The University of New South Wales

School of Computer Science and Engineering

Learning a Strategy for Multi-Player Chess

Joseph Barcham

Bachelor of Engineering
(Software Engineering)

November 2013

Supervisor: Alan Blair
Assessor: Malcolm Ryan

Table of Contents

Table of Figures

Abstract

This thesis attempts to apply the recently developed TreeStrap algorithm to create the first machine learning A.I. player for the chess variant Duchess. Self-play was used to learn the foundation of a feature based heuristic function for evaluating non-terminal board states in Duchess. This served as a platform to examine the nature of Duchess as a game as well as test the effectiveness of the TreeStrap algorithm in dealing with larger search spaces than had previously been explored.

Introduction

Developing algorithms to play games is an area of study which dates back to before the first computers were even built. As computer hardware has improved, so too has the techniques and algorithms for playing games.

Some games such as tic-tac-toe or checkers have been either completely solved or solved for later stages of the game. Meaning that from some position the line of best play is known with complete certainty, and hence can be played perfectly by simply looking up the best move for the current position with a minimal amount of computation.

However for most games, such as chess, this is not the case and determining the optimal move from a given position would take an infeasible amount of time to compute. In these cases algorithms are employed which attempt to produce a good estimate of what might be a line of best play. One method of producing this estimate is to have a heuristic function which gives an approximation of the strength of positions which follow down the various potential lines of play, these can then be compared to obtain a 'best guess' as to the true line of best play.

In the case of chess this heuristic is often a single layer neural network comprised combination of a set of 'features' (Beal, 1997), which include metrics such as which pieces each player has remaining on the board and where they are positioned. These feature values then need to be combined to produce a single heuristic value for a position. The weights given to each of these features can be assigned either by human experts or calculated using machine learning. Until recently a combination of the two had been needed, a human had to provide reasonable weights as a starting point for the machine learning process.

It was recently shown (Veness, et al., 2009) that the TreeStrap algorithm is able to learn to play master level chess without requiring weights initialised by a human expert, instead using randomly initialised weights. It does this by updating the weights in an offline fashion, meaning that it does not only use positions which are actually observed or are even necessarily on the line of best play. This thesis aimed to test the performance of the TreeStrap algorithm in the more complicated search space of Duchess, which is a variant of chess designed to be played by up to six players. If the algorithm performs well in this case it could enable or improve the use of machine learning techniques in areas where they have traditionally struggled due to search space complexity.

Background

The game of Duchess is a variant of chess played with six players on a hexagonal style board with each player controlling a set of pieces similar to those used in chess. Like chess, Duchess is a zero-sum, sequential, discrete game where all players have perfect information. It is also team based with the players being on one of two fixed teams. With reference to Figure 1 the teams are: Red, Pink, and Yellow on one team against Green, Blue, and Cyan on the other.

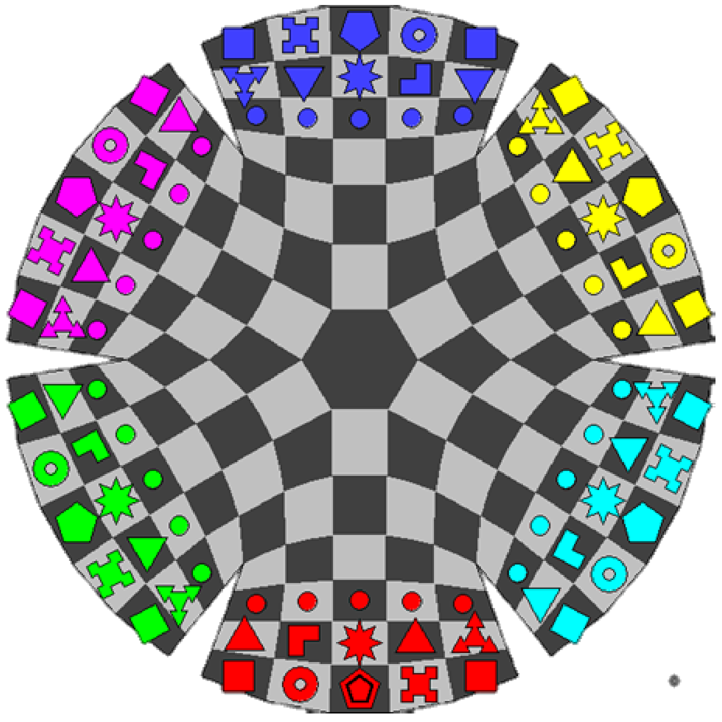


Figure 1 Initial setup for six player Duchess

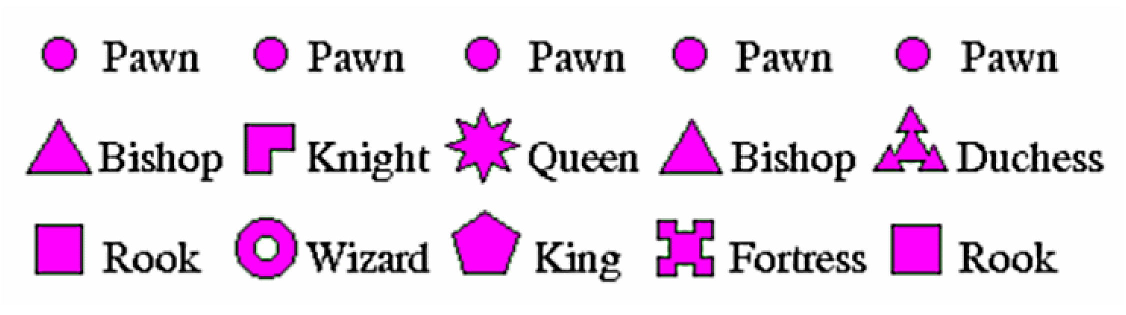


Figure 2 Initial piece layout for each player

There are three pieces used in Duchess which are not in standard chess, they are:

- The Duchess which on each turn can move in the style of either a bishop or a knight.
- The Fortress which on each turn can move in the style of either a rook or a knight.

