School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Game tree learning enhancements for multiplayer Chess

by

William James Archinal

Thesis submitted as a requirement for the degree of

Bachelor of Engineering (Software Engineering)

Submitted:	24 October 2016	Student ID:	z3376581
Supervisor:	Dr Alan Blair	Topic ID:	3379

Abstract

This paper investigates the development of an Artificial Intelligence (AI) player for the multiplayer Chess-like game Duchess. Adjustments and extensions are made to traditional Chess-playing techniques to develop the player, and experiments are conducted on game tree search optimisations to improve its learning potential. These experiments determine that the history heuristic best suits the game of Duchess, and other optimisations such as lookup tables and incremental evaluation are also helpful in increasing the depth of the game search tree.

The TreeStrap machine learning algorithm is used to train the AI player to a competent skill level via self-play. The player is trained in various stages, and each of these stages are competed against each other to evaluate the utility of the feature sets they use. Various techniques are additionally evaluated to reduce the frequency of draw results in an effort to improve future training, including stochastic game-play and new sets of features.

Acknowledgements

Thank you to Alan for your ongoing mentorship. I am extremely fortunate to have a supervisor who consistently makes time in his schedule to welcome and encourage my ideas and input. The success of this project is indebted to your endless and contagious enthusiasm.

Thank you to Hamish for being a comrade and sounding board throughout this project. Having consistent opportunities to discuss new ideas and hear second opinions has been an invaluable asset.

I'm grateful to Nick for acting as an impromptu guinea pig for my presentation and for helping me play-test against the AI. Your interest in the game was a breath of fresh air.

Finally, thank you to Lucy for your endless help and encouragement. Your support has been both academic and personal, and you've turned stressful times into manageable ones, and routine occasions into joyous ones.

Contents

Chapter 1

Introduction

The age of the smart phone and the increased proliferation of hand-held computing devices has led to a vast increase in multiplayer gaming. As more people are playing games there has been an increased demand for sophisticated Artificial Intelligence (AI) players to provide an immersive and on-demand multiplayer experience.

This increased demand has been met with consistent breakthroughs in AI game play. In March 2016 the world Go champion, Lee Sedol, was defeated by AlphaGo in four games to one, the first time a computer was able to defeat the human world number one.

Developments in game playing AI are also directly applicable to real-world situations. In particular, improvements in adversarial planning in more complex spaces could inform strategy with war games and battle simulations. It is therefore important that we strive to apply AI to more complicated games than were previously possible. In particular, because Chess and Go have been "solved" we may see an increase in demand for AI that works well in games with more than two players.

This paper will examine the mutliplayer Chess-like game "Duchess" with six players (three players per team). Because of its wider problem space it is harder for AI players to plan ahead and Duchess has therefore not yet seen an AI player who could perform

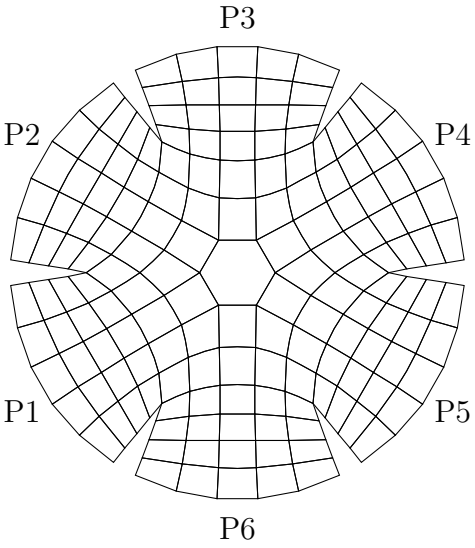

Figure 1.1: A six-player Duchess board

at a master level.

1.1 Duchess

1.1.1 Outline of the game

Duchess is a Chess-like game that is played out on a hexagonal board modified to include up to six 4x5 square flaps attached to its vertices. The board is populated by two teams of up to three players each.

Each player controls the pieces used in traditional Chess (King, Queen, Bishop, Knight and Rook) as well as 5 modified Pawns (which may move one square in any straight line, and take on any diagonal), a Duchess (which moves like a Bishop or a Knight), a Fortress (which moves like a Rook or a Knight) and a Wizard (which enables pieces to teleport from squares adjacent to a friendly Wizard to other friendly Wizard-adjacent squares).

At the centre of the board is the Vortex, a hexagonal place which, when entered by a

pawn, promotes the pawn, thereby replacing it with any piece belonging to the player which has already been taken.

The aim of the game is to put each of the opposing players in Checkmate. Unlike traditional Chess, there is no official concept of Stalemate. While a game between humans may only be officially drawn upon unanimous agreement, because of the complexity of the game we also maintain that a game played by an AI is considered a draw when either:

- The game has been played for more than 660 turns (110 turns per player).
- In the last 24 moves each of the 6 players has made moves in the sequence M, M', M, M' where M is one move and M' is another move which is the inverse of M (such as moving a Rook back and forth).

The full rules of the game can be found on the Duchess website[1].

1.2 Applying AI to Duchess

An ideal AI player for Duchess shares many similarities with existing Chess playing agents. When it is the AI's turn to make a move, it will imagine all possible legal moves it can make from the board state and evaluate the strength of the position that leaves them in. Advanced AI players plan many moves ahead in this way, with their strength being determined largely by two factors:

1. The accuracy of their evaluations of how favourable a certain board state is for them, determined by a heuristic.
2. How far ahead they are able to plan (since a board state that looks favourable three moves from now may actually result in disaster two moves later).

Modern Chess programs typically employ an α-β search to look ahead, and evaluate the strength of board states using a linear set of features with pre-determined weights[2].

1.2.1 Differences from Chess

Some common techniques employed by Chess AI are not applicable to Duchess. A key example is the quiescent search used by many Chess programs[3]. While this technique may improve the search capabilities of Chess players, its unsuitability is discussed in Section 2.1.

Another useful technique for games with smaller branching factors is to generate early-game or end-game move sequence databases. The merits of this are further discussed in Section 2.1.3.

1.2.2 Features

A linear set of features can be used with pre-computed weights as a single layer neural network to determine the strength of a board state. The feature sets used in this work include combinations of the following types:

1. Material: Which pieces remain on the board
2. Partner-Wizard: How many Wizards still exist on each team
3. Bias: An always-active feature that offsets training bias (further explained in Section 2.2.2)
4. Pawn-Distance: How far each pawn is from the Vortex (only used in networks where Piece-Square weights are not used)
5. Check: Whether a certain player is in Check
6. Checkmate: Whether a certain player is in Checkmate
7. Attacking: Whether a piece is attacking a particular enemy piece
8. Defending: Whether a piece is defending a particular friendly piece
9. King-Distance: How many moves separate a piece from each of the enemies' Kings

10. Piece-Square: A particular piece being on a particular square (e.g. Queen on Vortex)

While this feature set contains thousands of features, evaluation is rapid because for any given board state only a very small subset of features will be active. The board state is therefore evaluated by an informed multiplication of features and weights rather than a naive dot-product of the two sets.

1.2.3 Planning and lookahead

In order to determine a move to play, the AI must use the board evaluation function alongside a search through a tree representing all possible game states branching from the current state. The evaluation is performed from the perspective of the player whose turn it is to play using a negamax implementation of an α-β search. Because Duchess presents such a large branching factor (well over 100 in some cases due to the mobility afforded by Wizard pieces), several optimisation strategies must be used to improve the ability of the AI to plan ahead. The branching factor of a typical game of Duchess is illustrated in Figure 1.2.

1.3 Training an AI Player

The AI player's ability to accurately analyse the strength of a particular board state is not only dependent on the features used in the evaluation, but also on the weights applied to each feature. Ultimately a sigmoid function can be applied to the weights to produce an estimated probability of winning from a certain game state. These weights must be developed and improved over time to improve the AI's ability to play the game.

While humans would learn by playing against multiple opponents in multiple games and tweaking their strategy, with a more niche game like Duchess this approach would be inefficient and impractical. Temporal Difference (TD) learning can be applied to train these weights via self-play.

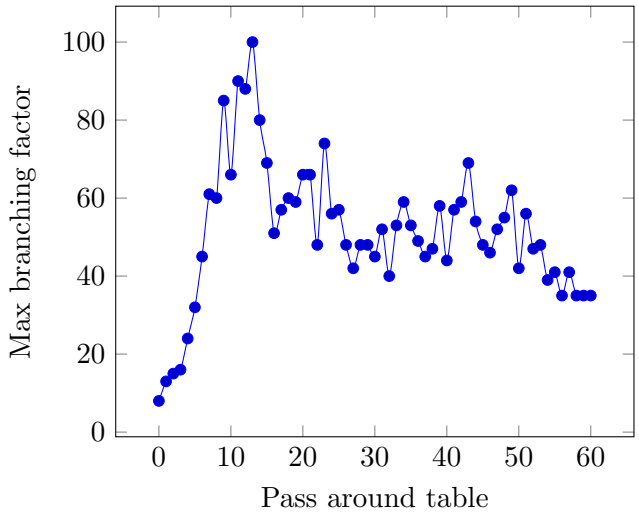

Figure 1.2: The branching factor in a sample game of Duchess. The points plotted represent the maximum branching factor encountered in one set of turns around the table.

As the AI player evaluates the board states at each node in the α-β search tree it can use leaf nodes of the search tree to train weights higher up the tree. There are several strategies that can be used to perform this training, such as TDλ and TreeStrap. The merits of each of these strategies is discussed in Chapter 2.

Chapter 2

Background

The issue of applying an AI player to Duchess is not an entirely new one. Significant work has been done in the past to introduce AI to two-player adversarial games, and a proof-of-concept Duchess player was developed in 2015 by Real and Blair[4] for the four-player version of the game.

