Automated Suicide: An Antichess Engine

Jim Andress and Prasanna Ramakrishnan

1 Introduction

Antichess (also known as Suicide Chess or Loser's Chess) is a popular variant of chess where the objective of each player is to either lose all of her pieces or be stalemated. To facilitate this goal, capturing is compulsory if possible, and the king has no "royal powers" (that is, it behaves as a regular piece, can be captured, and there is no notion of castling or check).

For our project, we have developed an engine capable of playing Antichess at roughly an expert level of play. We chose to focus on playing Antichess because of unique properties which separate the game from standard chess and other chess variants. In particular, the presence of forced captures results in a significantly smaller branching factor for the game tree of Antichess as compared to chess and other variants. In fact, the game's low branching factor made it possible for researchers to weakly solve the game in October 2016 [1].

Our process for solving the problem was fairly standard, drawing on techniques from both search and reinforcement learning. First, we developed a model and corresponding code framework to cast the problem as a collection of states and actions which can be searched by Minimax. We then created a linear state evaluation function whose weights were computed by TD-learning on an online dataset. Finally, we fine tuned our Minimax implementation to be especially efficient for Antichess.

In terms of evaluating the strength of our engine, we were confident that our engine would be able to consistently beat an engine that either plays random moves or plays moves with little information (i.e., uses Minimax search, but with a very simple evaluation function). Our more advanced goals were for our engine to be able to beat (or at least mimic) a strong human player or engine. To a large extent we accomplished these goals.

2 Literature Review

Since 1968, when John Good first postulated a strategy for building a chess computer [2], the methodology for solving chess games has always been based on some sort of learning to make an accurate board evaluator and a version of Minimax with Alpha-Beta pruning to search the game tree. About three decades later, in 1997, IBM's chess computer Deep Blue was able to beat Chess world champion Gary Kasparov thanks to its powerful computing and ability to selectively extend search to go deeper into particular lines [3].

Current state of the art chess engines have harnessed stronger computers and more accurate board evaluators. They've also been able to calculate a number of positions beforehand using opening books and endgame tablebases. Based on what we've seen from other open source Antichess engines (Stockfish, Nilatac,

Sjeng), the problem solving paradigms being used are the same as those for Chess. However, Antichess is unique in that its opening book is far more extensive than it is for chess; some opening moves have been solved entirely. In particular, Nilatac has shown that at least 8 of the 20 opening moves for white are losing [4].

For our implementation, we chose to use TD-Learning to train our board evaluator. First described by Sutton [5], TD-learning is a technique which finds optimal feature weight values by processing large sets of recorded games. TD-learning was successfully applied by Tesauro to play backgammon [6], and similar ideas were recently applied in Google's AlphaGo engine [7].

3 Model

We've modeled Antichess as a typical game with states and actions. The states are represented in code by a Board data structure which stores the current position of each piece in the game grid, as well as the player whose turn it currently is. Although the above is a minimal representation of the game state, we also supplement this minimalist state with other convenient information like the number of available moves. The actions at any given state are the valid moves for the current player. In our implementation, we've represented a move as a (start position, end position) pair, where the start is where the original piece was and the end is where it is moved to. During capture moves, the piece replaces whatever was previously in the end position (if there was anything there at all). Figure 1 provides an example in which state s (the start state) can transition to states s', s'', or s''' via the actions $a2a4$, $b1c3$, or $e2e3$, respectively. Note that the figure shows only a fraction of the available actions.