- The Wizard allows pieces located on tiles adjacent to it to teleport to any other square adjacent to it or any other wizard belonging to that player's team.

For the full details on the rules of Duchess refer to http://www.cse.unsw.edu.au/~blair/duchess/rules.html

The primary inspiration for the A.I. Duchess player which was developed came from computer chess players such as KnightCap (Baxter, et al., 1998). Such programs have vast bodies of research behind them and are skilled enough to out-play even Grandmaster level human players. The theory was that there is sufficient similarity between chess and Duchess that chess players, the research behind them, and the techniques they use should provide a good starting point for developing a Duchess player.

The key reason developing such a player is interesting is that the game tree of Duchess has a much higher branching than that of standard chess. This poses some problems when it comes to using machine learning to train a heuristic function. Using an approach such as temporal difference learning would struggle to learn effectively since it only learns from positions which are actually observed. This means that for a tree with a high branching factor there would be sections that TD explores very rarely, if ever, even when a large number of games are played. This will result in poor or unpredictable behaviour under extra-ordinary circumstances, not unlike the problems with the training of ALVINN (Pomerleau, 1989) in the early development of neural networks. TreeStrap alleviates such issues, as explained later.

Note that with three instances of the same deterministic computer player playing on one team then that team can also be thought of as a single player controlling triple the normal amount of pieces. This is the case because for any given position each will see the same line of best play and hence be able to perfectly predict the moves the other two will make, meaning they 'think' as if they were a single entity. The problem with this is that the algorithm may find a strong sequence of moves, one or more of which are risky and/or must be made by teammates. If a teammate does not make the expected move for any reason then the strong sequence is broken and may leave the team in a much worse position.

Solving this problem requires some amount of player modelling to be done so the algorithm can decide whether it's teammate will follow up on the risky complicated stratagem, or whether it should stick with something safer and simpler which does not have the same reliance.

Experiments in Parameter Learning Using Temporal Differences (Baxter, 1998)

A key design decision when creating a heuristic is choosing a suitable set of features. Certain trade-offs are necessary, primarily the cost of calculating the feature value verses the improvement in accuracy that feature gives to the final value of a position. This paper looks at the feature set used by KnightCap and discusses their effectiveness.

The decision to base the feature set for TreeStrap on board position and material was made on the basis that this paper suggests they are the core and most important feature sets used by KnightCap.

Note that although the feature set is important developing an ideal feature set for Duchess is not one of my main goals since improving the feature set is only useful if it is already backed by a good learning algorithm. It is however an interesting area which could be explored much more deeply in future work.

Bootstrapping from Game Tree Search (Veness, et al., 2009)

This paper introduces TreeStrap and demonstrates its advantages over similar methods such as Temporal-Difference learning (TD). The authors successfully trained a master-level chess player using randomly initialised parameter weights, although they do not detail what exactly those parameters represented.

TreeStrap uses minimax search with alpha-beta pruning to produce a game tree rooted at the current position. Alpha-beta is used to speed up the search and allow deeper exploration, this will be especially useful with Duchess' large branching factor.

TreeStrap avoids the exploration problem TD suffers by using all states in the computed game tree to update the weights meaning that it learns more efficiently and is able to learn about positions even if they are never played. Thus when an unorthodox series of moves is played a heuristic function trained by TreeStrap sees a position unlike anything it has encountered previously but is still able to provide a strong evaluation.

A problem arises due to using alpha-beta which is that not all states have a target value to adjust the heuristic value towards since when alpha-beta triggers a cut the position at which it occurs has only a range with an upper and lower bound on its value. This is solved by increasing or decreasing the value towards the nearest bound if it is outside the range. If the value is within the range no adjustment is done.

Design

The TreeStrap algorithm is fundamentally built using an iterative deepening minimax depth-first search with alpha-beta pruning. The heuristic is a set of weighted features which are combined yielding a value for a single board state.

Minimax search is an algorithm used in to determine a line of best play in two player zero-sum games. For every board state the current active player is considered to be trying to maximise their return, while the other player tries to minimise the current player's return. For games with more than two players similar algorithms such as Max-N exist, but are not necessary since each team is considered to be controlled by a single entity and noted earlier.

Alpha-Beta pruning is an optimisation of minimax search which is used to avoid searching unnecessary lines of play, it is proved optimal (Pearl p. 1982) meaning that a search done with alpha-beta pruning will always return the same result as one done without it. It works by remembering the lowest upper-bound (beta) and the highest lower-bound (alpha) of all previously explored moves, if a move is found from a given position which is outside of these bounds (i.e. beta ≤ alpha) then any remaining moves from that position are not explored since the position is strictly worse than some previously explored position and hence will never be reached.

