Learning a Multi-Player Chess Game with TreeStrap

Diogo Real and Alan Blair
School of Computer Science and Engineering
University of New South Wales
Sydney, 2052, Australia
Email: d.real@student.unsw.edu.au, blair@cse.unsw.edu.au

***Abstract*—We train an evaluation function for a multi-player Chess variant board game called Duchess, which is played between two teams of two or three players, and is similar to Chess but with extra pieces, larger board size and significantly greater branching factor. Leaf positions in the alpha-beta search tree are evaluated with a linear combination of features, whose values are trained by self-play using the TreeStrap algorithm. We find superior performance can be achieved with an incremental approach, where the material values are learned first, followed by the attacking and defending values, and finally the piece-square values. To speed up the search, we evaluate board positions in a cumulative manner – identifying only those features that have changed, compared to the position at the previous move, and adjusting the evaluation accordingly.**

I. Introduction

In this paper we explore the learning of a single-layer neural network evaluation function for Duchess – a multi-player Chess variant board game for up to 6 players. Duchess was invented in 1984, and game-tested as a networked Java applet on the Internet in 1996. The feedback from users was generally positive, but a number of them suggested that an automatic player should be made available – to supplement the humans, or to take over in the event of a human player having to quit the game on short notice. Although considerable advances had already been made in Chess, Checkers, Backgammon and other board games, the computing speeds available at that time made it infeasible for these techniques to be adapted to a game like Duchess – which is similar to Chess but with multiple players, larger board size, additional pieces and consequently more expensive position evaluation and significantly greater branching factor.

Recent years have seen the introduction of a number of new machine learning techniques for strategic games [?], [?], [?], [?], [?]. Inspired by these advances – together with a general increase in computing power – we describe in the present work our preliminary attempts to learn an effective evaluation function for Duchess, using the TreeStrap algorithm introduced in [?] to train weights for features adapted from Chess and Checkers. To speed up the training, features are introduced incrementally in three stages of training.

II. Duchess

Duchess is played on a rose-shaped board by two teams, with either 2 or 3 players per team (Fig. ??). In addition to the King, Queen, Knight, Rooks, Bishops and Pawns, each player also has a Duchess (which is capable of Bishop and Knight moves), a Fortress (capable of Rook and Knight moves) and a Wizard (capable of King moves). Any piece next to

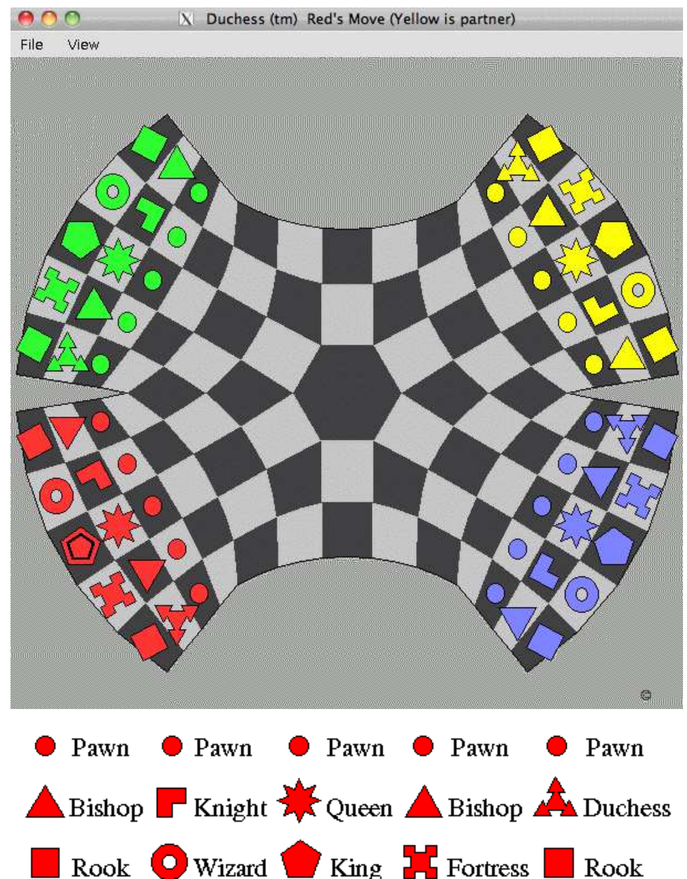

Fig. 1. The Game of Duchess.