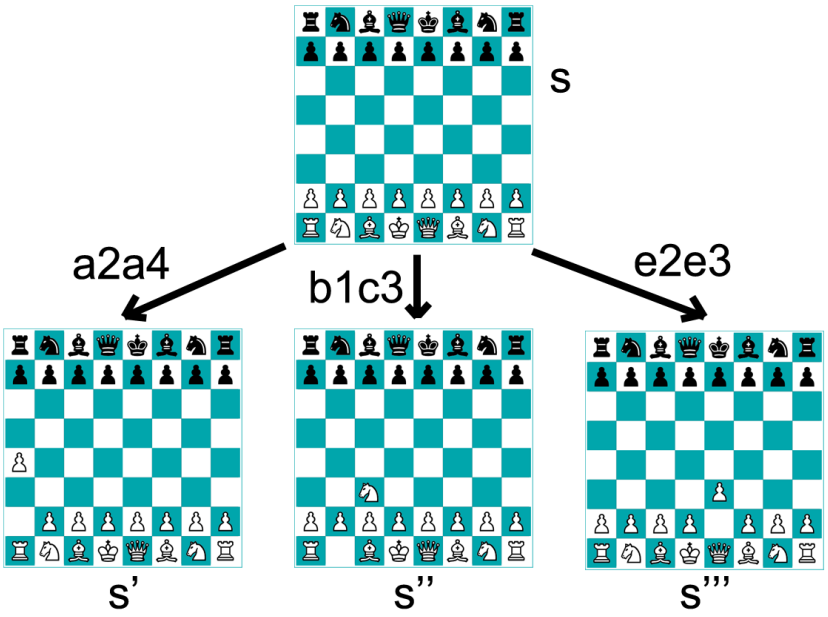

Figure 1: The starting state s in a game of Antichess, as well as three possible actions and their corresponding successor states.

Developing this framework consisted of building a "Board" interface for the computer to process a position and know the possible moves allowed at the position. This infrastructure involved a non-trivial amount of work given the number of different pieces in chess, each with its unique movement rules and special cases to consider.

3.1 Feature Extraction and Evaluation

In order to tractably model the state of a game board, we decided to extract a set of numerical features from the positions of the various game pieces. Choosing the specific features to use was an essential step in the modelling process and a non-trivial one, as we sought to capture the important aspects of a configuration without risking over-fitting.

In our approach, we first turned to the standard features used in regular chess analysis: material, mobility, king safety, and center control. Of course, king safety is not particularly relevant in Antichess since the king has no royal powers, and we found that including center control resulted in the computer keeping all of its pieces irrationally far away from the center (center control doesn't translate as well to Antichess because the "control" and "attack" dynamics behave very differently than they do in regular chess). As a result, the main features that we chose to focus on were material and mobility.

For the material features, we simply counted the number of each piece that each player had. We wanted to make sure that the resulting features could be applied symmetrically when playing as either white or black, so we chose our feature templates to be "my pieces x" and "opponent pieces x", where x could be any one of the six types of pieces.

For the mobility feature, we began by simply including the number of available moves for the given player and board configuration. However, we soon observed that because in each position a player either must take a piece or cannot, the number of legal moves is always at one of two extremes; it is almost always less than 5 when a player must take a piece, and at least 20 otherwise. These extreme values meant that simply the number of legal moves did not provide a good descriptor of the true board situation, since its impact on the position is not linear (the difference between having 1 move and 6 moves is not the same as having 30 moves and 35 moves). To normalize the extremes onto a more linear scale, we added the square root of the number of legal moves as a feature (we also tried to use log rather than squareroot, but found that log didn't capture the lower extremes well enough).

4 Learning

Because the game tree in Antichess is so deep, it is in general not possible to search all the way to leaf nodes while evaluating states. We therefore took the standard approach of limiting our search to a certain depth (3 plies in our case) and approximating the value of a state with a linear evaluation function. We first extract the set of numerical board features described above and then use a linear combination of these features to form an approximation of the value of that board configuration.

In order to determine the best weights to use in this linear combination, we turned to reinforcement learning techniques. In our specific problem, the number of possible board states is extremely large, as is the number of possible actions which can be taken. We therefore decided to avoid techniques like Q-learning,

which try to compute the value of each $(state, action)$ pair. Instead, we turned to TD-learning. TD-learning tries to approximate the value of just a *state* and works on the intuition that the evaluation function in adjacent states ought to be highly correlated.

As an example, suppose we are in a state s_t at time t, and suppose further that our current set of feature weights are such that the evaluation function gives state s_t a value of 100. We then take what we believe to be the optimal action and transition to state s_{t+1} in time $t+1$. However, the evaluation function value for state s_{t+1} is only 5. The presence of such a large disagreement between evaluation function values in consecutive time steps suggests that our feature weights are incorrect. TD-learning will therefore update the weights in such a way that the predicted values for states s_t and s_{t+1} will be much more similar to each other.

Specifically, if we begin in state s_t at time t and take an action which leaves us in state s_{t+1} with reward r at time $t+1$, then in standard TD-learning the weights will be updated according to the rule

$$w_{t+1} = w_t + \eta \left(r + f(s_{t+1}) - f(s_t)\right) \nabla_w f(s_t) \tag{1}$$

where f is the evaluation function, η is the learning rate, and w are the weights. Although this update rule will eventually lead to the weights converging to values consistent with the game rewards, it can be quite slow in practice since feedback about a state is only propagated back to the state which came directly before it. We therefore decided to instead use the variant TD(λ). In this modified version, the update rule becomes

$$w_{t+1} = w_t + \eta(r + f(s_{t+1}) - f(s_t)) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w f(s_k)$$

Now, feedback about each state is propagated back to all predecessor states, and the new parameter $0 < \lambda < 1$ controls how that influence drops off over time. In our specific implementation we use a value of $\lambda = 0.7$ which was described in the literature as a good baseline value to use.

