USING ADAPTIVE LEARNING ALGORITHMS TO MAKE COMPLEX STRATEGICAL DECISIONS

ASHANTHI MEENA SERALATHAN

Haverford College, Computer Science Department

Advisors:
Dr. Deepak Kumar
Dr. Douglas Blank
Dr. Stephen Lindell

April 25, 2011

Date: Spring 2011.

1. Introduction 4
1.1. Chess 4
1.2. Deep Thought and Deep Blue 5
1.3. Minimax Search 6
1.4. α-β Pruning 7
1.5. The Horizon Effect and Quiescence Search 8
1.6. Verified Null-Move Pruning 8
1.7. Neural Networks 9
1.8. Evolutionary Processes and Genetic Algorithms 10
2. Previous Work 12
2.1. Temporal Difference Learning and TDLEAF(λ) 12
2.2. Explanation-Based Neural Networks 12
2.3. Population Dynamics 13
3. Experimental Design 15
3.1. Content of Chromosomes 15
3.2. Crossing-Over 16
3.3. Chess Engine 16
3.4. Results 16
3.5. Analysis 18
3.6. Future Work 19
3.7. Conclusion 20
4. References 21
5. Appendix A: Player Chromosomes 22
6. Appendix B: Code 24

Contents

Abstract

Traditionally, artificial intelligence (AI) algorithms have not been built on particularly adaptive principles. Systems were created using complex collections of rules that were created specifically for the purpose at hand, and whose flexibility was wholly dependent on what flexibility the programmer incorporated within the rules. As a result, this thesis examines many different algorithms for decision-making, particularly for playing chess. It surveys a number of different techniques for creating a chess-playing system, and finally begins an altered implementation on the genetic algorithm-inspired algorithm that uses Population Dynamics to train a system to understand how to rank board states in a game of chess, which includes more genes than the original algorithm.

While still a work in progress, the process of creating the system has already demonstrated some advantages other other algorithms for learning evaluation functions for chess (such as the flexibility of the algorithm), and futher work could lead to interesting insight on whether a chess system built using a modified version of Population Dynamics can lead to a system whose skill is comparable to the likes of other chess systems, or even to human players.

1. Introduction

This thesis focuses on a number of different strategies utilized for making complex decisions: Deep Blue, an example of a chess-playing system that utilized no adaptive learning strategies to play chess, as well as three popular algorithms for learning evaluation functions for chess. All three learning algorithms rely on a minimax search (which is sped up with various methods of pruning the move space at a given turn) that returns the best found move. Each algorithm has a distinctly different method of developing its evaluation function: two algorithms (Explanation-Based Neural Networks and TDLEAF(λ)) use neural networks in some manner to develop the evaluation function; one (Population Dynamics) uses genetic algorithm techniques to develop the function.

Finally, this thesis begins to analyse the effects of altering the Population Dynamics algorithm by slightly altering how it searches through the chess board space, as well as adding more parameters to the player's chromosome in order to have the system determine even more about factors that are important to playing chess.

It must be noted that these algorithms all have uses outside of chess, and could be slightly altered to fit a number of different applications, such as natural language processing and generic decision-making systems.

1.1. **Chess.** Chess is an interesting game to choose for studying learning algorithms (due to its complexity), as it has yet to be solved. Winning the game does not simply become a matter of having the hardware to quickly find the moves necessary for the perfect strategy: some amount of strategy is required for a system to be able to play on-par with intelligent beings.

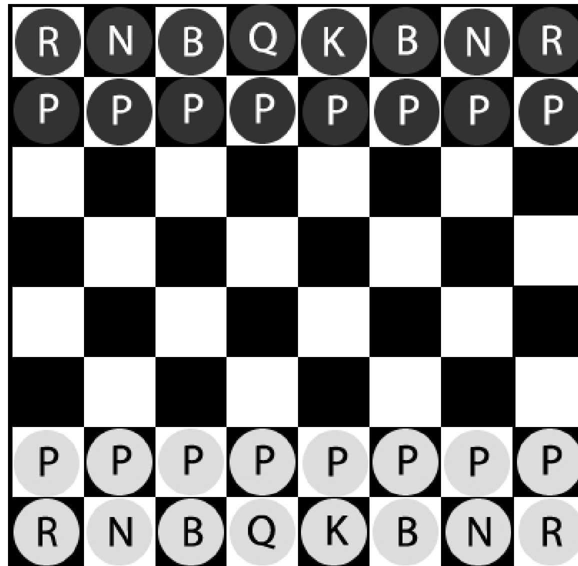

Figure 1. A typical chess board layout.

In chess there are six different types of pieces, each with a different set of rules as to how it can move (see Figure 1). Some pieces, such as kings, can only move one space at a time, while other pieces, such as queens, can move several spaces in several directions. If a piece can move to a space occupied by an opponent piece, then it may capture that piece. One piece, the king, is not allowed to be captured, and the player is required to find a way to get its king out of harm's way if it is threatened (called a "check"). The object of the game is thus to find a way to capture the king, which requires trapping the King to the point where no matter how the player moves, it cannot save its king from being captured (called a "checkmate"). Along with this are various stalemate criteria, which include a player who is not in check being unable move without putting itself in check, that may be taken into account. In addition, there are many special moves that certain pieces can only make under specific circumstances, such as castling.

The system therefore has to keep track of each piece, keep track of where that piece can move, keep track of whether a piece's movement will put its king in check, and then (as there will be often be dozens of possible moves for each player) it must make strategic decisions about which possible moves would be most advantageous at a certain point in the game. How the computer chooses these moves is the topic of discussion in this thesis.