the Wizard can move "by teleportation" to a square next to a partner's Wizard. Pawns can move forward, backward or sideways and capture in any diagonal direction. When a Pawn reaches the central "square", called the *Vortex* (which is really hexagonal) it can be promoted to any piece which has previously been captured. Full details of the rules can be found at `duchessgame.com`

In this paper we concentrate on the 4-player version of Duchess, although the approach could readily be extended to the 6-player version. We do not consider the 3-player version, which is played in *melee* mode where teams and alliances can change throughout the game. Melee mode presents its own particular game-theoretic and technical challenges, which have previously been explored in the context of other games [?] including an alternative multi-player Chess variant [?].

III. BOARD EVALUATION

Because it is deterministic, fully observable, and played between two fixed teams, techniques developed for Chess and Checkers can be adapted to the 4- or 6- player version of Duchess.

We adopt the framework – common to most modern computer chess programs – of enhanced alpha-beta search, with leaf positions evaluated using a one-layer neural network (although a variety of alternative approaches have also been tried, including multi-layer network evaluation, sometimes in combination with evolutionary computation [**?**], [**?**], [**?**], as well as stochastic search techniques such as Monte Carlo Tree Search [**?**]).

The network applies a sigmoid function to a weighted sum of discrete features, to produce a value between 0 and 1 – interpreted as the probability of winning from the specified position. The range of features typically include the following [**?**], [**?**], [**?**], [**?**]:

1) material (which pieces are remaining on the board)
2) check (whether a certain player is in check)
3) attacking and defending (whether a piece is attacking an enemy piece, or defending a friendly piece)
4) piece-square (a particular piece on a particular square)

These features number in the thousands, but they are "sparse" in the sense that for any given position, the number of active features is quite small, thus allowing rapid evaluation via array lookups. (For example, since the White King can only be on one square at a time, only one of the 64 piece-square features for that piece would be active in any given situation.)

The following table gives a rough estimate of the number of weights required for each type of feature, in Chess and Duchess. The 4-player version of Duchess is played on a board with 117 squares, with 9 different kinds of pieces (compared to 6 in Chess).

	Chess	Duchess
material:	6	9
check/checkmate:	2	$2\times 4 = 8$
attack/defend:	$6\times 2\times 6\times 2 = 144$	$9\times 4\times 9\times 4 = 1296$
piece-square:	$64\times 6\times 2 = 768$	$117\times 9\times 4 = 4212$
total:	920	5525

A small number of additional features need to be added, to capture the characteristics of a multi-player game. In Duchess, although a player in checkmate is unable to move, the game is not finally lost until all players of the same team are in checkmate simultaneously. We therefore add extra features to indicate whether each of the players is in checkmate.

Other types of features are sometimes used in Chess programs, for example mobility (the number of moves available to each player). But, we excluded these from the present study, mainly because they are too costly to compute in a game like Duchess.

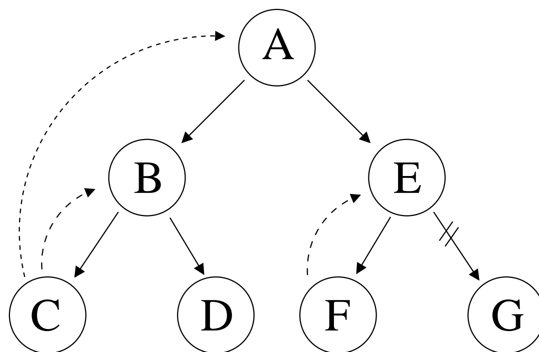

Fig. 2. TreeStrap algorithm. The values for Nodes A and B are both trained toward that of Node C. Although Node E is not on the line of best play, its value is still trained toward that of Node F. If F causes a *cutoff* (thus pruning G) then E can be trained toward the lower or upper bound provided by F, if its current value is on the wrong side of the bound.

IV. LEARNING THE FEATURE WEIGHTS

Once the features have been determined, the weights for these features can be learned by various stochastic gradient descent methods, all of which involve training the value of earlier board positions toward the value of subsequent or successor positions.

