ARTIFICIAL INTELLIGENCE FOR DRAUGHTS

Degree: BSc Computer Science
Supervisor: Dr Renato Amorim
Second Assessor: Dr Javier Andreu

Harry Edmund John Brewer
Hb16680@essx.ac.uk
1602734

Abstract

Artificial Intelligence(AI) in games is a well-researched and explored topic, and due to this, there are many approaches used to create different algorithms. This project's goal is to evaluate the effectiveness of two types of algorithms, Heuristic and Alpha-Beta Pruning(ABP). The Heuristic algorithm is rule-based and uses a list of predetermined rules to decide what move to make and the ABP algorithm that evaluates game trees to determine what move is the best choice.

Both algorithm types were tested on a game of Draughts. The game has been designed to allow three different modes to be played, a human player against an algorithm, an algorithm against itself or two different algorithms against each other.

The advantage an algorithm has over a human player is that when appropriately programmed, the algorithm will never miss a move as it will evaluate all possible moves before deciding what to do. This is a common flaw with human players who may not be able to visualise all potential legal moves. This is due to the ability of the algorithms to process the game significantly quicker than a human can. An artificial wait timer had to be added so that the moves would be slow enough so that a human player could follow them.

Acknowledgements

Firstly, I would like to thank my project supervisor, Dr R. Amorim, for support and guidance throughout this project.

For help with proofreading, I would like to thank Michael Brewer and Jane Ayres.

I would like to acknowledge Grammarly, which I used for grammar and spell checking on this report.

Finally, I would like to thank my parents and friends for their support and encouragement when I came across problems during the research and writing of this project.

Contents

List of Abbreviations

ABP	Alpha-Beta Pruning
AI	Artificial Intelligence
BCL	Binary Code License
BCS	British Computer Society
CBR	Case-Based Reasoning
GUI	Graphical User Interface
IDE	Integrated development environment
OOP	Object-Oriented Programming
UML	Unified Modelling Language

Table of Figures

1 Introduction

This project implements different AI algorithms that will play a game of Draughts. This will require programming a game of Draughts that will allow the player to choose different algorithms and allow those algorithms to interact with the game. Two different algorithms, using various methods to get their results, will be compared against each other to show how the methods have different efficiencies; this allows for easy comparison. There needs to be a system that allows for natural interaction between the different actors of the program which has a simple method of transferring data so that all systems are running on up to date data. The game itself will have the standard English Draughts rules, and there will be a simple GUI for the player to interact with. This allows for most of the development time to be spent on the algorithms themselves.

There are many aspects to AI that have developed over the years. This has led to the development of numerous methods to achieve similar results. For this project, I am interested in game evaluation algorithms. Within these, there are different types that work towards choosing the best move possible. These methods have been used to create algorithms that can defeat the world champions in games such as Chess and Go [1]. In a lot of these cases, machine learning was used to improve the AIs ability.

The primary objective of this project is to develop the ability to compare two AI algorithms against each other. This requires a game to play and for the two algorithms to be completed by the end of the project. There needs to be a simple method for the player to interact with the game that is easy to learn. The two algorithms should have two different methods of choosing the next move.

I have an interest in game development, but due to my level of experience, I wanted to select a project that I knew I would be able to learn from and accomplish within the time constraints that this project needed to be completed in. I have a deep interest in AI which helped my decision as this project is based more on the AI side than game development and lends itself more to meeting my objectives. I have some prior experience in simple AI and wanted to improve my knowledge and skills within AI and programming in general.

This report includes a literature review that details the level of research undertaken for this project, including research into different algorithms and how they compare to each other. There is a system design and implementation section that covers what the program includes, and the methods used to create the different sections. In the evaluation section, there is a comparison of the two algorithms and which one is more effective at playing the game of Draughts.

2 Aims and Requirements

2.1 Aims

As mentioned in the introduction, the aim of this project was to develop a working version of Draughts with the standard English Draughts rules (see section 2.3) implemented within it.

The game should allow a player to play against another human, against an AI or allow two AIs to play against each other. The game will have an integrated GUI for the player to use to choose these options.

There should be two algorithms that the player can choose to play against; they need to have different methods of selecting the next move. This will allow them to be directly compared against each other to find out which method is more effective.

2.2 Requirements

The program shall have:

- A player interface that allows the player to choose different options that will change how the game runs.
- The player shall have the option to choose how many human players are playing.
- If the player is playing against an AI, they should be able to choose from a list of different algorithms.
- The program should have other features that make the game a better experience for the player, such as a scoring system and an end screen.
- There should be a game board that allows for the game pieces to be easily identifiable and for the king pieces to be distinguishable from the standard pieces.
- When it is a player's turn, it needs to be clear which pieces can be moved, and when a valid piece is clicked on, it should show where it can be moved to.
- There should be different game algorithms that the player can play against.
- There needs to be the ability to allow two algorithms to play against each other.

2.3 Definition of Game Rules

Each player has twelve playing pieces that are placed on the black squares of the first three rows on their side of the board. The game aims to capture all the opponent's playing pieces or to block them from moving. Playing pieces can only move diagonally, one square at a time. The playing pieces must remain on the black squares for the entire game. A player can capture an opponent's piece by jumping over them diagonally, into an empty square behind their piece. A player can make multiple captures with the same playing piece within the same move by leapfrogging various opponent pieces, once it lands after taking the first piece. If there is the option to take another piece from its new position, then you can do so, and if a player can take an opponent piece, they must do so. Once a piece reaches the far side of the board, it becomes a King. A king can move both up and down the board but always diagonally, whereas a standard piece can only move diagonally forward, both of which must only ever occupy black squares. These rules have been summarised from the European Draughts Confederation [2].

3 Professional Considerations

3.1 Legal issues and Intellectual Property

Due to the nature of this project, as the project is self-contained with the development of a game with AI alongside it, there are no legal issues. There is no data gathered or stored about the player playing the game. The only data that is stored involves the data about the game itself. Therefore, this project isn't in violation of the Data Protection Act 2018 [3]. The British Computer Society (BCS) Code of Conduct [4] is a set of rules and regulations that ensures that those within the IT industry follow correct practices on ethical, legal and social issues. This project ensures that it follows these rules and regulations.

The game itself uses generated objects, and no sprites are used for the visuals. Draughts were created before the introduction of copyright, so there is no issue using the game rules and base design for this project.

This programming of this project is done in Oracles Java 8. During the project, I have ensured that I have followed the terms of use for the software. As seen in Oracles term of use page, under the use of software section it states that "the Software may be used solely for your personal, informational, non-commercial purposes" [5]. The software is free to use if the company's terms and conditions are accepted and followed, which in this project they are. The licence that covers this is the Oracle Binary Code License (BCL).

I used the JetBrains IntelliJ IDE for the development of this project. Use of the community version of IntelliJ is free for all users, to access the software JetBrains's terms and conditions must be accepted. JetBrains's terms of use are similar to Oracle's terms of use. It is required that there must be no attempt to tamper with the software or to redistribute it [6].

Other than the properties listed above, there are no other intellectual properties used within this project.

3.2 Ethical Issues

When it comes to ethical issues that can arise from a project, human involvement is the most important one. For this project, there was no external testing by anyone that wasn't involved with the development of this project. As mentioned in section 3.1, no data was gathered or stored about anyone who plays the game. Within the game, there is no offensive, shocking imagery or text that could affect the player.

4 Literature Review

4.1 Game Design

While looking at different games for the game design, there seemed to be a typical pattern that most online games of Draughts use a very basic design for the game. As a standard, the games allow the player to choose between one player and two players. Some will enable the player to select the piece colour, and there was one case that allowed the player to select the difficulty of the computer player.

One such example is “247checkers” [7]. The current playable pieces are highlighted for the player when a player clicks on a piece to move, the locations it can move to are highlighted. When it is the algorithms’ turn, the piece that is moved gets highlighted briefly to show the player what piece has just been moved. All the piece movements have animation, and all the pieces have a graphical asset. This game has a good options menu, allowing the player to customise the game they are going to play.

“CardGames.io” [8] uses a similar basic design to “247checkers” [7]. The game automatically starts when the page loads and the AI takes their turn first. The pieces that can be played are highlighted for the player and once selected, the locations it can be moved to are highlighted. The overall design is rather simplistic. The algorithm used in this game is Minimax (See Section 4.3.1). JavaScript was used to program this game.

A further example was created by Matěj Doležal [9]. This student carried out a similar Bachelor final report to this project, designing a very intuitive and user-friendly game. There is an extensive menu, with options to change the functionality of how the game works and allow the player to customise how they want to play. The game algorithm used is Minimax and Java was used to create the game. The Minimax search depth is set to 5, which is a compromise between the time it takes to run the algorithm each turn and to get an optimal result. Due to the project being written in a report format, it allows a detailed analysis as to how this game works compared to the other examples I’ve referenced [7] [8]. This student’s project is more focussed on the game creation and the one algorithm compared to my project which is focussed on comparing two algorithms.