1.2. **Deep Thought and Deep Blue.** The first truly successful chess-playing artificial intelligence (AI) systems were Deep Thought and Deep Blue (created by Carnegie Mellon University in 1988 and IBM in 1996, respectively)[1]. Deep Thought and Deep Blue both relied heavily in optimizing the system hardware, with the intent of being able to do very fast searches and to use carefully researched evaluation functions to choose how to rank boards. Deep Blue in particular was a sophisticated parallel system that relied on 30 IBM RS/6000 SP processors containing a total of 480 chess chips dedicated to speeding up the process of searching for possible moves (each chip was able to see 2-2.5 million possible positions per second). One of the 30 processors served as the master processor, which started searches through the move space and pushed deeper searches onto the other processors. These processors would then use their chess chips to do the last few levels of searches, before evaluating individual boards and sending their findings back up to the master processor.

It also relied on pre-computed and heavily researched evaluation functions that themselves relied on over 8000 features specific to chess games (such as how safe the king was, where specific pieces were located, and so on). By combining sophisticated hardware specifically designed for optimized chess-playing with a carefully constructed set of components to an evaluation, and with highly optimized minimax searches through the possible move space, Deep Blue was able to play at a level that was competitive with human Grandmasters, and was even able to beat one (Garry Kasparov) in 1996. It averaged scanning 126 million positions per second (with a range of 100 million to 200 million positions per second, depending on the point of the game).

While certainly a breakthrough in the field of AI, it must be noted that though it was able to beat a human player, Deep Blue was given almost every bit of information that it may ever need to play a game of chess against a human player, with the fastest, most sophisticated hardware that could be mustered at the time, and was left to automatically compute its movements based on all the hard-coded information it was handed. In consequence, the Deep Blue algorithm as a whole was rather inflexible in terms of its application to other real-world problems. Each rule was hand-crafted by Deep Blue's programmers to only improve the chess-playing process, and would not easily work with any other problem.

1.3. **Minimax Search.** Minimax searches rely on an efficient algorithm for making sure the player's move is optimized, even if the opponent moves in a manner most beneficial to itself.[2] It is applicable to any two-player game where there is a finite number of possible moves and where good choices by an opponent can directly detract from the player's ability to win, and thus is commonly used in chess to determine whether a certain move is the best, regardless of how good the opponent's next few moves are.

The algorithm begins by finding all the player's possible moves, and expanding the search tree for each node (in a manner similar to iterative deepening, or a bounded breadth-first search). When a move is expanded, the possible moves of the opponent player are found, and then each is expanded in the same fashion. The algorithm switches between expanding possible player moves and possible opponent moves until the search has reached a bound, at which point the possible moves generated are all scored by an evaluation function (generally provided to the system). Bounds typically allow for at least one search ply (a ply being the expansion of player moves followed by the expansion of opponent moves; in other words, two levels in the search tree).

Once each move has been scored, it is compared to all the other moves at that level of the search tree, and the best-scoring move is passed back up the tree, where its score is compared to the best score at the next-highest level of the search tree in a similar fashion. By passing these values up, the system gets a total score for that path (which represents how effective the player's move along that part of the tree may be). Ultimately, the player's moves are scored positively and the opponent's moves are scored negatively; therefore, if the opponent's move is better than the last move the player made, then the rating for that path of the search tree will be negative, and vice-versa. And the end, each of the player's next moves will be scored, and the best scored move is chosen (see Figure 2).

```
def minimax(node, depth, player):
        if depth > maxDepth:
                return evaluate_node()

        next_moves = node.next_moves()
        if game.player == player:
                best = -infinity
                for move in next_moves:
                        best = max(best, minimax(move, depth+1, game.opponent))
                return best
        else:
                worst = infinity
                for move in next_moves:
                        worst = min(worst, minimax(move, depth + 1, game.player))
                return worst
```

FIGURE 2. Pseudocode for the Minimax algorithm.

This algorithm is an effective way of determining how good a move will be, and of determining for how long a good move may remain a good choice in the future. However it requires looking at every

possible move of both players over the span of multiple turns, which can easily amount to hundreds or millions of moves to evaluate, depending on the game and on how the game has progressed. In consequence, other algorithms are used along with a minimax algorithm to reduce the number of moves that must actually be evaluated.

1.4. α-β **Pruning.** α-β pruning is a method of reducing the number of moves the system must evaluate, by determining whether the net score of a particular search path could ever be better than that of other paths around it. It works roughly the same as a normal minimax search, except that it also incorporates two new variables (α and β). α represents the net score of the opponent, while β represents the net score of the player.

As the scores of each move are returned up the search tree, in addition to calculating the scores of each move, the system also sets the α (if the moves being scored are the player's moves) or the β (if the moves being scored are the opponent's moves) to the score sent up to that level. If the α is not smaller than the β when the system is scoring the player's moves, then the system will stop evaluating the rest of the player's moves. This is due to the fact that α being smaller than β equates to the opponent's score at that point in the tree being lower than the player's score; if α is not smaller, then the likelihood that the move will ultimately return a low score is so high that the algorithm can simply cut it and its branch from the search entirely. A similar comparison is made when the system is scoring the opponent's moves; if β is larger than α, then the branch being looked at will be cut from the search (see Figure 3).

```
alpha_beta(node, state, bound, alpha, beta):
     if (bound > maxSearchDepth):
          return evaluate_node();

     nodes = node.children;
     if (state == maxNode):
          foreach child in nodes:
               alpha= max(alpha, alpha_beta(child, bound+1, alpha, beta));
               if (alpha >= beta):
                    break;
          return alpha;
     else if (state == minNode):
          foreach child in nodes:
               beta = min(beta, alpha_beta(child, bound + 1, alpha, beta));
               if(beta <= alpha):
                    break;
          return beta;
```

FIGURE 3. Pseudocode for the α-β pruning algorithm.

