English Draughts AI

Benedict Morris – c1324829

Supervisor – Dr. Yukun Lai

5th May 2017

ABSTRACT

A variety of AI players capable of playing English Draughts have been implemented, along with a method of testing their effectiveness. A Genetic Algorithm for tuning weights has also been implemented, to some success. A fully playable game of Draughts and graphical interface has also been created, in which Humans can play against the different AI.

ACKNOWLEDGEMENTS

I would like to thank Dr. Yukun Lai for his help during this project. The weekly meetings kept the work focused and on track. Further thanks go to my lecturers for teaching me the skills necessary to carry out this work.

Table Of Contents

ABSTRACT 2
ACKNOWLEDGEMENTS 2
INTRODUCTION 4
BACKGROUND 4
English Draughts 4
Overview 4
Moving Pieces 4
Winning The Game 5
Artificial Intelligence and Human Games 5
Overview 5
Random Move Selection 6
Game Trees 6
Best First Selection 7
MiniMax Algorithm 7
Weight Tuning 9
SPECIFICATION AND DESIGN 10
Specification 10
Design 10
General Flow of Programme 11
Drawing The Board 11
Object Oriented Principles 11

Maintaining Game State 11
Game Logic 12
Types Of Players 12
Genetic Algorithm 12
IMPLEMENTATION 13
Packages 13
Utility Classes/ moveFinders classes 13
Drawing The Board 14
Starting/ Playing A Normal Game 14
Human Player 15
Random Player 16
BestFirst Player, MiniMax Player And Game Tree 16
Genetic Algorithm 18
Player Testing 18
Results and Evaluation 18
Random Player vs Random Player 19
Random Player vs BestFirst Player 19
Random Player vs MiniMax Player 20
BestFirst vs MiniMax 20
MiniMax vs Human 21
Genetic Weights Testing 21
Evaluation of Results 22
Future Work 23
Conclusion 24
References 24
Appendix 25

INTRODUCTION

Developing Artificial Intelligence (AI) capable of beating expert Humans at Human games is an important goal in AI research. Attempts to do so have resulted in a wide range of approaches, from generating thousands of possible moves and counter moves and analysing which move to choose a result, to attempting to make computers learn the best strategy through reinforcement learning or evolutionary methods.

This project explores two of the aforementioned methods- creating and searching game trees of moves and countermoves, and finding strong methods of evaluating the board via an evolutionary algorithm.

BACKGROUND

English Draughts

Overview

English Draughts is played on an 8x8 chequered board. Both players start with 12 pieces each, for a total of 24 pieces in play at the beginning of the game. All pieces start as “man” pieces, meaning they can move forward diagonally one square at a time, or take opposing pieces that are diagonally in front of them. If a man piece reaches the opposite side of the board to its starting side, it is “crowned” and becomes a “king” piece. King pieces can move both forwards and backwards diagonally, and also take both forwards and backwards diagonally. King pieces are very powerful. The objective is to take all of the opponent’s pieces, or leave the opponent with no legal move.

Moving Pieces

There are two moves within English Draughts. A “simple move” consists of moving a piece to a diagonally adjacent square that is unoccupied. A man piece can only move forwards diagonally. King pieces can move either forwards or backwards diagonally. However, when performing a simple move, both man and king pieces can only move one square per turn. Completing one simple move will end that player’s turn. A “jump” move allows the capture of opposing pieces, and can only occur when:

1. An opposing piece occupies a square that would otherwise be reachable by making a simple move, and
2. The square immediately diagonally adjacent, in the same direction of movement to that occupied square, is empty.

After completing a jump, the opposing piece that was jumped over is “taken” and removed from play. If a jump move is possible, it must be taken, regardless of the state of the rest of the board. If jump moves are possible for multiple different pieces, the player may choose which piece to move.

If, after completing a jump, another opposing piece can be jumped over, that opposing piece must be taken- this continues until no more jumps are possible. If a piece is kinged after a jump, the turn ends immediately.

The rules of movement are illustrated in fig. 1.

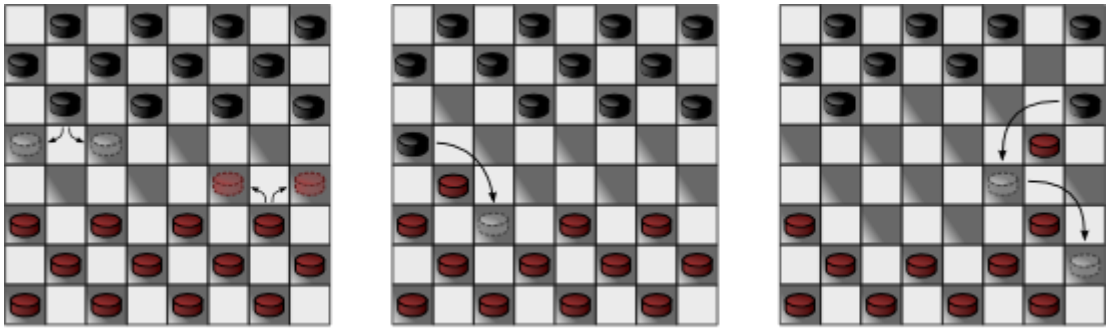

Figure 1. Source http://www.boardgamecentral.com/rules/checkers-rules.html

Winning The Game

The game is won by taking all of the opponent’s pieces, or leaving them without a legal move to make.