Temporal Difference learning [**?**] is a general-purpose algorithm which trains the value of the current position toward that of one or more subsequent positions in the game. It has successfully been applied to stochastic games like Backgammon [**?**]; but, for highly tactical games like Chess, better performance can be achieved through specialized methods which combine reinforcement learning with game tree search. The first such method, now known as TD-Root, was introduced in Samuel's 1959 checkers player [**?**] where the heuristic function for the current position was trained toward the result of a minimax search from a subsequent board position. This approach was later refined in TD-Leaf(λ) [**?**] where the leaf node along the line of best play from the current position is trained toward the leaf node along the line of best play from a subsequent position (see also [**?**]). The TreeStrap algorithm [**?**] does not restrict its training to the root or a single leaf node but instead trains the value of every (non-leaf) node in the search tree toward the value of the leaf node along the line of best play in the subtree rooted at that node (Fig. **??**). In doing so, it refines its evaluation not only of the "good" positions that were selected, but also of the "bad" positions which were considered but rejected. When combined with alpha-beta pruning, the recursive evaluation sometimes returns only an upper or lower bound rather than an exact value. TreeStrap handles this by training towards the upper or lower bound, if the current value is on the wrong side of the bound.

Previous work has shown that TreeStrap could learn Chess by self-play from random initial weights and achieve Master level play, whereas TD-Leaf(λ) could only achieve amateur level play under the same conditions [**?**]. For this reason, we use TreeStrap in the present work on Duchess.

V. Training and Search

We employ a negamax implementation of alpha-beta search, with the evaluation always from the perspective of the player whose turn it is to move. For the 4-player version of Duchess, the board shape from the perspective of the even team is different from that of the odd team (see Fig. **??**). Therefore, two separate sets of weights are maintained, and the weights are selected based on the player whose turn it is to move.

From among the standard techniques for speeding up alpha-beta search, we employ iterative deepening search and the killer move heuristic, but not transposition tables or quiescent search.

The killer move heuristic is based on the assumption that if a move at depth d in some part of the search tree is good enough to cause a *cutoff* (thus pruning off some branches) then the same move is likely to cause a cutoff at depth d in some other part of the tree. So, for each depth d, the move at depth d that most recently caused a cutoff is retained, and that move is tried first (if it is legal) at subsequent depth-d nodes in the tree.

Chess programs typically search about 14 moves ahead in the mid-game, and the same position may re-occur many times in different parts of the search tree. Positions are therefore stored in a *transposition table* with Zobrish hashing for fast lookup. When a previously searched position is encountered, the stored evaluation is used instead of re-searching. In our current implementation of Duchess, due to the significantly larger branching factor, it is only practical to search 5 to 7 moves ahead in the mid-game and 9 to 13 moves in the end-game, which amounts to only one or two moves for each player. For this reason it is unlikely that the same position would occur many times within the tree, so there is not so much benefit to be gained from transposition tables.

Quiescent search refers to the practice of extending the game tree in situations where a King is in Check or a valuable piece is threatened, until we reach a *quiescent* position. This technique is not easily applied in Duchess, because threats to a King, Queen, Fortress or Duchess occur at almost every move in the game, so the tree could potentially be extended without limit. Instead, we must rely on the attacking and defending features to provide an effective evaluation even for non-quiescent positions.

A. Cumulative Evaluation:

In Chess, heuristic evaluation is normally computed from scratch for each new position in the tree. But for Duchess, we find it is more efficient to compute it *cumulatively* from the previous position, by adjusting only those features which have changed as a result of the latest move. In general, when Piece X is moved from Square A to Square B, the evaluation is achieved by the following steps:

1) Examine all squares one Knight move away from Square A; if a Knight-capable piece is found, subtract the feature for that piece to be attacking or defending Piece X (or vice-versa if Piece X is Knight-capable).
2) Scan along each diagonal from Square A until you find an occupied square, or the edge of the board. If a Bishop-capable piece is found (or a King, Wizard or Pawn in the case of a unit Bishop move) subtract the feature for that piece to be attacking or defending Piece X; also check for possible *discovered* attacks along the diagonal passing through to the other side of Square A.
3) Repeat Step 2) but this time scanning along rows and columns, searching for Rook-capable pieces as well as pieces attacked or defended by Piece X, and discovered attacks along the row or column passing through Square A.
4) Repeat Steps 1) - 3) for Square B, this time looking to add features for Piece X to be attacking or defending other pieces (and vice versa) as well as *blocking* of Rook or Bishop attacks passing through Square B.
5) If a piece on Square B is captured, subtract the features involving the captured piece.
6) If the move results in a new attack on a King, or the removal of an existing attack, abandon the cumulative approach and instead compute the entire feature vector from scratch, for this board position.