Using this method, the system can cut a large number of moves out of the search, and therefore will not have to waste time calculating scores for bad moves. More specifically, α-β pruning can reduce the average branching factor of the search by $\sqrt[4]{b^3}$, and may reduce the search to $\sqrt{b}$ (where b is the number of children of a particular node).[2]

The only main concerns with this algorithm are that even with this, for a game like Chess there could still be a large number of moves that must be evaluated; it is estimated that chess has approximately 10^{40} game states, suggesting that even with an optimal reduction the program may have to scan hundreds or thousands of game states in order to make a move.[2] In addition, sometimes the algorithm can cut branches that could actually have led to worthwhile moves (often because the search was not performed far enough to make an informed decision about when to cut the branch). These limitations are typically not enough to abandon using the algorithm, however, and these concerns can be ramified with the addition of other algorithms to the search process, such as quiescence and null-move pruning.

1.5. **The Horizon Effect and Quiescence Search.** In games such as chess, in which the nature of a game can change in a matter of moves, it can be crucial to know the outcome of a heated part of the game if the system hopes to choose the best move to make in such a situation. This is described as the horizon effect; the system is only able to see so far into the future, and therefore cannot understand the full effects of its choices, depending on how far it is able to search. As stated before, a normal α-β-pruned minimax search uses a strict search bound that does not take into account the nature of the game, and thus it is very easy for the system to miss crucial information that could be revealed in the next move (i.e., over the horizon).

This is where quiescence is effective; by using quiescence techniques, the system will extend the search on a particular branch if, by the time it hits the normal bound, the game has not "stabilized."[3] A stable game can mean anything from neither player being in check, or not having particular pieces in jeopardy. If the game is found to be unstable for any reason, the search is extended a little further, and this process can be repeated until the system has extended it to a stable game state. By using this, the system lessens the number of times it misses important consequences of its moves, as it continues the search at points in the game where making the right move is crucial.

1.6. **Verified Null-Move Pruning.** In order to shrink the number of game states examined even further, an algorithm called Null-Move pruning may be implemented.[4] Null-Move pruning is based on the assumption that at any point in the game, it should be more beneficial to make a move than it would be to do nothing (i.e., a null move). Thus the algorithm begins with the player testing out a null move; it pretends the player did not make a move, and continues a reduced search (typically a reduction of 1-ply, or R=2) down this new branch of the search tree. This results in a score that is reflective of the null move, and which can be compared to the score of the game at the point before the null move was made. If the score of the null move turns out to be higher (i.e., better) than the score prior to the null move, the system ignores the rest of this game branch (under the assumption that since it was better to do nothing than to make a move, this move branch is undesirable). Moves that fall under these category are called "fail-high" moves.

While this algorithm works well in the middle game, when it is rare that not making a move will be beneficial, it can be detrimental once the game moves closer to the end, as less pieces on the board can lead to less of a benefit for moving those pieces. The horizon effect can also come into play when the algorithm does not search far enough along the null move branch to realize that the null move is not as beneficial in the far future as it seems in the near future. Thus the algorithm of verified null-move pruning was developed; it works to allow further reduction of the null-move search, while alleviating the effects of the horizon effect and of the nature of chess endgames.

Verified Null-Move pruning does not cut a branch immediately after finding a fail-high move; rather, it uses the result of a standard null-move search as a sort of flag, which informs the rest of the normal minimax search. It first performs a null-move search, where R=3, and determines whether the results indicate that a node is a fail-high; if this is the case, then rather than cutting off the node, the depth of the overall search is reduced by a ply, and standard null-move pruning is performed on that node's subtree (i.e., any node found to be a fail-high after this point will be cut; if the algorithm finds no need to cut the nodes, the depth of the search is increased by a ply, and continues as though the node had not been flagged as a fail-high). This way, a node's branch is only cut off if it consistently shows signs of leading towards a detrimental move, rather than at the first indication that the node may be a poor choice. This also allows for a shorter null-move search (R=3 rather than R=2), which more than makes up for the extra verification that occurs in the algorithm (see Figure 4).

```
search(alpha, beta, bound):
     if (bound > maxSearchDepth):
          return evaluate_node()

     if (!in_check and null_move_ok() and (!verifying or bound < maxSearchDepth)):
          make_null_move()
          result = search(alpha, beta, bound + R + 1) # R = the depth reduction factor
          if (result >- beta):
               if(verifying):
                    bound ++
                    verifying = false
                    fail_high = true
               else:
                    return result


re_search():
     result = search(alpha, beta, bound)
     if (fail_high and result < beta):
          depth --
          fail_high = false
          verifying = true
          re_search()
```

FIGURE 4. Pseudocode for the Verified Null-Move Pruning algorithm.

1.7. **Neural Networks.** Many machine learning algorithms employ some form of neural network. Based on a model of how synapses in a human brain function, a neural network is essentially a fully connected, weighted directed graph with three levels ("layers") that each have their own function within the network.[5] The first layer, known as the input layer, receives the information being given to the network, and sends it to the second layer, called the hidden layer. As the information moves from the input layer to the hidden layer, it is altered by the weights on each edge from the input nodes to the hidden nodes. In this sense, as every input node is connected to every hidden node and each edge has its own weight, each hidden node will receive a different transformation of the same data (in other words, each hidden node possesses a different perspective on the data). Data

is then manipulated by functions within the hidden nodes, and then passed from the hidden nodes to the output nodes (after being altered again by the set of weights between the hidden and output layer). Each output node is (before the network runs) mapped to a specific value or answer to the question the network is attempting to solve (such as what move to perform given a board); in the case of chess, this could mean that if the network is set up to take in a board and output what the board would look like after a chosen move, the output layer could represent every possible move that could be generated from the board taken in (with the number of nodes in the layer equalling the number of possible moves). The system would know which move to choose based on which node yielded the best value; typically, the best value is deemed to be the largest. This means that once a board is given to the network, a value will be sent to each output node, and the node with the highest value will represent the move that the system should make (see Figure 5 for a diagram of a neural network).