Artificial Intelligence and Human Games

Overview

The development of AI capable of both playing and beating humans at human games has been an important part of AI research since its inception in the mid 20th century[1]. Arthur Samuel, an early Computer Scientist and AI researcher, created the first AI capable of playing Draughts that successfully defeated the U.S.A’s number four ranked player in 1962[2]. Since then, AI for games such as Draughts, Chess and Go has rapidly started to outclass top human players worldwide[3][4]. The motivation for AI development in gaming is twofold. Firstly, board games present an achievable goal (winning), along with inputs in the form of the game state and a changing environment in the form of gameplay. This represents a laboratory- style controlled environment for testing AI design ideas in. The underlying principles of these game playing intelligences can be applied to other, real world problems. Secondly, the video game industry is worth vast sums of money, so there is strong financial incentive for creating suitable AI for humans to play video games against.

Therefore, the creation of game playing AI is a worthwhile pursuit. This brings with it many challenges that must be overcome if a suitably effective AI is to be built, ranging from hardware

limitations to the difficulty of emulating something as complex as a Human brain. The approaches implemented in this project will now be discussed in detail.

Random Move Selection

The most simplistic player possible is one that chooses a move randomly each turn. This was implemented in the project to provide a benchmark against which to judge supposedly more sophisticated AI players. If an AI cannot beat a random player, then that AI design or implementation clearly is not working as intended.

Game Trees

Generating a game tree is one of the first steps in creating an AI capable of playing games such as Draughts. Computers can simulate a game's progression into the future, much like humans imagine the impact of moves they make in a game- e.g. moving a piece now in order to get an advantage of some kind in the future. This can be done up to a certain number of moves in the future, with a tree as simple as looking one move forwards from the current board state to trees as complex as ten, twenty, or even more moves ahead for every possible move from the current board state. The limiting factor to tree generation is the computer's memory. An example of a game tree is provided in figure 2, using the game of Noughts and Crosses.

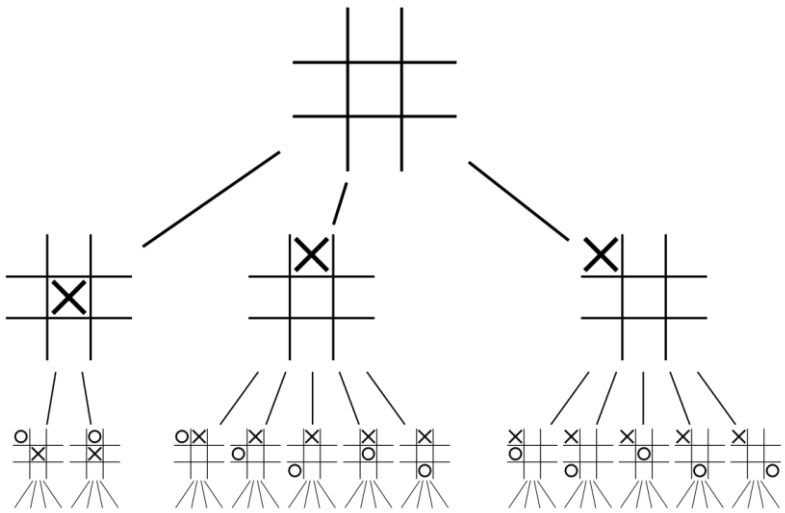

Fig 2. Source https://en.wikipedia.org/wiki/Game_tree

The computer can simulate its own moves from every possible legal move on the board currently, followed by its opponent's response, and its own response to that, and so on. Once a tree has been generated, the leaf nodes of the tree can be evaluated using an evaluation function, which is designed by the programmer and can take into account as much or as little information on the board as the programmer sees fit. The evaluation function used here is a polynomial expression. For example, in the case of Draughts, an evaluation function may increase a leaf node board state's

score if that board state has more pieces in the computer's control rather than under the opponent's control, or if that leaf node state has more kings under the computer's control than the opponent's. An example of a simplistic function in Draughts would be:

Ef(boardState)=numOurPieces+numOurKings

This expression will score board states more highly than others if the computer has more pieces or more kings than those other states. Further complexity can be added by introducing scalar "weights" to each variable in the above function. This represents the importance the computer places on each variable in the function. A higher weight on a variable means that the computer will take that variable into account more when evaluating the board state. An example of an evaluation function in Draughts with weights is:

*Ef(boardState)=w1*numOurPieces+w2*numOurKings*

Adding weights increases the sophistication of the computer's evaluation, allowing for experimentation to fine tune these weights and discover just how important each variable is when evaluating a board state.

Ultimately, a fully realised game tree will consist of possible future game states with leaf nodes evaluated using an evaluation function. This tree can then be used by the computer, in order to attempt to reach the highest scored future game states. In an ideal world, the computer could simply select the moves that lead to this highest scored state without interference. However, the opponent cannot be expected to allow this to happen, and it can be assumed the opponent will not make moves that allow the computer to reach this most desirable game state. Thus, an extra layer of sophistication must be added in order to achieve the highest scored game state possible whilst taking into account the opponent's intervention.

Best First Selection

A simplistic application of a game tree is found in a "best first" AI player. This strategy creates a tree of depth 1, meaning it just considers its own next move for every possible legal move. An evaluation can then be applied to these generated states, and the move taken is simply the one that ends in the highest scoring state.

This approach is beatable by most Human players as most people are fully capable of thinking more than one move ahead.

MiniMax Algorithm

The MiniMax algorithm was developed by Von Neumann in 1928[6].

> *"For every finite two-person zero-sum game there exists at least one optimal mixed strategy. Therefore, there exists a game value v, such that by applying the optimal strategy the first player guarantees for himself a payoff not worse than v, while the second player guarantees for himself a payoff not worse than −v."*[6]

MiniMax utilises a game tree and evaluation function, and provides a way for a computer to identify the highest scoring board state it can realistically reach given the opponent's interference. Starting with the leaf node's score, generated by the computer's evaluation function, MiniMax iterates through the game tree and assigns a score to each intermediary state between the root of the tree and the leaf nodes. The score it assigns to each state represents how desirable moving to that state is for the computer.

