CS1006 Tutor: Ian Miguel Student ID'S: 090010763 + 00000000

Project: Othello

Background:

History:

Competition:

The very first tournament of Othello between computers was held in 1979. This was 8 years after the game Othello had been defined from the change of the rules of the game reversi (even though these were only slight changes). Following this, in 1980, the first competition of human versus AI's in the Othello world took place. Peter Frey, from Northwestern University, organized this. The result of this tournament was a victory by the world champion Hiroshi Inouie. Although no computer won the competition it did shave a significant outcome. The AI, "The Moor" beat the champion in one of the games. This marked the first victory of the game for an AI against top skilled players.
Over the next decade the development of Othello machines became popular, with countries such as the UK, France, the Netherlands and the US approaching the challenge. By 1989 the AI's began to dominate the AI versus human events, 3 of the 4 best players of all time were defeated by AI's. In 2 events that took place in 1992 and 1994 it was clear that the AI's had taken over from humans at the game Othello. The machines were winning at ratios of up to 6 to 1. The AI LOGISTELLO, created in 1997, set the final statement of AI's dominance by beating the current world champion 6 games to zero in a match of six timed games. (History Cit)

Modern Research:

AI's development in more modern research is targeting to increasing the move quality of the subject in a limited time frame for moving. This had led to research into the development of smarter evaluation functions, open book algorithms and more selective searching.
The evaluation functions are used to map the different game position into a value, which would give an estimation of the change of winning with this move. The value becomes a higher quality the further into the game the AI can predict, in other words, how far down the game-tree of moves it can look. To achieve this program will use techniques such as alpha-beta pruning to reduce the amount of unnecessary searches, optimizing the performance of the search. (See alpha-beta pruning below).
Open book algorithms are used to have preset knowledge of the beginnings and ending of different games. There are some movements at the beginning of the game whereby the AI can achieve the best results for later in the game, which can be mapped and saved without repeating searches and with relatively low

memory storage. This same principle applies to the end of the game. When approaching the end of the game there are few moves left and the entirety of the game tree can be seen. These outcomes can be stored so that the perfect end game can be seen, so in certain situations the AI will know if it can win or lose. History Cit:)

Min Max Algorithm:

In a game tree we search the outcome of both players moves. It consists of all the possibilities of each move within the game. As each outcome is set a certain weighting or value such in terms of the 'quality' or preferred result in the game state we can analyze these values to find the outcome we desire. The min-max algorithm searches down the tree and when it is the players move it takes the maximum value or best result from this section. For each level down it switches to searching for the mi or max, as the next level would be the opponents turn. So the opponent is going to play the move most harmful to the player or beneficial to himself. Therefore the min value is the value given to the least benefit to the player but most to his opponent. This is assuming the opponent is intelligent to choose the move most beneficial to himself.
(Min-Max algorithm Cit)

Alpha-Beta Pruning:

When using a mini-max algorithm, we are analyzing a range of different values. Searching for either the maximum or the minimum. The process of alpha-beta pruning is used to cut out or not calculate area's where a value will be returned within the current max or min, knowing that it will never be taken into consideration. It is used to optimize the decision tree algorithms to allow faster processing times making it possible to go deeper within a set time frame.
(Alpha-Beta Pruning Cit)

Aim:

To create the game Othello (reversi).

There are 3 main aims within this project: firstly to implement a GUI in which 2 players can play the game, allowing only legal moves.

Secondly to implement an AI which plays the game automatically against another human or AI.

Finally to, using the Java RMI (Remote Method Invocation), implement networking so that moves can be sent to a remote machine. This has to conform to the standard set by the given code. This must also support all combinations of player between human and AI players.

Specification:

Basic:

- Create the game Othello with the rules stated in the lecture.
- Ability to play the game between 2 local humans.
- Created by completing the given othelloModel package.
- Has A GUI
  (Remote interface package not necessary)

Intermediate:

(In addition to basic)

- A simple AI player
- The ability to play over the network using the remote Interface package, for both the human and AI players.
- A button to play the Ai moves so that the game doesn't happen too fast to observe.