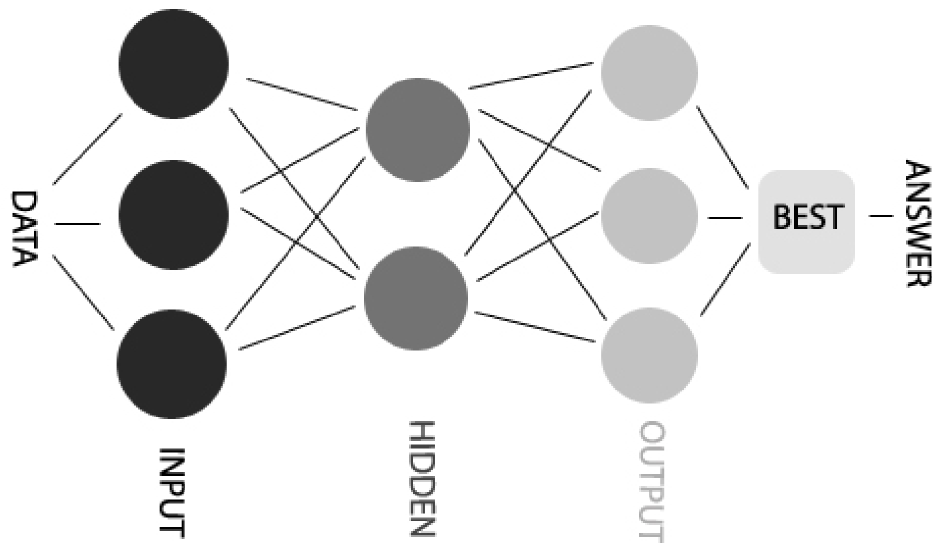

FIGURE 5. A neural network is a complex weighted graph that can be used to retain information about data propagated through it.

Neural networks are typically initialized with a random set of weights (as the best weights to use generally change depending on the problem being solved). Because of this, the network will need to be taught how to solve the problem correctly, and will have to know how to change its understanding of the data (i.e., change its weights) accordingly, so that it may correctly answer the problem. There are several different ways to train a network and to update its weights; typically, a value is added to the weights, which is proportional to how incorrect the network's answer was in comparison to what the trainer specifies as the correct answer (also known as error backpropagation).

1.8. **Evolutionary Processes and Genetic Algorithms.** Genetic algorithms are another subset of algorithms often used for machine learning, and are based on the biological phenomenon of evolution.[6] In the algorithm, a population is generated, where each member of the population

serves as a random possible answer to the question the system is trying to answer. Members have the ability to asexually reproduce, and may be removed from the population over the course of a population's lifetime, to be replaced with the more fit members of the population in future generations. Sometimes this fitness value may be a probability that the member can make it to the next generation, which is related to the strength of the member's genes, and other times it may be dependent on criteria that the member must fulfill in order to live past its current generation, such as successfully performing a procedure based on the contents of its personal genetic code.

Each member has its own "gene," which is typically a collection of values which, when coupled with an equation or other method of evaluation, can produce an answer that may or may not answer a question accurately. Answers are evaluated in some manner and, similar to Darwinian evolution, the best answers are allowed to persist in the population, while the worst answers will eventually be removed from the population (see Figure 6). Members of the population eventually create progeny, which are either clones of their parents, or who have their "genes" altered in a way similar to biological genetics (i.e., mutations and cross-overs). Such alterations are random and typically have a specified probability of occurring within the population. As long as the alterations are performed in a gradual manner, the population will eventually converge such that all the members will give a similar answer to a question. And, so long as alterations are random, there will be enough variety in the population such that the system will be making a fairly informed decision about what the answer should be.

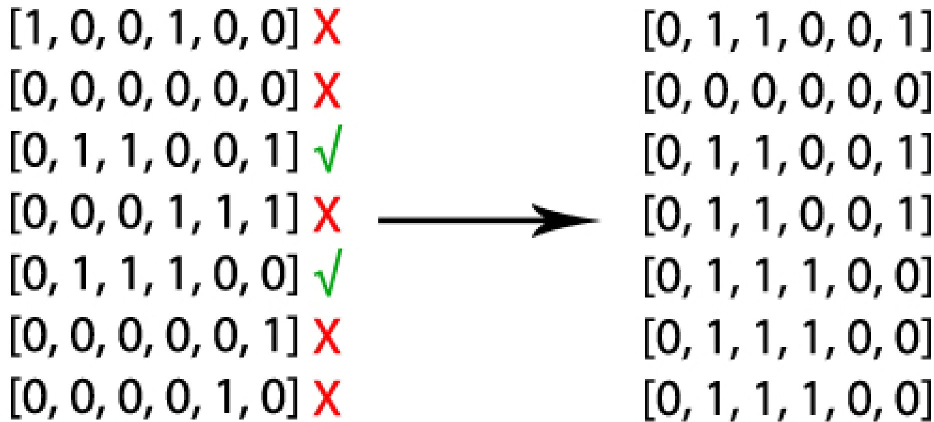

FIGURE 6. After being evaluated, the genes that best solved the problem are allowed to persist and reproduce, while the worst genes are given less of a chance of persisting in the population, and are less able to reproduce.

Genetic algorithms are viewed as more flexible than many algorithms due to the fact that the system does not need to have significant knowledge of the domain of a problem in order to attempt

to solve it. The only portion of the algorithm that relies on any context from the question is the portion that assesses the fitness of a particular member, and thus the programmer does not have to integrate knowledge of the specific problem into every portion of the system. As such, a system trained on genetic principles may be used to solve multiple unrelated problems simply by altering the method of testing a member's fitness to suit the problem at hand, rather than having to adjust the internal mechanisms of the algorithm to suit the nature of the new problem.

2. Previous Work

2.1. **Temporal Difference Learning and TDLEAF(λ).** Temporal Difference Learning focuses on training a system so that it may make informed decisions about how to react to something by being able to predict the future cost of its decision.[7] The algorithm requires a set of end states that can be evaluated in some way (in a game like chess, a set of moves), and sets each state to have a probability value $p(x_t, x_{t+1}, a)$ that corresponds to the likelihood that the state will occur (for example, the likelihood that a move will occur in a chess game). Using this, the system can assign a reward value to each end state, and use these values to make the best decision. The system also scores itself depending on whether its predictions were successful or not (1 when it is successful, 0 when a neutral outcome is reached and -1 when unsuccessful). The system can use this value to adjust the reward values for each end state.