MiniMax takes the opponent into account by setting the score of an intermediary state, where the opponent is deciding which move to take, to the minimum score of the resultant game states. In other words, when it is the opponent's move from an intermediate state, the opponent is assumed to move to the lowest scoring state for our computer player. Fig 3 illustrates this.

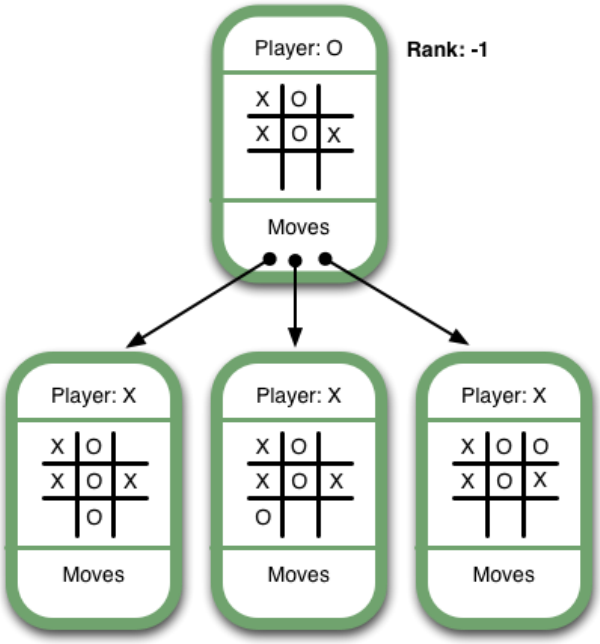

Fig. 3. Source http://www.flyingmachinestudios.com/programming/minimax/

As shown in Fig 3, the opponent has to make a move from the state at the top of this tree. The opponent will want to win, thus the opponent can be assumed to make the move resulting in the far left leaf state. The score of this leaf state is set to -1 by our evaluation function, because it is undesirable for our computer player. MiniMax therefore sets the score of the parent state to -1 as well, because entering this state is equivalent to giving the win to the opponent. The score of an intermediate state where the opponent is making a move is equivalent to the minimum score of all child states reachable from that intermediary state.

MiniMax applies the opposite logic to intermediary states where our computer player has the choice of moves. If the game enters these states, our computer player can choose the maximum scoring child state from all available child states. Therefore, the score of an intermediate state where the computer player is making a move is equivalent to the maximum score amongst all child states reachable from that intermediary state.

This logic is applied to the entire tree, until the children of our root node have scores assigned. The computer can then select the move that reaches the highest scoring child of the root. In perfect information games such as Draughts, combining game trees with the MiniMax algorithm is extremely powerful and normally creates a strong enough AI to beat most casual Human players. However, a MiniMax player is only as strong as the evaluation function it uses to score leaf node states.

Weight Tuning

The weighting of variables in the evaluation function will alter the computer's effectiveness. To use Chess as an example, if the evaluation function does not weight the Queen highly enough, that computer player is likely to lose its Queen during the course of the game. That computer will likely lose games as a result due to the Queen's known importance in Chess[7]. How high, then, should each variable in the evaluation function be weighted? The answer to this question can be provided by a number of methods. This project attempted an evolutionary approach.

Genetic Algorithms (GAs) are a family of algorithms that draw inspiration from natural processes[8]. Specifically, GAs are based on the Darwinian process of evolution through natural selection, including mutation and "mating" of genes to produce subsequent generations. Starting with a pool of possible solutions, in which the solutions consist of various components that are analogous to genes, in general a GA will test the solutions in some way and assign a "fitness" to each solution based on how effectively it attacked the problem. The solutions will then be recombined in some fashion, perhaps with more effective solutions disseminating their genes more than less effective solutions, and a new "generation" of slightly modified solutions will be created. The process then repeats. Hypothetically, each subsequent generation will consist more and more of the most effective genes of the previous generations, meaning a better solution is found each time. GAs have been applied effectively to real world problems- for example, NASA developed an antenna for the ST5 mission spacecraft using an evolutionary approach[9]. Furthermore, to prevent stagnation after a few generations as the best genes spread, random mutation is introduced by tweaking some solutions randomly each generation. This keeps competition fresh, as new genes introduced by mutation may be strong and thus improve the solutions generated.

GAs are not without issues. GAs can provide solutions that are "locally optimal", meaning the solution is more effective than other solutions in its gene pool, but is in fact not a strong solution overall. Since there can be a huge variety of potential solutions, identifying a globally optimal solution can be extremely difficult, if not impossible for some GAs in some applications. Fortunately, a globally optimal solution is not necessarily always needed.

In the case of English Draughts, the problem we seek to overcome is how exactly to weight our variables in the evaluation function. A solution would be a set of weights. Our genes are therefore individual weights that together comprise a solution. Mating and mutation involves the crossover and random changing of the numerical value of these weights.

SPECIFICATION AND DESIGN

It was decided early in the project that an object oriented approach would be used, implemented in Java. Java was selected as it was the most familiar language to the author, and no special language requirements were identified.

Specification

The project was required to meet 2 overall goals:

1. Create a virtual game of Draughts, playable by both Human and AI, along with a suitable GUI and Graphics.
2. Create computer players of varying complexity, starting with a simplistic random player capable of playing a random legal move, up to a player capable of tuning its own weights for its evaluation function.

Due to the wide variety of approaches to AI in games, from different tree searching algorithms to machine learning approaches and genetic approaches, a lot of freedom was provided over the exact details of implementation, provided the general goal of demonstrating increasingly complex players was followed throughout the project.