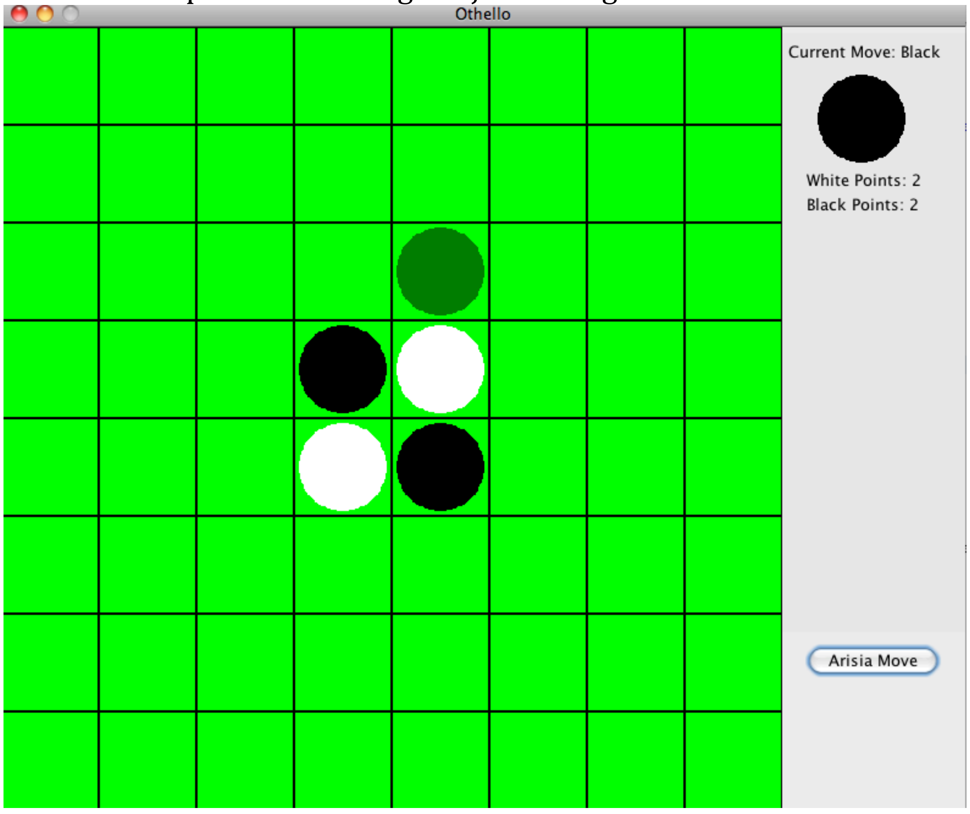
Advanced:

(In addition to intermediate)

- A smarter AI which looks further than one move ahead based on the mini-max algorithm
- Anything more beyond thisStructure:

GUI:

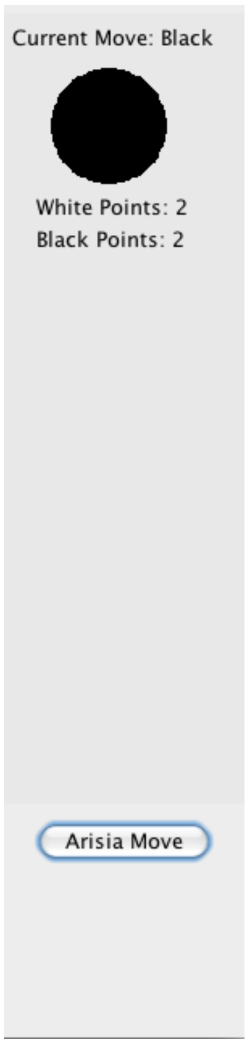
The GUI is implemented using the java Swing and AWT classes.



The main interface consists of 3 main components.

Firstly is the board area. This is an 8 by 8 grid of the Othello game. The board is where the player can make moves by clicking on the desired square. This converts the location of a click into an integer value, which corresponds to a relative place inside a 2d array.

When the user moves the mouse across the board it is refreshed with every move. This allows the displaying of a transparent piece of the current colors turn to be shown. A red cross is displayed on top of this if it is an illegal move showing the user that they cannot make this move. The constant refreshing and display of the transparent piece gives the user updated feedback of his input giving the user more information about what he is doing. Along with the red cross for illegal moves it gives the player a better experience using the game, since he will understand if he cannot make moves or not.



The second main part of the interface is the stats panel. Here information of the current game state is displayed. It tells the user the current move's color whether it is black or white and displays the image of the current move color as well. Underneath this is the amount of pieces of each color that are on the board in the games current state. This is updated when a move occurs so the points and current move are always up to date. The display of the image is useful as if you change the piece displayed to a different picture this one will change as well, so it can be customized to a different look from the simple black and white.

At the button of this panel there is the AI button, this will call the AI to play the next move. The AI will automatically play the move and the Stats and the board will both be updated.