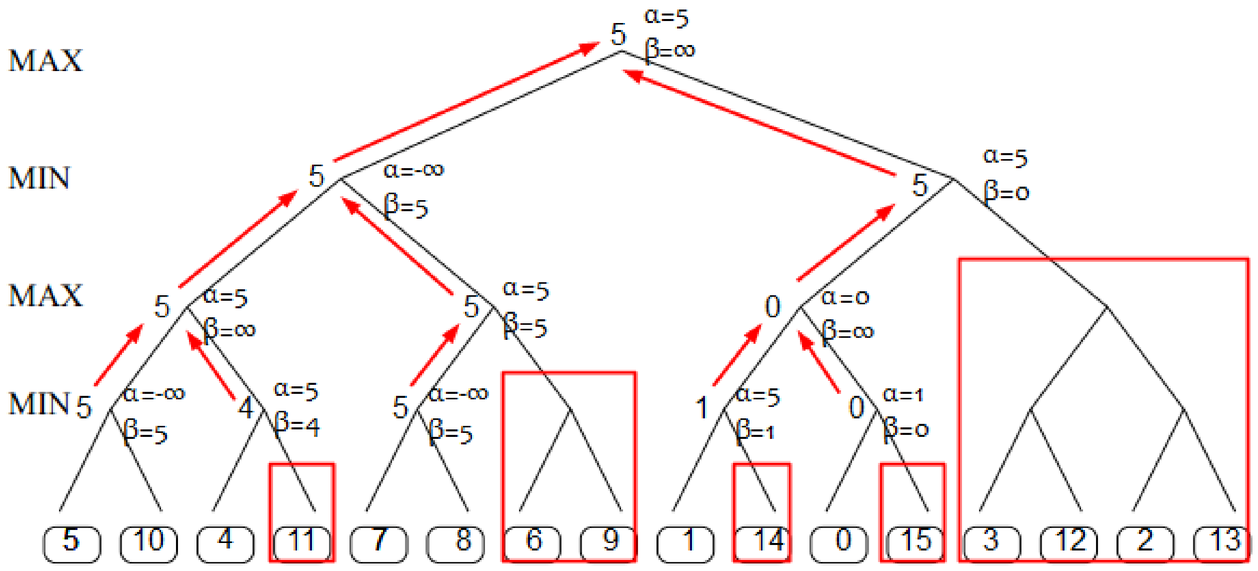


Figure 3 Example Minimax search with A-B pruning

Iterative deepening search is an expansion on depth-limited, depth-first search (DFS) (Korf, 1985). By repeatedly re-running a DFS with an increasingly deep limit iterative deepening is able to simulate breadth first search while retaining the memory usage of DFS. Extra overhead is incurred since a search to depth *d* will cover level *d-1* twice, *d-2* three times, etc. however is overhead amounts to only around 11% of total search time and is preferable to the vast memory requirements of breadth-first search.

Each turn the game tree is generated starting with the current position, from this tree the line of best play and feature weight updates are computed. Based on the line of best play the next move is chosen. At each node in the tree a weight delta is calculated which would bring the heuristic value of that node closer to those observed further down the tree. These deltas are combined and used to update the heuristic function, note that this is done after the tree has been fully generated otherwise the update would affect the line of best play in an unpredictable way.

Courtesy of Alan Blair there already exists a Duchess engine and frontend which allows humans to play against each other. It consists of a client and server written in Java. Players can play against each other by using their respective clients to connect to a running server, choosing a game and joining it.

This was very useful for development since the existing server could be left untouched, and a computer player is able to connect to an existing, unmodified server and play against either humans or other computer players. This made self-play training easier and removes the need to write additional code in order to have the computer player be portable.

Implementation

For the implementation of the computer player agent, much of the existing client code was reused, including network code albeit with significant modification, board state initialisation and storage, move generation, and display functions for debugging purposes. Additional components which were to be created from scratch include the feature set evaluation, update, and management, state storage and reversible moves for search tree exploration, and of course the tree search and agent core.

Existing Code Reuse

All pre-existing client and server code was written in Java (most likely version 2, based on use of APIs which are now deprecated). The scope of the project originally included porting most or all of this code to C. The advantage of using C being that it is better optimised and does not have Java's garbage collection overhead. Since search depth is one of the areas that tends to have the largest impact on the strength of game players (Sturtevant, 2003) and improved running speed would serve to increase potential search depth, C would have been the preferred option.

Work was started on porting the client code, however it was not completed due to schedule concerns. The port itself would not here directly contributed to the project goals, rather it may have provided a qualitative speed improvement. In addition, development of Oracle's HotSpot™ JVM which uses JIT (Just In Time) compilation has made it possible for Java code to achieve execution speeds comparable to native code. For these reasons development proceeded in Java rather than C.

Additions

The first module developed was reversible moves to avoid storing an entire board state when doing tree searches (which would take up large chunks of memory, most of which would be redundant). This was done using a wrapper around the existing board class. The wrapper stored the information which makes up each move and was able to roll back the piece location array and propagate that change through the rest of the structures which comprise the board state.

With moves being reversible a minimax search tree could be constructed and alpha-beta pruning implemented on it. For this a recursive negamax search call was written which ran in two main steps, generate a list of all moves at the current state, then recurse on each move in turn reversing the move once the recursion returns. The terminating condition on the recursion was a depth limit, if the top level search returned with sufficient time remaining in the current turn the depth limit would be increased and the search repeated, i.e. iterative deepening.