Design

The major aspects of the programme can be broken down as:

1. Overall object oriented design (displayed above) and principles
2. Creating, maintaining and updating the game state accurately as the game progresses
3. Provision of game logic to ensure legal moves are taken, pieces are removed as they are taken, pieces are crowned correctly etc.
4. Provision of different types of players that, given a game state, can then interact with that game state and update it according to the type of player it is. For example, the HumanPlayer class can update the game state via mouse clicks. An AI class such as the MiniMaxPlayer simulates the game before making a move that does not require mouse clicks.
5. Provision of classes that implement a game tree for the AI players that need it to use.
6. Provision of classes that implement a genetic algorithm for weight tuning.

These aspects of the programme are kept separate via use of packages, the exact details of which are contained in the Implementation section.

General Flow of Programme

Once started, the programme can run three different modes- a normal game of draughts, a test mode where two players play one thousand games against each other, or a genetic algorithm mode which runs the genetic algorithm. The mode to run depends on the command line arguments used. Exact detail is in the Implementation section.

When playing a normal game, two players are instantiated and player 1 starts their turn. Each player ends their turn by telling the other player to start theirs. This continues until no legal moves are possible at which point one of the players will have won.

The player testing version does not create a UI. Instead, two players that are defined within the code itself (not through command line input) are played against each other 1000 times, with the number of games won by player 2 displayed at the end. This exists in order to easily test how effective players are against each other.

Finally, the Genetic Algorithm option will start the GA with a new random gene pool. All setting for the GA are kept within the code, not through command line arguments.

As the project continued, it was decided to forgo some user friendliness in order to concentrate on developing correct AI players and a functioning GA. Ideally, the programme would allow easy altering of GA settings, or restarting normal games with different players etc, through a user friendly interface that does not require restarting through the command line each time or altering of settings within the code itself. This would make testing and using the programme easier in general.

Drawing The Board

Separate classes to game logic and game playing are responsible for the graphics needed when playing a normal game. The Board, DraughtPiece and CandidateSquare classes are responsible for all graphics seen.

Object Oriented Principles

Early attempts at design made poor use of the full strength of Object Oriented programming. For example, the Board class was an individual class, but the DraughtPiece class did not exist. Instead, Draught pieces were recorded as part of the Board class and were not separate entities. This quickly resulted in difficulty, as accessing information about the draught pieces through the Board class resulted in a cluttered Board class and an overall non-intuitive, hard to read programme. After a redesign, strictly separating out different elements of the game into separate classes where possible, the code became far simpler and more understandable. This reflects the principle of high cohesion, resulting in focused classes that, where possible, perform one task and one task only.

Maintaining Game State

It was decided early that a class, GameState, would maintain the real state of the game whilst it is played. This would mean the AI players could take a copy of the game state from this GameState class, use that copy for simulations as necessary, and then update the real state within the GameState class. This ensures that the real state of the game is untouched as the AI simulates moves. Each player class can take the game state, change it a needed, experiment with it, or otherwise alter it, without destroying the real record of the current game state.

Game Logic

The logic for calculating the legal moves available to players, the pieces taken and consequences of a move, and the option of multiple jumps is all contained within the moveFinders package. The idea behind these classes is to have a set of fully reusable classes that can be called upon by any player class and then "forgotten" when or if necessary. If, for example, a player needs to find the legal moves available for a certain draught piece, it can call on the methods in the LegalMoveFinder class which return this information. No other code is needed- all of the classes in this package are designed to be "utility", like a tool to be picked up and used and then forgotten about when not needed.

The pieces legally movable by each player are highlighted with a gold border. The squares legally reachable by each piece are highlighted with a green border.

This design meant the creation of new player classes was greatly simplified. As these utility classes were designed to be accessible by any player type, it promotes reuse of code and simplifies the code base. When creating a new player, the knowledge that these well tested and robust utility classes would return the correct information took a lot of work out of the process. If not for this style of class, players would either have to calculate game logic themselves (resulting in much bigger and less cohesive player classes) or rely on some other method that could be more complex.

Types Of Players

The player classes have the same general methods and fields, with some variation depending on the type of player (e.g. AI, Human etc). By keeping the structure of the player classes similar where possible, it meant that less specific details about each individual player class had to be remembered when making changes or trying to access information. Each player was intended to use the utility classes in the moveFinders package to process game logic.

Genetic Algorithm

This algorithm was designed to propagate stronger genes by favouring them when crossover is performed, and introduce new genes via a small number of mutations each generation.

The algorithm creates an initial gene pool of randomly generated weights, where each group of weights forms a single solution. After fitness has been assessed by a tournament, only the top half of players create child solutions for the next generation. The bottom half of players are lost from the

gene pool completely. The best players from each generation are preserved and displayed once the maximum number of generations is reached.

This design was taken because, hypothetically, by propagating only the strongest genes from each generation, child solutions will inevitably get stronger as the generations continue. Mutation is necessary to keep the environment fresh with new competition, by introducing potentially stronger solutions that then start to spread through the gene pool.

IMPLEMENTATION

Please see Appendix for instructions on command line arguments.

Packages

The programme consists of 6 packages:

1. "board" contains classes responsible for drawing the graphics of the game as it starts and progresses.
2. "gameControl" contains classes that start the game, initialise the board, and options for how the game is run. Responsible for creating the game's JFrame and JPanels as well.
3. "genetic" contain classes that create and run the genetic algorithm for weight tuning.
4. "moveFinders" contain classes that provide game logic, such as working out what moves a piece can make, if a piece is taken etc.
5. "players" contains the different types of players the game supports, such as Human or AI and all variations thereof.
6. "tree" contains classes that generate the game tree for AI players.

Utility Classes/ moveFinders classes

The utility classes are used by all players, so a brief overview of their function will help understanding of how the players interact with the game board.