A feature of the GUI that has been implemented is animation. When a piece is played the state of the board previously is compared with the new state. If there has been a change it decides whether a flip is necessary and if it is, how it must be carried out (E.g. if a flip from white to black or from black to white is necessary). The animation occurs by reducing the size of the image from its full width to 0 then the new piece starting from zero to its full width. Between every pixel change it sleep for a set amount of milliseconds. This is useful, as it does not allow the user to click any more so they have to wait until the animation is complete. The flips occur one after the other, in a spiral sequence based from an algorithm inside the update method.

AI:

The basis of our AI is using the min max algorithm. This effectively finds the best worst-case scenario over a number of moves ahead of the current game state. To make this possible we have implemented a recursive tree search to find out all of the possible game states at the end of x turns. The recursion involves a node (game state after a possible move), which then creates new nodes for all of the possible moves from that node (and of course, each of those nodes will then create their own set of nodes). Recursion needs to have a base case which enables to search to end and to not continue infinitely; If the computing capabilities were within reach, then ideally the base case would be a game over state, however as we do not have access to super computers and/or the ability to live hundreds of years, this is not possible due to the sheer number of possible game states within Othello. Due to this restriction, the base case we have chosen is simple a limit on the depth that the search goes. We have implemented this limit as a simple count, which is passed down the branches and decreased by one each depth. When this count reaches zero, the new branches will stop going any further, and the current node will become a leaf. Note that as the game draws close to an end, it is feasible to calculate all of the remaining moves, in this case both the count and the game being over are used as the base case for the recursion.

Min max works by going to the leafs of the tree, then if its your turn, find the best move. It then goes up a level and picks the worst of those options (assuming the player you benefits when you lose out) and then goes up a level and then picks the best case for you. This is where it gets the name min max. For this to work there needs to be a way of scoring the game state after a move. A simple way of doing this is to simply add up the number of pieces that max has on the board. We thought that this was a bit primitive, so we decided to delve a bit further into the game state and make the scoring take into account the position on the board. We have done this by implanting a 2D array of 'weightings', which are associated with each square. If a square is worth having (for example a corner piece) the square is given a weighting >1. If the square is not worth having, or a bad square, such as the squares adjacent to the corner square, then that square is given a weighting less than 1. The score of the game state at this point is then multiplied by this weighting, given a new score associated with that particular game state.

We also decided that if a move leads to a the opponent forfeiting a move, then this move should be taken as at worst, it simply givens the player an extra piece and allows the AI to think again a pick a new move.

To make the code more efficient we implemented an addition to the min max algorithm. This addition is known as 'Alpha, Beta, Pruning'. This works by having two values known as alpha and beta. Alpha represents the current highest max score, and beta keeps track of the current lowest min score. If a min score is come across which is lower than beta, beta is updated and the rest of the nodes branching downwards are ignored. This is because the AI known that whatever scores are given to the other nodes, its irrelevant because the opponent (in theory) should take the lowest score, and this lowest score is lower than an alternative elsewhere in the tree. This saves time on calculating the score of the other nodes, and in theory, could save a large proportion of time overall, although it drastically varies depending on the (unpredictable) sequence in which scores are found in the tree. At worst is will take as long as the normal min-max algorithm, but in certain situations it can massively improve the efficiency.

When we started coming up with ideas on how to make the code more efficient, we came up with a few ideas, which we didn't implement, but we believe would have greatly increased the impact of our AI. The main one was to use multiple computers to calculate the tree. This would have worked by using the min max algorithm still. The way in which we thought it would be easiest to implement would be to pass the first set of children nodes one at a time to different clients running on different computers along with the timer deadline (therefore we would need as many computers as there could be children, experiments revealed at any one point it was very unlikely that more than 12 would be needed). Each client would then search down the tree as far as it could go using min max still, and then after the timer, return the end result to the main computer which would then pick the best move from the returned value and play that move. Using this method would allow the tree to go down at least one extra level than before. We decided not to try implementing this as we decided that even if it worked, it would be an unfair advantage and not very practical outside of the lab.

Another simple way we thought we might be able to reach further down the tree with was to use the opponents allotted calculation time as well as our own to search down the tree in our prediction of their next move. This would mean that the AI got at least twice as long to calculate the tree as before (assuming the opponent played the move we predicted, otherwise the tree would be restarted and no gain would be had). We decided not to implement this as again, we decided it would not be fair, especially with the implementation of 'AI Move' buttons, as the AI would not only get their calculation time (which we thought would be somewhat fair) but any time in between moves (such as pauses, talking, interference or 'referee' checking moves – and of course we could cheat by deliberately holding up the game between moves).