The method by which it was determined if 'sufficient' time remained to run a deeper iteration of the search was to remember to time taken by previous searches and use that as an estimate of how much time might be required to run the next search. This caused somewhat irregular usage of the allowed time as discussed in the results below. This could be improved by having the search be interruptible so it could consistently run for the full duration on each turn. This would entail modifying the recursive search call such that when time expired it would immediately return with the current best known result. However this is not categorically better, it is simply more consistent, the current implementation covers an equal amount of search space overall but is shallower on some turns and deeper on others.

In games outside of the training environment the search could be further improved by utilising the time taken up by other players turns, gaining up to a sixfold increase in available search time. During this time when another player makes a move that move will almost certainly already be in the search tree and already have some amount of exploration beyond it, in which case all other possibilities at that level can be pruned from the tree. This retains already known information and reduces wasted computation time. For training purposes within this implementation however, there is no such thing as wasted computation. Since the agent is the only player taking up 'thinking' time, and since TreeStrap learns from all explored positions, there would be no benefit to continuing to search on other player's turns as doing so would detract from the equivalent search efforts of the same agent controlling a different player. For this reason searching was left limited to each player's actual turn time.

At the maximum search depth for a given iteration a heuristic value for the current state was determined. This used a basic feature set consisting of material and board position as determined by a value given to each square for the piece on said square. These values had the sigmoid function applied to them and were then returned as the heuristic value for that state. In chess the king does not need to be given a material value since losing one's king equates to losing the game, in Duchess however one king can be in a checkmate position without the game being lost. It is therefore necessary for the king to have a material value, however although a single king can the threatened, put in check, or even checkmate, it can never be actually be taken so the material value would seem not to be effective since the king is never removed and the drop in sum total value is never observed.

This problem is addressed by pretending the king can be taken for the purposes of the search, but never generating the move to actually take it as a candidate move. Thus the agent is incentivised to attempt to capture the king, but in doing so it is actually working to get closer to a true checkmate position which has minimax's usual extreme positive/negative value associated with a win/loss which were set to 10000 and -10000 respectively. In addition stalemates were considered negative outcomes in an attempt to encourage the agent to more aggressively seek check and checkmate states. This solution is not ideal however and the further implications of it will be discussed in the results section.

Following evaluation of states TreeStrap's unique update step occurs. This entails calculation of the difference between the heuristic value of a position and the values of subsequent positions. These differences are accumulated and used for the update of the feature weights. See the paper discussed earlier, *Bootstrapping from Game Tree Search* (Veness, et al., 2009), for a full explanation of the update component of the algorithm.

In all, 1413 parameters were used to make up the feature set. This comprised of, for each of the 157 squares, a set of 9 values one for each type of piece. Both Rooks, both bishops, and all five pawns were condensed to 3 piece types. Bishops are interesting since in Duchess they are able to switch colours by using the wizard's teleport to move from a light square to a dark one or back, thus it is possible to have two light, two dark, or one of each bishop at any given time.

It has been shown that TreeStrap is able to learn from randomly initialised weights, however in an attempt to expedite the process the weights were initialised as follows, plus or minus a small random variation per square.

Piece Type	*Initial Value*	*Notes*
Pawn	5	Slightly higher than chess value, since Duchess pawns are more versatile being able to move in many directions.
Bishop	10	Similar to chess values, these were used as the base to choose value for other piece types.
Knight	10	
Rook	15	
Wizard	15	Teleport is powerful, but piece is otherwise little better than a pawn. A bonus for having multiple wizards in play could be an effective addition since this increases teleport power.
Duchess	20	A duchess is a bishop + a knight.
Fortress	25	A fortress is a rook + a knight.
Queen	25	Because of the unique board layout, a queen often has similar mobility potential to a fortress and is hence weighted equal.
King	50	a large value is given to the king to encourage putting it in check

When a game is initially setup the agent reads existing parameter values from a file which is stored from the point of view of chair zero, these values are read into the agent's parameter array using the formula

```
 (target_square + (chair * SQUARES_EACH)) % TOTAL_SQUARES
```

which takes advantage of the rotational symmetry of the board to alter the perspective from that of chair zero to that of the current player's chair. Saving parameters used the same operation with `-1 * chair` to reverse the rotation.

Once the update is performed, the agent would be conclude its turn by sending the best move found by the search to the server and original code would handle subsequent moves until the agent's next turn.

Results and Evaluation

The performance of the TreeStrap based agent developed suffered from one major stumbling block. As mentioned earlier, the condition for a Duchess game to end is quite different from chess in that in requires all three opposing kings to be held in checkmate simultaneously. This turned out to be much more difficult than expected. Highly ranked chess players will commonly use some combination of an end game database, an independent end game engine (Hauptman, 2005), or a separate end game feature set, as in the case of KnightCap. For Duchess none of these previously existed and developing them was never included in the scope of this thesis.

The TreeStrap agent could play a sound mid game thanks to significant search depth and human-initialised weights. However training sessions which were setup in self-play scenarios would consistently lead to stalemate with both sides loosing most of their pieces and being unable to put all three opposing kings in check simultaneously. Similarly in investigative games against human players the agent would perform with a strength empirically equal to or better than humans sometimes gaining a significant material advantage, but would be unable to close out the game eventually leading to stalemate.

When training was switched to the agent controlling one team, against a team making random moves the outcomes improved dramatically, while the agent rarely achieved victory in the early or mid-game, it was obviously much stronger than a random player and would eventually capture most of the random team's pieces, obtain far superior board control and subsequently be able to secure a win.