Figure 4-1 Screenshot of Matěj Doležal’s game *[9]*

In the case of game design as seen in Figure 4-1, Matěj Doležal uses a complex design that is also very user-friendly; this allows the player to choose a lot of options as remove to how the game runs.

All of the example games use Minimax as the algorithm to some extent, although when considering the examples where the search depth is known, Matěj Doležal had the highest depth of 5. When it comes to gameplay, all the examples use similar methods and have a similar style. Matěj Doležal's overall design and functionality were the most advanced.

4.2 Programming

There are many different programming languages that I could select to use for this project, the different options that are commonly used for this kind of project include: Java, C++ and Python. They each have their benefits and drawbacks [10].

Java is a large high-level object-oriented programming (OOP) language; it has access to many libraries which are easy to implement. It can be used to make objects easily. It has extensive online support, and it is probable that there is already documentation for any errors that appear. Java is also platform independent, allowing most programs to work across platforms.

C++ is well suited to general use and low-level programming; it is widely used in game programming. C++ has a fast run time and is ideal when latency is an important issue. C++ is well suited for various applications and systems and useful in situations that require hardware integration.

Python is an easy to read and maintain programming language. Unlike other languages, Python emphasises code readability. Python is very flexible, not requiring variables to be declared before using them and still allows programs with errors to run.

Due to my project's specification and my abilities, using Java has many benefits. The majority of my program relies on objects interacting with each other, sending information about the state of the game. As the game is not the focus of the project it only required basic graphics and a simple GUI attached to it, this could be done quickly in Java.

4.3 Algorithms

4.3.1 Minimax

Minimax is a tree search algorithm; it follows two steps to achieve the best possible result. Minimax works by maximising the gain for the players move while minimising the loss from the opponent's turn [11]. In an ideal situation, the entire tree would be explored, and then the best possible move would be taken, but for a game such as Draughts that has large trees, this would take a longer processing time, so the depth that the algorithm searches to each time is quite low. However, this is dependent on the problem the algorithm is made for and how quickly each decision can be made. For Draughts, due to the need for the time for each turn to be quite short, the search depth will be in single digits. This algorithm relies on an evaluation function that scores each game path to decide which is the best one to take. This can be done in different ways, but if carried out poorly can easily lead to an ineffective algorithm.

While looking at different examples of Draughts games, it appears that the most common algorithm used is Minimax [7] [8] [9]. This is probably because it doesn't take much development time to make and it has been widely proven to be a useful method for the game. If the evaluation method is made correctly, it can be quite efficient as an opponent to most standard players. The problem that comes with this is that using Minimax on its own can be ineffective and leads designers to limit the tree search depth so that the loading times do not deter the player from wanting to play. This is the case for the cardgames.io [8] game of Draughts which only has a two-deep search process, allowing the player not to worry about waiting for the computer to take its turn, but this does lead the algorithm not to choose the best move in many cases.

There are different Minimax methods for tree searching; each has its benefits and flaws that make them ideal for different situations. Two common versions are depth-first and breadth-first [12]. These two methods have different approaches that in different situations can either save or increase the search time.

Depth-first search starts from the root and then follows the first child of each new node it discovers until it reaches the end of the tree [13]. It then backtracks to the most recent node with an unexplored child and proceeds till the whole tree is searched. Breadth-first search starts from the root and then discovers all the children of each node at the current depth before moving onto the next depth and repeating the process.

For a game with a large game tree, depth-first will take longer to find a good solution as it may need to follow many paths before it finds a good one, whereas breadth-first can find a good a solution a lot quicker by only partly exploring all the paths.

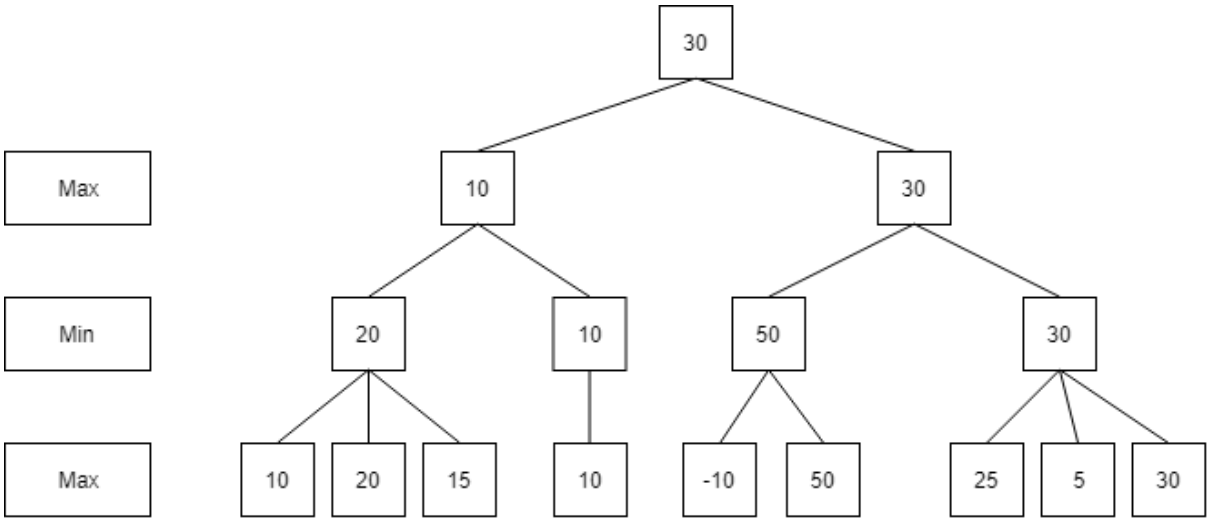

Figure 4-2 Minimax game tree

Figure 4-2 shows a visual breakdown of how Minimax works. At each parent, either the highest value child's or lowest value child's value is moved up the tree. Until it reaches the root, the next move is then made in an attempt to follow the highest value path; this process is repeated each move.

Many different improvements can be made to Minimax [14]. Such as ABP which works by removing the need to search all the nodes in a tree by ignoring branches that are known to be non-beneficial. There are other methods that work on improving a different aspect of the Minimax function.

4.3.2 Alpha-Beta Pruning (ABP)

A typical improvement on the standard Minimax algorithm is to use ABP. This is a method that aims to reduce the branches that a tree search method must make [11]. A branch stops being evaluated when at least one of the other children of the current parent is better than the move that is being currently assessed. Two values are kept as the tree is traversed, Alpha and Beta. If the value of a move is either less than Alpha in the maximisation stage or more than Beta in the minimisation stage, the move and any branches that follow from it are ignored. `

As time can be a strict constraint in many problems that use this algorithm, ABP is an important improvement to Minimax as it allows the algorithm to spend more time exploring paths that have potential to be optimal solutions and less time on paths that are known to lead nowhere. However, due to paths in Minimax, and extension ABP, having estimated scores associated with them to

determine their value if the scoring is inaccurate ABP can remove good tree paths leading to a less optimal solution being chosen.

4.3.3 Case-Based Reasoning (CBR)

Case-Based Reasoning (CBR) solves new problems by adapting previously successful solutions to similar issues [15]. CBR doesn't require a specific domain base, as its knowledge is based on previous cases. This can be gathered through performing the decided task until it collects a large enough data set to make informed decisions. The system can generalise previous cases to determine what is the best course of action to perform well. CBR remembers previous problems it has come across and uses the knowledge to get past them.

CBR allows for ease of knowledge; the cases that are saved are easy to recall and isolate, whereas rules can be challenging to separate. Due to the cases being stored from previous experience they cannot be affected by the player's bias which can change the effectiveness of the system. CBR can work with a limited case base unlike other learning algorithms and adds to the case base as the system is run. This skips the dedicated learning period. CBR systems learn over time by acquiring new cases to base future solutions from; this makes maintaining the system easy and allows for the method of improving the system by just running it.

There are, however, drawbacks to this method, it needs a significant memory requirement to store the case base and is time-consuming to collect the cases required to make the algorithm useful. As the case base cannot cover all possibilities, it can leave the algorithm with no solution to new problems. If a problem requires the best solution each time CBR will not work, as CBR generally gives good enough solutions.

4.3.4 Heuristic

Heuristic in AI is a quick method of problem-solving when other algorithms are too slow or when the knowledge base is too limited. The speed is gained by sacrificing optimality or completeness (See section 4.3.5). It works on producing a solution that is good enough within a short time frame, but it may not find the best solution for the problem. Heuristics can be used in different ways; they can be used on their own or as part of another algorithm.