While it was able to play a competent, limited style of Duchess, the 2015 Duchess player was limited to the four-player variant of the game and was unable to achieve high-level play. Using lessons learned from past research, we will attempt to address some of the conclusions made by [4] and train a sophisticated Duchess player for the six-player edition of the game.

2.1 Past Research

The development of artificially intelligent players in adversarial games has been influenced by a great deal of research. Samuel's introduction of machine learning to the game of Checkers in 1959 proved the feasibility of training an AI to play a game using temporal difference learning[5]. Since the 1990s a variety of AI players have defeated world champions in two player adversarial board games such as Checkers, Backgammon and Chess[6, 7, 8, 9].

At the same time, a great variety of research has gone into optimisations for AI players. The aim of this work is to experiment with optimisation assumptions made by [4] and to improve the efficiency of the board evaluation and planning stages, thereby improving the ability of the player to plan ahead in the game.

A great deal of past research, both in AI player development and in adversarial planning optimisation, has been considered in the development of this Duchess player. We consider this research to be helpful to us because of the parallels between Duchess and other games like Chess and Checkers. A key difference between Duchess and two-player games, however, is that Duchess is more accurately a two-team game, where teams consist of two or three players. Because all players on a team share a common goal, however, teams can be treated somewhat similarly to a single entity.

Chess in particular has been successfully played by a variety of AI players who employ a minimax search (such as *KnightCap*, by Baxter et al)[8], or an α-β search (such as *Meep*, by Veness et al)[10]. These players typically use a set of features weighted by a set of weights in order to determine the favourability of a particular board state (specifically, an estimation of the chance of winning the game, given a board state).

Similarly, artificially intelligent Checkers players have seen great success by employing an α-β search, as well as some miscellaneous optimisations, to plan a line of play. Most notably, Schaeffer was able to create the Checkers program *Chinook* in 1992, which went on to defeat several world champions in 1994[6].

Ultimately, the success of these players is determined by two main factors:

1. The distance to which they are able to plan ahead
2. The accuracy of their estimated chance of winning on a given board state

The former is an issue of optimisation and efficiency, while the latter relies on machine learning techniques. Both of these areas of study have therefore been researched in the planning of this Duchess AI player.

2.1.1 Planning Optimisation

While a minimax search obviously lends itself well to planning in competitive games (indeed, it was used very successfully by *KnightCap*[8]), a natural improvement is the α-β search, which intelligently determines when a particular branch of the search tree is no longer worth evaluating concluding that it would be irrational for any of the subsequent moves to be made. The α-β search is used commonly in adversarial planning, and was highly successful when used by *Meep*[10].

The negamax implementation of the α-β search is an appropriate choice for Duchess, as the moves that benefit the player equally disadvantage its opponent. This implementation was proven to be feasible by [4] in their Duchess player.

The key benefit of the α-β search (and subsequently negamax) is that instances of pruning parent nodes in the tree greatly limit the complexity and size of the search tree as a whole. As such, several heuristics exist to influence the order in which leaf nodes of the tree are expanded in the hopes of identifying cutoffs earlier and more frequently. Generally speaking, these heuristics "improve the order in which branches are considered at interior nodes"[11], as evaluating higher-value moves first can greatly limit the size of the evaluation tree.

A common heuristic in Chess programs called the "killer-move heuristic" was also employed in Real and Blair's 2015 Duchess player. The killer-move heuristic was introduced by Akl and Newborn in 1977[12]. The core idea behind this is that moves which cause a cutoff in one section of the search tree are likely to cause a cutoff if evaluated along another path of the tree.

In 1989 Schaeffer introduced a popular extension on the killer-move heuristic called the "history heuristic"[11]. While similar to the killer-move heuristic, the history heuristic keeps track of a weighted sum of how many cutoffs a move has caused in the evaluation tree, and prioritises moves which create more cutoffs. Schaeffer summarises this by explaining "a history is maintained of how successful each move is in leading to the highest minimax score at an interior node".

Another key heuristic often used in Chess and Checkers is the "best-reply heuristic" introduced by Uiterwijk[13] (also known as the "counter-move heuristic"), which operates on the assumption that some moves have a natural reply, and that a move b which worked well as a reply to a move a is likely to be a good move if a is encountered elsewhere along the search tree.

It is worth noting that Chess players often use a popular extension to the α-β search called quiescent search. Quiescent search involves "expanding the search just enough, and only just enough, to avoid evaluating a position where tactical disruption is in progress"[3]. Unfortunately, this is not feasible in Duchess, as a tactical disruption (e.g. a player being in check or a major piece threatened) is almost always in progress because of the expanded player number and the presence of the Wizard piece. As such, the quiescent search extension would result in an impractical lengthening of the search tree.

2.1.2 Training via Machine Learning

In order to produce reliable board evaluations it is important that the AI player is able to closely approximate its chance of winning at any given board state. This chance is determined by an evaluation function which typically includes several categories of features, each of which has its own weighting.

The combination of a linear set of features along with weights is a common and successful approach to board-state evaluation, having been used in many of the AI players mentioned previously[8, 10, 4, 14, 9].

In order to learn the weights to apply to each of these features, a common approach is to use some variant of Temporal Difference (TD) Learning that can be employed by the AI to train the weights via self play. After its introduction by Samuel in 1959 for Checkers[5], variants of TD Learning have been used by Tesauro's Backgammon player[7] (TD-Gammon), Baxter et al's *KnightCap* Chess program[8] (TD-λ) and Veness et al's *Meep* Chess program[10] (TreeStrap). Real and Blair also used TreeStrap in

their 2015 Duchess player, and reported some success[4]. In 1999, Beal et al observed that TD learning was particularly effective for developing weights for piece-square features for Chess[15].

Because Duchess is a niche game without widespread adoption, it is infeasible to plan to train the AI player by playing highly skilled opponents, as Baxter et al did for their Chess player. Instead, the player will need to train itself via self-play.

The work of Veness et al showed that a Chess player was able to play at grand-master level after learning via self-play, beginning from only a small set of pre-computed material weights[10] (Baxter et al noted that attempting learning by self play from a zero-knowledge state ran the risk of premature convergence of weights, reinforcing a suboptimal player[8]).

Another core benefit of the TreeStrap learning algorithm discovered by Veness et al was the improved rate at which the player was able to learn. Because the player was training against the evaluation values of all child nodes in its own tree, as opposed to training against the root node of the subsequent move's tree (as in Samuel's Checkers program), the player was learning from a richer source of information.

In addition to learning the optimal values for weights, it is important to identify aspects of the game that lend themselves well to being used for features. Naive Chess players often only consider the raw total value of their pieces, however sophisticated Chess programs use several extra sets of features, each consisting of potentially many thousands of individual features. A list of features employed in this project is included in Section 1.2.2. It should be noted that King-distance features have not been tested before in Duchess, and their utility will be evaluated as a part of this work.

2.1.3 Miscellaneous optimisations

Opening and End-Game databases

In order to optimise their early game and late game play, some AI players make use of a database of opening move sequences (and their best replies) and end-game play. IBM's *Deep Blue* and Schaeffer's *Chinook* both used early game move databases, and *Chinook* was also famous for its extensive end-game database[9, 6]. Because Duchess has a significantly wider branching factor than Checkers and Chess, and because the ultimate aim of the project is to release a playable version of the AI for consumer devices, it is impractical to maintain and employ these move databases.

Lookup Tables

Another common optimisation in the evaluation of a board state is to use lookup tables to assist in determining which piece-square features should be switched on[10], as well as to instantly determine the piece-distance values. These tables greatly speed up the α-β search, and can therefore assist in both the depth of learning and the player's ability to plan ahead. Furthermore, in Duchess, since the board is not arranged as a simple grid, rather as a rose shape, using a lookup table which stores pre-computed square adjacencies, and Knight-moves from each square may speed up play. This hypothesis will be validated as part of the work.

Move Ordering Functions

In cases where the search heuristics do not result in a cutoff in the α-β tree, the player would traditionally naively evaluate all of the leftover legal moves in no particular order. A common family of optimisations for this process are move ordering functions. The purpose of these functions is to sort the list of legal moves to be evaluated, preferring the evaluation of promising looking moves first. Since these functions have not been

examined before, this work will examine only one such function to indicate whether or not these hold promise for future work.

In this case we will examining the Most Valuable Victim (MVV) ordering. This ordering operates similarly to how a naive chess player may choose a move, by examining the moves which result in the capture of the highest-value enemy pieces first. In this case the value of the pieces will be determined by the related material weights used by the player.

Incremental Evaluation

A key optimisation employed by Real and Blair was to evaluate the board features incrementally "by adjusting only those features which have changed as a result of the latest move"[4]. This "incremental" evaluation was referred to as "cumulative" evaluation in [4]. While a naive evaluation method simply re-evaluates everything about the board state on each turn (which we refer to as "De Novo" evaluation), incremental evaluation works as follows:

$$Features(B_{t+1}) = Features(B_t) + Feature\Delta(M_t)$$

Where B_t represents the board state at time t, and M_t represents a move on the board state B_t.

To calculate $Feature\Delta$ we re-evaluate only the aspects of the game that could have changed in a certain move. For example, if we move a piece p owned by player P and take an enemy piece q owned by player Q we would only:

1. Update the material features removing q from the list.
2. Recompute attacking/defending vectors from p according to where it can legally move.
3. Clear all attacking/defending vectors extending from q.

4. For all attacking/defending vectors that used to terminate at p, retrace them and update them to now potentially reach a new piece r.

5. Switch all attacking/defending vectors that were terminating at q to instead terminate at piece p (such that an attacking vector becomes a defending vector and vice versa).