1. LegalPieceFinder provides a list of pieces legally movable by the player using it. Its methods work irrespective of player colour. Furthermore, the class will correctly only return pieces that can take another piece if that is the current situation on the board.
2. Three of the classes in this package work in tandem- TurnCalculator, LegalMoveFinder and JumpFinder. The classes are tightly coupled with each other. These classes calculate the square a piece can legally reach, and due to the jumping rules of Draughts, their implementation is somewhat complex, as there is number of variables to take into account such as piece colour and king status.
3. The moveFinder class is responsible for moving a DraughtPiece. Given a DraughtPiece, it will return a DraughtPiece with a new row and column depending on the move taken. The move taken is also an argument for this class. This class also returns the piece(s)

taken by the moved piece, and will return null if no piece has been taken. The moveFinder class is used in the doMove() method of most players.

Drawing The Board

The Graphics class was used by the Board, CandidateSquare and DraughtPiece classes. All 3 classes draw themselves when instantiated, which gives the illusion of movement for the CandidateSquare and DraughtPiece instances. The CandidateSquare are the green highlights around legally reachable squares.

Starting/ Playing A Normal Game

If the command line argument “normalGame” is supplied when running the programme, the main() method in the EnglishDraughts class instantiates a new EnglishDraughts object, which extends the JFrame class. This provides a JFrame and JPanel for the game board to be drawn on. The dimensions and settings of the JFrame and JPanel are set in the EnglishDraughts class.

EnglishDraughts then instantiates a new Board object, which draws the game board and creates the 24 draught pieces by instantiating the DraughtPiece class 24 times. An ArrayList of these 24 Draught pieces is then used to instantiate the GameState class. This instance of the GameState class will now persist for the entire game, and will be used to maintain the correct game state. The user is then asked for the type of opponent they would like to play against. Once the correct input is entered, one player of each type is instantiated in the EnglishDraughts class, and the Board and GameState objects are passed to them so they can tell the board to repaint and update the game state as the game progresses. Player 1’s start() method is called- this begins the game.

Each player class contains the “start()”, “doMove()” and “end()” methods. The parameters of these methods varies from player to player. The method names were kept the same in order to maintain some similarity between how players take a turn, so that once understanding the flow of data and processing in one player class helps in understanding the flow of data and processing in another class. A player’s turn starts when its start() method is called. Each player then does game logic, using the utility classes in the moveFinders package, finding the pieces it can legally move, and then decides a move in some way. How the move is decided is different from player to player. The move is then executed using another moveFinders class- the MoveFinder class.

Once a move is made by a player, the game state in the GameState object is updated and the board is repainted. The end() method in each player class then calls the start() method of the opponent, thus starting the next turn in the game. Eventually, one player will be unable to make a legal move, either through having no pieces or being blocked by the opponent. At this point one player is victorious.

The implementation of the various Human and AI classes will now be explained in detail.

Human Player

The human player turn is driven by mouse clicks. To implement this, a nested class MouseDetector that extends the MouseAdapter class is contained within the HumanPlayer class. Fig 4 shows this nested class

```
143
144     public class MouseDetector extends MouseAdapter{
145         public void mouseClicked(MouseEvent e) {
146             int[] clickedSquare=findClickedSquare(e.getX(), e.getY());
147             doTurn(clickedSquare, this);
148         }
149
150         private int[] findClickedSquare(int mouseX, int mouseY) {
151             int[] squareCoords=null;
152             int originX=195;
153             int originY=30;
154             for(int row=1; row<9; ++row) {
155                 for(int col=1; col<9; ++col) {
156                     Rectangle2D square=new Rectangle2D.Double(originX+(62*col), originY+(62*row), 62, 62);
157                     if(square.contains(mouseX, mouseY)) {
158                         squareCoords=new int[2];
159                         squareCoords[0]=col;
160                         squareCoords[1]=row;
161                         break;
162                     }
163                 }
164             }
165             return squareCoords;
166         }
167     }
168
169
```

Fig 4. MouseDetector class nested in HumanPlayer class.

The class is nested so that the mouseClicked() method can invoke the doMove() method in the HumanPlayer class.

The row and column of each mouse click is checked against the current legal pieces and current legal squares. If a legal piece is clicked, then legal squares are generated and displayed for that piece. If a legal square is selected, the piece that can move to that square is moved. The turn is ended with the following check (Fig 5) to see if the move made is part of a chain of jumps or not. If it is part of a chain of jumps, then the player must continue jumping as per the rules of the game.

```
89      if(moveFinder.pieceTaken(oldGameStatePieces, newGameStatePieces)) {
90          if(moveFinder.pieceKinged()) {
91              end(mouseDetector);
92          }
93
94          else {
95              legalPieces=legalPieceFinder.findPossibleJumpPiece(movedPiece, gameState.getGameStatePieces());
96              if(legalPieces.isEmpty()) {
97                  end(mouseDetector);
98              } else {
99              board.repaintBoard(null, newGameStatePieces, legalPieces);
100             }
101         }
102
103     }
104
105     else {
106         end(mouseDetector);
107     }
108
109 }
110
111
```

Fig 5. The end turn logic checking if a chain jump is possible or not.

This logic is repeated in each player. It checks if a piece has been taken using the moveFinder class. If so, it checks if the taking piece has been kinged. If so, the turn ends, as per the rules of Draughts. Otherwise, it uses a special method in LegalPieceFinder to see if the moved piece can jump further. If LegalPieceFinder returns no legally movable piece, then the moved piece cannot jump further and the turn ends. Otherwise, the board is repainted to show the legal squares reachable by the moved piece and the current turn continues.

Ending the turn calls the current opponent's start() method.