Heuristics are important as its methodology is similar to how humans think, and problem solve. When it comes to solving problems that have no algorithmic solution, intelligence and insight need to be used to create a solution [16]. This can be modelled in an algorithm that uses different rules that check for different states and combined to form a solution.

Rule-based AI is limited by the size of the rule base; if there aren't enough rules for the algorithm to find solutions to problems, then the system will attempt to solve the problem with the rules it has and may produce a solution that is not optimal. These types of systems are not ideal for complex domains or use across many different domains.

A heuristic method can be improved as time goes on, more rules can be added, and previous rules can be changed to improve the performance of the algorithm. For problems that will not change in the future, for example, classic board games, heuristic algorithms work well as it is known that the system won't become unusable or outdated.

4.3.5 Comparison

AI Algorithms that use game trees can have their performance evaluated by measuring four different methods [13].

- Completeness is the measure of whether an algorithm is certain to return a solution if there is one.
- Optimality is the measure of whether the algorithm is certain to find the best solution to the problem.
- Time complexity is how long the algorithm takes to find the solution.
- Space complexity is how much memory is used during the search to store all the data used by the algorithm.

The two methods for Minimax search, depth-first and breadth-first, were compared. Depth-first is complete if the tree is finite, and it is not optimal. Depth-first has a time notation $O(b^m)$. Depth-first has a space notation of $O(bm)$ which is significantly less than breadth-first. Breadth-first is a complete search; it is also optimal. Breadth-first has a time notation of $O(b^d)$, which is a smaller time complexity than depth-first. Breath-first has a space notation of $O(b^d)$. Overall Breadth-first is better, but if the tree is finite and space is an issue, then Depth-first can be a better choice [13]. In this case, "b" is the branching factor, "d" is the depth of the shallowest solution and "m" is the maximum depth.

Comparing the other algorithms is more difficult as they don't have standard complexity values. When it comes to domains that are well defined and have complete knowledge, it is better to use a Heuristic rule-based method, whereas when it comes to domains that have missing or limited knowledge, it is better to use a CBR system [17].

When it comes to choosing which algorithm to use, the domain needs to be analysed. Concerning Draughts due to its complete knowledge base, a Heuristic rule-based algorithm would work well. Due to the games set turn-based gameplay, a game tree search algorithm would work well, however, due to Draughts branching factor and maximum game depth the time complexity would become a serious factor in the effectiveness. This is where a CBR algorithm would work well as it wouldn't have a large time complexity or a complex rule base, but it would also need a well-designed case system with many cases designed before the system could be set to learn.

5 System Design and Implementation

5.1 Programming Language & Libraries

All of the programming for this project was programmed in Java 1.8. It was chosen due to it being the language I am most fluent in allowing me to spend more time learning how to tackle the more difficult problems I would come across. Using my experience, I was able to create a game that allowed for easy integration of the AI algorithms.

The program only uses the default libraries within Java, for all of the visuals I used swing. This was used to make all the GUI systems and all the game board visuals using the 2D graphics. The main three libraries used throughout the program are javax.swing [18], java.awt [19] and java.util [20]. For each of these classes, the online documentation provided by Oracle was used extensively to get information and examples of the different methods and functions that were used in the programming of this project.

5.2 Architectural Design

The following figures, 5-1 to 5-4, show the break down of the class structure of Draughts program using Unified Modelling Language (UML) diagrams. Figure 5-1 shows a simple class diagram of the overall structure; this is so the whole system can be seen at once. Figure 5-2 shows the GUI and game portion of the program, showing a detailed view of the different methods and the essential variables within each class. Figure 5-3 shows the AI algorithm section of the program, showing the same level of detail as figure 5-2.

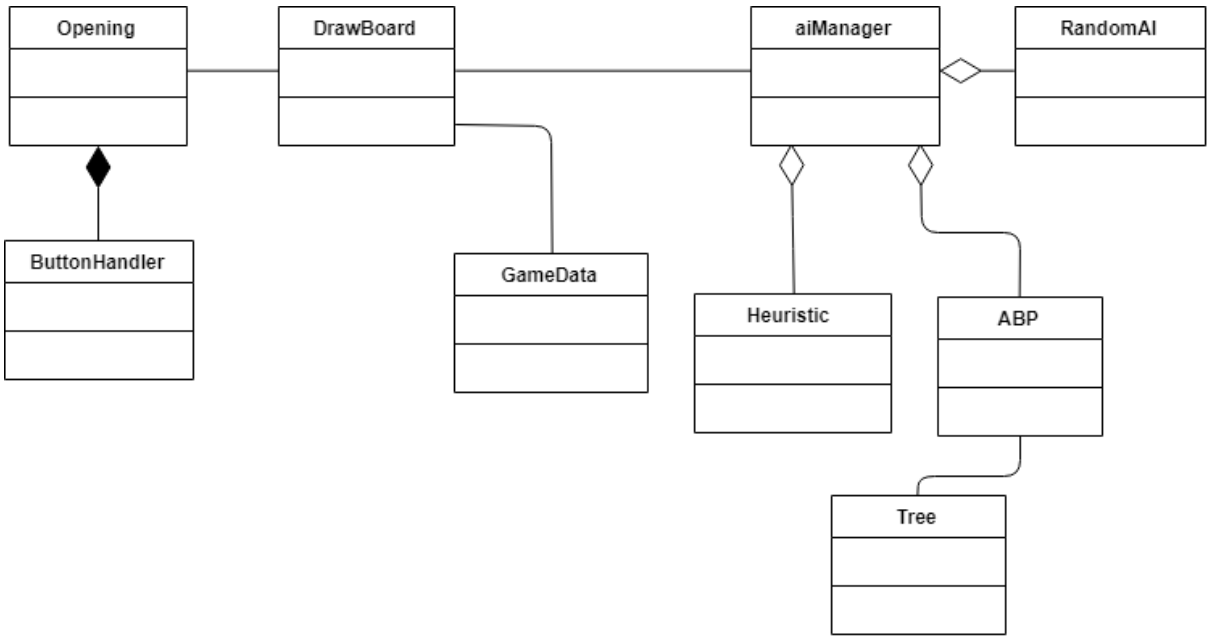

Figure 5-1 Draughts project class diagram (class names only)

Figure 5-1 shows the links between all the different classes within the project. The program can be split into two main sections, the game and the AI, with the connection between those two sections being between DrawBoard and AIManager. This simplified view enables the whole project can be seen as a whole. Although as will be seen in Figure 5-2, there are two objects that are not included in this class diagram. This is due to the fact that those two objects are used in 5 different classes, so they were kept separate from the class diagram to make it easy to read.

Looking at the relationships of the different classes, it can be seen that the ButtonHandler has a composition relationship with the Opening class, it exists only to handle the events of the buttons on the GUI. The three algorithms all have an aggregation relationship with the AIManager; each one uses the AIManager to communicate with the game but exists on its own. All the other relationships are associations.

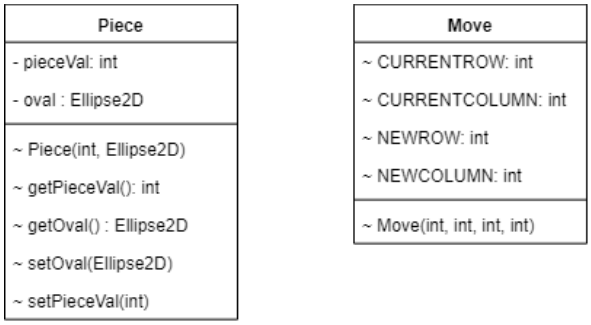

Figure 5-2 Piece and Move classes

Figure 5-2 shows the two main objects that are used to transfer data throughout the game. These were kept separate from the main class diagram as they are used in almost all the classes which would make the class diagram difficult to read. The Piece object, which stores the data about each game piece is used within, DrawBoard, GameData, Heuristic, ABP and is also used as a variable in the Tree object. The Move object, which stores the data for a valid move of a piece is used within: DrawBoard, GameData, Heuristic, ABP, Random and is also a variable within the Tree object. This shows how integral these objects are for the running of the game, although for the most part a specific instance of one of the objects will not move between more than two classes, they are used in the majority of the classes.