In order to run, the TD-learning algorithm requires a large collection of full games, including all states and actions taken. To quickly gain access to such a dataset, we scraped full games of Antichess from the Lichess.org [1]. We then ran TD-learning as the winning player, which should in theory allow our feature weights to converge to values leading to wins in the training games.

5 Search

For the search portion of the engine, we used a standard Minimax search with a number of optimizing modifications. The first modification we made was Alpha-Beta pruning, which allows us to limit our search by evaluating positions only if there's a chance that it would be part of the optimal path chosen by a usual Minimax search. That is, it keeps lower and upper bounds on what evaluation the search has already guaranteed, and does not search nodes whose evaluations lie outside those bounds.

By observing many Antichess games, we found that often times the advantage would go to one player when she could force her opponent to take pieces as often as possible. Eventually, the material is so unbalanced

[1] `https://en.lichess.org/games/search`

that the winning player can always find an opponent's piece that can take each of her pieces.

Originally, we thought that we could account for this dynamic by having a large weight on the "legal moves" related features, but for the reasons described above these features turned out to be less informative than we thought. With that in mind, we decided to encode this intuition through the search algorithm by allowing Minimax to search a little farther whenever a forced move was encountered. That is, in the recursive call for each position with one legal move, we incremented the depth by a small ε (we used 0.25 or 0.1 depending on how much time we wanted the computer to use) rather than decrementing it by 1. These features of our search algorithm are shown in Figure 2.

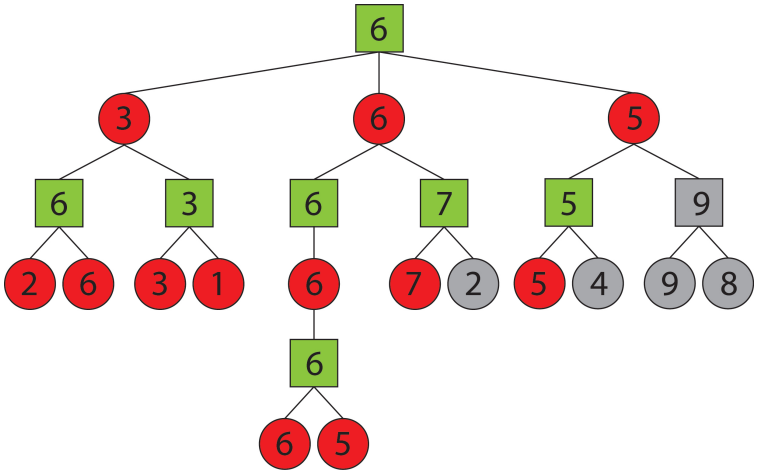

Figure 2: An example game tree demonstrating features of our search algorithm. The green nodes represent the maximizing agent, red nodes are the minimizing agent, and grey nodes are those that would be ignored due to Alpha-Beta pruning. Additionally, we can see that one branch of the tree is longer since it involves nodes containing only a single valid action.

Finally, we made a modification that allowed the algorithm to maintain a game tree, and hence not need to recompute the possible moves from each position every time it is the computer's turn. This modification gave us another boost in efficiency by allowing us to fully take advantage of Alpha-Beta pruning. Because of the nature of the pruning algorithm, if we happen to search better lines first we can prune more aggressively later on. Keeping track of a game tree allows us to easily order the states to search by a heuristic (such as the number of successors a state has, or the previous evaluation at that state), when otherwise we would have to compute all of these values by scratch every time the computer is prompted for the next move.

6 Results

First, we show the result of our TD-learning in Figure 3. The weights learned from our Lichess dataset agree with our general intuition about Antichess. In particular, our engine's material weights are negative, meaning it wants to lose its pieces, and its square root mobility weight is positive, meaning it wants to maximize the number of moves it has available. The opposite holds for the opponent weights.