B. Scaling of Learning Rate by Depth

When considering the learning rate for TreeStrap, we need to bear in mind that weight updates are applied not only at the root but at every non-leaf position in the (pruned) search tree – thus introducing a *multiplier* effect which could scale exponentially with the depth of the tree.

In the present work, we mitigate this effect by introducing a scaling factor which assigns an exponentially lower learning rate to positions further away from the root. Specifically, the learning rate is

$$\mathrm{LR} = \mathrm{LR}_0 \lambda^d,$$

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 (typically 10^{-4} or 10^{-5}). There are two reasons for the exponential scaling: Firstly, the nodes near the root of the tree are being trained on the basis of a more extensive subtree, thus intuitively providing "better quality" information. Secondly, it may happen that the best move at one leaf of the search tree is also the best move at thousands of other leaves within the tree. Although the individual positions will differ, the *difference* between each leaf node and its parent (in terms of which features are active) may be substantially preserved, thus amplifying the modification to those particular weights. For example, if a Pawn is promoted to a Queen at the final move, the value of the Pawn will be lifted toward that of the Queen (and may even exceed the value of the Queen if the exponential scaling were not applied).

As an additional precaution, we enforce that no individual weight is allowed to change by more than 0.001 in a single move. This kind of per-weight limit was also imposed in previous work on Chess [?], where the potential "blowing up" of weight updates was already recognized, but was mitigated to some extent by the use of transposition tables, ensuring that identical positions occurring multiple times in the tree would only be updated once.

In order to encourage diversity, the first four moves of each game are played randomly. After that, the "best" move is always chosen, according to the network's current evaluation and search. The weight updates are applied at the end of each move, so the player's strategy changes as each game progresses.

There is a tradeoff between depth of lookahead and speed of training. With four players in the game, 5-move lookahead is the minimum depth required for the network to observe the effect of two consecutive moves by the same player – for example, moving the same Knight twice, moving two mutually protective Pawns "in formation" so that they continue to protect each other, or promoting a Pawn to a Queen, Fortress or Wizard and then using that piece to good effect. With only 5-move lookahead, a significant number of games end in stagnation (indicated by 200 consecutive moves without a capture or promotion). In order to increase the number of games ending in a result, we insist on at least 7-move lookahead when the number of pieces is less that 20, and 9-move lookahead when the number of pieces is less than 8.

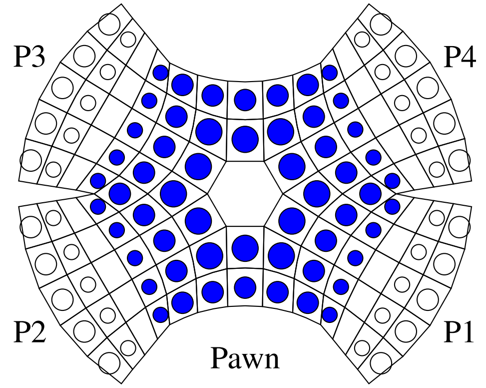

Fig. 3. Initialized pawn-square weights. The odd team (P1 and P3) are playing against the even team (P2 and P4).

TABLE I. TRAINED MATERIAL WEIGHTS

Piece	Weight	Feature	Weight
Fortress	0.12	P1 check	-0.05
Duchess	0.11	P2 check	0.08
Wizard	0.06	P3 check	-0.04
Wiz(×2)	0.05	P4 check	0.04
Queen	0.13		
Rook	0.09	P1 checkmate	-0.33
Bishop	0.07	P2 checkmate	0.26
Knight	0.05	P3 checkmate	-0.22
Pawn	0.06	P4 checkmate	0.24

VI. EXPERIMENTS

Stage 1: Material, Check and Checkmate

In the first stage of training, we include only weights for material, pawn-square, check and checkmate, as well as a bias weight.

We insist that the material weights must be the same for all players. That is, the value of your partner's Queen must be equal to that of your own Queen, and the value of an enemy Queen must be equal in magnitude (but opposite in sign) to that of your own Queen.