It seems likely that adding checkmate related features such as king mobility, squares around each king which are threatened, threats to pieces which are themselves threatening the king or the area around it, etc., would provide improved mating performance, but implementing and quantitatively evaluating the benefits these provide and the cost they add in additional computation was unfortunately not a luxury the schedule allowed.

Learning

The lack of the ability to reliably reach an end state against non-random players severely impacted the learning ability of TreeStrap. The fundamental basis of TreeStrap and other Temporal-Difference based algorithms is that the heuristic function they use and improve upon is an estimation of how close the current state is to a win or loss state. When one such state is found the positive/negative value associated with it is back-propagated and in some way used to improve the heuristic function. Without common end states TreeStrap was unable to effectively update its heuristic.

A limited amount of information was learned from the games against random players, a sample of which is shown below from the lower right player's perspective, however it is from a small number of games played against a very weak opponent and would need much more refinement before being considered a sound representation of good board positions.

None the less even from this small data sample some trends begin to appear. In the diagrams below green represents a high value, the heuristic value of a board increases if the appropriate team's piece occupies a high value square, conversely red represents a low value square which is less favourable. The values shown below are normalised for display purposes so the red squares are of equally low value as opposed to some lowest possible value. From this it seems the rook is more powerful from its team's starting positions and from the centre region, while being in a file which an enemy rook starts in tends to have a negative outcome. For the wizard, a more tactical piece primarily useful for its teleport, staying away from opposing areas seems preferable, occupying the centre appears beneficial perhaps allowing other pieces to teleport in strong attacking positions.

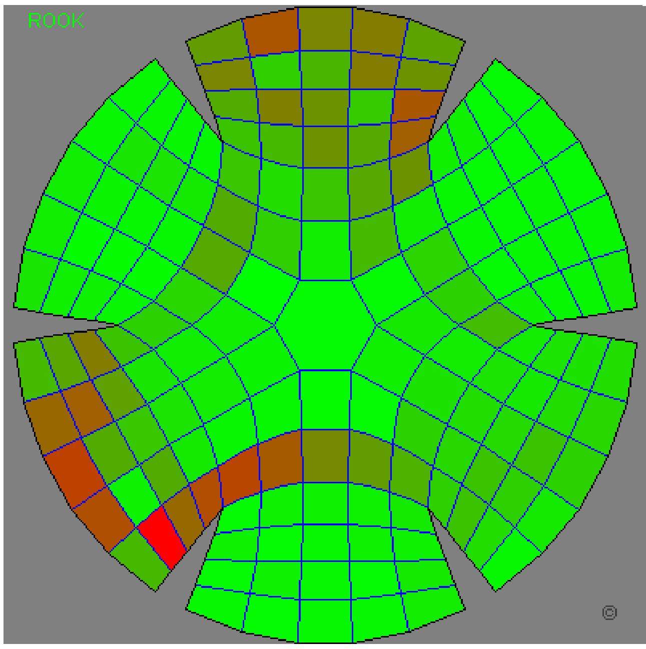


Figure 5 Rook Position Weights

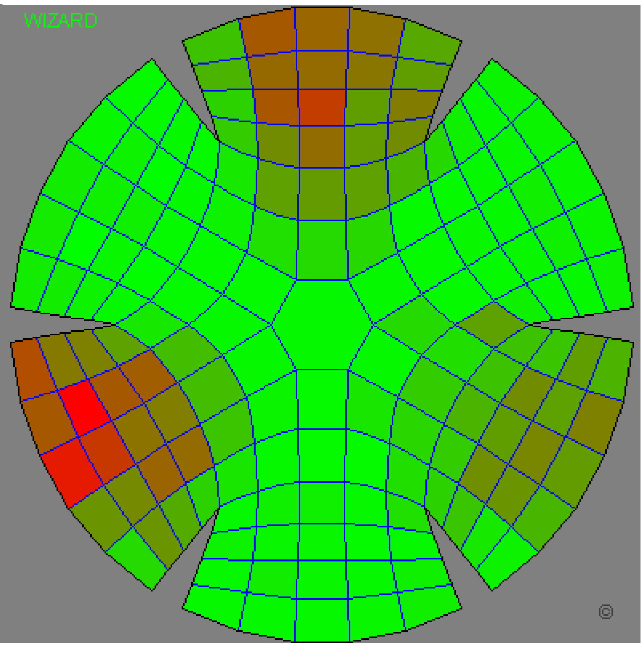
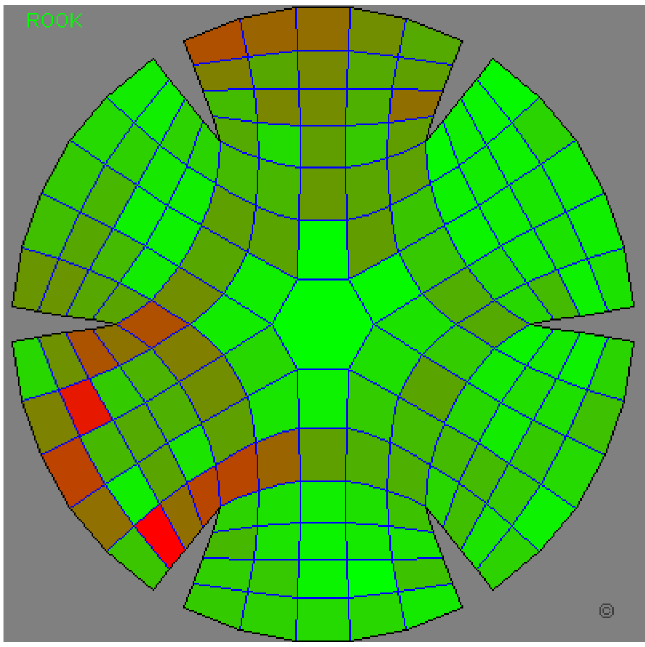