We evaluated the success of our Antichess engine by comparing its performance against three different types of opponents: simple baseline computer models, human players, and advanced third-party Antichess

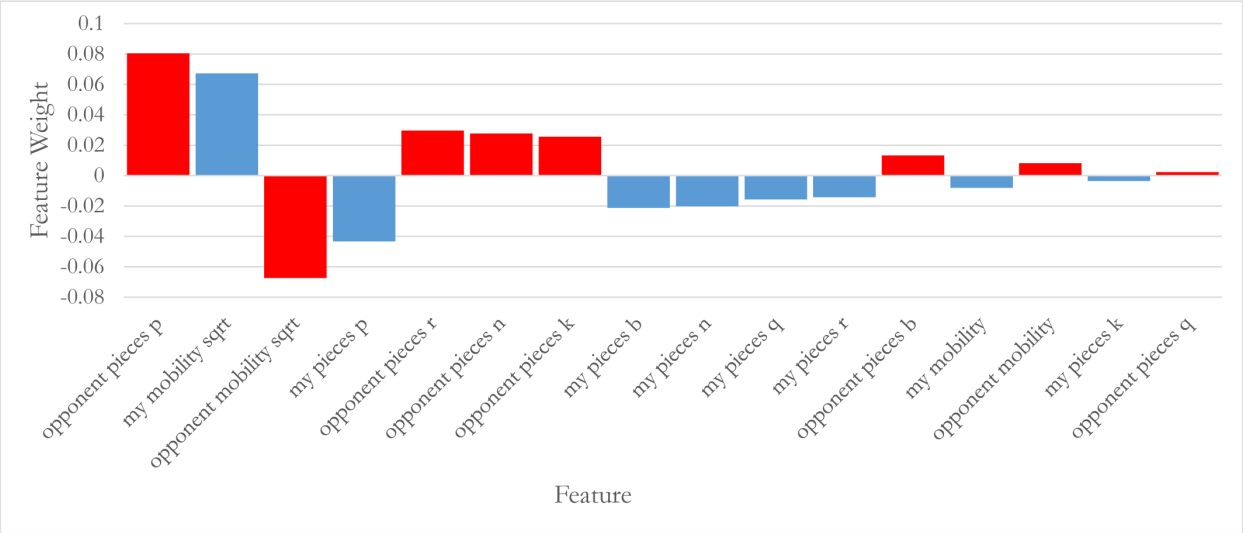

Figure 3: The learned feature weights, sorted by decreasing absolute value. The red weights correspond to features of the opponent's pieces while blue weights correspond to features of our engine's pieces.

engines. The first category, baseline computer models, was intended to validate the correctness of our algorithm. We implemented several baseline algorithms of increasing complexity:

- A randomize engine which simply chose a move uniformly at random from the set of available moves.
- A simple Minimax engine with a basic evaluation function. Rather than using complicated board features with weights learned from data, the evaluation function for this engine simply returned the number of available moves.
- An advanced Minimax engine with a complex evaluation function. This engine uses the same features and weights as our final engine, but it does not explore forced lines any deeper than normal moves.

By playing these engines against each other, we confirmed that they had the relative levels of difficulty that we expected; that is the simple Minimax engine beat the randomized engine, and the advanced Minimax engine beat both of the other two.

Given the same board configuration, our engine will always extract the same feature values and combine them using the same feature weights, meaning that a board will always have the same evaluation function value. Thus, our engine is deterministic: given any specific board configuration, it will always choose the same move. In order to test the true strength of our engine, we ran our experiments with an added random component. During these tests, each time our engine made a move there was a small probability ε that it would simply choose a random move rather than its usual move. By varying ε, we are able to test how much better our engine is than the baselines since we can see how often it wins despite random errors.

If we were to set $\varepsilon = 1$ then our engine would be identical to the randomized baseline. Thus, for any $\varepsilon < 1$ we see improvement over the randomized engine, since then at least some fraction of the engine's moves are optimal. The results of our tests for the simple and advanced Minimax engines are shown in Figure 4.

