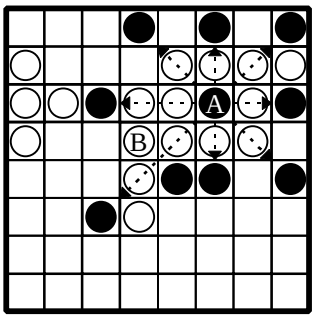



*Figure 7*

8. Play continues until one of the following occurs.

- The board is filled.
- One player has no pieces of his colour left on the board.
- No further moves are possible for either player.

9. At this time the discs are counted and the player with the most pieces of his colour on the board is declared the winner.

## APPENDIX B - SAMPLE DOS OUTPUT

The following output is from the final DOS/UNIX version including the same computer player as used in the Windows version.

```
Enter player type:
1. Human
2. Computer player
1

Enter player type:
1. Human
2. Computer player
2
Enter number of lookaheads: 2

 12345678
1........
2........
3........
4...XO...
5...OX...
6........
7........
8........

Enter x co-ordinate (0 to pass):
5
Enter y co-ordinate:
3

 12345678
1........
2........
3....X...
4...XX...
5...OX...
6........
7........
8........

I move (6, 3)

 12345678
1........
2........
3....XO..
4...XO...
5...OX...
6........
7........
8........
```

## APPENDIX C - MINIMAX CODE

The following code implements the static evaluation, MINIMAX lookahead strategy and Alpha-Beta pruning. A typical call to the function would be of the following form.

weighting(0, board, m_colourColour, FALSE, -INFINITY, INFINITY, move)

This starts the algorithm at level zero (so that at least one lookahead will be performed) with alpha set to -∞ and beta to +∞, where infinity is approximated by a large integer.

```
double CComp::weighting(UINT lookAhead, const CBoard& board, EColour player,
                        BOOL previousPass, double alpha, double beta, CMove& bestMove)
{
  EColour opponent = (player == BLACK) ? WHITE : BLACK;

  // If this is the last lookahead return the board weighting (static evaluation)
  if (lookAhead == m_nLookAhead)
  {
    // Return static evaluation score using weighted sum of strategies
    return (m_lFrontierWeighting*(frontier(board, player)-frontier(board, opponent)))
      + (m_lLongSequenceWeighting*(longSequence(board, player)-longSequence(board,
opponent)))
      + (m_lMaxDiscWeighting*(maxDisc(board, player)-maxDisc(board, opponent)))
      + (m_lAvoidEdgeWeighting*(avoidEdge(board, player)-avoidEdge(board, opponent)))
      + (m_lSpecialSquareWeighting*(specialSquare(board, player)-specialSquare(board, opponent)))
      + (m_lMobilityWeighting*(mobility(board, player)-mobility(board, opponent)));
  }

  UINT playableMoves = 0;
  double weight = 0;
  CMove move;
  move.pass = FALSE;

  // At base of search tree set alpha to minus infinity
  if (lookAhead == (m_nLookAhead - 1))
  {
   alpha = -INFINITY;
  }

  // Search each board square
  for (move.x=1;move.x<9; move.x++)
  {
    for (move.y=1;move.y<9;move.y++)
    {
      CBoard tempBoard(board);

      // See if move is possible by playing it on a temporary board
      if (tempBoard.makeMove(move, player))
      {
        playableMoves++;

          // Perform lookahead on this temporary board swapping and negating alpha and beta
          weight = -weighting(lookAhead + 1, tempBoard, opponent, FALSE, -beta, -alpha,
bestMove);
```

```
        // Check if best move so far
        if (weight > alpha)
          {
          // At top of tree record move
          if (lookAhead == 0)
          {
            bestMove.x = move.x;
            bestMove.y = move.y;
            bestMove.pass = FALSE;
          }

          alpha = weight;

          // Alpha-beta pruning (or winning move)
          if ((alpha >= beta) || (weight >= 8003))
          {
            return weight;
          }
          }
      }
    }
  }

  // If no available moves
  if (playableMoves == 0)
  {
    // At top of tree record a pass
    if (lookAhead == 0)
    {
      bestMove.pass = TRUE;
    }

    // Check if both players have passed
    if (previousPass)
    {
      // Score is a large bias towards the winner plus the difference in the number of pieces
      UINT blackPieces, whitePieces, difference;
      board.countPieces(blackPieces, whitePieces);
      difference = (player == BLACK) ? (blackPieces - whitePieces) : (whitePieces - blackPieces);
      return difference + ((difference > 0) ? 8000 : -8000);
    }
    else
    {
      // Lookahead having passed
      alpha = -weighting(lookAhead + 1, board, opponent, TRUE, -beta, -alpha, bestMove);
    }
  }

  return alpha;
}
```