6. For each king in the game that is being attacked, mark that player in Check.

7. For each player who is marked as being in Check, re-evaluate whether or not they are in Checkmate.

8. Update other relevant features (King-distance, piece-square) using lookup tables.

To our knowledge, this approach is not used in Chess players, as on a standard Chess board one would need to trace Rook, Bishop, and Knight moves at both the origin and destination, which is roughly comparable to computing the move vectors of two Rooks, two Bishops and two Knights non-incrementally. In addition to this, there are three extra computational overheads:

1. We must check whether the potentially-attacking pieces encountered can actually attack in the manner specified.

2. In the case of a King or Pawn, we must check that the distance between the two pieces is only one unit distance, and that it is attacking in the correct direction (e.g. the Pawn must be moving forward diagonally).

3. We must also check for attacks are are blocked or unblocked by the move.

As such we hypothesise that incremental evaluation may not improve the performance of the search in Chess, which therefore makes it important that we justify our decision to use (or not use) the optimisation in Duchess.

Round-Off Error Avoidance

The implementation of the α-β used by [4] includes a concession to avoid round-off errors. It was hypothesised that because board states would be evaluated to many significant figures, round-off errors associated with the multiplication of `double` types may cause significant problems when checking for cutoffs in the search tree. As such the following adjustment was made to the cutoff conditions:

Traditional α-β search algorithm:
$if(\alpha \geq \beta) : cutoff()$

Algorithm adjusted to mitigate false-negative cutoff conditions:
$if(\alpha + \epsilon \geq \beta) : cutoff()$
Where $\epsilon = 10^{-8}$

Although this adjustment was hypothesised to reduce the number of false-negative cutoffs, its efficacy was not tested. We will validate this hypothesis as part of the experiments in this work.

2.2 Reflection on past work

2.2.1 Improving Real and Blair's 2015 Duchess player

As a result of the promising results of Real and Blair's 2015 four-player Duchess player it is natural to examine the design decisions of the 2015 player as a starting point for planning development.

The most apparent issue faced by the player during training was that the α-β search tree was only able to be evaluated down to a depth of 5 during the mid-game. This training was for the four-player variation of Duchess, so this meant that the player was able to evaluate its move, all other players' moves, and then its move once more.

During the end-game, because of the reduced branching factor, search was extended to a depth of 7. The authors acknowledge that in the future they "hope to increase the depth of lookahead, which may ... lead to improved learning"[4]. This shallow search depth resulted in a high number of Duchess games being "drawn" (games between the stage 3 network and the stage 2 network had a draw rate of 9%) which may have been able to be overcome if players could search further ahead. A draw was defined in [4] to occur after a sequence of 200 consecutive moves without capture or promotion. It is therefore crucial that we are able to optimize the planning such that the player can plan further ahead.

More efficient planning is even more important in the six-player version of the game, which has a larger branching factor than the four-player version, as a result of the increased number of squares and Wizard pieces. Furthermore, as the six-player version of the game begins with 50% more pieces on the board, there are significantly more attacking and defending features to be considered, and the process for determining legal moves is therefore more computationally intensive (as no move may leave the player's King in Check).

As such, we must therefore attempt to improve on the search optimisations used by the 2015 Duchess player. The main optimisation used by this player was the killer-move heuristic in the α-β search tree. Another key feature was to evaluate the board state in the tree incrementally, as an alternative to De Novo evaluation, as outlined above.

To evaluate the usefulness of these optimisations we will quantitatively test different combinations of optimisations against each other, and record their results. In particular we will examine the difference between incremental and De Novo evaluation, as well as heuristics including best-reply, history and killer-move. The combinations of the optimisations that will be tested are further outlined in Section 4.1.4.

In addition we will also experiment with introducing a move ordering function, and will quantitatively determine its effectiveness, as well as the effectiveness of lookup tables and round-off error avoidance.

Another way in which we will mitigate the risk of stagnation resulting in a draw is by introducing King-distance weights. It is hypothesised that most of the draws took place because each player had only a few low value pieces (such as Bishops or Pawns), and with the limited look-ahead was unable to determine any path of play which results in an improved board state (either by promoting a pawn on the Vortex, or attacking an enemy King). We will therefore introduce King-distance weights to subtly encourage players' pieces to approach the enemies' Kings, especially in the end game. To speed up the evaluation of King-distance features, we will employ extra lookup tables of the which contain solutions entries of the form $Distance_{pij}$, where p is a type of piece, i and j are two positions on the board, and the $Distance$ is the number of moves from the piece p that separate the two positions.

Finally we will evaluate the effectiveness of adding a stochastic component to the evaluation. To discourage cases where each player makes repetitive sequences of two moves M and M', where M' is the inverse of M (e.g. moving a Rook back and forth), we will experiment with introducing a 5% chance of the AI performing a random move. It is hypothesised that this may help the AI break out of these cycles, which may assist more games in reaching a conclusion, and subsequently improve the quality of the training.

2.2.2 Machine Learning Considerations

The advantages of the TreeStrap learning algorithm are outlined in section 2.1.2 above. TreeStrap was shown to be a useful training method by both [4] for Duchess, and [10] for Chess. Because TreeStrap seems promising, but has not been widely used in other games, this project serves as a good opportunity to further evaluate its usefulness for adversarial planning games with high branching factors.

Several considerations in the implementation of TreeStrap for Duchess were discussed in [4], including a diminishing learning rate of:

$$LR = LR_0 \lambda^d$$

Where $\lambda = \frac{1}{3}$ is the discount rate, d is the depth of the node in the tree and LR_0 is the root learning rate. The root learning rate used previously was 10^{-5}.

The exponential scaling was used for two main reasons. The first reason was because nodes close to the root of the tree are being trained by the more extensive sub-tree. Thus nodes closer to the start provide more reliable information than nodes further down the tree. The second reason was that nodes may have their weights trained towards the weights of their child nodes when this may not be an appropriate action (as in the case of a move in which a Pawn is promoted to a Queen, where we would run the risk of starting to overvalue Pawns). For this second reason, a maximum limit of 0.001 was also given to how much a weight can train in any one move[4].

In addition a bias feature was included to prevent infinite growth of the material weights. In cases where a board state has value V for a player p_i and the player moves and captures and enemy piece of value x (and there is no change to the other features) the board evaluation after the move is $-V - x$ from the perspective of player p_{i+1} (or $V + x$ from the perspective of player p_i). Therefore V will be trained towards $V + m$ in all cases where pieces are taken, and all piece values will continually increase. If we include a bias of b, however, the initial evaluation is $b + V$, and the subsequent evaluation is $b - V - m$ for player p_{i+1} (or $-b + V + m$ for player p_i). In this case $b + V$ will be trained towards $-b + V + m$, i.e. b will be ultimately trained towards $\frac{m}{2}$ where m represents the overall probability of a piece being captured. This feature particularly important in the first stage of training, where only material weights are included (the addition of extra features in subsequent stages of the network help this bias factor to decrease).

Following the advice of the research discussed above, it is also prudent for us to train the weights for sets of features in stages, starting with a basic network which employs a small subset of the total feature set, and gradually adding in new sets of features. This will help us train using deeper trees for the initial sets of features (as it is much more efficient to evaluate a search tree using only smaller sets of weights, such as material weights only). Furthermore, this will also give us an opportunity to enter the player

into competitions against its previous editions, giving us quantitative data on the utility of feature sets.

Chapter 3

Solution Development

At the commencement of the project the intention was to develop an artificially intelligent agent for playing six-player Duchess using code from the work of Real and Blair[4] as a foundation. As clear experimentation goals were established, however, it was decided that it would be more sustainable to develop a brand new Duchess library that would easily facilitate our research requirements.

The library developed in [4] was sufficient as a proof of concept, however the code was tightly-coupled, sparsely documented, relied on global variables and side-effects, and used such little abstraction that it was impossible to even compete two of the AI players against each other without connecting to a third-party Java server as a proxy. A great deal of re-engineering would have been required to make the player work with six-player Duchess, and as no automated tests existed in the project it was impossible to be confident that changes applied to the code would work as expected.

3.1 Objectives

When it was decided that the Duchess library would be re-developed from scratch, several objectives were outline to guide the solution's design decisions:

- **Performance:** The library must be written in an efficient and performant way so as to enable deeper game tree search.

- **Modularity:** The project should be organised in a highly modular way to facilitate clear automated testing, highly re-usable components, incremental development, and simple customisation for experimentation (such as using different search heuristics which employ different cache structures).

- **Extensibility:** The code should be organised in such a way that it can be easily extended in future work. In particular the use of new move ordering functions and search heuristics should require little to no changes to the existing code base.

- **Testing:** Since the code will be written in a highly modular way, it should be easy to test the code at both a granular and an abstract level. These tests will facilitate rapid development, and enable changes and optimisations to be made with a high degree of confidence.

- **Multi-platform:** While the player developed in [4] only worked reliably on Linux systems, this library should work across all platforms. This not only makes development easier, but it also enables the resulting AI player to operate on any system that can run the Java GUI.

- **Configurability:** Because the AI player will need to perform under a variable set of conditions, it should be constructed in such a way that options like its evaluation mode, search heuristic, and feature set should be easily configurable.

- **Integration:** The library should be able to communicate with the existing Java client over TCP. This will allow human players to interact with the AI in a visual way.

3.2 Design Decisions

3.2.1 Performance

Performance

In order to construct a library with the best performance possible, the library was developed in C++. C++ was chosen as an alternative to Java because of its overall improved performance, lack of reliance on the JVM, and improved memory management. It was also chosen over C because of how it easily facilitates object-oriented design patterns, and because of the in-built Standard Template Library (STL), whose features were used extensively in this work.

Abstraction on Native Types

During development native types were often used in an abstract way not only for improved readability, but also to allow for easy type-adjustment to improve performance. An example of this is the abstract type `BoardIndex`, which is a number that uniquely identified a square on the Duchess board in the range [0, 157]. It was initially defined as an alias for a `short`, via:

```
typedef short BoardIndex;
```

This definition reserved at least 16 bits for each `BoardIndex`, facilitating values in the range $(-2^{15}, 2^{15}))$. Later, the definition was changed to:

```
typedef unsigned char BoardIndex;
```

This definition instead reserved only 8 bits for each `BoardIndex`, facilitating values only in the range $[0, 2^8)$. This was a significantly more memory-efficient definition which resulted in a small performance gain.