The TDLEAF(λ) algorithm adapts this strategy to a chess-playing system that uses a minimax algorithm to evaluate moves in a chess game. Rather than simply being given a set of states, the algorithm takes the leaves from a minimax search and assigns reward values to each, and then scores other moves based on their relationship to these leaves. In this sense, the reward values are propagated up the minimax tree, such that the system gets a better understanding of how a possible move will turn out, based on how its predicted reward values have turned out.

The algorithm is markedly slower than Deep Blue (it plays 6,000 times slower than Deep Blue), however it is still able to play at a highly sophisticated level. After setting most of its evaluation function parameters to 0 (leaving only piece rankings in the equation) and allowing the system to play human and computer chess players, it played at a B-grade human level (around 1650); once the TDLEAF(λ) part of the system was enabled, the chess system's ranking rose 500 points (2150) in its first 300 games, and became even better afterward. And because the system learns all the parameters to its evaluation function, the system could be left to develop an optimal evaluation function over time, rather than requiring extensive research on how to set each parameter (as was necessary with Deep Blue). The algorithm is also extremely successful with games such as Backgammon.

2.2. **Explanation-Based Neural Networks.** The explanation-based model for learning chess was motivated by the desire to reduce the amount of time needed to train a network that is going to play chess.[8] Explanation-based methods rely on "domain knowledge" of the problem being tackled in order to form generalized opinions as to how to carry out the task.

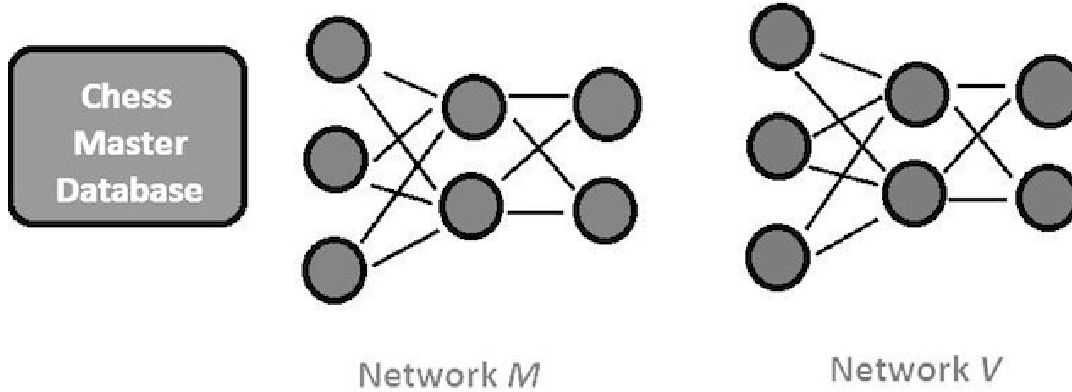

FIGURE 7. Components of the Explanation-based algorithm.

In the case of chess, this model is utilized with neural networks; one network, called the chess model M, is trained to keep track of this domain-specific knowledge by training itself on large databases of grand-master chess games, and learning how to take in a board and predict what the next move will be. Specifically, it receives a board s_t (the board at the start of the player's turn) and maps it to s_{t+1} (the board at the start of the next player's turn). The system then uses M to affect how it learns an evaluation function V (which is represented by a separate neural network which is trained using the TD algorithm), such that

$$\frac{\partial V^{target}(s_t)}{\partial s_t} \approx \gamma \cdot \frac{\partial V(s_{t-2})}{\partial s_{t+2}} \cdot \frac{\partial M(s_t)}{\partial s_t}$$

where $V^{target}(s_t)$ is the value the evaluation function will give at board s_t , γ is a discount factor that favors early accuracy over late accuracy (i.e., a stronger starting game rather than a strong end game), $V^{target}(s_{t+2})$ is the value the evaluation function predicts will be valid at the player's next turn, and $M(s_t)$ is based on the information M has about board s_t, based on the chess games it was trained on. This algorithm saves time because V is the network that will be used in the chess-playing system, and with this algorithm it does not have to spend time training on games first-hand every time it must train a player (instead, a researcher trains M once, and uses it to train V repeatedly as necessary). See Figure 7 for a diagram of this structure.

This algorithm was tested against GNU Chess (a popular and well-developed open-source chess application), and won approximately 13% of its games. The largest problem with the system were its extremely poor opening strategy, and its inability to play out short endgame strategies; however, the system was able to generalize how to protect important pieces and how to do intelligent trades.

2.3. **Population Dynamics.** One of the only chess-playing learning algorithms that does not rely on a neural network framework, this algorithm uses an algorithm based on evolutionary principles in order to calculate what the system's evaluation function should be.[9] The algorithm uses a minimax

algorithm using α-β pruning and quiescence. The minimax search evaluates boards based on the following equation:

$$\sum_{y=0}^{6} W[y](N[y]_{white} - N[y]_{black})$$

where N is a vector containing the number of pawns, knights, bishops, rooks, queens, and legal moves for the current board, and where W represents the weights of all these parameters. A small constant "bonus" is granted in cases where the player controls the center, or when pawns have the potential to be promoted.

It then randomly generates a population P, which consists of chess-playing systems that have their own randomized weight vector (or chromosome), v_i, where $v_i = \{\text{weight}_{knight}, \text{weight}_{bishop}, \text{weight}_{rook}, \text{weight}_{queen}, \text{weight}_{legalmove}, \text{weight}_{pawn}, \text{weight}_{king}\}$. Weight_{pawn} and weight_{king} are always fixed to 1 and ∞, respectively (and, due to being fixed, do not need to be added to the chromosome or used in calculations). Each member plays against another member of the population twice (once as black and once as white). Opponents are selected based on the relative strength of the two systems. Each player is represented in a vector, V, and a variable, c, is used to iterate through the vector. A member $V[c]$ is pitted against a randomly selected player in the remainder of the vector ($V[c+1]$ - $V[\mu]$), and then c is incremented. When the vector has been iterated through a generation has passed, and the vector is reversed, which guarantees that the best player of that generation is placed at the beginning of the vector. This ordering holds due to the fact that the later players will have played more games than their predecessors, and thus will have improved more rapidly than their predecessors. This also ensures that the best player will persist in the population longer than other other players, as it will not be challenged often enough to allow for many unlucky losses.