Since the main value of the Wizard is to enable teleportation, two Wizards provide more than twice the benefit of a single Wizard. Therefore, we introduce a new feature to indicate whether the team has both its Wizards still on the board. (For the 6-player version, an extra feature for three Wizards would also be needed.)

The bias weight serves two purposes. First, a slightly positive bias accommodates the fact that an enemy piece may be captured on the next move. Second, the bias for the even and odd weights can differ – thus allowing for the fact that the even team has a slight advantage over the odd team (at least in the opening game) because the team members are able to directly attack their closest enemy, without the opportunity for other players to intervene.

Although TreeStrap was able to learn a good evaluation for Chess from random initial weights [?], the training for TreeStrap and other methods can be sped up considerably by initializing the material weights [?], [?]. With this in mind, we initialize the material weights to 0.1 for the Queen, Duchess and Fortress, and 0.05 for the Knight, Rook, Bishop and Pawn. We assign an initial weight of 0.05 for each Wizard individually and an additional 0.05 if a team has both their Wizards on the board. The weights for check and checkmate are initialized to 0.05 and 0.25, respectively.

In order to encourage Pawns to move toward the Vortex for promotion, we include a weight for the location of each Pawn, which is

$$0.04 - 0.01\,d$$

where d is the distance (number of King moves) from its current location to the Vortex (see Fig. ??).

This network was trained for 8 hours (140 games) with a root learning rate of 10^{-4}. The values that emerged are shown in Table ??.

Pawns in Duchess are much more valuable than in Chess because they can move or capture in any direction, and have a greater likelihood of being promoted. The Knight has a low value compared to the Bishop (and even the Pawn) due to its limited mobility, in relation to the larger board size.

Checkmate has a large (negative) value for the player whose turn it is to move (P1) and steadily declines for Players P2, P3 and P4. The weight for P2 being in check is relatively large, because P1 may have a chance to capture one of his pieces. The previous player (P4) can only be in check if he is also in checkmate, so these two weights are only active in combination.

Stage 2: Attacking and Defending Weights

In the second stage, we add additional weights to indicate whether one piece is attacking or defending another piece.

To avoid a blowout in the number of activated features, we do not include attacking and defending when it is achieved by teleportation. Thus, a piece next to a Wizard is considered to be protected by the Wizard itself, but not by any other piece next to the Wizard (or the partner's Wizard).

As a starting point, we took the existing weights from the (trained) Stage 1 network and initialized the (new) attacking and defending weights to zero. The full set of weights in this new (Stage 2) network was then trained for 20 hours (220 games) with a root learning rate of 10^{-5}.

We compared the two networks by having them play a number of games against each other and counting the number of wins, losses and draws. The first four moves of each game are chosen randomly. Games are played in duplicate pairs, with the two networks swapping roles for the second game in each pair, and both games beginning with the same sequence of four moves (chosen randomly).

Out of 100 games played under this protocol, the Stage 2 network achieved 85 wins, 13 losses and 2 draws against the Stage 1 network.

In situations where it is not possible to force a material advantage within the search horizon, the Stage 2 network is able to choose moves which lead to a favorable arrangement of pieces attacking and defending each other, giving it a significant advantage over the Stage 1 network.

The attacking and defending weights from the Stage 2 network are shown in Fig. **??**.

Row 1, columns 2 and 4 are active when pieces belonging to the player whose turn it is to move (P1) are attacking enemy pieces (P2, P4). High weight is assigned to a Queen, Fortress or Rook attacking an enemy piece, or to any piece attacking an enemy Queen, Fortress, Rook, Wizard or Pawn. Attacks on the King are supplementary to the check feature being active.

The four blocks of weights along the diagonal in Fig. **??** are active when a player's own pieces are defending each other. High weight is given for a King next to a Wizard (allowing teleportation out of check) and for two Pawns defending each other in a mutually protective formation. Blocks P1 $\leftrightarrow$ P3, P2 $\leftrightarrow$ P4 reflect the benefit of defending your partner's pieces – either by moving into the center of the board, or by teleporting a piece (especially the Queen) to a square next to your partner's Wizard.

Similar to what has previously been reported for TD-Leaf(λ) [**?**], [**?**], networks trained directly using the attacking and defending weights combined with material weights (with either high or low learning rate) perform poorly compared to the Stage 2 network trained using the Stage 1 weights as a starting point. The reason seems to be that the untrained material weights lead to a number of games ending early, with the winning team having several of their pieces remaining in their initial locations. The attacking and defending weights are therefore being used as a "proxy" for the material weights.