Memory Safety

A key shortcoming of the library developed in [4] was its unsafe use of memory. It is hypothesised that its inability to run on OSX environments was due to unsafe memory access which resulting in segmentation faults at runtime. When running the code through `valgrind` a significant number of warnings were raised.

In response to this all automated tests for the new player were run via:

```
$ valgrind --leak-check=yes --quiet [EXECUTABLE]
```

In this way the testing process would not only notify us of failing test cases, but would also immediately notify us of any memory leaks or unsafe memory access. In this way the code was always kept in a state that used memory safely and without leaks. The result of this is that the player works reliably on any environment where it can be compiled, satisfying our requirement for cross-platform compatibility.

Code Profiling

To address runtime bottlenecks and improve the overall performance of the player, `gprof` was used to profile the code at runtime. `gprof` produced an analysis of which parts of the code were run the most frequently, and which sections caused congestion. This analysis was then used to carry out informed adjustments to the code, improving overall performance.

3.2.2 Game Modelling

The new library takes a highly object-oriented approach to modelling the game of Duchess. Each conceptual part of the game is represented in a model, making the code-base easy to understand for any developer familiar with the game. The subsections below provide a high-level overview into how functionality is abstracted across models.

Representation of a Board

- At the highest level, a `Board` is comprised of `Pieces` and `Positions`. `Pieces` track their own `Position` (although the `Board` also keeps a lookup table mapping `Positions` to `Pieces` to improve performance).
- Each `Piece` is defined by its type (e.g. Pawn), its `Position` and its owner. `Pieces` can track the `MoveVectors` they're currently involved in, including which pieces they're attacking and defending, and how they're being attacked and defended.
- `MoveVectors` are used to simplify legal move generation to ensure no move leaves a player's own King in Check.
- Each `Position` is defined by its flap, column, and row, but can also be defined by its unique `BoardIndex`, a number between 0 (which refers to "Off Board") and 157 (which refers to the Vortex).
- Position arithmetic (e.g. ascertaining the positions adjacent to a given square) is performed by a `PositionUtils` library which is able to operate without knowledge of any irrelevant concepts like `Pieces`. This keeps functionality loosely coupled and easily testable.

Representation of a Game

- A `Game` object tracks a set of six `Players` and a `Board`. The `Game` object is responsible for asking `Players` for `Moves` to apply to the `Board`.
- Each `Player` can either be controlled by an AI or a Human. Humans can simply input moves via a command line interface, and AI players use a `Brain` object to determine a `Move` to play. Each `Player`, given only a `Board` state, can generate a `Move` object. This allows for the `Player` object to be easily used by both a `Game` and a `Server`, the interface used to connect to the visual Java client.
- A `Move` object defines any information relevant to a move in Duchess, including the start and end positions, the piece taken (if applicable), and the piece with

which it was replaced (in the case of Pawns moving onto the Vortex). The `Move` contains enough information to be easily applied and undone, which makes the game tree search simpler to conduct.

- A `Brain` object holds any information necessary for the AI to compute a move (such as the set of feature weights). It also holds the `HeuristicCache` related to the AI's search heuristic. An advantage of keeping a separate `Brain` object is that multiple `Player` objects can therefore share one `Brain`. This means that during training the AI player is able to maintain a more rich `HeuristicCache` that tracks information across all move searches, not just the searches performed by, for example, Player 1.

3.2.3 Automated Testing

The modular approach to development allowed for tests to easily be written. The inclusion of these tests allowed for development to be done with confidence, and greatly reduced the impact of bugs introduced via code refactoring and optimisations. In total nearly 4000 lines of test code was written.

Testing took place at three main levels:

- **Unit testing:** These tests assess the low-level functionality of each modular part of the system. Examples of this include tests that check the correctness of functions like `PositionUtils::getDiagonalPositionIndices`. Every single public function of each object was unit tested.

- **Integration testing:** These tests tested functionality at a much higher level, with the intention of ensuring the proper integration of various components of the system. For example, the test `TestPlayer::testAlphaBetaSearchMakesSmartMoves()`, constructs a simple board state in which Player 1 is able to take a Queen or a Knight. The test ensures that the player not only chooses to take a piece, but prioritises the Queen over the Knight. This test relies on correct functionality

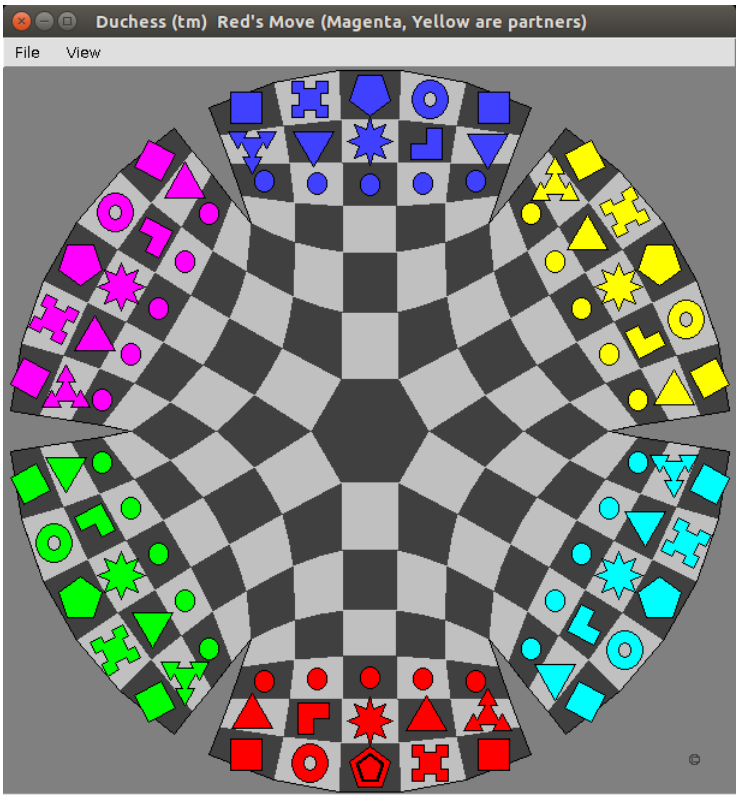

Figure 3.1: A screenshot of the Java Duchess GUI

spanning from valid legal move generation all the way to application of feature weights and board state evaluation.

- **Runtime testing**: Many sections of the code include `assert` statements to validate the data passed to the function. An example of this is the function `Brain::multiplyFeaturesByWeights` which checks that the size of the feature set and the size of the set of weights are the same. This is intended to catch cases where the system has loaded an empty or incorrect set of weights on start-up. These assertions can help bugs be caught further upstream, further reducing the time taken to fix the code. The `assert` macro is disabled at runtime to improve performance by defining `NDEBUG` at the top of each file.

3.3 Integration with Graphical User Interface

Because a key goal of this work was to develop a player that can compete against humans, it was critical that it be able to integrate with the existing Java Duchess GUI. This integration was achieved over TCP by a `Server` interface, which operates as follows:

1. The `Server` listens for communications from a Java client.
2. Upon startup, the Java client connects to the `Server` and sends it a board state as a stream of bytes.
3. This interface interprets a byte stream from the Java client and constructs a `Board` object from it.
4. The `Server` passes this `Board` object to an AI `Player`, who returns a `Move` object.
5. The `Server` translates this `Move` object into a stream of bytes understood by the Java client, and sends it back via TCP.

Chapter 4

Experiments

At a high level, experimentation took place in three main stages:

1. Experimentation of optimisations to improve the naive player's ability to search the game tree, and the development of an ideal set of optimisations
2. Training of the AI player and introduction of new feature sets
3. Evaluation of the utility of feature subsets via competitions between various stages of the network

4.1 AI Development Outline

4.1.1 Evaluation of Optimization Techniques

One of the core areas for improvement of Real and Blair's Duchess player was its speed and the subsequent effect that had on it's ability to learn, particularly in the mid-game.[4] Although some optimisation strategies were implemented in this project (namely the killer-move heuristic), the hypothesis that these optimisations would prove useful was not validated. To address this knowledge gap we ran experiments benchmarking and comparing combinations of the following optimization techniques.

α-β Search Optimization Heuristics

1. **Killer-move heuristic**[12]
2. **Best reply heuristic**[13]
3. **History heuristic**[11]

Board state evaluation methods

1. **De Novo evaluation** on each node of the tree
2. **Incremental evaluation** on each node of the tree

Miscellaneous optimisations

1. Position lookup tables
2. Move ordering function
3. Round-off error avoidance

4.1.2 Adjusting Weights Using Machine Learning

Once the player was able to use its material weights in its heuristic to search the α-β tree, assisted with the ideal combination of optimisations, we implemented the functionality required to allow the AI to refine these weights via self-play using the TreeStrap algorithm[10].

During training games we enforce that the first six moves of each game (one move per player) be chosen at random. Each player usually has eight legal moves on their first turn of the game, so this gives us over one million possible starting game states.

Feature subset	**1**	**2a**	**2b**	**2c**	**3a**	**3b**
Material	✓	✓	✓	✓	✓	✓
Partner Wizards	✓	✓	✓	✓	✓	✓
Bias	✓	✓	✓	✓	✓	✓
Pawn-Distance	✓	✓	✓	✓		
Check / Checkmate	✓	✓	✓	✓	✓	✓
Naive Attacking/Defending			✓			
Attacking/Defending		✓		✓	✓	✓
King-Distance				✓		✓
Piece-Square					✓	✓

Table 4.1: Staged composition of the network

4.1.3 Extending Feature Set and Training

Once the AI was capable of training itself via self-play we extended the subsets of features it used to evaluate the strength of a board state. We expected that the extension of the feature set would drastically improve the performance of the player, as observed by [4].