The winner of the two games is allowed to stay in the population, while the losing member is expelled from P, and a mutated version of the winner is put in its place. If there is a draw, then both members are left in P and mutated. The mutated members of the population have their chromosomes change by first calculating the standard deviation of each parameter:

$$\sigma_{(y)} = \sqrt{\frac{\sum_{n=1}^{\mu}(V_{n(y)} - average_{(y)})^2}{\mu - 1}},\ \forall y \in v$$

(where y is a particular evaluation parameter (i.e., gene), $V_{n(y)}$ is the parameter for player V_n, $average_{(y)}$ is the average of that parameter across the entire population, and μ is the size of the population), and then using this value in a final mutation equation:

$$V_{(y)} = V_{(y)} + ((RND(0...1) - 0.5) * R * \sigma_{(y)}),\ \forall y \in v$$

where R is a fixed value that is determined depending on how well the player performed:

$$\text{R} = \begin{cases} 2, & \text{if player won both games} \\ 1, & \text{if player lost and is becoming a mutated version of the winner} \\ 0.2, & \text{if player won a match and tied another match} \\ 0.5, & \text{if both players tied both times} \end{cases}$$

In this sense, the mutation of the genes reflects the trends in the population, and isn't purely random, which helps in keeping the values close enough so that they can converge at some point. This also allows the values to mutate differently depending on how well the player has played recently.

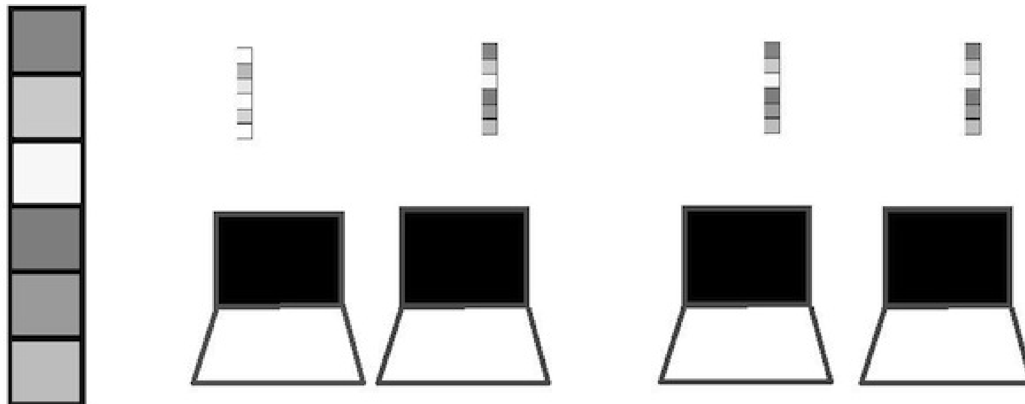

FIGURE 8. In the Population Dynamics algorithm, each member has an array of parameters that it uses to determine its next move; the best set of parameters leads to wins, which in turn leads to the member remaining in the population.

In Kendall and Whitwell's experiment[9], $\mu = 50$, and the genes were randomly set to values between 0-12. One member of the population was seeded with values known to be optimal (knights = 3, bishop = 3, rook = 5, queen = 9, legalmoves = 1), both to speed up learning and to prove that the best member of the population would not be lost during the learning process.

Their experiment yielded the following results:

TABLE 1. Kendall and Whitwell's Best Players

	Knight	Bishop	Rook	Queen	Legal Moves
Average	3.22	3.44	5.61	8.91	0.13

3. Experimental Design

For the purpose of this experiment, the algorithm utilizing Population Dynamics was implemented and tested; due to the inherently flexible nature of genetic algorithms in general, it was deemed to be the most simple to alter, relative to many of the other algorithms, which either would require extensive previous knowledge of how best to analyze each and every board, or would require even more training time and calculations to properly train the chess-playing system.

3.1. **Content of Chromosomes.** Kendall and Whitwell's algorithm[9] uses piece rankings and the number of legal moves as genes in their chromosomes, and incorporates control of center and pawn promotion as constants in the search algorithm itself; the effects of removing the constant bonus for control of the center and promotion and adding these as genes to each member of the population,

as well as adding genes for other more abstract strategies or types of moves, such as castling, were not examined in their experiment.

Considering that the algorithm as it stands would not allow the system to truly figure out how important the fixed parameters are (in comparison to how important it is to keep bishops in play or to have a significant number of legal moves), it would seem that an alteration of this sort would only improve upon the algorithm: instead of relying on programmer's intuition to set these values to a number that will not cause the system to ignore these parameters too often or give them too much priority, the system itself would be able to not only come up with its own values for these parameters, but would have done so in a manner that naturally relates these values with any other factors the system pays attention to while playing.

Adding to the number of genes in each chromosome also should not lead to a problem, as genetic algorithms in general are designed with the ability to manipulate large amounts of data at once; a typical algorithm may be altering hundreds of weights at once. Thus it is doubtful that the addition of these two factors, along with a few abstract moves, would hinder the system's ability to converge to a final set of genes.

Thus for the purposes of this experiment, $v_i = \{\text{weight}_{knight}, \text{weight}_{bishop}, \text{weight}_{rook}, \text{weight}_{queen}, \text{weight}_{legalmove}, \text{weight}_{controlOfCenter}, \text{weight}_{promotion}, \text{weight}_{castling}\}$.