Figure 5 Wizard Position Weights

Assessing the learning ability of TreeStrap is difficult with the limited data gathered, however it is clear that it does demonstrate rapid and consistent development. An effective learning rate of λ = 0.02 was used which is unusually large for this type of algorithm, but necessary in this case since games with a successful outcome were less common and there was not sufficient time to play the thousands of games normally required for comprehensive training. TreeStrap was able to refine the parameters shown above to the following in a mere six games (keep in mind that a game of Duchess is often 2-300 moves, much more than many games).



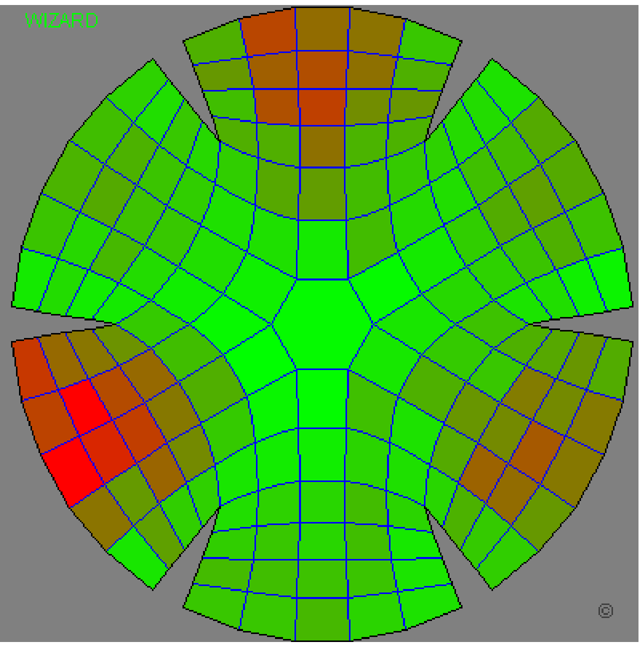


Figure 6 Refined Wizard Position Weights

Figure 7 Refined Rook Position Weights

Even this small number of games makes a noticeable difference, in both cases the value of the centre remained high while the starting positions have decreased in value. Interestingly for the rook the 2nd file from the centre has continued to worsen in value, while for the wizard the edges of the board appear to be emerging as good positions. These rapid and consistent changes to the heuristic parameters are strong evidence for the successful application of TreeStrap in learning a good heuristic, and with further training and improvements to the feature set it would likely be very challenging to defeat.

One other note on TreeStrap's learning, throughout the training sessions the agent very rarely, if ever, moved pawns backwards from their initial starting positions even when using the initial uniform weights, simply due to the initial board setup and the fact that such moves would be zero-sum in all but the rarest circumstances. For a more conventional TD-based learning algorithm this would mean that the weights for the squares behind the initial pawn positions would not have their values updated. Looking at the top-left, top-right, and bottom centre starting positions in Figure 8 below, however, shows that TreeStrap's ability to update based on hypothetical positions has allowed it to begin learning weights for these positions despite little or no data from actual play.

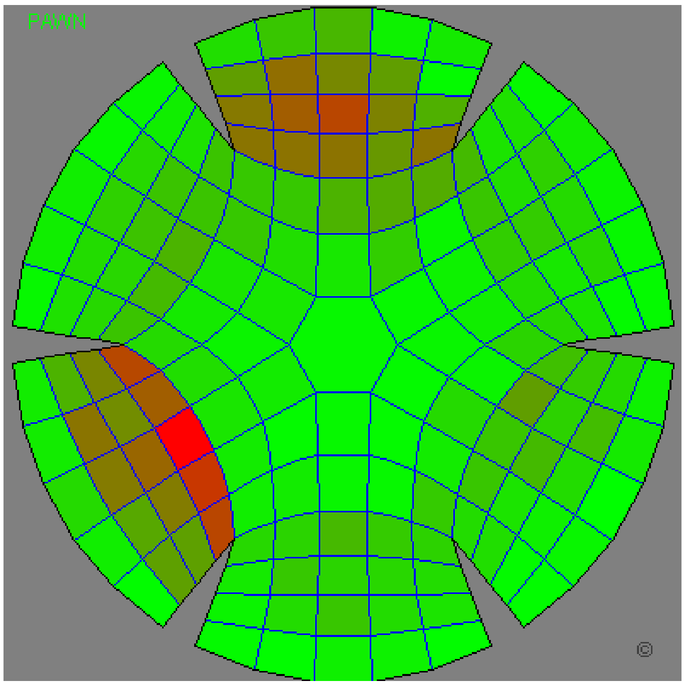


Figure 8 Pawn Position Weights

Branching Factor

This is one of the defining characteristics of Duchess and one of the key reasons learning a strategy for it was interesting and difficult. Initially the branching factor of Duchess was estimated to be many times greater than that of chess. Chess is commonly stated as averaging somewhere in the vicinity of 35 possible moves during the bulk of the game, with the opening and end-game having fewer.