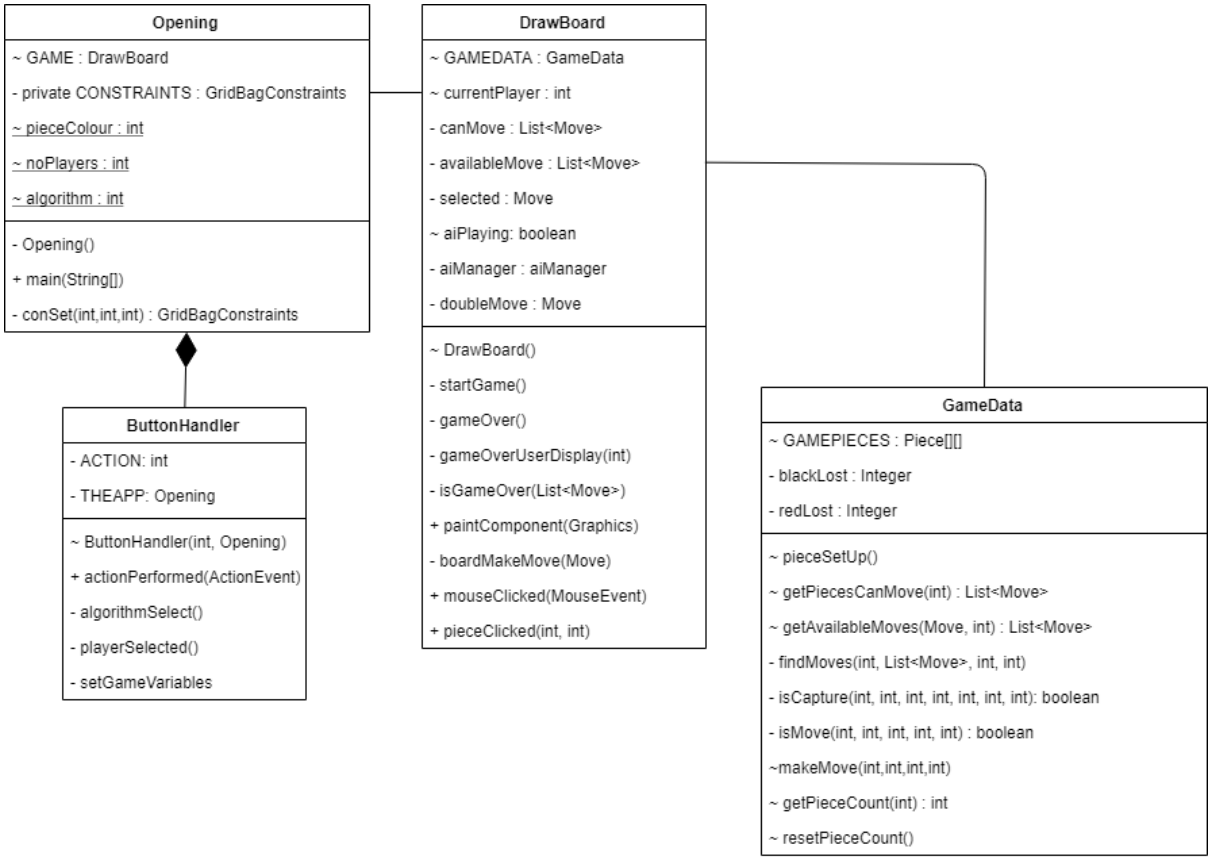

Figure 5-3 GUI and game class diagram

Figure 5-3 shows the fully detailed classes of the GUI and game. The opening class handles all the GUI side while DrawBoard and GameData are what runs the game, each one running a different part of the main functionality. Once the player starts the game and leaves the GUI, they interact with the DrawBoard class which then communicates with the needed areas.

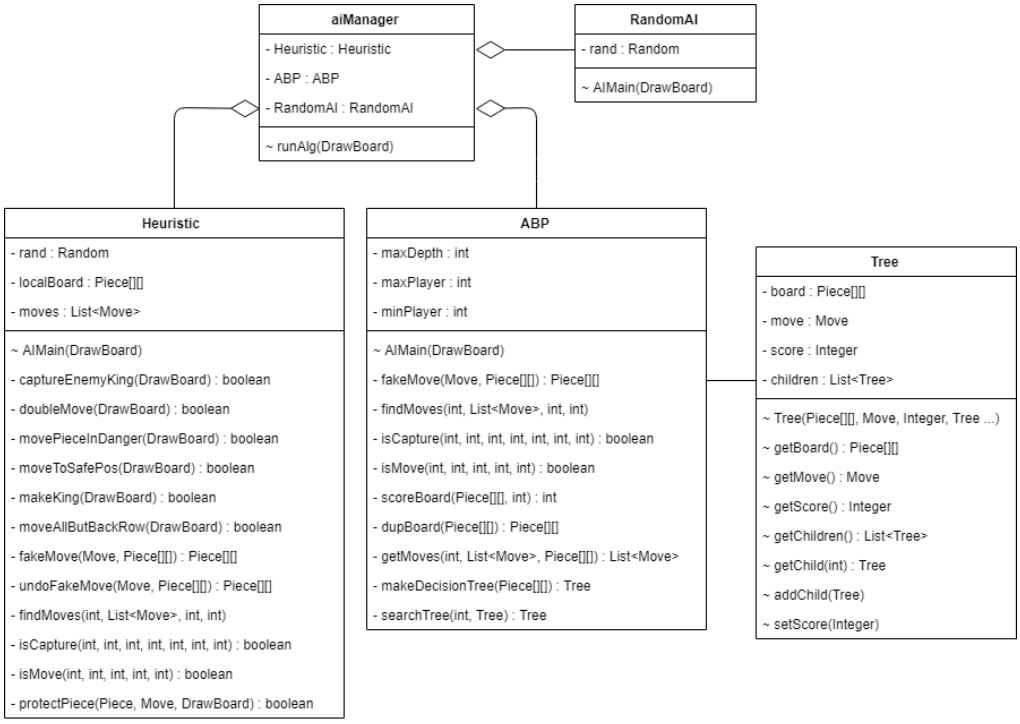

Figure 5-4 AI algorithms class diagram

Figure 5-4 shows the AI portion of the program, with the AIManager being the central control point that links to the game. When an algorithm is required to make a move, the AIManager links the appropriate algorithm to the game so that it can make the move. As a basis, each algorithm takes the current game instance and uses it to return a move that will be shown for the player. Each algorithm has its own method for generating this move; more detail of this can be seen in section 5.3.

5.3 Programming Components

5.3.1 Draughts

The Draughts game was made in different sections so that it was easier to keep track of what functions were within the program knowing that this would start to become a problem later into development when more functionality was added. All the GUI was kept in one main class with some of the functionality spread out into sub-classes.

The GUI has different menus that allow the player to change what game mode is being played. This includes allowing the player to change how many human players there are, and the player can choose which AI algorithms they want to play the game. This allows the player to test different algorithms to see their effectiveness.

All the GUI is run by having each menu on a different JPanel; these are swapped out when the player wants to move between different menus. The game is also running as an instance on a JPanel which allows for the game to be quickly restarted if the player wants to do so.

There are two objects that are used to transfer data during gameplay. The first being a piece object; this object, which is stored in a 2D array, stores the value of the piece and its corresponding Oval piece. The Oval piece is used to track its location precisely on the game board. When the piece is placed on the board, an oval is created which has the pieces' exact X and Y coordinates on the

player's screen, this allows for more precise mouse clicking. This object is used to keep track of the current game board and is always saved in a 2D array format.

The second object that is used across the game is the move object. This object is used to save data about a piece's move. It stores the location of the piece before and after the move. It is a simple object and only has the data that it needs to have saved. This object doesn't contain any data on what piece is being moved or if it is capturing a piece. A move object will always be a valid move due to how they are made. This object is used to show what moves are valid for a player and when an AI is in play, it is used so the algorithms can choose a move to make.

The game itself is split into two main sections, DrawBoard and GameData. DrawBoard handles most of the actual game playing and the drawing of the game; it houses the mouseListener to track what the player wants to do and controls all the initialising of a game. GameData controls most of the checking of movement; it contains all the move finding and validation. When an AI is playing the game, the AIManager keeps track of which algorithm is being used and when each one needs to make a move.

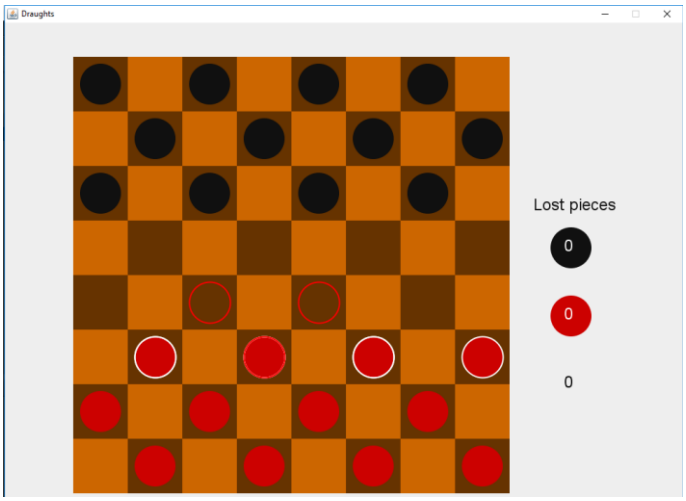

Figure 5-5 Screenshot of the game

Figure 5-5 shows a screenshot from the game, it shows the player highlighting one of the pieces that can be moved this turn which can be seen by looking at the red piece highlighted in red. This screenshot shows different aspects of the game that have been implemented, with the highlighting of possible moves, and where those pieces can be moved to. There is also the lost pieces counter on the right-hand side that shows how many pieces each player has lost so far. As can be seen in the requirements section 2.2, this follows the plan for what needed to be included in the game section of the project.