Testing:

GUI:

Stats panel:

When the board state changes such as the amount of white pieces or black pieces change the relative scores update.

When a move is played, the current move changed to the opposing team and the correct image is displayed.

Board panel:

The board panel displays the correct state of the board and is updated when an action occurs.

It also displays the correct images.

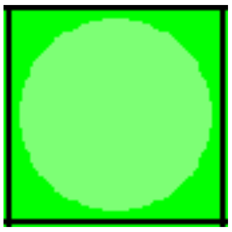
Networking as Server:

When hosting as a server if a client connects a game can be played. It correctly updates o the local GUI with the new move and send the appropriate data for the client as well. This works vice versa so when the client sends us data it is updated on the GUI correctly also.

Networking as Client:

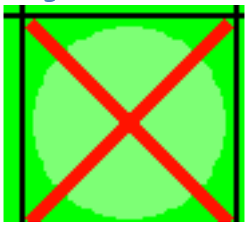
As the client we can connect to a server based on an IP address or mac name. When doing so a game can be played over the network. The local GUI is updated with new moves and appropriate data is sent to the server for there updating of game state. When the server sends us data it correctly updates the client GUI.

Legal Moves:



When the player moves there mouse over a legal move it will display the correct transparent piece over the square. This changes when you move to a new square. If the user clicks a square like this it will allow the user to carry out this move.

Illegal Moves:

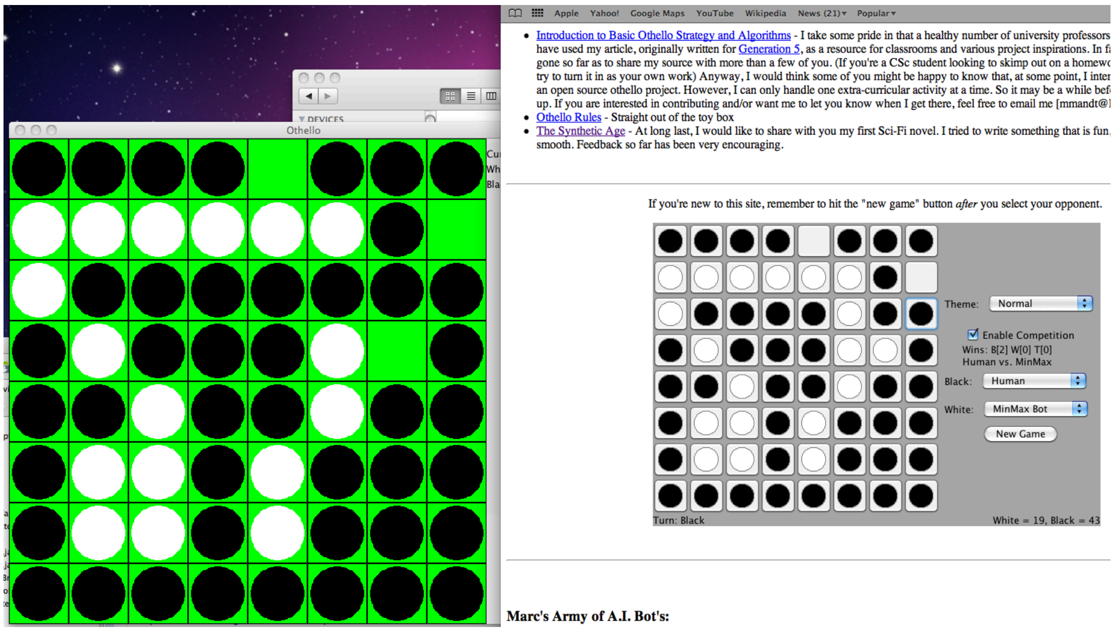


if the player moves over an illegal move the transparent image with a cross through it is displayed. It will not allow the user to play this move even if they click the mouse on this square.
This image and function is displayed on illegal moves based on the rules stated in the lecture.

Online Testing:

To help the testing of our AI I created a class which allows us to automatically test our AI against almost any other Othello AI (whether online or not). To do this, I created the WebBroker class. This class stores the model of the game from our clients point of view and carries out the interactions with the other client. To do this, all it needs to know is the pixel co-ordinate of the top left of the other clients board and the bottom right pixel of the opponents board. It then calculates the width in between each square on their board. It then uses these two pieces of information to create an array of points which are all of the squares on their board. The class can then use the awt.Robot class to click on one of the squares, or to find out the color of the piece on that square. When our AI plays a move, it moves the mouse onto the square of the other clients board and clicks. This initiates the move on their client. The class then waits for a set time (to allow the opponent to make a move). It then reads in the colors of all of the squares on their board, and finds out that if there is a piece on their board that isn't on ours, then that must be the move they have made, and updates the model within our client.