3.2. **Crossing-Over.** In Kendall and Whitwell's experiment[9], crossing-over is not incorporated in the training process at all. In order to examine the effects of crossing over on such an algorithm (such as whether or not it speeds up the rate of convergence, or whether it creates a fitter population), another population will be trained in which chromosomes have a fixed probability of 0.1 of performing crossovers during the mutation process. Crossovers only occur at one point in the chromosome, and involve a swap of genetic material at a random point on the chromosome.

3.3. **Chess Engine.** The chess game engine is a modified version of a ChessBoard program[10], which handles finding valid moves and checking the state of the game, but does not come with a built-in artificial intelligence mechanism for choosing moves. The system is written in the Python programming language. It will be appended with the population dynamics system (which would handle training the population, keeping track of player chromosomes, and mutations), as well as a minimax search algorithm that uses α-β pruning. For the sake of speeding up training and testing the strength of the algorithm, the search depth was reduced to 1-ply, and quiescence was disabled. The evaluation function would be preserved from the original algorithm, with the exception that it for control of center, promotion, and castling, the weights in the chromosomes would be controlled with boolean values (i.e., $v_6 = \text{weight}_{controlOfCenter}*1$, if the player has control of the center; if not, then $v_6 = \text{weight}_{controlOfCenter}*0$; i.e., the weight is ignored). The way in which these values are calculated varies from the way the original genes were used simply due to the fact that these genes are only concerned with the player still having possibility of performing the move, and are not concerned with immediately quantifiable values, such as how many moves the player has.

3.4. **Results.** In Kendall and Whitwell's experiment[9], the initial standard deviation of the chromosomes in the population was 3.8; because I incorporate three new genes in each chromosome, the initial standard deviation for my players begins higher, at around 10 (10.01 for the population without crossing over, and 9.84 for the population with crossing over).

A player with 51 members in its population was trained for 13 generations; it converged with an average gene deviation of 0.13.

Table 2. Player A

	Knight	Bishop	Rook	Queen	Legal Moves	Control of Center	Promotion	Castling
Best Player	-6.65	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Average	-6.77	263.53	6.81	740.31	-3.92	-3.27	9.32	11.33

Values are rounded to two decimal places. These genes indicate that this player values having many Queens first and foremost; then it becomes concerned with the number of Bishops, castling, promotion, the number of Rooks, the player's control of center, the number of legal moves, and the numberof Knights, in that order. The average player has roughtly the same goals in mind.

A player with 51 members in its population and crossing-over enabled was also trained; the addition of the crossing over may have given the system some difficulty in converging as quickly, as even after twice as many generations had passed, the system had not converged (and, in fact, the genes had an average deviation of 5103.02). By the end of the 26th generation the population had the following characteristics:

Table 3. Player B

	Knight	Bishop	Rook	Queen	Legal Moves	Control of Center	Promotion	Castling
Best Player	16255.52	-5994.23	28.88	-1259.47	-127.51	7446.18	853.11	-54.99
Average	10002.75	-1123.14	-138.56	-540.27	-24.70	7538.78	435.20	-124.32

These genes indicate that this player values having many Knights more than anything else; then it becomes concerned with the player's control of the center, promotion potential, the number of Rooks, castling potential, the number of legal moves, the number of queens, and finally the nymber of Bishops, in that order. The average player mostly agrees with this hierarchy (the average chain is Knights, control of center, promotion, legal moves, castling, Rooks, Queens, and Bishops; Rook and legal moves swap precedence).

In both of these models, winning members were not mutated (only the replacement chromosomes were mutated, at a rate of 0.5). See Appendix A for tables of all the genes of both populations.

For the purpose of testing their ability to play chess (regardless of whether or not their genes were optimal), these two players were both pitted against random players (i.e., players who choose moves at random), using the same two-game format used in the training. The results are as follows:

Table 4. Player A v. Random Player

	Result	Moves Required
Game 1 (White)	Lost	140
Game 2 (Black)	Draw (3-Repetition Rule)	180

Table 5. Player B v. Random Player

	Result	Moves Required
Game 1 (White)	Lost	232
Game 2 (Black)	Draw(3-Repetition Rule)	189

These results remain consistent if the player is allowed a rematch; the players consistently lose their first game and then tie the second game.

These players also played against players pre-fitted with the piece rank genes used in the seed for Kendall and Whitwell's experiment ([3, 5, 5, 9, 1, 8, 1, 1]):

Table 6. Player A v. Seeded Player

		Result	Moves Required
Game 1 (White)		Draw (3-Repetition Rule)	12
Game 2 (Black)		Draw (3-Repetition Rule)	47

Table 7. Player B v. Seeded Player

		Result	Moves Required
Game 1 (White)		Draw (3-Repetition Rule)	12
Game 2 (Black)		Draw (3-Repetition Rule)	66

For each player, the results are identical if the player and the seed play again.

A similar set of games was played between a random player and the seeded player:

Table 8. Seeded Player v. Random

		Result	Moves Required
Game 1 (White)		Draw (3-Repetition Rule)	110
Game 2 (Black)		Lost	125

3.5. **Analysis.** As can be seen through the data, the results of the system's evolution differ greatly from the results of Kendall and Whitwell. Many of the genes for piece ranking are inflated to a significant degree in comparison to the "ideal" values. It is currently unclear as to whether this is the result of the added genes having an effect on the values in general, or whether the system as it stands is evolving incorrectly.

It is possible, for example, that the fact that the values for the piece rankings are not only dependent on each other, but are now dependent on the new factors, would cause the values in general to deviate from the values one would expect with only the five piece rankings in play. It is also possible that

the system, as it is currently written, is simply changing too quickly, thus causing abnormally high (or low) values for many of the genes.

And while the results of the played games would indicate that the non-optimal values indeed lead to a non-optimal level of gameplay (as the players could not win against a random player), the fact that the seeded player also could not beat the random player suggests that the weight may not be the only cause of weak play. There may currently be a bug in the search procedure that does not allow the player to truly choose its best move, causing even the seeded player to have difficulty beating the random player.