Duchess proved to have slightly less than suggested by the initial estimate, with values between 60 and 100 being most common, as shown in Figure 9. The highest single value observed was 156 possible moves from a given position. These figures give Duchess a branching factor 2-3 times greater than that of chess during regular play and up to 4 times greater in worst case scenarios.

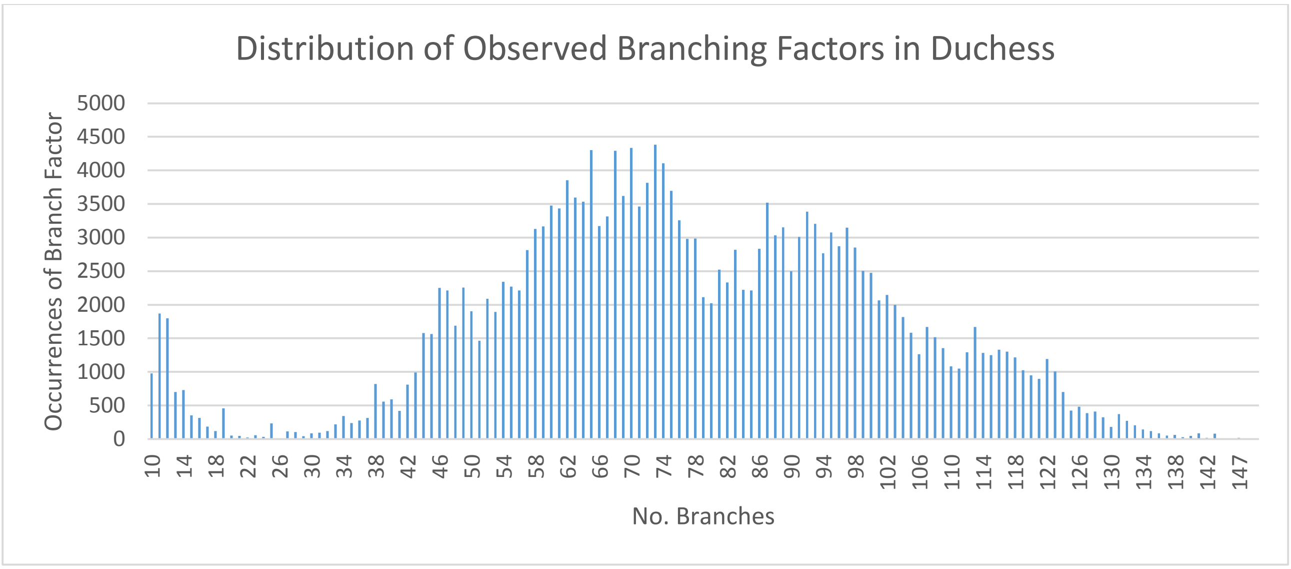


Figure 9 Distribution of Observed Branching Factors in Duchess

Figure 9 represents 215,582 data points obtained from a single training session. Values between 0 and 10 have been excluded since they relate almost exclusively to opening and end-game situations, a significant number of values lie in the 10-20 range however they were not excluded on the basis that these numbers could represent relatively unobstructed movement for 2-3 pieces and hence could constitute a mid-game position. The reason for the lower values in the 80-85 range is unknown, this and the other trends observed here are consistent across all sessions.

Search Time and Depth

Arguably the single greatest factor determining the strength of an A.I. player is the depth it is able to reach in its game tree search. For this Duchess player a target depth of 7 was stated originally since that would represent each other 'chair' making one move and the active chair making the initial move as well as a follow up. Intuitively this makes sense, keep in mind however, that in this case due to the nature of the agent using a deterministic self-similar model for other places the game is effectively being played by only two players and hence the depth required for a 'response' move in only 2.

The searching was limited by a timer akin to a 1, 1 chess clock, meaning the allowed time for a given turn is 1 second plus the accumulation of left over time from previous turns which is initially set to 1 second. Due to the lack of interruptible search, however it was uncommon for this time to be fully utilised on every turn with the average search time across 3334 turns being a mere 365 milliseconds. However this meant that a significant build-up of extra time would occur leading to occasional turns were drastically more time was spent searching, in one instance 8805ms was spent on a single turn.

The impact on search depth is profound with many short turns skewing the results heavily towards shallower searches of 4 or 5 levels while the less common multi-second turns yield depths up to 12 on rarer occasions, see Figure 10 below.