## APPENDIX D - SAMPLE GENETIC ALGORITHM OUTPUT

The following output is taken from a sample run of the evolution program with the verbose output option set. Additional comments have been added to clarify the process and the formatting has been changed to make it more suitable for the A4 page.

```
Select
(1) Evolve special squares
(2) Evolve weightings
1
```

Having selected that we wish to evolve the weightings for the special squares rather than the strategy weightings we must then enter the initial data. All individuals in the population do the same number of lookaheads, selected to be two here. There are four squares whose weightings we wish to decide. One of these has its value set at a constant level throughout the evolution and the other three are entered as genes below. This initial individual is then added to the population.

```
Input lookahead : 2
Input gene 1 : 5
Input gene 2 : 5
Input gene 3 : 5
{5,5,5}, added to population

GENERATION : 1
```

The quality is calculated by playing each individual against all the other individuals (including itself!). It plays as both black and then white, scoring three for a win and one for a draw. The diversity rank is set to zero as no selection has taken place.

```
Initial population
  {5,5,5} quality 3, quality rank 1, diversity 0, diversity rank 0, total rank 1, rank fitness 1
```

The initial population is first mutated. Here the second gene has been randomly selected and one has been randomly subtracted. No crossover takes place as there is only one individual.

```
Mutation
  {5,5,5} mutates to {5,4,5}, added to population
```

As the new individual is added to the population it is played against all the current individuals and all the quality scores are updated. Here it can be seen that the new addition wins both games against the opposition and one of the games against itself, giving a quality score of 9. The fraction for the rank fitness calculation can be seen to be two-thirds.

```
New population
  {5,5,5} quality 6, quality rank 2, diversity 0, diversity rank 0, total rank 2, rank fitness 0.333
  {5,4,5} quality 9, quality rank 1, diversity 0, diversity rank 0, total rank 1, rank fitness 0.667
```

As there are less than four individuals in the population all of them survive into the next generation.

```
All survive
```

GENERATION : 2

Although the results of all games are not stored they are carried through from one generation to the next so no more games need to be played before the next generation can begin to evolve.

Initial population
{5,5,5} quality 6, quality rank 2, diversity 0, diversity rank 0, total rank 2, rank fitness 0.333
{5,4,5} quality 9, quality rank 1, diversity 0, diversity rank 0, total rank 1, rank fitness 0.667

Mutation
{5,5,5} mutates to {4,5,5}, added to population
{5,4,5} mutates to {5,5,5}, already exists

There are now two individuals in the initial population so crossover can take place. The individuals created are however already members of the population so are not added.

Crossover
{5,5,5} crossed with {5,4,5} to get {5,5,5}, already exists
{5,4,5} crossed with {5,5,5} to get {5,5,5}, already exists

New population
{5,5,5} quality 6, quality rank 3, diversity 0, diversity rank 0, total rank 3, rank fitness 0.110889
{5,4,5} quality 12, quality rank 1, diversity 0, diversity rank 0, total rank 1, rank fitness 0.667
{4,5,5} quality 12, quality rank 2, diversity 0, diversity rank 0, total rank 2, rank fitness 0.222111

All survive

GENERATION : 3

Initial population
{5,5,5} quality 6, quality rank 3, diversity 0, diversity rank 0, total rank 3, rank fitness 0.110889
{5,4,5} quality 12, quality rank 1, diversity 0, diversity rank 0, total rank 1, rank fitness 0.667
{4,5,5} quality 12, quality rank 2, diversity 0, diversity rank 0, total rank 2, rank fitness 0.222111