The stable AI state with which we began this step allowed us to experiment with different feature sets to better understand the important factors in a game of Duchess.

We began the training by finding reliable weights for smaller subsets of features (such as material weights only). We then used the known-passable weights as we introduce new sets of features to speed up the learning process and to avoid the immature convergence problems that Baxter et al have observed from training via self play[8]. The combinations used in various stages of the network are described in Table 4.1.

4.1.4 Stages of the Network

Once an ideal set of optimisations had been compiled and functionality was added for the AI to learn via self-play, the following sets of board features were used in different stages of the network. While the original intention was only to have three core stages of the network, for reasons discussed in Section 5.2 different versions of each of these stages were later developed.

Explanation of feature subset variations

The above table contains some variations on standard feature subsets. Network 2b involves the introduction of *Naive Attacking/Defending* features. Because there is a chance that the AI player will in many cases fail to plan ahead far enough to foresee the actions of the player to their right (e.g. Player 1 thinking about the resulting actions by Player 6), they may learn a set of Attacking and Defending weights that underestimates the threat posed by some players (for example, Player 1 may see no threat moving its Queen into a position where it is attacked by Player 6's Pawn). To evaluate this hypothesis, we constructed a Network 2a which used the traditional Attacking/Defending features, and a Network 2b which used a naive variation on the features, wherein it only evaluates attacking and defending features between teams rather than on a per-player basis. We then ran Network 2a in a competition against Network 2b to evaluate the usefulness of this adjustment.

Later on in the project the King-distance weights were added in an attempt to reduce the number of "Draw" results encountered. These weights were introduced at the same time as the Attacking/Defending weights for a new Network 2c, and Network 3b was subsequently trained to also include piece-square weights. The result of this enhancement is also evaluated in Section 5.2.3.

Chapter 5

Results

5.1 Optimisation Experiments

For each of the experiments in this stage of the project a fixed game of Duchess was run with no search heuristic and no other optimisations. To ensure consistency across the experiment the AI player was forced to search to a depth of exactly 4 moves (such that when planning a line of play, Player 1 would examine moves up to and including the first move by Player 4). No randomness was introduced. In this way the AI played the same game in all cases and we were simply able to examine the amount of time it took to plan its moves. We thus define a successful optimisation as one that significantly decreases the average and/or median search times of the player.

It is important that the AI was forced to play a consistent game, as if randomness or variable (and inconsistent) search depths were used, it is possible that a successful optimisation would result in a divergent path of play that results in an increased branching factor, and would thus be false-negatively flagged as an unhelpful optimisation.

For the miscellaneous optimisations only the first 120 moves of the fixed game were examined due to the slow rate of play. Once a helpful set of miscellaneous optimisations were established (and the speed of play subsequently increased), the search heuristic

experiments were played for a full game (which ended after 349 moves).

5.1.1 Miscellaneous optimisations

The following combinations of miscellaneous optimisations were tested:

Set #	**Evaluation mode**	**Lookup tables**	**Round-off error avoidance**	**Move ordering function**
Base	De Novo			
1	Incremental			
2	De Novo	✓		
3	De Novo		✓	
4	De Novo			✓

Each experiment was then run by enabling the corresponding optimisation in the library. The average and mean search times were then recorded, and an overall percentage change from the base experiment set (no optimisations) was produced.

Set #	**Average search time (ms)**	**Mean search time (ms)**	**% Time change**
Base	46927	15993	
1	7012	2175	-85.06
2	16516	2256	-64.8
3	46039	16040	-1.89
4	13385	10556	-71.48

From the above results we first see that, as hypothesised by Real and Blair[4], the use of incremental evaluation methods as an alternative to De Novo evaluation significantly improved the amount of time taken to search through the game tree. We also find that pre-computing position adjacencies via lookup tables as an alternative to algorithmically ascertaining this data results in a significant speed-up of search.

Interestingly the addition of ϵ in the cutoff evaluation for the α-β search does not seem to have a large impact on the player's ability to search the game tree. In fact, we observed that in this example the addition resulted in a slight addition to the median time taken to search for a move. Because the resulting computation times are so close to the average and median search times observed in the base set, however, we conclude that the round-off error avoidance strategy does not have a discernible impact on the search time.

Finally we see that the introduction of the Most Valuable Victim move ordering function provides a significant speed up on the game tree search. Although the impact on the mean search time is not as significant as those introduced by lookup tables or incremental evaluation, the move ordering function provides a drastic improvement on the average search time. This perhaps suggests that although the move ordering function does not pay off as consistently as lookup tables or incremental evaluation, when it does cause an earlier cutoff in the search tree the benefits can be immense.

Based on these results we define an ideal set of optimisations as including:

- Incremental evaluation over De Novo evaluation
- Position lookup tables
- A Most Valuable Victim move ordering function

Note that we do not then implement the round-off error avoidance strategy, as it was not shown to be helpful.

5.1.2 Search Heuristics

Using the ideal set of optimisations defined above, we then ran experiments to examine the use of different search heuristics on evaluation time. Similarly to the previous experiments, no random element was included in these games, and the AI player always

Heuristic	**Average search time (ms)**	**Mean search time (ms)**	**% Time change**
None	678	284	
Killer Move	669	283	-1.33
History	660	280	-2.65
Best Reply	695	294	+2.51

Table 5.1: Search times for various heuristics

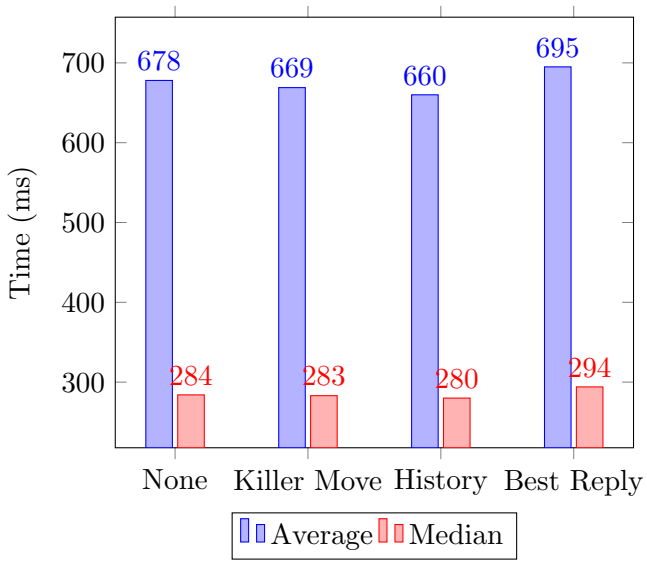

Figure 5.1: Average and median search times for heuristics

searched to a depth of exactly 4. As a result, a consistent game of Duchess comprising of 349 moves was played.

Table 5.1 gives the average and median search times observed. These times are compared against the base case of not using a cutoff heuristic. These times are further displayed in Figure 5.1.

We see from these results that although [4] was correct in hypothesising that the killer-move heuristic would account for a small speed up in the search, the History heuristic provides a marginally better benefit. While the two heuristics operate in the same way, the added granularity of the history heuristic seems to pay off in the long run, especially when the results of the α-β search are cached and re-used in later searches.

We also note that the added computational complexity of the best reply heuristic does

not provide a benefit in the search. Because the mapping of best replies were stored in a hash table, it is likely that they were not producing cutoffs enough to outweigh the cost of maintaining the information.

As a result of these experiments, we chose the history heuristic as the search heuristic for training the network.

5.2 Results of Training and Competitions

Once an ideal set of optimisations was developed and a search heuristic was selected, we began to train the network in stages. The composition of each stage of the network is defined in Section 4.1.4. Upon completing new stages of the network, competitions were run against prior versions of the network to evaluate the efficacy of the new features. In each of these competitions the two networks swapped sides each round (between teams of even-numbered players and teams of odd-numbered players) to mitigate any small advantage or disadvantage one might receive from being the first team to play (though no results indicated that such a bias exists).

5.2.1 Search depth

During the training and experiments the player was able to search to a depth of 5 in the early game, then during the mid game it searched to a depth of 3 or 4. Finally towards the end of the game it was able to consistently search between 6 and 9 moves ahead.

The player determined a depth to search by incrementally increasing the max depth of an α-β search until it estimated that the next depth in the tree would take too long to search. The soft time-limit given to the player to search for a move was 750 milliseconds. The amount of time taken to search to depth d given a board state S was estimated by:

Piece	**Value**
Pawn	0.0645
Bishop	0.0737
Knight	0.0522
Queen	0.1354
Rook	0.0933
Duchess	0.1144
Wizard	0.0637
Fortress	0.1233

Table 5.2: Material weights

$$EstimatedTime(d) = RealTime(d-1) * \sqrt{BranchingFactor(S)}$$

The square root function mitigates the effect of the branching factor of S, making the optimistic assumption of maximal speedup from α-β pruning.

5.2.2 Network Stage 1

For the first network we trained only the core set of features that were expected to be critical to producing a player capable of finishing a game. These feature subsets are outlined in Section 4.1.4 and are explained in Section 1.2.2. This network was trained for 12 hours (roughly 100 games of roughly 400 moves each) with a learning rate of 10^{-5}. The resulting weights learned by the network are shown in Tables 5.2, 5.3, 5.4, 5.5.

Material Weights

It is interesting to note the high value of the Pawn in the game of Duchess compared to Chess. Because the board converges on the vortex, and because the vortex is highly accessible from all sides, it is far more common for Pawns to be upgraded, particularly when a Wizard piece moves next to the vortex. In this way Pawn pieces are valued higher than Knights. Although Wizards have a similar material value to Pawns, they are also prioritised by the Partner Wizard features, effectively raising their value. Thus

Player	Value
1	-0.0422
2	0.0769
3	-0.0403
4	0.0415
5	-0.0395
6	0.0378

Table 5.3: Check weights

Player	Value
1	-0.3268
2	0.2582
3	-0.2197
4	0.2406
5	-0.2189
6	0.2379

Table 5.4: Checkmate weights

only in cases where we only have one Wizard on a team do we ultimately value them similarly to Pawns.