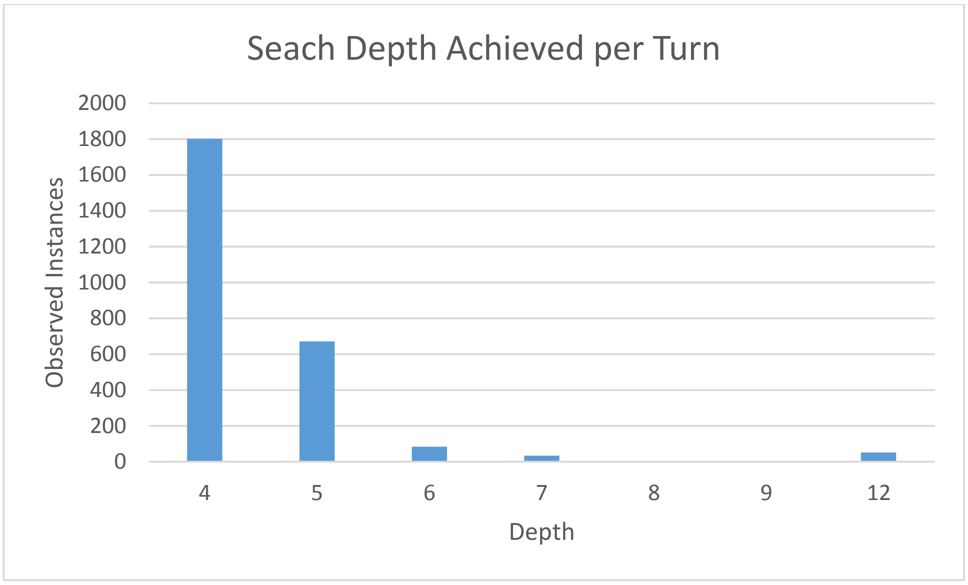


Figure 10 Search Depth

The implementation of alpha-beta pruning did provide a significant improvement in speed. Of 11,317,056 evaluated positions, a prune occurred in 1,189,427 cases. This equates to some portion of the search tree being pruned in 10.51% of all evaluations.

A modified search which computes alpha-beta cut-offs but does not act on them, instead performing a full search and counting states which would have been pruned, was used to find an estimation of the number of states which are pruned by alpha-beta. This suggested that out of a total of 30,068,490 states across many search iterations; 27,160,221 of them would have been pruned from the tree and not evaluated. That would be reduction of 90.33%. Owing to the large branching factor and, on average, shallow search depth a large reduction should be possible, however over 90% seems unreasonably high and may be the result of an error in calculation.

At the time of writing the majority of the data on which all of results calculations are based, as well as other related tools and information is available online at
http://www.cse.unsw.edu.au/~jrba250/Duchess/

Test Environment

All testing was on a single machine, running both the server and clients simultaneously. The server is very minimal, essentially acting as a broadcast hub, it performs no meaningful verification or other processing on moves received from clients meaning that running it alongside the clients removed network latency and other networking related hassles without adding significant additional burden on the processor.

Relevant test machine specifications:
CPU: Intel Core i7 3667U (Ivy Bridge) running at 3GHz with 2 physical and 4 logical cores.
8 GB RAM
64bit Windows 8 Pro
Java version "1.7.0_17"
Java SE Runtime Environment (build 1.7.0_17-b02)
Java HotSpot 64-Bit Server VM (build 23.7-b01, mixed mode)

There was sufficient RAM to accommodate six client agents and a server running simultaneously without extensive paging to disk being necessary, for this reason hard drive performance is deemed irrelevant and is not included.

Conclusion

Prior to this work TreeStrap and, especially, Duchess had not been explored in great depth making this inherently exploratory in nature. As a result much useful information was learned about both.

Duchess is just as complex as was suspected, more so in the case of checkmate difficulty. This makes it a useful testing ground for newly developed learning algorithms such as TreeStrap. Although it is unlikely to compare to Go with regards to shear breadth of state space, it boasts chess like qualities in a more challenging form, or it could very well have applications as a true multiplayer game for use with agents that perform modelling of their teammates and opponents with the aim of predicting their play.

TreeStrap, despite being able to learn master-level chess play starting from random weights, does retain the basic requirement of having to be able to reach an end state. It is possible it may have reached a win or loss state a sufficient number of times to learn a rudimentary heuristic using random starting weights which was equivalent to the human-supplied starting point given enough training games and perhaps a more advanced feature set. Thus allowing it to go on to learn a strong heuristic in the same way as it did for chess, however the required time and number of training games would be orders of magnitude greater than what was possible in this case. With a little assistance getting started however, TreeStrap showed very promising results with even a basic feature set, suggesting it has the ability to perform very strongly in Duchess and many other domains in which temporal-difference learning is applicable.

References

Baxter Jonathan, Andrew Tridgell, and Lex Weaver Experiments in parameter learning using temporal differences [Journal] // International Computer Chess Association Journal 21.2. - 1998. - pp. 84-99.

Baxter Jonathan, Tridgell Andrew and Weaver Lex Knightcap: A chess program that learns by combining td(λ) with game-tree search [Journal] // Proceedings of the 15th International Conference on Machine Learning. - 1998. - pp. 28-36.

Beal D. F. and Smith, M. C. Leaning Piece values Using Temporal Differences [Journal]. - [s.l.] : Journal of The International Computer Chess Association, 1997. - September.

Hauptman A. & Sipper, M. GP-endchess: Using genetic programming to evolve chess endgame players [Report]. - [s.l.] : Springer Berlin Heidelberg, 2005.

Korf R. E. Depth-first iterative-deepening: An optimal admissible tree search. // Artificial intelligence 27.1. - 1985. - pp. 97-109.

Pomerleau Dean A. ALVINN, an autonomous land vehicle in a neural network [Report]. - [s.l.] : Computer Science Department, CMU. Paper 1875., 1989.

Sturtevant Nathan Reed Multi-player games: Algorithms and approaches. - [s.l.] : Diss. UNIVERSITY OF CALIFORNIA Los Angeles, 2003.

Veness J [et al.] Bootstrapping from game tree search [Journal] // Advances in neural information processing systems, 22. - [s.l.] : Advances in neural information processing systems, 22, 1937-1945., 2009. - pp. 1937-1945.

Winands Mark Henricus Maria Informed search in complex games. - [s.l.] : Maastricht University Press, 2004.