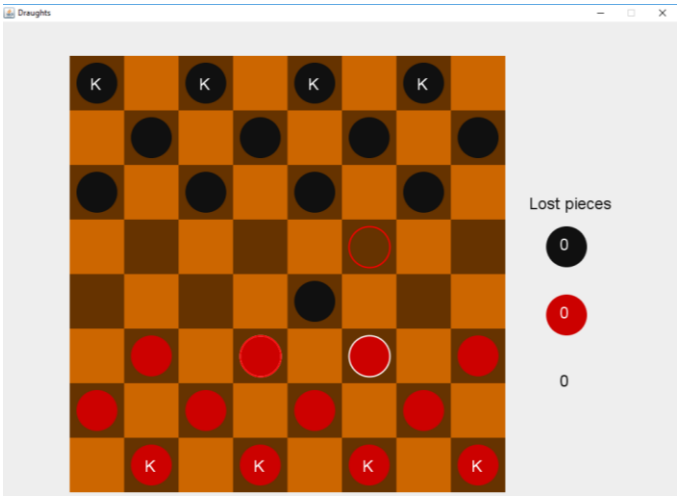

Figure 5-6 Second game screenshot

Figure 5-6 is a second screenshot, this shows other aspects of the game from figure 5-5. This shows an example of the difference between the king pieces and the standard pieces. It also shows an example of the force capture rule. Even though other pieces can be moved, there are x pieces that can perform a capture, so they must do so.

5.3.2 AI Management

When the time came to begin programming the algorithms, I had to add the framework that would allow the game to interact with the algorithms. This started by adding a basis for how the layout of each algorithm should be and how they would each return a move. Within the game there is a tracker for each player of the game, whether it be a human or an algorithm, then on each turn it references this to see where the next move needs to come from.

Each algorithm has the same main function within it to keep everything uniform. They all call the same function within the game to make the next move. This has been designed to recognise that an algorithm is making a move as there are differences as to how the game runs when it is a human's turn, such as how the visuals work.

5.3.3 Heuristic Search

When creating the Heuristic search algorithm, an attempt to keep as much of the algorithm modular was made. As a minimum, each rule was kept separate, and if possible, a rule was broken down into multiple methods that could be reused throughout the algorithm. This allowed for parts of the program to be reused, for example, the method that would check a list of moves for which ones were safe to make without the possibility of the piece being taken.

Initially to determine what the different rules should be and in what order to play them, I played many games and wrote down my reasoning and methodology for why a move was made. Using this I was able to break and translate them into simple rules that could be programmed. To get the rules I also looked at how other people play games and tried to design rules that would counter these moves. I started by adding the rules that were most straightforward making implementation and testing easier. This led to adding in rules where there were holes in the algorithm that made it ineffective against a human player. Rules were added until the algorithm was at a point that it was able to hold its own against a human player; once this was achieved, I moved on to the second algorithm.

Looking at the rules that were made they can be broken into two sections; rules that affect capturing and rules that affect movement. Each rule is ordered with regards to the importance of gaining the most value for the algorithm; once each rule has been run, it will either make a move if it is suitable or the next rule will run until a move is made.

There are two rules that work on improving how the algorithm works with capturing pieces. In order of importance, the first rule looks for the ability to capture opponent kings; the second is looking for double jumps. Both of these rules perform a simple task, but both highly improve the efficiency of the algorithm in gaining the most value out of a turn.

There are more rules that affect standard movement, some of which are interlinked with each other to create more complex rules. The first is interested in locating pieces that are in danger of being taken by the opponent on the next turn, this rule first checks if a piece in danger can move to a safe location, if this isn't possible the rule looks to see if a friendly piece can move to block the capture from taking place. This rule is one of the more advanced rules in the algorithm and once it was added it made a substantial change to how good the algorithm was. The following rules include moving a piece forward without moving it into a place that will get it captured, moving a piece so that it becomes a king, and the last rule tries to avoid moving the back row as this prevents the opponent getting kings.

5.3.4 Alpha-Beta Pruning

ABP works on the basis of Minimax tree search; this algorithm was started by first creating a working Minimax algorithm. This was then developed to create a better, more efficient algorithm that used ABP to improve the run time and therefore allow the tree searching to reach a higher depth.

The tree object is used to create a full game tree of the current game up to a specific depth. Each object contains the current game board of that state and the move that was made to get the game to that state. It also includes the score of the game state. It then has the option to store an undetermined number of children. This is used to create the tree allowing each node to have all its children added which then have their children added and so on.

The game tree is created using a recursive method that allows the player to enter a specified depth that will then loop to building the tree. An option was added to the GUI that prompts the player to specify a depth that will run within a reasonable time on the device they are using. As the player needs to be swapped each time the depth increases, within the method there is a check to see if the depth is currently an odd or an even number, even being the player that needs to make the next move. This allows there to be one method that calls itself instead of two methods one for each player calling each other.

During the tree creation process, the ABP algorithm is used to improve the original Minimax algorithm. Two values are used in this process, Alpha and Beta. Before an item is added to the tree, its score is checked to see if it is more than the previous Alpha if so Alpha now equals the current score. Then if Alpha is more than Beta, the branch following from the current move is not explored. If the opponents move is being added, then Alpha and Beta are swapped, checking if the current score is less Beta and if Beta is less than Alpha. This is used to reduce the number of items in the tree that are useless as their score will mean that another child from their parent will move its way up the tree instead.

After the tree is created, it needs to be analysed to find the best move for the player; this is where the Minimax method is used. There is another recursive method that is used to look through the tree, returning the values for the moves that are to be chosen at each stage up the tree until the top layer is reached. At this point, the highest value out of all the possible moves is selected as the next move.

```
private void makeDecisionTree(Tree layer, List<Move> moves, int depth, int depthLimit, int
alpha, int beta) {
    for (Move move : moves) {
        Piece[][] temp = dupBoard(layer.getBoard());
        //Generate the next GameBoard after the current is made
        temp = fakeMove(move, temp);
        List<Move> nextMoves = new ArrayList<>();
        //If it is a even depth, i.e. maximum depth
        if(depth % 2 == 0){
            //Generate a new tree and its score
            Tree nextLayer = new Tree(temp, move, scoreBoard(temp, maxPlayer));
            nextMoves = getMoves(minPlayer, nextMoves, temp);
            //If the score just generated is more than Alpha overwrite it
            if (nextLayer.getScore() > alpha) alpha = nextLayer.getScore();
            //Alpha is more than Beta don't go any deeper
            if (alpha >= beta){
                layer.addChild(nextLayer);
                continue;
            }
            if (depth < depthLimit) {
                //If the depth limit hasn't been reached continue going deeper
                makeDecisionTree(nextLayer, nextMoves, depth+1, depthLimit, alpha, beta);
            }
            //Add the current tree being made to the one above
            layer.addChild(nextLayer);
        }
        //Else it is a odd depth, i.e. minimum depth
        //Works the same as above but Alpha and Beta are swapped
        else {
            Tree nextLayer = new Tree(temp, move, scoreBoard(temp, minPlayer));
            nextMoves = getMoves(maxPlayer, nextMoves, temp);
            //If the score just generated is less than Beta overwrite it
            if (nextLayer.getScore() < beta) beta = nextLayer.getScore();
            if (alpha >= beta){
                layer.addChild(nextLayer);
                continue;
            }
            if (depth < depthLimit) {
                makeDecisionTree(nextLayer, nextMoves, depth+1, depthLimit, alpha, beta);
            }
            layer.addChild(nextLayer);
        }
    }
}
```

Figure 5-7 ABP code snippet

The code above is the recursive method that generates the game tree for the ABP algorithm. Whereas a standard method of using Minimax is to do the searching while generating the new nodes within the same method, I used a tree object that would then be searched through to get the next move. I chose to do it like this way as I wanted to design and implement a method of Minimax that was more unique, that required me to plan how it was going to work. Looking at figure 5-7, it can be seen that the method is split into two halves, as during the tree creation each layer swaps between maximising and minimising. The two halves work almost the same except for when ABP is used. During a maximising path, Alpha is the variable that is checked, comparing this to Beta it can be determined if the current nodes branch is worth generating and evaluating.