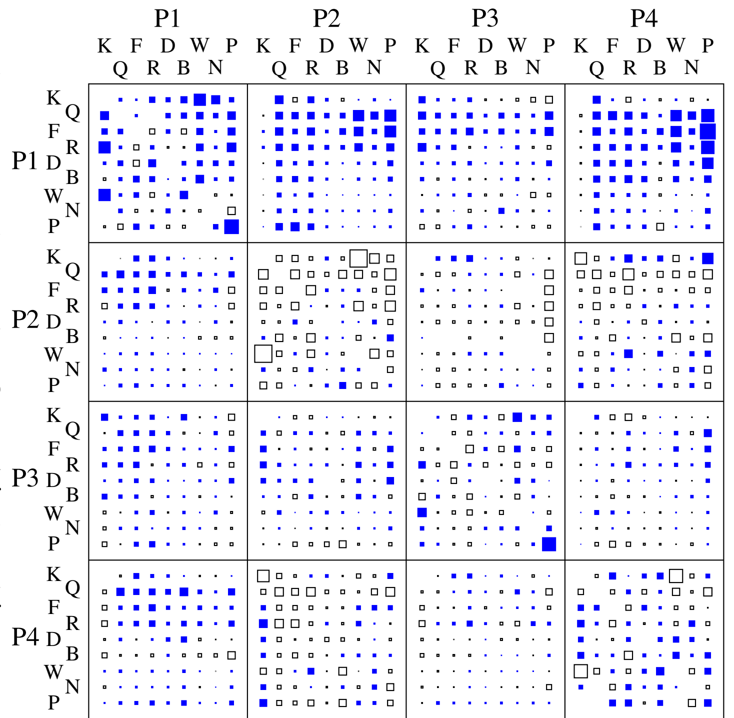

Fig. 4. Hinton diagram showing the weight of one piece (row) attacking or defending another piece (column), from the perspective of player P1 (whose turn it is to move). P3 is the partner of P1; the opposing team are P2, P4. Positive weights are indicated by filled-in squares; negative weights by empty squares.

For example, if the game is won with the Wizard and Knight having not moved at all, the weight for a Wizard protecting a Knight will be boosted unreasonably. By first training the material weights, we ensure that games continue long enough for most pieces to be deployed, and also ensure that the attacking and defending weights are properly tuned to the value of the piece being attacked or defended.

Looking again at column 1 of Fig. **??**, we see that some weights in rows 2 and 4 are positive when we would perhaps expect them to be negative. These weights might be acting to some extent as a proxy for the material weights, if the material weights are not quite settled. Another consideration is the mode of attack. We expect the weight for P2's Queen attacking P1's Queen to be positive, because it logically follows that P1's Queen would also be attacking P2's Queen, and P1 will have the advantage of moving first. When it comes to the Queen attacking the Fortress, this could be either good or bad depending on whether the Fortress is also attacking the Queen, i.e. whether the Queen is attacking with a Rook move or a Bishop move. In future work, it may be advantageous to introduce additional weights which distinguish these different modes of attack.

Stage 3: Piece-Square Weights

In the final stage, we add additional weights for the location of each piece on the board [**?**], and train the full set of weights in this new network for a further 4 hours (60 games). The resulting piece-square weights are shown in Fig. 5.