Random Player

This implementation is very simplistic. Legal pieces are found in the start() method, and a random one is chosen. The doMove() method in this class takes a DraughtPiece instance as a parameter and moves it using the moveFinder class.

Finally, the same end turn logic as shown in Fig 5 is used to see if the moved piece has jumped and can continue jumping. If jumping can continue, then doMove() is called again. Otherwise, the turn ends.

BestFirst Player, MiniMax Player And Game Tree

These players are grouped together because both utilise a game tree in a similar fashion to make moves.

The game tree is made of MiniMaxNode instances. Each node has an ArrayList<MiniMaxNode> of its child nodes. Each MiniMaxNode is also contains a game state. This game state is resultant from a simulated move. For example, a MiniMaxNode 3 levels from the first MiniMaxNode will be 3 moves in the future from that first MiniMaxNode. Giving child nodes to a starting MiniMaxNode, and then giving those child nodes children (and so on) results in a traversable game tree.

Both BestFirstPlayer and MiniMaxPlayer create a root MiniMaxNode. This node is given the real game state at the start of the player's turn. Then, both players want to simulate the game in order to generate a game tree that can be scored using an evaluation function. This is accomplished in the simulateGame() method. This method uses the ChildNodeFinder class to do all the moves legally available in the current game state. ChildNodeFinder functions like a player class in that it uses the moveFinder package to find the moves of all legal pieces in the given game state.

BestFirst player creates a tree of one level, effectively just looking at the states resulting from all moves it can possibly take. It does not simulate opponent moves. MiniMaxPlayer looks ahead 3 levels. This choice of levels was arbitrary.

Both players have an evaluateState() function, which takes a state and gives it a score based on the weights supplied to each player when they're instantiated. The BestFirst player simply sends all children of the root node to this method, then selects the highest scorer, or a random one if no state is a higher score than the current one. The MiniMax player scores the leaf nodes of its 3 level tree, then applies a MiniMax algorithm to the tree in order to determine its move. Fig 6 shows the MiniMax algorithm used. It is based on pseudocode from reference [10].

```
143    private Double miniMax(MiniMaxNode inputNode) {
144        Double bestScore=0.0;
145
146        if(inputNode.getChildren().isEmpty()) {
147            bestScore=inputNode.getMiniMaxScore();
148        }
149
150        else if(inputNode.getPlayerType()=="maximiser") {
151            bestScore=Double.NEGATIVE_INFINITY;
152
153            for(int i=0; i<inputNode.getChildren().size(); ++i) {
154                MiniMaxNode child=inputNode.getChildren().get(i);
155                Double childScore=miniMax(child);
156                bestScore=Math.max(childScore, bestScore);
157
158                inputNode.setMiniMaxScore(bestScore);
159            }
160        }
161
162        else {
163            bestScore=Double.POSITIVE_INFINITY;
164
165            for(int i=0; i<inputNode.getChildren().size(); ++i) {
166                MiniMaxNode child=inputNode.getChildren().get(i);
167                Double childScore=miniMax(child);
168                bestScore=Math.min(childScore, bestScore);
169
170                inputNode.setMiniMaxScore(bestScore);
171            }
172        }
173        return bestScore;
174
175
```

Fig 6. MiniMax algorithm.

The evaluation function for these players is as follows:

w1*differenceInMenPieces+w2*differenceInKingPieces+w3*(ourPiecesInCentre/totalOurPieces)

where w1, w2 and w3 are adjustable weights set at amounts estimated to be effective by the author. For the purposes of testing the players, both players use exactly the same weights.

Genetic Algorithm

This GA was loosely based off of reference [11]. Instantiated by EnglishDraughts when given the command line argument “runGA”, it creates a pool of 30 players with randomly generated weights. The evaluation function that uses these weights is as follows:

w1*numOurMen+w2*numOpponentMen+w3*numOurKings+w4*numOpponentKings

+w5*(ourPiecesInCentre/totalOurPieces)

The evaluation function was modified from the BestFirst/MiniMax function to provide more genes to vary.

The algorithm pits all 30 players against each other once in the evolvePlayers() method. The fitness function is the number of games won in a single tournament. Then, the crossoverPool() method sorts the players based on how many games they won. The top half of players are then bred together, creating two children per two parents, with the best two players creating four children. The bottom half of players are eliminated from the gene pool. The best player from this generation is saved as an elite, and carried over to the next generation. The best player is also saved in a separate int[] array, where the best players from each generation will be recorded. The mutatePool() method then introduces some mutation in the new generation by randomly modifying one weight of some randomly selected children. evolvePlayers() then iterates again, repeating the process for this new generation. The number of mutations and number of generations is easily modifiable in the GeneticAlgorithm class. However, the number of players is set to 30 and modifying this requires some changes in the crossoverPool() method in order to successfully pick out the top half of players.

The class also contains some methods for determining who won a game, and for sorting a gene pool based on number of games won.

The players pitted against each other in the tournament are instances of the GeneticPlayer class. This class essentially provides a MiniMax player that uses the longer evaluation function described above. A small addition is a turn limiter in the GeneticPlayer’s start() method. This prevents a stack overflow error, in cases where a game would continue for a long time because neither side could win (e.g one king present on both sides, both chasing each other around the board). In cases like this, the players call each other’s start() methods infinitely and cause the stack overflow error.

Player Testing

This mode is accessible by suppling the “testPlayers” argument to the command line when activating the programme. It runs a loop 1000 times, playing off two players with each iteration. Player 2’s wins are then printed along with the total number of games played. This mode is intended to test the effectiveness of various AI strategies against each other.

Results and Evaluation

All tests conducted were performed with the 1000 game test mode, accessible by supplying the "testPlayers" argument when executing the programme at the command line. Players are hard coded in the EnglishDraughts class for one test. Then, when new players are to be tested, the new players are hard coded over the old ones. No command input exists for instructing the programme what players to test.