5.3.5 Random

For testing purposes and to help establish a baseline, a completely random algorithm was created. This was done to allow the tracking of progress of the Heuristic and ABP during their creation and to see how efficient the algorithms were against an opponent that will always play consistently. This was more useful when testing the Heuristic search as the more rules that were added would improve its effectiveness. Using the Random method to check the other algorithms allowed for games to run quickly compared to playing these games manually.

6 Artificial Intelligence Evaluation

6.1 Overview

To compare the Heuristic search and ABP algorithms, I implemented the ability for the two to play against each other without human interaction as this made it easier to record the results after each game finished, it also made testing quicker. I originally ran 100 games with a draw counter (the number of movements after a piece is taken before a stalemate) of 50. This draw counter value turned out to be too low as on many games the two algorithms would get down to both having 2 or 3 pieces and would keep moving the pieces back and forth. In the next version, the draw counter value was changed to be higher, allowing the algorithms to have more time to move towards each other trying to win the round. For gathering the full set of results, the draw counter value was set to 100. All of the results that contained the ABP algorithm the tree depth was set to 4, this was due to the limitation in processor power on the device that the program was run on.

Results that were recorded included who won the game or if it was a draw, along with the number of pieces each side had lost. This allowed for two different types of evaluation. The first being who won more over a set amount of games and the second being how effective they were at winning by showing on average how many pieces were lost by each player.

Over the development of the project, I ran multiple tests to see the progress regarding how useful the algorithms were. When I was developing the Heuristic algorithm, I did this testing against both a random algorithm and against human players. This was until the start of the ABP algorithm, during the ABP development I programmed it in sections so that it could be tested at different stages to track its progress.

To get a full set of results that would clearly show the effectiveness of each algorithm 200 games were played for each pairing of the three algorithms, which included swapping between which algorithm played as which colour. This lead to the following three pairings of heuristic verses ABP, Heuristic versus Random and ABP verses Random. For each paring there were 400 games played, compiling this into graphs gave a clear view of the effectiveness of the different algorithms.

6.2 Game results

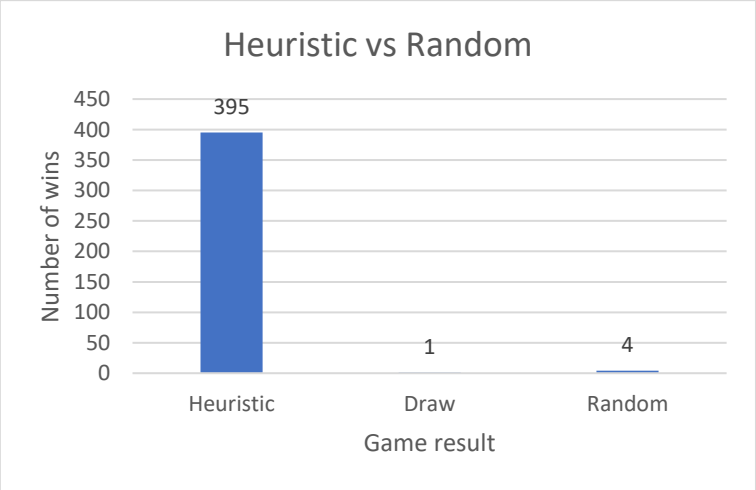

Figure 6-1 Results of the Heuristic and Random games

Looking at the first set of Heuristic versus Random, Figure 6-1. The results were so conclusive that the graph is mostly irrelevant, over the 400 games the Heuristic algorithm won 395 times, that equates to a 98.75%-win rate. The Heuristic algorithm was expected to win most of the games it

played against the Random algorithm, but it was not expected to win such a high percentage of them.

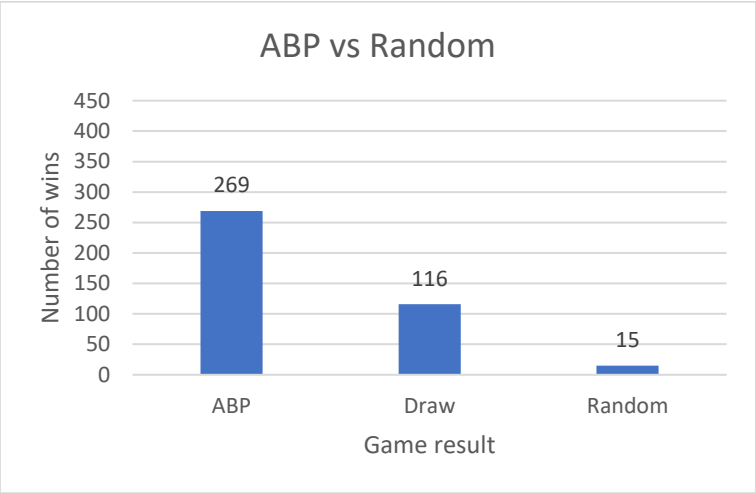

Figure 6-2 Results of the ABP and Random games

Next set of testing is ABP verses Random, Figure 6-2. These results are not as extreme as the previous set. Although the ABP algorithm still won most of the games played with a 67.3% win-rate, and a large proportion of those that weren't won by ABP were drawn. As can be seen in Figure 6-2, the Random algorithm won 15 games out of the 400 played. This is still a significant win for ABP, and it is closer to the kind of results that were expected compared to the previous set.

So far, the previous sets have just shown that the two algorithms are better than a completely random algorithm, which doesn't show easily how effective they are. Although comparing to a random algorithm does still give a good view of how good an algorithm is and whether the implementation of it is partially successful.

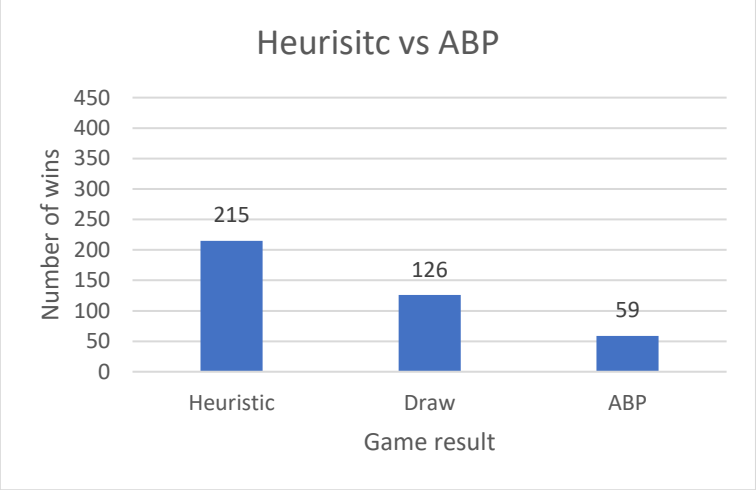

Figure 6-3 Results of the Heuristic and ABP games

The final set is the matchup of Heuristic and ABP, Figure 6-3. It can be seen in the graph, that Heuristic won over half of the games played against ABP with a 53.8% win rate, and that over a quarter of the time 31.5% the game ended in a draw. At face value, this shows that Heuristic is a more effective algorithm, although it is not significantly better as it does still lose or draw almost half or 46.3% of the time. The results may be a result of the limitation of the search depth of ABP; if this was set to a higher depth, it is possible that it would win or draw more often.

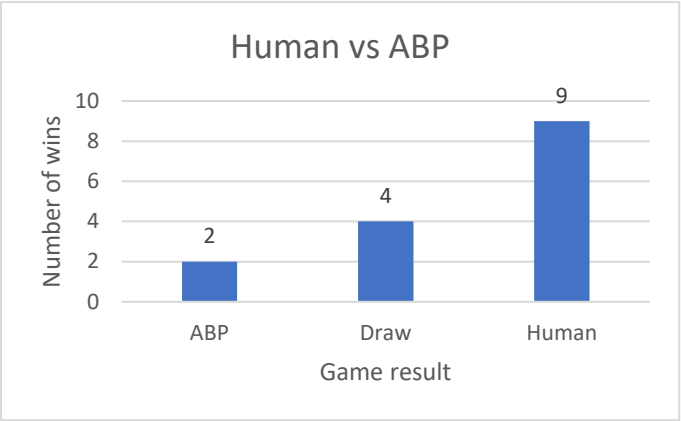

Figure 6-5 Results of the Human and ABP games

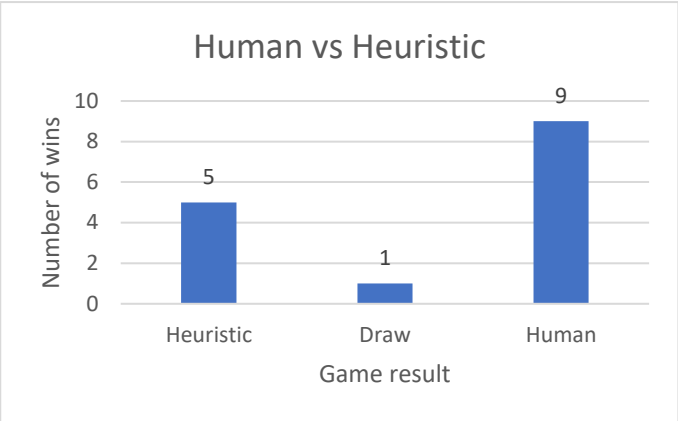

Figure 6-4 Results of the Human and Heuristic games

Results of the Human and ABP games Figure 6-4 and 6-5 shows the data from the games played against the two algorithms. Each of the data sets are out of 15 games. The total data set is a lot smaller than when two algorithms are played against each other due to the amount of time each game takes to play manually. The moves made by the algorithm are slowed down so the player can track them, and the time a player takes to make a move is significantly longer than the time taken by algorithms.