Check Weights

Note that the player numbers are relative to the player about to move, such that each player thinks of themselves as "Player 1".

We observe from the weights in Table 5.3 that the AI player cares comparatively less about the players to its right (Players 5 and 6) and more about itself and Player 2. This could be a result of the limited search depth in the mid game (where it may perhaps underestimate the effects of check on players to its right) or could be because of the volatile nature of the game, causing Player 1 to acknowledge that there's a lower chance for Player 5 or 6 to still be in Check when it becomes their turn.

Feature	**Value**
Partner Wizards (friendly)	0.0511
Partner Wizards (enemy)	-0.0443
Bias	0.0933
Pawn distance	0.0051

Table 5.5: Other weights

Checkmate Weights

Similarly to the Check weights, we note in Table 5.4 that Player 1 cares most about its own welfare and the status of Player 2, and comparatively less about the other players on the board. In particular we note that Player 1 may frequently be able to directly observe negative consequences as a result of being in Checkmate itself (such as not being able to avoid having its pieces taken).

Other Weights

We see from the weights in Table 5.5 that the player prefers to remain in situations where it has at least one partner Wizard. These partner Wizard weights, combined with the basic Wizard material weight, effectively give the Wizard piece a dynamic value, which changes based on the mobility it affords.

5.2.3 Network Stage 2

To train the second stage of the network, initially both Network 2a and Network 2b were developed, employing the standard approach to attacking/defending features, and a naive team-based approach to attacking/defending features respectively. Both networks were trained over roughly 100 games over 12 hours and were competed against each other to determine a canonical "Network Stage 2". The competition results in Section 5.2.5 show that Network 2a won this competition, and so the traditional approach to attacking/defending features was subsequently chosen as a foundation for later-stage networks.

The attacking and defending weights used by Network 2a are shown in Figures 5.2 and 5.3 respectively. The values of these weights closely reflect how we would imagine a human player would think about the board. Generally, a high value is assigned to the capabilities of Queen, Fortress and Duchess pieces, which matches their high values in the material weights. The player mostly cares about things that it is attacking and defending on its own and comparatively less about the status of its allies, perhaps due to the volatile and ever-changing nature of the game. An interesting thing to note in the defending weights is that cases where one Wizard is defending another are strongly discouraged. In these cases the Wizards must be adjacent, which causes the team's mobility around the board to be greatly hamstrung.

5.2.4 Network Stage 3

The third stage of the network involved the introduction of piece-square weights to the feature set. These weights provide a small value for a particular piece owned by a particular player residing on a given square. These weights encourage Pawns to move forward to promotion, and encourage high mobility pieces such as the Queen or Fortress to move to dominant positions.

These weights were trained for a period of 12 hours. The results of this training on the weights given to the position of the Queen piece are displayed in figure 5.4. The blue circles in this diagram show that the Queen is commonly used to move to highly flexible positions on the board, such as the Vortex, which significantly lower the opponents' options in play. The rest of the piece-square weights are visualised in figures 5.5 and 5.6.

For pieces capable of moving long distances we see that they are favoured in squares which maximise their mobility. As hypothesised, we see that the Wizard prefers either to move towards the Vortex of the board (thereby assisting in the promotion of Pawns) or to stay in its own base, which would also help improve the mobility of pieces around the board.

Attacking Weights

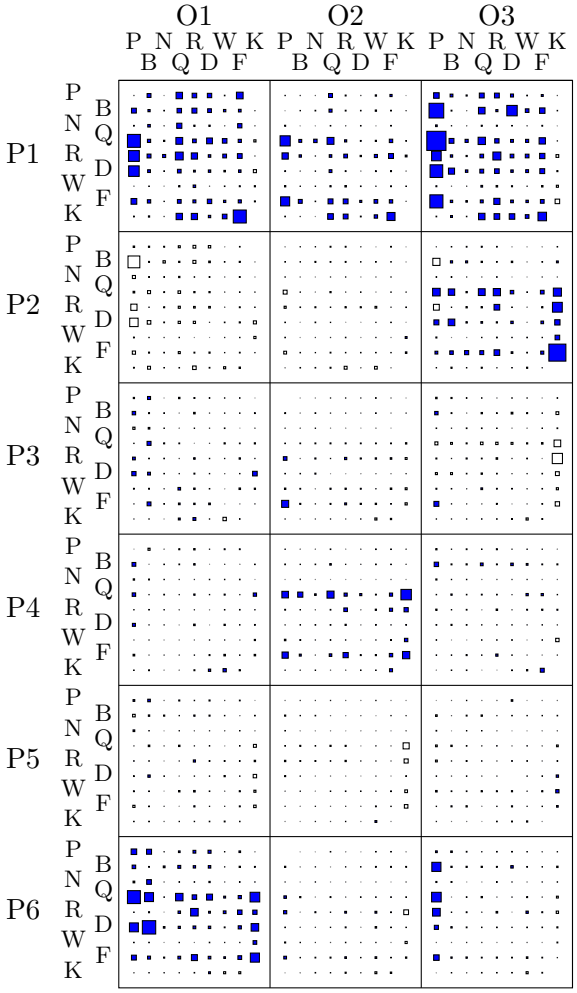

Figure 5.2: Hinton diagram showing the attacking weights used by Network 2a. This shows the value of one piece (row) attacking a piece owned by an opponent relative to that player, such that O1 is the closest opponent to any given player. Blue boxes denote positive weights while empty boxes correspond to negative weights.

Defending Weights

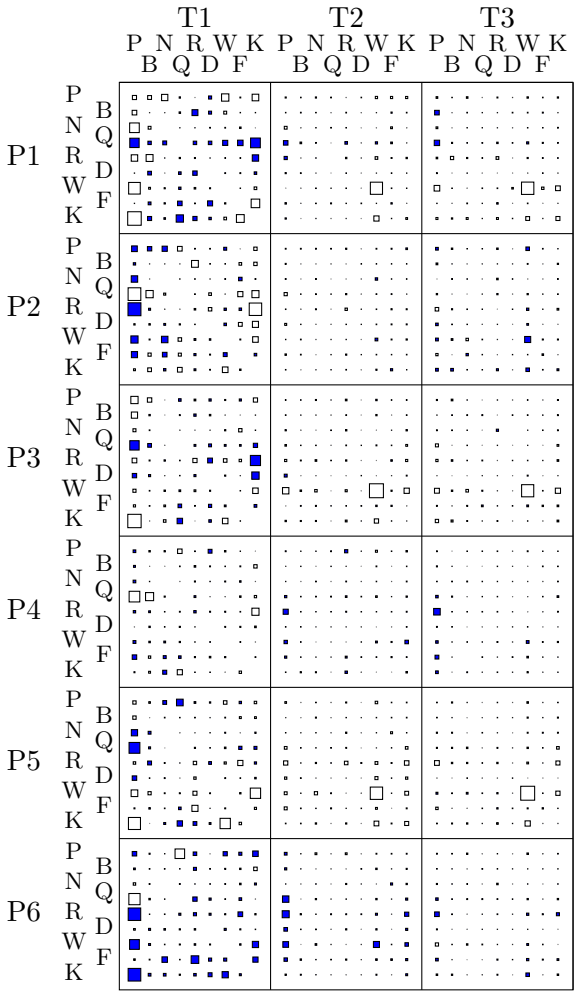

Figure 5.3: Hinton diagram showing the defending weights used by Network 2a. This shows the value of one piece (row) defending a piece owned by a team mate relative to that player, such that T1 is the player themselves. Blue boxes denote positive weights while empty boxes correspond to negative weights.

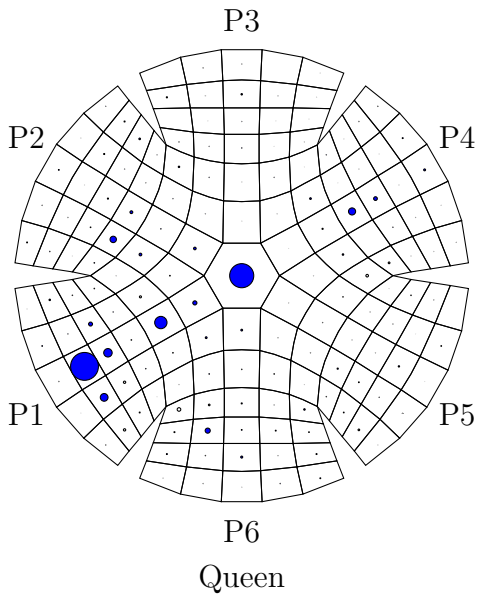

Figure 5.4: Piece-square weights for Player 1's Queen. The blue circles show the Queen's clear preference for playing from squares that offer it maximal mobility.

These diagrams do highlight some potential issues in the weights, which may account for the loss of Network 3 in competitions against Network 2 as reported in Section 5.2.5. In general we see that the player often prioritises movements towards Player 6's area of the board, perhaps because the consequences of such actions are not foreseen. Furthermore, we also see some cases where pieces prefer not to move from their original positions, particularly in the case of the Rook. This was also observed in the training performed by Real and Blair[4]. Just as [4] supposed, we suspect this may be a result of training via games where the player was able to win without needing to move these pieces.

5.2.5 Competitions

As new stages of the networks were trained, where appropriate different versions competed against each other in a competition of 100 games. In these games competitors took turns playing on both the even-numbered and odd-numbered positions, just as Chess players would compete from both Black and White ends of the table.