Random Player vs Random Player

This test is designed to show the effectiveness of the test mode. It was hypothesised that a random vs random player test run should show approximately 50% of games won by both players. If the test mode shows this, it will be considered reliable for testing players.

Method of test: In the simulateDraughts() method in the EnglishDraughts class, players 1 and 2 were instantiated as follows

```
RandomPlayer player1=new RandomPlayer(board, 1, gameState);

RandomPlayer player1=new RandomPlayer(board, 2, gameState);
```

3 testPlayers runs were conducted:

First Run: Player 2 won 505/1000 games.

Second Run: Player 2 won 522/1000 games.

Third Run: Player 2 won 501/1000 games.

Player 2 won 509 games per 1000 on average, equivalent to a 51% winrate. This supports the testPlayers mode's effectiveness in testing AI players against eachother.

Random Player vs BestFirst Player

If the BestFirst player is working as intended, we would expect to see a higher than 50% winrate against a random player.

Method of test: In the simulateDraughts() method in the EnglishDraughts class, players 1 and 2 were instantiated as follows

```
int[] weight1=new int[] {100, 200, 50};


RandomPlayer player1=new RandomPlayer(board, 1, gameState);

BestFirstPlayer player2=new BestFirstPlayer(board, 2, gameState, weight1);
```

Kings were weighted as twice as important as men pieces, and central board control as half important as men pieces. These weights are estimates by the author.

3 testPlayers runs were conducted:

First Run: Player 2 won 638/1000 games.

Second Run: Player 2 won 597/1000 games.

Third Run: player 3 won 628/1000 games.

The BestFirst player won 621 games per 1000 on average, equivalent to a 62% winrate. This supports the conclusion that the BestFirst player plays with some intelligence. If it were random, a winrate close to 50% would have been shown. However the lack of dominant victory shows how simplistic this AI is- although untested, a Human player would most likely have close to a 100% winrate against a random player.

Random Player vs MiniMax Player

A MiniMax player would be expected to perform very strongly vs a Random player, given the strength of MiniMax players against Humans.

Method of Test: In the simulateDraughts() method in the EnglishDraughts class, players 1 and 2 were instantiated as follows

```
int[] weight1=new int[] {100, 200, 50};

RandomPlayer player1=new RandomPlayer(board, 1, gameState);

MiniMaxPlayer player2=new MiniMaxPlayer(board, 2, gameState, weight1);
```

3 testPlayers runs were conducted:

First Run: Player 2 won 991/1000 games

Second Run: Player 2 won 989/1000 games

Third Run: Player 2 won 992/1000 games

The MiniMax player won 991 games per 1000 on average, equivalent to a 99% winrate. This demonstrates the effectiveness of this MiniMax player- if the winrate were less than almost perfect, the MiniMax player would most likely not be functioning as intended.

BestFirst vs MiniMax

As this BestFirst player is not designed to be particularly strong, a dominant result from the MiniMax player is expected. Given the results of the MiniMax vs Random and the BestFirst vs Random tests, it

was expected that MiniMax would not dominate quite as strongly as it has against the random player, but it should still win the vast majority of games.

Method of Test: A turnCount variable was added to the MiniMax class that prevents a MiniMax player from taking more than 150 turns per game. This was due to stack overflow errors when conducting this test- likely due to games lasting too long.

Both players used the same weights.

In the simulateDraughts() method in the EnglishDraughts class, players 1 and 2 were instantiated as follows

```
int[] weight1=new int[] {100, 200, 50};

BestFirstPlayer player1=new BestFirstPlayer(board, 1, gameState, weight1);

MiniMaxPlayer player2=new MiniMaxPlayer(board, 2, gameState, weight1);
```

3 testPlayers runs were conducted:

First Run: Player 2 won 995/1000 games

Second Run: Player 2 won 983/1000 games

Third Run: Player 2 won 985/1000 games

Player 2 won 987 games per 1000 on average, equivalent to a 99% winrate. Whilst it was expected for the MiniMax player to perform very well against the BestFirst player, it was not expected to have an equivalent winrate as against the Random player. This does however further show this MiniMax implementation's strength.

MiniMax vs Human

Although testing vs a Human was not conducted on a 1000 game run, 20 games were played by the author against the MiniMax programme. The MiniMax programme won 20/20 games. However, this test was very dependent on the author's ability, which is not experienced or particularly strong. Ideally, exhaustive tests would be conducted against multiple different Human players, but this was out of scope for the project.

Single games against the MiniMax player still demonstrate its strength of play.

Genetic Weights Testing

The Genetic Algorithm was run for 50 generations, and the final elite player gave the weights:

420, 150, 107, 368, 188, 363

These weights were tested against 4 different sets of weights from the initial gene pool of that run. Hypothetically, if the GA has improved the weights, then the above set of weights will have an above 50% winrate against the initial weights in the gene pool.

The weights from the initial gene pool were:

166 222 355 211 112 444

490 302 410 466 369 303

246 12 45 186 58 467

101 228 231 227 129 229s

Method of test: In the simulateDraughts() method in the EnglishDraughts class, players 1 and 2 were instantiated as follows

```
int[] p1Weights=new int[] {166, 222, 355, 211, 112, 444};
int[] p2Weights=new int[] {420, 150, 107, 368, 188, 363};
```

```
GeneticPlayer player1=new GeneticPlayer(1, gameState, p1Weights, board);
GeneticPlayer player2=new GeneticPlayer(2, gameState, p2Weights, board);
```

The testPlayers test was run with the elite weights vs the 4 sets of initial weights, in order from top to bottom as the initial weights are shown above. The initial weights were saved as the “p1Weights” variable, the elite weights as the “p2Weights” variable.