From this data, it shows that although the human player wins over half of the games played 60% in both cases, the algorithms still are able to win games or cause a draw. This shows that although the algorithms aren't advanced enough to always win against a human player, they are advanced enough to not lose every game. As a comparison, there were also 15 games played by the player against the random algorithm; the player won all 15 games.

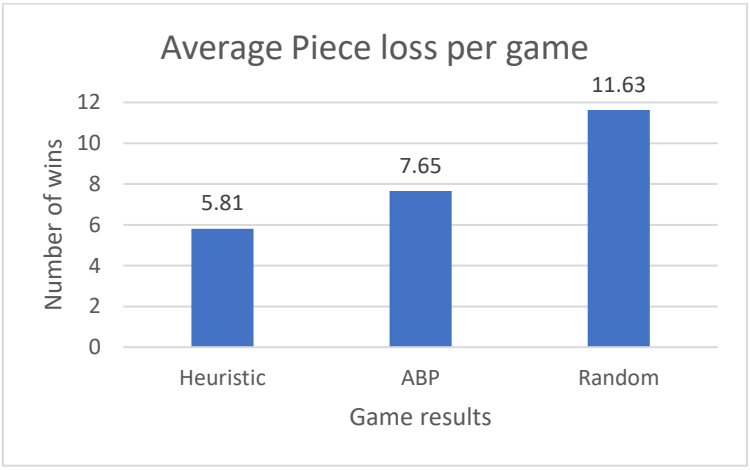

Figure 6-6 Average piece loss per game

Figure 6-6 shows the average number of pieces each algorithm lost in a game. This data is captured automatically because at the end of each round the number of pieces each player lost is recorded, these numbers are the averages created by each of the three pairings of the algorithms. Each bar represents the average gathered from 800 games. This data shows how successful each algorithm is at protecting its own pieces while being able to capture the opponent's pieces.

6.3 Algorithm comparison

When looking at the results and directly comparing the Heuristic and ABP algorithms the data capture proves which one is better overall. The Heuristic algorithm performs better overall in all

three sets of tests, against a human, against Random and against ABP. Although in the games against a human there were the same amount of lost games, there were more wins than ABP was able to achieve. Looking at Figure 6-6, the average piece loss also helps reinforce this result, as on average out of all the testing it ends each round with the lowest number of pieces lost.

There are multiple reasons why the Heuristic algorithm can be seen as better as can be seen in section 6.4. A significant flaw with ABP is that it can require a complicated evaluation function; without this, the algorithm can be very ineffective. The development of this function, however, can be quite difficult, so it can easily lead to this algorithm to remove not being as effective. There are other limitations of ABP that are mentioned in section 6.4. Another advantage of Heuristic is that is can be easily improved upon over time. Adding new rules only requires designing and programming them, whereas once the main functionality of ABP has been completed, improvements all take time to implement and merge into the existing algorithm.

This is not to say that the ABP algorithm is not good, but from the data collected from the testing of my implementation of the algorithms and the Heuristic search is better. Certain factors explain why the results are what they are, such as the program is running on a standard computer, without enough processing power to have ABP run to a depth that would allow true prediction of move tracking to take place.

6.4 Limitations of the algorithms

There are different limitations with each of the two main algorithms which affect the potential of the algorithms. On the whole, the limitations that are part of the algorithms are due to restrictions within the project Due to the time constraint and the need to have two algorithms, the time spent on development was split roughly between the two which ultimately limited the potential of each algorithm.

Looking at the Heuristic algorithm, the main limitations are that being a rule-based algorithm more rules can always be added. When programming it, the choice had to be made what to add, in what order and how long each rule would take to program. This led to many rules being planned but not being added due to their complexity and the amount of time it would take to program them. Examples of such rules included controlling the centre of the board which is seen as a position of power, seeking out opponent pieces near the end of the game, and when ahead in pieces using the force capture rule to gain a victory. To decide what rules needed to be added and which were not as important, I evaluated which ones would add the most value for the algorithm. I started adding rules and then looked at how the algorithm would lose most of its pieces and then made the rule that would counter that issue.

The original method of how the rules were designed was that the whole system worked on trailing IF statements, with the rules being in order and running through the IF statements until one succeeded in making a move. This had many issues with it; it required the majority to exist within the IF statement and didn't allow for rules to be skipped or called upon multiple times. A new design was made where each rule would be isolated, and all the rules could be stored into a table allowing for more flexibility. The final product resembles something similar to this, which each rule being its own method and a main method calling each one at a time until a move has been chosen.

The main limitation with the ABP algorithm is due to its evaluation method; the method that is used to give a score to each game state. The evaluation method is rather basic which leads to the algorithm not always choosing the best move. This was mainly due to the fact that it was more important to program a fully working algorithm that had a robust method of tree creation and tree

searching. The evaluation method was created to be sufficient for the task although it could be improved upon. The other limitation with the ABP algorithm is more to do with the whole algorithm itself, Minimax, and in extension, ABP. ABP isn't an ideal solution to Draughts, due to it requiring such a large amount of processing to truly complete the tree for each move, it has to be limited so that it is playable.

The ideal solution to creating an AI for Draughts is to use Machine Learning. Using a static algorithm that cannot learn will always be a limitation as it cannot improve without manually programming more rules or better methods etc. However, the trade-off for using Machine Learning is the long development and learning time it takes to make such an AI. When choosing what kind of AI, I wanted to use for this project, I decided that it would better to be able to make two algorithms that could be evaluated against each other rather than having one learning algorithm.

7 Project Planning

For the project planning of this project, I used software programs Jira Kanban and Git version control. Jira is a schedule system used for tracking issues. It allows the categorization of different types of issues and assigning users tasks. I used it to track tasks for different sections of my project and to help plan and identify work assignments to be done. Git is used to track changes and to keep a backup of work. Both were used to keep track of the progress of work and to help keep me focused on finishing individual tasks and the project.

Jira has a complex issue system that allows a lot of flexibility for the planning and tracking of tasks. Epics are used to track large portions of the project, for this project the game and the different algorithms each had an epic to separate the tasks. Each epic contains a mixture of tasks, stories and subtasks. Each one tracks a different feature or task for the project and is used to track the progress of it.

Looking at the path of an issue, it starts by creating an issue, after assigning it to an epic it appears in the backlog. I used the backlog as a planning section filling it up with issues that may be moved to the next stage. The Kanban board has three sections; to do, in progress and done. Different tasks can be allocated to each section at the same time and once a task is moved to the Done section the time spent on the issue can be set.