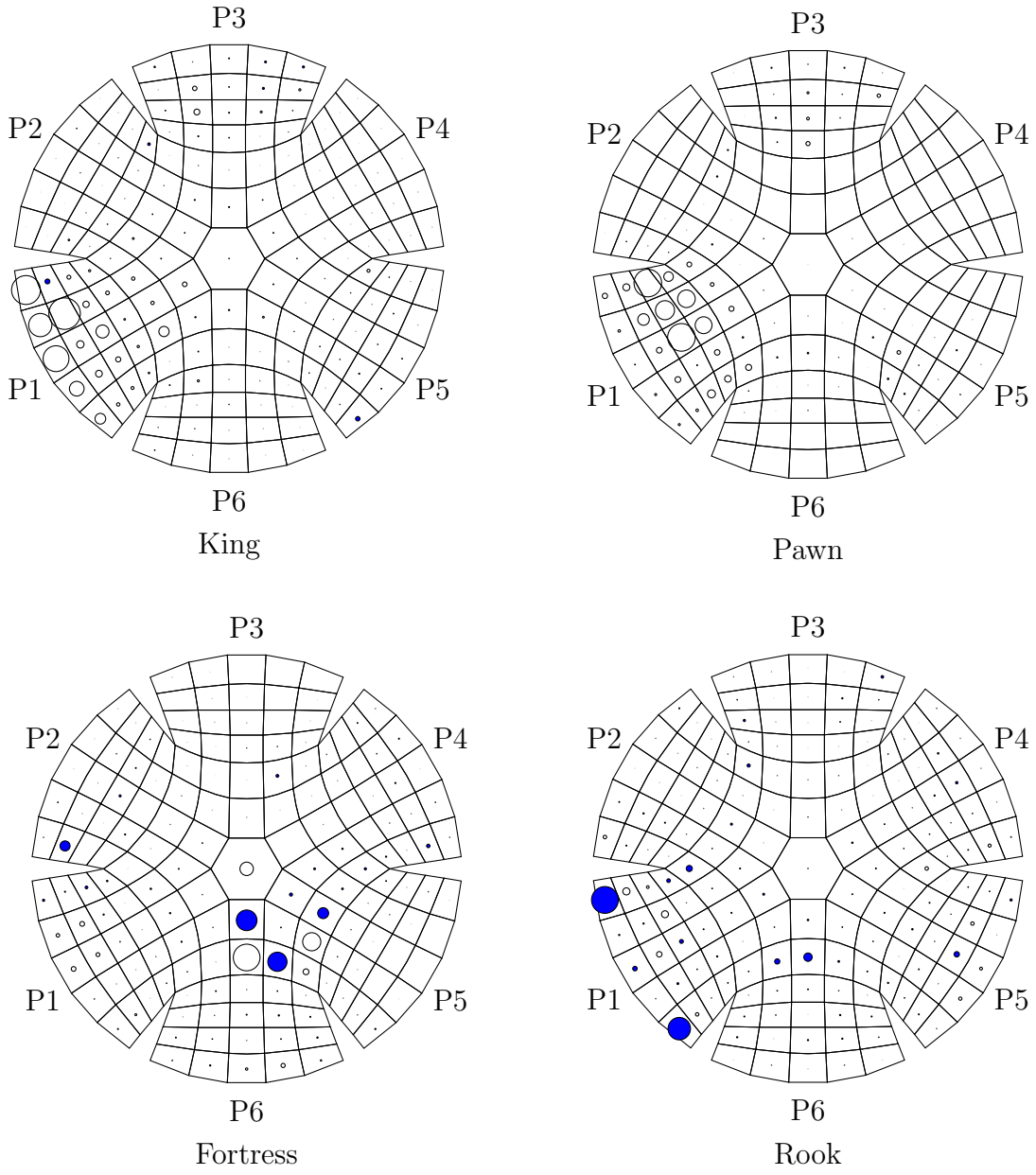

Figure 5.5: (Part 1) Trained piece-square weights corresponding to pieces owned by Player 1. Empty circles denote negative weights and blue circles denote positive weights.

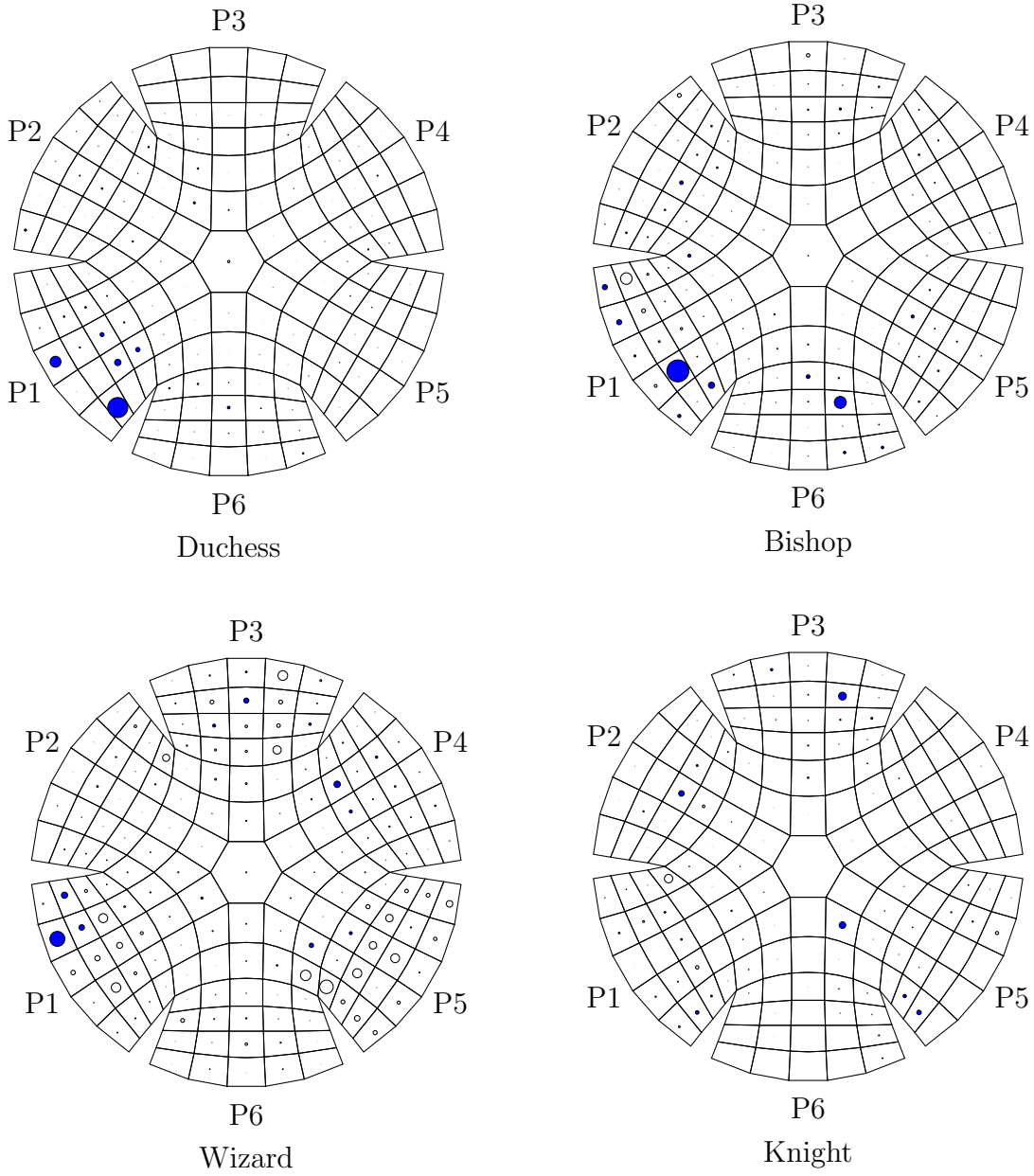

Figure 5.6: (Part 2) Trained piece-square weights corresponding to pieces owned by Player 1. Empty circles denote negative weights and blue circles denote positive weights.

Team 1	**Team 2**	**Result**
2a	2b	16 / 19 / 65
1	2a	11 / 16 / 73
2a	3a	13 / 11 / 76

Table 5.6: Results of initial competitions

The results of these games are shown in Table 5.6. Entries are of the format Won / Lost / Drew and are from the perspective of "Team 1". The name of the team corresponds to a network comprising of feature subsets defined in Section 4.1.4.

The first result from the competitions in Table 5.6 shows that traditional attacking defending features are marginally superior to the naive implementation of the feature subset. In this way we conclude that cases where the player was able to plan sufficiently far ahead during training were adequate in teaching it useful attacking and defending weights on a per-player basis. We therefore chose Network 2a as our canonical "second stage" network, similarly to [4].

We also observe similar results to [4] in the improvement that attacking and defending features provide for the player, however the difference is not as pronounced here. In cases where the player is not able to foresee how to arrange a material advantage the attacking and defending weights allow it to place its pieces in favourable positions that may facilitate such a manoeuvre on the next round of play.

Finally we observe that the addition of piece-square weights in Network 3a does not seem to offer a visible benefit to the player.

A key issue we see here is that there are significantly more "Draw" results than were observed in four-player Duchess. This is likely due to two main factors:

1. The conditions required for a win in the six-player version of the game are inherently more complex than those of the four-player version (as there is a whole extra King piece which must be put in Checkmate). The win condition becomes exceptionally difficult to achieve in late stages of the game where players have lost a significant number of their pieces.

2. The 50% increase in the number of players in the game drastically affects the player's ability to plan ahead. While a game tree of depth 9 was sufficient in four-player Duchess for a player to plan three of its own moves, six-player Duchess requires a depth of 13 for this. This, combined with six-player Duchess' naturally higher branching factor, makes intricate end-game plans significantly more difficult for the AI to produce.