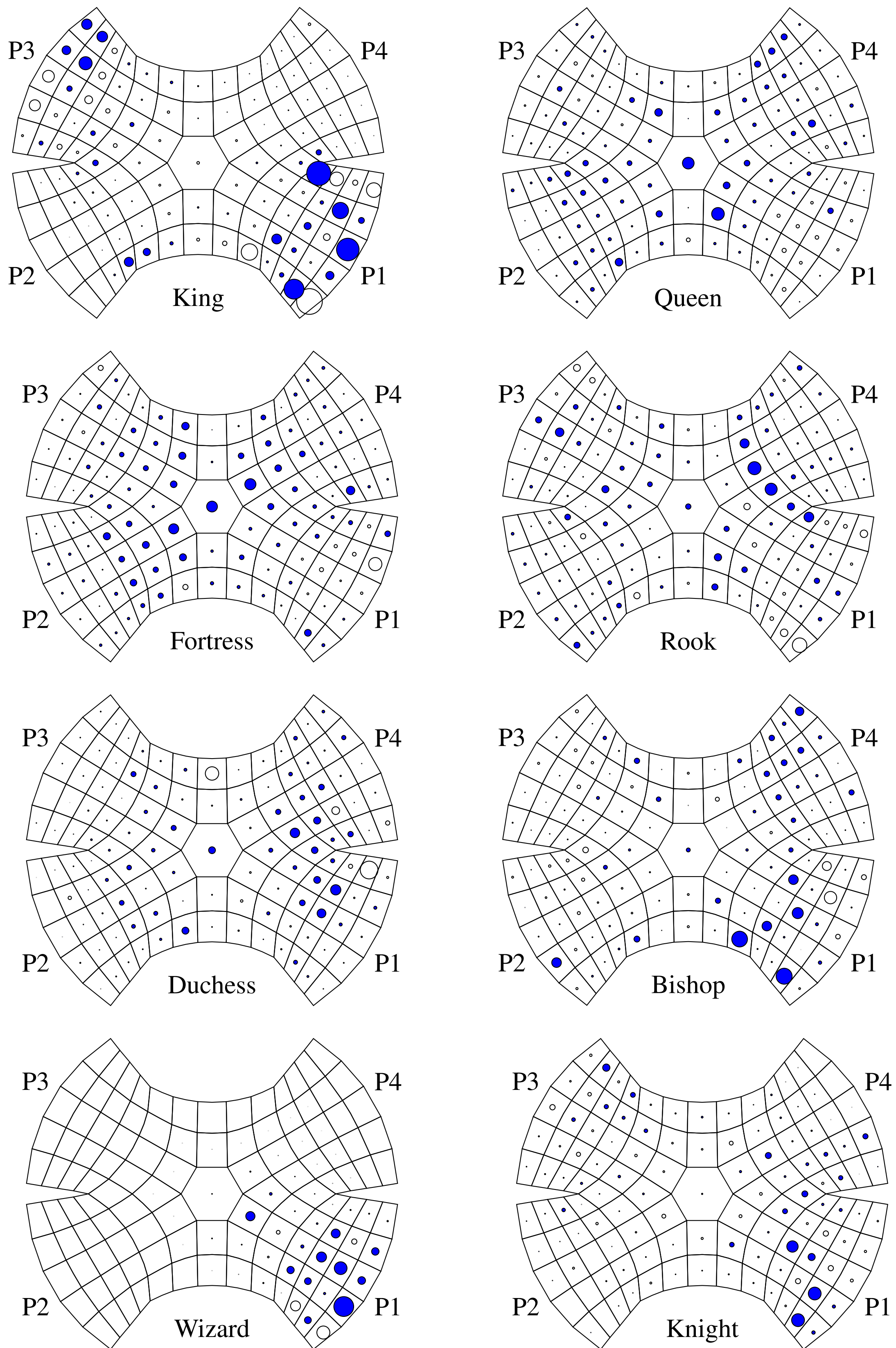

Fig. 5. Trained piece-square weights for pieces belonging to P1, whose turn it is to move. Positive weights are indicated by filled-in circles; negative weights by empty circles.

We see that the initial square(s) for the Fortress, Duchess, Queen and Rook end up with a negative weight, thus encouraging these pieces to move away from their initial location and toward the center of the board. The Vortex is highly favored, along with other squares from which enemy pieces can most easily be attacked.

In contrast, the network has developed a suspicious preference for the King, Wizard and Knight to remain in their initial locations – perhaps a consequence of the fact that a triumphant player can sometimes capture enemy material, or force a checkmate, with these pieces not having moved at all. For the Bishop and Pawn, some of the initial locations are positive while others are negative.

The Rook, Queen, Fortress and especially the King are encouraged to teleport to P3's section of the board. Locating the Wizard next to the Vortex is also beneficial, because it allows Pawns to teleport directly to the Vortex for promotion.

In a contest of 100 games, the Stage 3 network won 53 games, lost 38 games and drew 9 games, against the Stage 2 network. Thus, the inclusion of the piece-square weights appears to provide some additional benefit, compared to the material and attacking-defending weights on their own, but we cannot say at this stage that it is a significant benefit.