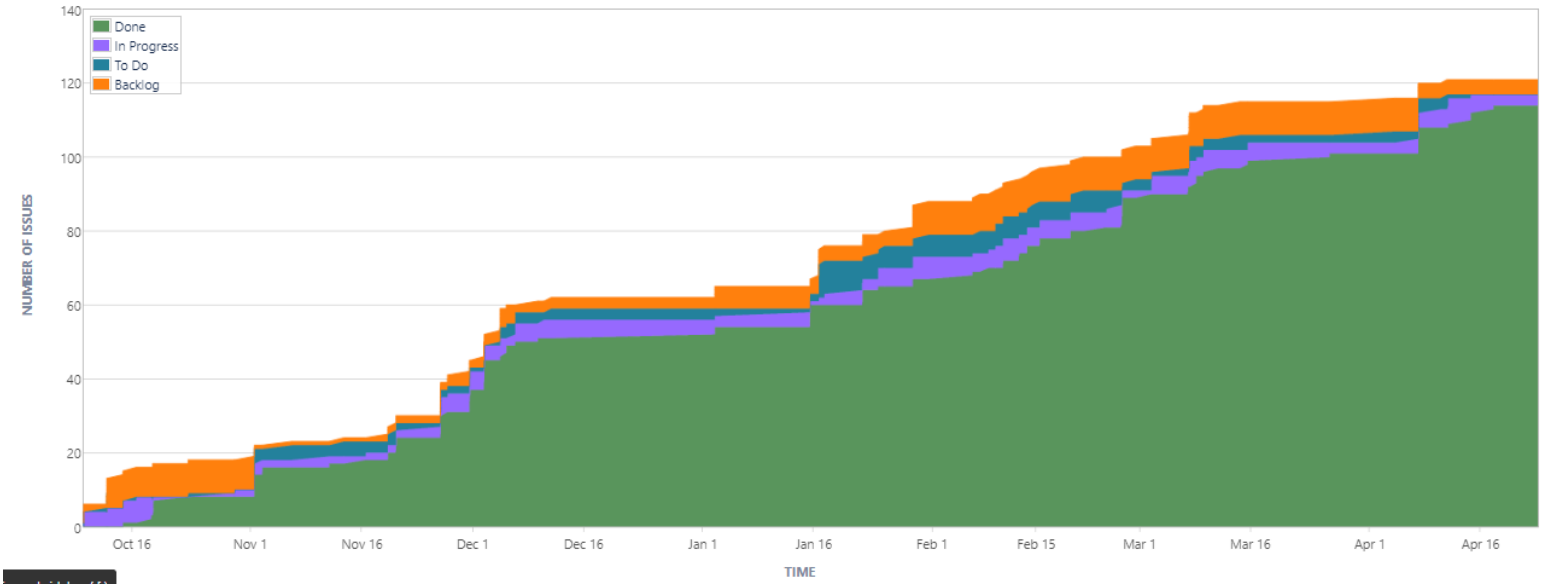

Figure 7-1 Jira issue graph

Figure 7-1 is a graph of my Jira Kanban issues throughout this project. It shows quite a gradual progression of issues moving between the four categories in figure 7-1. Although throughout there are occasions where there is a jump of work expanding, this happened when I planned a new set of issues and add them all at once.

In many cases issues were either overestimated or underestimated, meaning that the time an issue was expected to take was hard to predict how complex a task was from the surface. Once a task was started it was often split into smaller tasks that were more manageable, this became less of an issue as the project continued and I got better at predicting how complex tasks were.

Just like any other project this project also has a certain level of risk management to consider. With a programming project that has many complicated aspects to it, time management is important to ensure that all parts of the program work. When I was creating the two main algorithms, I had to ensure there was a balance to how long I spent on each to ensure that they were equally matched, if more time was spent on the Heuristic search than ABP for example, then it would be sure that the Heuristic search would be more efficient.

8 Conclusion

In conclusion, I am pleased with the outcome of the project as a whole and with what has been achieved, considering at the beginning of this project I only had a basic knowledge of AI. I have throughout the project's duration learnt a lot about both the theory and the implementation of different AI algorithms. Accepting that Draughts is a basic game, and I had no previous game development experience I am proud of what I was able to implement.

The final results show that in my implementation of the two main algorithms, in this particular instance the Heuristic search was more effective. This algorithm has a good degree of adaptability, for details see section 6.3 if further development was undertaken it is expected that the algorithm could be easily improved by adding new rules.

When referring to the original game requirements detailed in section 2.2, the contents of the report demonstrate these requirements have been met. Figure 5-5 and 5-6 in section 5.3.1 show that the game follows the requirements set out at the beginning by establishing how the game should work and the features it should have. The report identifies in several sections that there are three AI algorithms. These have been programmed so that the game allows a player to play against another human, against an AI or allow two AIs to play against each other, as covered within section 2.1 Aims.

When reviewing the goals set in the initial report, several original goals have been followed, completed and expanded upon. The plans for the game have been followed and implemented. However, the goal for the algorithms changed with the plan to create two algorithms that would then compared against each other added later into the project plan. The choice of algorithms also changed since the initial report as an outcome of the research undertaken within the Literature Review, with Heuristic search becoming the focus for the main algorithm along with the choice of ABP as being ideal for a direct comparison of different methods to Heuristic search. The initial plan did not contain detail plans of the programming, as the project progressed the plan was adapted as the programming developed and transitioned into the final version.

Once the development of the game reached the point of allowing two human players to play a complete game of Draughts, it was necessary to research how to implement the algorithms, which for Minimax/ABP was a simple task as it has been implemented countless times. When it came to programming ABP, although I was influenced by other methods, I made sure I designed my own method of implementing it. This is why I decided to use a Tree object to generate the game tree; this created a challenge that I had to overcome that would distinguish my finished product from others. This was not as much of an issue when it came to the Heuristic search as there were very few actual examples of this method, mainly just research on how it could be done. Using this research, I planned how the algorithm would be implemented while making many changes during the actual implementation process.

There are things about this project that could be improved in the future. The goal of developing the game was to get it to the point that had a basic level of functionality for the user to interact with. Time could be spent enhancing some smaller features that would make the game better for a user to play, such as it is obvious which piece was moved by an algorithm. The algorithms themselves have some potential for improvement, with many rules not being added to the Heuristic search due to time restraints. With some more work, they could be added to create a more advanced algorithm. There are also some small bugs that didn’t occur very often that were hard to track down their source.

However, as they didn't cause a problem for the player, they were not deemed a priority within the time constraints of the project.

On reflection, there are things that if given additional time I would consider doing differently. The testing methodology I adopted gave results that overall were not as helpful as they could have been. As I implemented the algorithms, I have gained significant knowledge regarding how they work and what their flaws are. If I were to do the project again, I would set up other human testers, although by the time this had become apparent it had become too late to design and implement a testing process to allow it to be used in this project.

If I repeated this project, I would aim to try and start the implementation of the algorithms earlier, due to the time constraint limit it became a rush to get both the algorithms in working order, I feel that this may have limited their potential. With more time both Heuristic search and ABP could have been improved to become more advanced.

Overall the I am happy with the outcome of the project and the opportunity it has presented me. On a personal level, I have gained valuable experience that I can use in the future as I now have a much better understanding of Algorithms, Programming, Research methods, Project planning and an appreciation of the importance of time management.

9 Bibliography

[1] “Deep mind AlphaGo,” DeepMind, 06 12 2018. [Online]. Available: https://deepmind.com/blog/alphazero-shedding-new-light-grand-games-chess-shogi-and-go/. [Accessed 09 04 2019].

[2] “Europe Draughts,” European Draughts Confederation, [Online]. Available: http://europedraughts.org/edc/general-rules-of-draughts/general-rules/. [Accessed 03 04 2019].

[3] “Data Protection Act 2018,” [Online]. Available: http://www.legislation.gov.uk/ukpga/2018/12/contents/enacted. [Accessed 12 04 2019].

[4] “BCS Code of Conduct,” British Computer Society, [Online]. Available: https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/. [Accessed 12 04 2019].

[5] “Terms of use,” Oracle, 22 04 2016. [Online]. Available: https://www.oracle.com/uk/legal/terms.html. [Accessed 11 04 2019].

[6] “Terms of use,” JetBrains, 18 05 2018. [Online]. Available: https://www.jetbrains.com/company/useterms.html. [Accessed 11 04 2019].

[7] “247checkers,” 247 Games LLC, [Online]. Available: https://www.247checkers.com/. [Accessed 05 02 2019].

[8] E. Egilsson, “CardGames.io,” CardGames, 17 05 2018. [Online]. Available: https://cardgames.io/checkers/. [Accessed 05 02 2019].

[9] M. Doležal, “Checkers with Artificial Intelligence,” Czech Technical University in Prague, 2015.

[10] L. Prechelt, “An empirical comparison of seven programming languages,” *Computer,* vol. 33, no. 10, pp. 23-29, 2000.

[11] A. A. Elnaggar, M. A. Aziem, M. Gadallah and H. El-Deeb, “A Comparative Study of Game Tree Searching,” *International Journal of Advanced Compiter Science and Applications,* pp. 68-77, 2014.

[12] R. E. Korf, “Optimal Path-Finding Algoirhtms,” in *Search In Artificial Intelligence*, New York, Springer-Verlag, 1988, pp. 223-267.

[13] P. N. Stuart Russell, Artificial Intelligence: A Modern Approach, Third Edition, Prentice Hall, 2009.

[14] J. Schaeffer, “The history heuristic and alpha-beta search enhancements in practice,” *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 11, no. 11, pp. 1203-1212, 1989.

[15] M. Pantic, “Intelligent behaviour understanding group,” [Online]. Available: https://ibug.doc.ic.ac.uk/media/uploads/documents/courses/syllabus-CBR.pdf. [Accessed 12 02 2019].

[16] M. J. Apter, The Computer Simulation Of Behavior, Hutchinson, 1970.

[17] L. D. Xu, “Case based reasoning,” *IEEE Potentials,* vol. 13, no. 5, pp. 10-13, 1994/1995.

[18] “Package javax.swing,” Oracle, [Online]. Available: https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html. [Accessed 17 04 2019].

[19] “Package java.awt,” Oracle, [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html. [Accessed 17 04 2019].

[20] “Package java.util,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html. [Accessed 17 04 2019].