This attempt at speeding up testing against other AIs (opposed to manually updating both clients) turned out much more successful and adaptable than we anticipated and even allowed us to play multiple games in a row to find out who wins how many games out of x games. The screen shot bellow demonstrates us playing against an AI for three matches; we won all three matches (as black). Obviously you can't see the moves being made, but all three games have played out all on their own with no user interaction and with no errors.



- Introduction to Basic Othello Strategy and Algorithms - I take some pride in that a healthy number of university professors have used my article, originally written for Generation 5, as a resource for classrooms and various project inspirations. In f gone so far as to share my source with more than a few of you. (If you're a CSc student looking to skimp out on a homewo try to turn it in as your own work) Anyway, I would think some of you might be happy to know that, at some point, I inter an open source othello project. However, I can only handle one extra-curricular activity at a time. So it may be a while bef up. If you are interested in contributing and/or want me to let you know when I get there, feel free to email me [mmandt@
- Othello Rules - Straight out of the toy box
- The Synthetic Age - At long last, I would like to share with you my first Sci-Fi novel. I tried to write something that is fun, smooth. Feedback so far has been very encouraging.

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

Theme: Normal

[✓] Enable Competition
Wins: B[2] W[0] T[0]
Human vs. MinMax
Black: Human
White: MinMax Bot
New Game

Turn: Black
White = 19, Black = 43

Marc's Army of A.I. Bot's:

Specification versus Outcome:

Basic:

Create the game Othello with the rules stated in the lecture.
Upon running the program creates the game of an 8 by 8 board. It conforms the set rules of Othello whereby you have to play moves that cause at least one flip and have to be next to another piece in an adjacent square etc. These rules are the ones stated in the lecture.
Spec = met.

Ability to play the game between 2 local humans.
You can play a game on the local machine on one interface. This who be carried out by both players using the mouse to click.
Spec = met.

Created by completing the given othelloModel package.
The implementation uses the othelloModel package given. It has completed the game while conforming to this.
Spec = met.

Has A GUI
A Swing/AWT interface has been implemented and is used for both local and network play.
Spec = met.

Intermediate:

A simple AI player
By clicking the AI button on the GUI the AI will play a move, this works for both local and network play.
Spec = met.

The ability to play over the network using the remote Interface package, for both the human and AI players.
Both as a server or client we can play the game across a network with a specified address. We cannot play moves, which we are not allowed to, we are assigned a color that is others to play and cannot play the other color.
Spec = met.

A button to play the Ai moves so that the game doesn't happen too fast to observe.
In the stats panel part of the GUI we have a button with makes the AI calculate and play the next move. Therefore it can be pressed as slowly as the user desires allowing it to be observable.
Spec = met.

Advanced:

<u>A smarter AI which looks further than one move ahead based on the mini-max algorithm</u>
Our AI implemented a mini-max algorithm with added alpha-beta pruning; it searches through a large amount of possibilities to calculate the best move possible. It has different weightings for certain squares imported from a file.
Spec = met.

Area of improvement/incomplete areas:

Animation failures: We testing the AI with animation it does not consistently work. Some cases it will animate the correct pieces but for the majority of the plays it will either not animate or only animate a few pieces. We trouble shot but could not find the source of the problem. This however, does not effect the legality of the game and still renders the correct pieces. It is a problem isolated to the combination of the AI button and animation. Animation works correctly with the human playing via mouse click.

Conclusion:

Overall I would say that the outcome of the project is very good, we have met all the specification and added a few extra elements of our own. It is disappointing we could not get the animation to work in conjunction with the AI, however, this is a small part of the program and does not effect the actual game so is only a minor problem. The implementation of the AI has shown to perform well against online components so this is a positive feedback in turns of results. This project as a whole is implemented to a standard which we are happy with.

Our AI is called Arisia in memory of the schools Server, who when down during the creation of our project.

Collaboration Note:

During testing we played against several of the other students in the module, these students include: Conrad, Gordon, Simone, David, Jamie and Antii.

References:

History Cit:
The evolution of String Othello Programs, Michael Buro.

Min-Max algorithm Cit:
homepages.cwi.nl/~rdewolf/simonlowerbound.pdf

Alpha-Beta Pruning Cit:
http://www.maths.nottingham.ac.uk/personal/anw/G13GT1/alphabet.html