This high draw rate and the relatively low number of games player means that it's difficult to obtain statistically significant results from these competitions. While the results observed here may indicate some improvements in the network, it would be preferable for future experimentation to observe distinct results across 1000 games. Because of the long running time of a single game, however, running so many competitions is at this stage impractical.

5.2.6 Mitigating Draws

A key side effect of the high-number of draws was that the player was trained with relatively little end-game experience compared to its experience in early- and mid-games. This lack of experience adds to the already complicated nature of the end-game of six-player Duchess. To mitigate this, and to hopefully produce a more successful AI player, two key strategies were tested in reducing the number of Draw results for the game.

King-Distance Features

The first strategy implemented was to introduce a new feature set to the second-stage of the network which tracks the number of moves that separate each piece from each enemy King on the board (with the concept of "enemy" being relative to the owner of the piece).

The values are indexed by a mapping of the form:

Team 1	**Team 2**	**Result**
1	2c	14 /16 / 70
2	2c	12 / 8 / 80
2c	3c	8 / 6 / 86

Table 5.7: Result of competitions involving King-distance weights

$p_i, j, d \rightarrow v$

Where p is a piece type owned by player i, j is an enemy of player i and d is a number of moves that separates p_i from j's King, and v tracks the status of this feature.

To facilitate efficient computation of the King-distance feature set, a new lookup table was generated that maps:

$p, i, j \rightarrow d$

Where p is a piece type, i and j are two positions on the board, and d is the minimum number of moves that p must use to travel from i to j.

The King-distance feature set was introduced in the second stage of the network as part of Network 2c, and was trained for 12 hours. The training resulted in some logical values, such as favouring positions where a Duchess was close to an enemy King, and discouraging a Pawn piece from being near an enemy King, however many of the weights were trained to illogical values, such as the Bishop being trained to avoid enemy Kings. An illustration of the Duchess weights is shown in Figure 5.7. The shortcomings of this approach to King-distance features, and a recommendation for adaptation for future use, are further discussed in Section 6.3.2.

Network 2c was also used as a pre-courser to a new Network 3b, and some competitions were run between these networks and the previously trained players.

We can conclude from the results in Table 5.7 that although the general concept of King-distance weights may hold some promise, the present implementation did not produce an improved player and did not noticeably reduce the number of draws in the game.

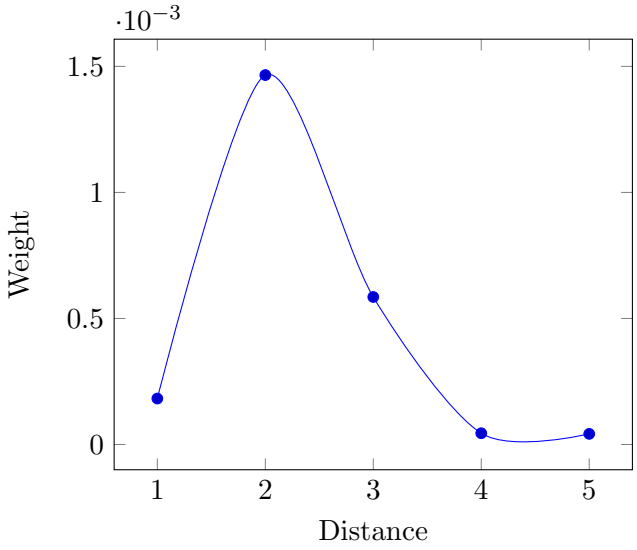

Figure 5.7: King-distance weights for a Duchess attacking the King of player opposite to them on the board.

Stochastic Gameplay

A second strategy to reduce the prevalence of "Draw" results was to include a stochastic element in the training of the player and in the competitions. The first approach to implement this was to force the player to choose a random move 5% of the time. We enforce this randomness after the move planning stage in order to keep the heuristic cache up to date.

Section 1.1.1 defines one of the criteria for a draw to be a sequence of 24 moves where the first 12 moves match the latter 12 moves. In these repetitive cases the random factor will help to encourage an unscripted move to potentially break up the repetition.

Networks 1, 2c and 3c were retrained using this random factor, and while competitions yielded the same victors as in Section 5.2.6, only roughly 70% of games ended in a draw, a minor improvement over the near 80% draw rate of the original competitions.

Because the random move events may be contributing to breaking up critical end-game plays, an alternative stochastic approach of using a Boltzmann distribution has been considered. In this approach a move m would be chosen with probability:

$$\frac{e^{Q[s,m]/T}}{\sum_{a \epsilon LegalMoves(s)} e^{Q[s,a]/T}}$$

Where $Q[s, m]$ represents the value of a board state s after applying move m, the Σ function operates on all legal moves a from the board state s, and T is a temperature controlling the bias with which we choose moves of a higher perceived value. The benefit of this approach would be that the player could relatively frequently choose moves that are not optimal, but are still favourable, while seldom choosing especially unfavourable moves. Further experimentation would be required to choose a suitable temperature value T.

Chapter 6

Conclusion

6.1 Heuristics and Optimisations

The results of the heuristic and optimisation experiments in Section 5.1 has shown that although some benefit can be gained by search heuristics, particularly the history heuristic, because of Duchess' complexity and subsequently reduced searchable game tree depth, greater gains can be made by applying incremental evaluation methods, lookup tables for position calculations, and by intelligent move ordering functions such as the Most Valuable Victim function.

6.2 Training

The use of the TreeStrap learning algorithm successfully trained an agent which could competently play the complex six-player version of Duchess. In human games played via the Java GUI, this AI player posed a significant challenge to human players with experience in Chess.

Similarly to Real and Blair [4], we have observed that the introduction of attacking and defending features greatly improves the success of the AI network over the stage 1

network. Unlike [4], however, we were unable to see a large improvement by introducing a set of piece-square weights in six-player Duchess that remain consistent throughout the game. It may be the case that in six-player Duchess different sets of piece-square weights should be used for distinct stages of the game, such as early game sets that encourage expansion across the board, and late game sets that encourage control of dominant board positions.

We also noticed that the player had a greater incidence of ending games as a draw than the player for four-player Duchess. To improve training in the future, compensation strategies could be employed to try to force results in the game, however the complexity of the end-game of six-player Duchess compared to the four-player variant may mean that equal prevalence of draws between the two games are impossible.

6.3 Future Work

6.3.1 Extending Search

The use of a move ordering function to increase the depth of the search was highly profitable. Future study should be conducted on the usefulness of other move ordering functions in speeding up the α-β search.

In order to search deeper in the tree it may be beneficial to develop a library which takes advantages of multi-core processors via threaded programming. Legal moves for a given board state could be computed concurrently, potentially resulting in significant performance games in the system. In contrast we do not expect that parallelising board evaluation using a GPU would speed up the search process. Because the number of features and weights is so high, the overhead associated with copying the relevant data to the GPU far outweighs the expected benefit it will present in the calculation of the board value.

Finally, it may be useful to experiment with applying transposition tables to store the result of previous searches, particularly during training. This may be useful because

Player n plans many move sequences for various states that Player $n+a$ may encounter (for some $a > 0$), but the applicable lines of best play previously computed by Player n are forgotten. If this previously computed information were re-used the player would have a "head start" in progressing down the search tree.

6.3.2 Reducing the Number of Draws

While the introduction of the random factor in Section 5.2.6 did show some promise in reducing the number of draws in the game, it may be more beneficial to implement a Boltzmann distribution to chose a somewhat-random move, as discussed previously. In particular experimentation would have to be performed to determine a useful temperature parameter T for this operation.

Finally, Section 5.2.6 noted that although in some cases King-distance weights were trained to logical values, several illogical anomalies emerged. It may be the case that King-distance weights, which are intended to encourage the player to put enemy Kings in Checkmate, are actively unhelpful in cases where the player is losing the game. It may therefore be helpful to only enable use of King-distance weights when the player is at a clear advantage (for example, when they have a material advantage over their opponents, the opponents has fewer than 15 pieces collectively, and the corresponding King is not in Checkmate).

Bibliography

[1] A. Blair and J. Kramer, "Duchess(tm) rules." https://www.cse.unsw.edu.au/ blair/duchess/rules.html.

[2] U. Lorenz and T. Tscheuschner, "Player modeling, search algorithms and strategies in multi-player games," in *Advances in Computer Games*, pp. 210–224, Springer, 2005.

[3] D. F. Beal, "A generalised quiescence search algorithm," *Artificial Intelligence*, vol. 43, no. 1, pp. 85–98, 1990.

[4] D. Real and A. Blair, "Learning a multi-player chess game with treestrap," 2015.

[5] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of research and development*, vol. 3, no. 3, pp. 210–229, 1959.

[6] J. Schaeffer, R. Lake, P. Lu, and M. Bryant, "Chinook the world man-machine checkers champion," *AI Magazine*, vol. 17, no. 1, p. 21, 1996.

[7] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[8] J. Baxter, A. Tridgell, and L. Weaver, "Knightcap: A chess program that learns by combining td (λ) with game-tree search," in *Proceedings of the 15th International Conference on Machine Learning*, 1998.

[9] M. Campbell, A. J. Hoane, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1, pp. 57–83, 2002.

[10] J. Veness, D. Silver, A. Blair, and W. W. Cohen, "Bootstrapping from game tree search," in *Advances in neural information processing systems*, pp. 1937–1945, 2009.

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

[12] S. G. Akl and M. M. Newborn, "The principal continuation and the killer heuristic," in *Proceedings of the 1977 annual conference*, pp. 466–473, ACM, 1977.

[13] J. Uiterwijk, "The countermove heuristic," *ICCA JOURNAL*, vol. 15, no. 1, pp. 8–15, 1992.

[14] M. Buro, "From simple features to sophisticated evaluation functions," in *Computers and Games*, pp. 126–145, Springer, 1998.

[15] D. Beal and M. Smith, "Learning piece-square values using temporal differences," *ICCA JOURNAL*, vol. 22, no. 4, pp. 223–235, 1999.