Mutation
{5,5,5} mutates to {5,5,6}, added to population
{5,4,5} mutates to {6,4,5}, added to population
{4,5,5} mutates to {4,4,5}, added to population

Crossover
{5,5,5} crossed with {5,4,5} to get {5,4,5}, already exists
{5,4,5} crossed with {4,5,5} to get {5,5,5}, already exists
{4,5,5} crossed with {5,4,5} to get {4,4,5}, already exists

New population
{5,5,5} quality 15, quality rank 4, diversity 0, diversity rank 0, total rank 4, rank fitness 0.0246297
{5,4,5} quality 18, quality rank 3, diversity 0, diversity rank 0, total rank 3, rank fitness 0.073963
{4,5,5} quality 24, quality rank 1, diversity 0, diversity rank 0, total rank 1, rank fitness 0.667
{5,5,6} quality 15, quality rank 5, diversity 0, diversity rank 0, total rank 5, rank fitness 0.00820168
{6,4,5} quality 21, quality rank 2, diversity 0, diversity rank 0, total rank 2, rank fitness 0.222111
{4,4,5} quality 12, quality rank 6, diversity 0, diversity rank 0, total rank 6, rank fitness 0.00409469

There are now more than four individuals in the population so not all can survive into the next generation. The individual with the highest quality ranking goes through automatically.

Fittest survives : {4,5,5}

The diversity from the one already selected for those remaining in the population can now be calculated. The next survivor is now chosen randomly but biased according to the rank fitness. It is an unfortunate coincidence that in this case the individual with the highest rank fitness has always been chosen. It should be noted though that this is not always an individual with the highest quality when diversity comes into play. This continues until all four survivors for the next generation have been chosen.

Remaining population
{5,5,5} quality 15, quality rank 2, diversity 1, diversity rank 1, total rank 3, rank fitness 0.667
{5,4,5} quality 15, quality rank 3, diversity 0.5, diversity rank 3, total rank 6, rank fitness 0.073963
{5,5,6} quality 12, quality rank 4, diversity 0.5, diversity rank 4, total rank 8, rank fitness 0.0122964
{6,4,5} quality 18, quality rank 1, diversity 0.2, diversity rank 5, total rank 6, rank fitness 0.222111
{4,4,5} quality 12, quality rank 5, diversity 1, diversity rank 2, total rank 7, rank fitness 0.0246297

Next survivor : {5,5,5}

Remaining population
{5,4,5} quality 12, quality rank 1, diversity 1.5, diversity rank 1, total rank 2, rank fitness 0.667
{5,5,6} quality 9, quality rank 4, diversity 1.5, diversity rank 2, total rank 6, rank fitness 0.036926
{6,4,5} quality 12, quality rank 2, diversity 0.7, diversity rank 4, total rank 6, rank fitness 0.073963
{4,4,5} quality 12, quality rank 3, diversity 1.5, diversity rank 3, total rank 6, rank fitness 0.222111

Next survivor : {5,4,5}

Remaining population
{5,5,6} quality 6, quality rank 2, diversity 2, diversity rank 2, total rank 4, rank fitness 0.110889
{6,4,5} quality 9, quality rank 1, diversity 1.7, diversity rank 3, total rank 4, rank fitness 0.222111
{4,4,5} quality 6, quality rank 3, diversity 2.5, diversity rank 1, total rank 4, rank fitness 0.667

Next survivor : {4,4,5}

GENERATION : 4

Initial population
{5,5,5} quality 12, quality rank 2, diversity 0, diversity rank 0, total rank 2, rank fitness 0.222111
{5,4,5} quality 12, quality rank 3, diversity 0, diversity rank 0, total rank 3, rank fitness 0.073963
{4,5,5} quality 18, quality rank 1, diversity 0, diversity rank 0, total rank 1, rank fitness 0.667
{4,4,5} quality 6, quality rank 4, diversity 0, diversity rank 0, total rank 4, rank fitness 0.036926

This process is repeated until the fittest individual surviving from one generation to the next remains constant.

## APPENDIX E - SCREEN SHOTS

The following screen shots from the completed application show the first few moves and endgame position. The computer is playing white on skill level 5 and the human playing black. The game ends in a convincing victory for the computer.