It may be that piece-square weights in Duchess are inherently less important than in Chess, because the board quickly opens up and the effect of the initial configuration is dissipated. Locations that are good in the mid-game might be bad in the end-game and vice-versa. This problem could perhaps be addressed by training different networks for different stages of the game, and switching when the number of pieces falls below a pre-defined threshold.

It may also be that greater search depth is required in order to get maximal benefit from the TreeStrap algorithm. If the search depth in the mid-game were increased to 9 or 13, the network would have the opportunity to learn from a sequence of 3 or 4 moves by the same player. In the endgame, the same piece may need to be moved three or four times in order to threaten an opponent's King; additional features such as the number of moves required to reach the King(s) may prove beneficial in this situation.

VII. Conclusion

We have successfully trained a Duchess-playing agent, using the TreeStrap algorithm to learn feature values for a single-layer neural network evaluation function combined with alpha-beta search.

The inclusion of attacking and defending weights provides a substantial benefit over material weights alone. Piece-square weights also provide some additional benefit, but the TreeStrap algorithm seems to have difficulty in fully optimizing these weights, possibly due to the limited search depth in the mid-game. In future work, we aim to optimize our implementation and extend the search depth to 9 or 11 moves in the mid-game, in order to measure the effect of search depth on the quality of the evaluation. We also plan to refine the attacking and defending weights, based on the mode of attack (Rook, Bishop or Knight), and explore additional features such as number of moves to the opponent King (particularly in the endgame) to see what additional benefit they may provide.

Alternative approaches such as evolution, MCTS and deep learning of board features would also be interesting to explore.

Acknowledgment

The authors would like to thank Joel Veness for helpful advice and discussions.

References

[1] Baxter, J., A. Tridgell & L. Weaver, 1998. Knightcap: a chess program that learns by combining TD(lambda) with game-tree search, *Proc. 15th International Conf. on Machine Learning*, 28–36, Morgan Kaufmann, San Francisco, CA.

[2] Beal, D.F. & M.C. Smith, 1999. Learning piece-square values using temporal differences, *Journal of the International Computer Chess Association*, 22(4), 223-235.

[3] Buro, M., 1998. From simple features to sophisticated evaluation functions, *Computers and Games*, Springer, 126-145.

[4] Campbell, M., A. Hoane & F. Hsu, 2002. Deep Blue, *Artificial Intelligence* 134, 57–83.

[5] Chellapilla, K. & D.B. Fogel, 2001. Evolving an expert checkers playing program without using human expertise, *IEEE Trans. Evolutionary Computation* 5(4), 422-428.

[6] David, O.E., H.J. van den Herik, M. Koppel & N.S. Netanyahu, 2014. Genetic algorithms for evolving computer chess programs, *IEEE Trans. Evolutionary Computation* 18(5), 779-789.

[7] Fogel, D.B., T.J. Hays, S.L. Hahn & J. Quon, 2004. A Self-Learning Evolutionary Chess Program, *Proceedings of the IEEE* 92(12), 1947-1954.

[8] Lorenz, U. & T. Tscheuschner, 2006. Player modeling, search algorithms and strategies in multi player games, in *Advances in Computer Games*, 210–224, Springer Berlin Heidelberg.

[9] Ramanujan, R. & B. Selman, Trade-Offs in Sampling-Based Adversarial Planning, Proc. 21st Int. Conf. Automat. Plan. Sched., Freiburg, Germany, 2011, pp. 202209.

[10] Samuel, A.L., 1959. Some studies oin machine learning using the game of checkers, *IBM Journal of Research and Development* 3, 211–229.

[11] Schaeffer, J., M. Hlynka & V. Jussila, 2001. Temporal difference learning applied to a high performance game playing program, *Proc. 17th Int'l Joint Conf. on Artificial Intelligence*, 529–534, Morgan Kaufmann.

[12] Sturtevant, N.R., 2003. *Multi-player games: Algorithms and approaches*, Doctoral dissertation, University of California Los Angeles.

[13] Sutton, R., 1988. Learning to predict by the method of temporal differences, *Machine Learning* 3, 9–44.

[14] Tesauro, G., 1992. TD-gammon, a self-teaching backgammon program, achieves master-level play, *Neural Computation*, 6, 215–219.

[15] Veness, J., D. Silver, W. Uther & A. Blair, 2009. Bootstrapping from game tree search, *Advances in Neural Information Processing Systems 19*, 1937-1945.