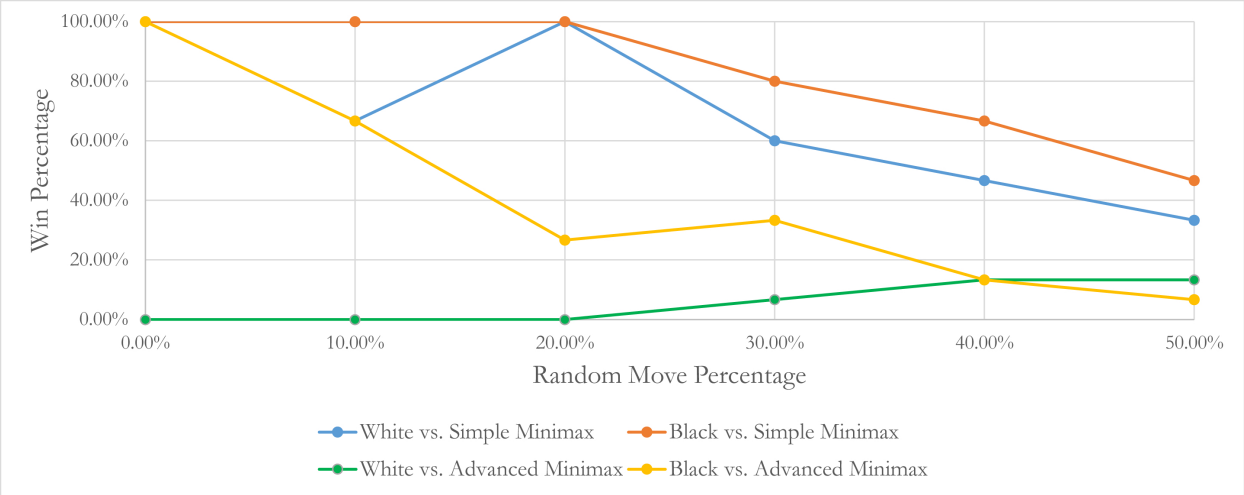

Figure 4: The win percentage of our engine against the simple and advanced Minimax baseline engines as a function of the percent of random moves made by our engine.

Once we had verified the correctness of our algorithm against baseline computer players, we next sought to verify that it held up in real-world situations. We therefore turned to games against human players. These opponents included both ourselves as well as willing challengers at the poster session. Neither of us have been able to beat the final version of the engine, and at the poster session the computer won 9 out of 10 games. The game it lost involved roughly 7 people working together, and it was their third attempt at playing the engine. With prior experience, the group was able to take advantage of the engine's determinism, and with many people, they were able to calculate farther than the computer was set to.

Finally, we set out to test the upper limit of our engine by running it against the Antichess engine found on Lichess.org. This engine is capable of playing at various different levels, with the strength of the engine given as an Elo rating (the standard metric used to rank chess players). We found that our engine was able to win against a player with a rating of 1900 when we played as black and 1700 when we played as white. Given that the standard cutoff for "expert" players is a rating of 2000, it is quite possible that our engine has reached expert level of play (at least when it moves second).

7 Discussion

The data in Figure 4 reveals several interesting details about our implementation. One quite striking feature is the fact that the engine performs significantly better when playing as black rather than white. Because white moves first, this difference suggests that while our engine is able to capitalize on errors made by the opponent, it makes suboptimal moves itself when it has many options available. It is likely that further improvement would require more complex features, allowing the engine to develop a better sense for Antichess strategy.

The graph also clearly shows the benefit of fully investigating forced lines. In the yellow curve, the two engines only differ in that black investigates these forced move lines to the end. The fact that the forced line

engine still wins more than 60% of the time even when 10% of its moves are random is a testament to the importance of these forced lines.

Although our engine has achieved an impressive level of play, the fact that it performs better when playing as black is evidence that there is still room for improvement. From the work of Watkins [1], we know that it is always possible for white to win at the start of a game of Antichess, which means that if our engine were perfect we'd see better performance when playing as white.

7.1 Game Analysis

Figure 5 demonstrates one case where the computer could almost be considered clever. In this position, knowing that Black would have to take the rook on $a3$, the computer placed its bishop on $g2$, so that it would have both the option of taking the $a3$ pawn or the $b7$ pawn. After the computer takes the $b7$ pawn, Black's bishop is forced out into the open. Afterwards, the computer will have to take the pawn on $a3$, but then will take the pawn on $e7$, possibly forcing either the other black bishop or the queen out into the open as well. At this point, the computer has gotten rid of both of its bishops (which are particularly dangerous pieces in the early game), and has no major pieces in the open, while Black has two pieces in the open. This suggests that the computer has a winning position, though the computer was able to confirm that with certainty through computation.

Figure 5: Position from sample game between the computer and a human. After the 8th move, the computer evaluated 13513 positions and determined that it could guarantee a win with the move $f1g2$.