The results were:

Vs first initial weights: Player 2 won 805/1000 games

Vs second initial weights: Player 2 won 501/1000 games

Vs third initial weights: Player 2 won 555/1000 games

Vs fourth initial weights: Player 2 won 921/1000 games

Evaluation of Results

The increasing effectiveness of the AI players has been shown. As the AI increased in complexity, the winrate against less complex AI also increased- from 50% as Random vs Random, to 99% as MiniMax vs BestFirst. Tests against Human players were not rigorous, but as early pilot experiments they support a strong MiniMax player.

Furthermore, the GA was shown to have at least slightly increased effectiveness of the AI player compared to its ancestors, although results appeared to vary drastically. A potential explanation for this would be that some of the ancestors that the final elite player is tested against themselves

contain elements of the elite's strategy. In other words, those ancestors against which the elite had only a slightly higher than 50% winrate against contained elements of the stronger strategy the elite is using. The elite's dominance against other ancestors shows how those ancestors did not contain any elements of the stronger strategy. It is possible that a greater generation run would result in greater winrates against ancestors.

Finally, the test used to determine the winner in a game was simplistic- it simply found the first piece in the game state and tested its colour. If it was Red, player 2 scored a win. If it was white, player 1 scored a win. This would miss wins where perhaps one piece was trapped yet happened to be selected for testing. This could account for the few losses experienced by the MiniMax player against the Random player.

The Genetic player was not tested against the MiniMax player, because both used different evaluation functions, not different weights for the same evaluation function. Furthermore, the Genetic player would only need to be tested against its ancestors to show whether or not it has improved.

Future Work

A first priority for future work would be the expansion of the Genetic Algorithm. A greater player pool and a much greater number of generations would hypothetically increase the effectiveness of the algorithm. The GA implemented in this programme tends to converge at a local optimum, resulting in different elite weights being presented at the end of different runs, rather than a global solution.

More extensive testing against Human players is also desirable. Ultimately, it can be said that as long as an AI can perform better than a Human at a given task, then the AI has been successful, even if it is not perfect. For example, a medical robot that achieves a 90% diagnoses success rate is still much stronger than a Human that can achieve an 80% success rate, even though the robot does not achieve 100% success. So, to show the true effectiveness of the Genetic Algorithm weights and the MiniMax player, exhaustive tests vs Humans would also be attempted.

From a usability perspective, the project stands to gain a lot. The usability was pushed to one side in favour of prioritising the experiments and results. This can make replicating the tests harder, and is therefore undesirable; it is important for experiments conducted to be replicable, so whilst effort has been made to explain the experimental method here, a user friendly interface with clear experimental options would go a long way to improve the programme.

Other machine learning approaches, such as Reinforcement Learning, could be explored and compared against the Genetic Approach. Results from experiments here could shed light onto which machine learning approaches are stronger for certain problems or situations.

Conclusions

The project set out to implement an increasingly strong set of AI players in a fully playable game of Draughts, culminating in a more machine learning style of player of some kind in order to tune the weights of an evaluation function. A genetic approach was chosen to achieve this.

A set of AI players was produced, along with a UI for playing against them. Along with this, a Draughts game was developed that was fully playable for either two Humans, two AI, or some mix thereof.

A Genetic Algorithm was implemented, and the best players of each generation saved and displayed at the end of each run. The final elite player was tested against its ancestors with some success.

Overall, the project has explored its goals and has either met them or made headway into meeting them. The Genetic Algorithm tuned weights were not quite as strong against their ancestors as hoped, although some improvement was shown. The MiniMax player played very strongly against all opponents, which was a satisfying result. The Random player and BestFirst Player also performed as expected. The game board itself was presented with interesting graphical highlights to show legal pieces and moves for Human players. Throughout the numerous games to playtest, there appeared to be no illegal or unusual moves allowed, implying the game logic is sound.

The project would have benefited from more attention to user friendliness. It is very much only usable by those more experienced with computing. Some of the game logic is complex and should be simplified if possible.

The project taught me the importance of good programme design and the power of Object Orientation; splitting up the elements of the game into different classes greatly improved the readability and usability of the codebase. It introduced me to important Modern concepts of machine learning and AI, and how AI research is impacting our lives today, even if it is not obvious in many cases.

References

1. http://www.livinginternet.com/i/ii_ai.htm
2. http://www.tug.org/TUGboat/tb11-4/tb30knut-samuel.pdf
3. http://www.popularmechanics.com/technology/apps/a19790/what-deep-blue-beating-garry-kasparov-reveals-about-todays-artificial-intelligence-panic/
4. https://gogameguru.com/tag/deepmind-alphago-lee-sedol/
5. https://www.cims.nyu.edu/~munoz/files/ml_optimization.pdf
6. http://wikizmsi.zut.edu.pl/uploads/d/d0/Min_max_en.pdf
7. http://www.chessinvasion.com/queen.html

8. https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_introduction.htm
9. https://ti.arc.nasa.gov/m/pub-archive/1244h/1244%20(Hornby).pdf
10. http://will.thimbleby.net/algorithms/doku.php?id=minimax_search
11. http://ieeexplore.ieee.org/document/592428/

Appendix

To run a normal game, use command line argument “normalGame”. When asked what opponent to play against, enter the lowercase letter corresponding to the opponents listed in the console.

To run the Genetic Algorithm, use command line argument “runGA”. Changing the mutation rate and total generations is possible by altering the value of the corresponding fields in the Genetic Algorithm class.

To run the test players mode, use command line argument “testPlayers”. To alter which players are being tested, redefine the player1 and player2 variables as the players you wish to test. Examples of this are provided in the Results and Evaluation section.