On the other hand, Figure 6 shows two fatal errors on the computer's part. Here, our engine is playing as white against the Lichess engine playing at the level of a 1900 rated player. White is clearly winning in the left position; after Black takes the bishop, if white pushes the pawn to $a4$ then black cannot prevent it from being taken. In fact, the Lichess engine, which calculates further than our engine, claims that White

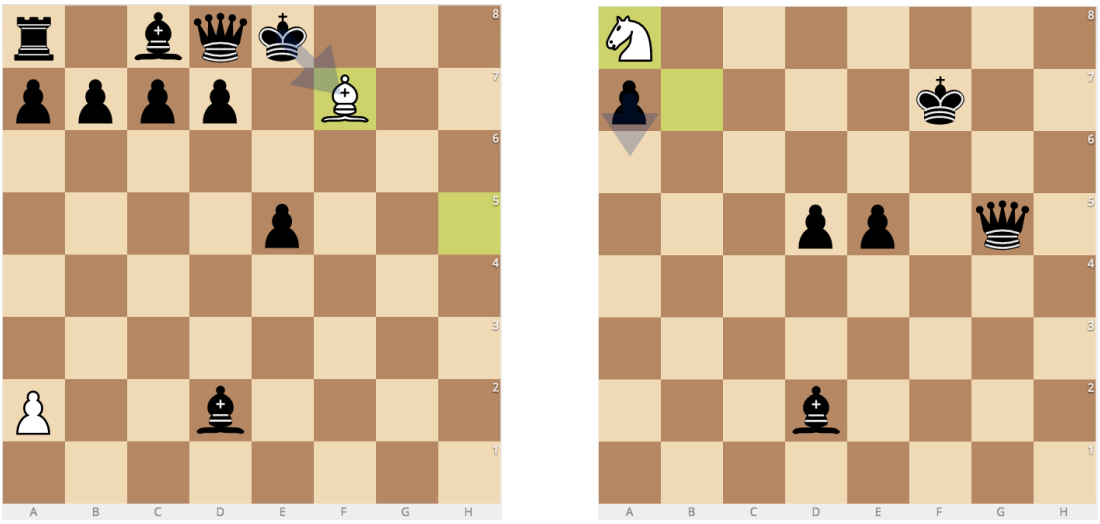

Figure 6: Game between our engine and the Lichess engine rated 1900. The left image is the position after 19 moves, and the right is the position after 24 moves. Full game: `https://en.lichess.org/3DjRqwIT#58`

can guarantee a win in 4 moves (7 half moves). However, our engine was unable to see that far, and because of the nature of the evaluation function couldn't see any difference between the pawn being on $a3$ and $a4$. It thus moved the pawn to $a3$ instead.

Black was then able to force White to promote the pawn. Again, because of the nature of the evaluation function and the limited depth, our engine chose to promote to a knight, which loses in 8 moves, rather than a bishop which would result in a draw under optimal play.

8 Conclusions and Further Work

We're quite pleased with the performance of our engine, since it was clearly superior to our baselines, and also performed quite well against human players and other engines. However, there are a number of ways in which our engine could improve.

- **Adding an opening book**. Our current engine operates with no previous knowledge of openings, and so its early moves are informed only by a low depth search of early positions. An opponent who knows her Antichess openings and is unlikely to make mistakes would be able to capitalize on this.

- **Endgame improvements**. As evidenced by Figure 6, our current engine is remarkably bad at endgames, since they often have set strategies and so can't be solved with our depth of search (especially since there are often a number of legal moves).

  Our engine's behavior in this situation suggests that the weights of our features and the features that we look at should be adaptable at the end of the game. Note that a queen isn't too dangerous early in the game when it is easy to lose, but if a queen is still around when there are fewer pieces an opponent can often force it to clear out the board. It might also be the case that if a player only has pawns left, it might be better to figure out how to get rid of each one independently by searching farther for each.

This issue could also be alleviated by using an endgame tablebase, which dictates what to do when a certain set of pieces is on the board (for example, two bishops against a knight and a rook). Knowing how certain endgames play out would be useful in guiding our engine through situations like those in Figure 6, where a mistake was made in the promotion choice.

- **Using Monte Carlo Tree Search and Neural Networks**. Another major problem with our engine is that it is deterministic. An opponent can then, over repeated play, correct her mistakes and eventually beat the engine by capitalizing on minor inaccuracies of the evaluation function. Often times, there is an objective best move, but in cases where there is not, a non-deterministic model might be better. We also think that because of the complexity of the ways in which the values of different pieces change over time, depending on the position, neural networks might be more effective than our linear approach.

References

[1] M. Watkins, "Losing chess: 1. e3 wins for white," 2016.

[2] I. J. Good, "A five-year plan for automatic chess," 1968.

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

[4] C. Frâncu, "Suicide chess book browser by nilatac," 2002.

[5] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

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

[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.