While it could be the case that the 1-ply search is proving to be a detriment to the system, as the low search bound will make it more difficult for a system to make an informed decision about future moves, one would assume that a weakly informed decision would still fare, on average, better than a completely uninformed decision, which is not the case with these players (as they consistently lose as White and draw as Black).

It also seems odd that the trained players lose when White (the color that moves first, and is therefore commonly expected to have a slight advantage) and draw when Black. More work would have to be done to determine how weak the opening move for the system is, as well as why a weak opening would cause poor gameplay when the player moves first, and cause a slightly better result when the player moves second.

3.6. **Future Work.** In order to verify the reason for the inflated genes, more training sessions would have to be run, and the average best player of each system would have to be compared. This would be required in order to verify that the inflated values in these preliminary training sessions were reflective of the system's behavior in general. In order to determine whether the new genes are the source of the weight inflation, it would be useful to go back and train a system that does not take the new parameters into account, and compare its results to the ideal set of weights. If similar to the ideal values, then it is possible that these values are close to normal, given the relationship of the new parameters to piece rankings. If the results were also equally inflated, then it could be concluded that the system itself is not evolving properly. Altering how the system's members mutate (lowering the rate of mutation or the magnitude of the mutations, for example) would possibly be the best ways to slow down the amount of change in the population, and keep the values closer to the ideal values.

The addition of more parameters in the chromosomes would also make for a more interesting system, which would hopefully yield a successful style of play. Adding parameters that take more complex strategies, such as knight-forking, into account would test the strength of the system when faced with abstract concepts and decisions (as opposed to the relatively straight-forward strategies and parameters incorporated in the current chromosomes).

Finally, in order to truly test the strength of the system's chess abilities, it would need to play against a wide variety of different players (human or computer). The system should also be compared to various other systems, including systems that incorporate neural networks or other structures in order to train their players. Through these tests, one could determine the advantages and disadvantages this approach holds over other other techniques, and could determine whether this approach is (or is not) the best approach to creating systems that can solve complex problems.

3.7. **Conclusion.** While there are not yet results that indicate that genetic algorithms create better predictions for tasks such as chess, in comparison to other learning techniques, the flexibility of the algorithm still appears evident, suggesting that the technique is still worth further study. The results thus far indicate that it may be the underlying playing algorithm, rather than the training algorithm, that could be causing non-optimal results; if this is the case, and if the learning algorithm itself does not need to be tweaked to create a more optimal chess-playing population, this would re-affirm the notion that genetic algorithms (and in particular, the population dynamics model) can be applied to a wide variety of problems with minimal need to hand-tailor it to the task at hand. This flexibility would vary widely from other techniques for playing chess, which either require the computer scientist to personally tweak with every chess parameter until they are optimal, or require using another sort of learning system, such as a neural network, that is more computationally costly (as it is solving a large system of equations at once[5], rather than one equation dependent on one set of weights [9]) and requires careful observation on the part of the computer scientist to ensure it does not train too much.

Additionally, Kendall and Whitwell's algorithm[9] suggests that a system trained with genetic can be training to play chess at a sophisticated level, indicating that the learning algorithm itself is capable of producing useful results for chess, and indicating that it is still worthwhile to study.

4. References

[1] M.Campbell, A.J. Hoane Jr., and F-H. Hsu. 2002. Deep Blue. Artificial Intelligence, 134 (1-2): 57-83.

[2] Nilsson, Nils J. 1998. Artificial Intelligence : A New Synthesis. Morgan Kaufmann Publishers, San Francisco: 197-207.

[3] http://en.wikipedia.org.

[4] Tabibi, O.D. And Netanyahu, N.S. 2002. Verified Null-move Pruning. ICGA Journal, 25 (3): 153-161.

[5] Cheng, B. and D.M. Titterington. 1994. Neural Networks: A Review from a Statistical Perspective. Statistical Science, 9 (1): 2-30.

[6] Koza, John R. 1992. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge: 1, .

[7] Tesauro, G. 1995. Temporal difference learning and TD-gammon. Commun. ACM 38: 58-68.

[8] Thrun. S. 1995. Learning to play the game of chess. In G. Tesauro, D. Touretzky, and T. Leen, (eds.), Advances in Neural Information Processing Systems 7, The MIT Press: Cambridge, MA: 1069-1076.

[9] G. Kendall and G. Whitwell. 2001. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In Proceedings of the 2001 Congress on Evolutionary Computation (CEC-2001), IEEE Press: 995-1002, 27-30.

[10] http://mainline.brynmawr.edu/cs371/chess.py.

5. Appendix A: Player Chromosomes

Table 9. Population A

		Knight	Bishop	Rook	Queen	Legal Moves	Control of Center	Promotion	Castling
Player 0		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 1		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 2		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 3		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 4		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 5		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 6		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 7		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 8		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 9		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 10		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 11		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 12		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 13		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 14		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 15		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 16		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 17		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 18		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 19		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 20		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 21		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 22		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 23		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 24		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 25		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 26		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 27		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 28		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 29		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 30		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 31		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 32		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 33		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 34		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 35		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 36		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 37		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 38		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 39		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 40		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 41		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 42		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 43		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 44		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 45		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 46		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 47		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 48		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player 49		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11
Player509		-6.64	258.36	6.68	725.80	-3.84	-3.21	9.13	11.11

Table 10. Population B

		Knight	Bishop	Rook	Queen	Legal Moves	Control of Center	Promotion	Castling
Player 0		21186.12	-94.47	-67.46	-151.08	4.50	8101.55	955.23	-101.64
Player 1		2626.51	323.88	-233.98	-497.36	78.62	-1584.32	42.14	-267.57
Player 2		2626.51	323.88	-233.98	-497.36	78.62	-1584.32	42.14	-267.57
Player 3		6534.92	-413.87	-9.93	-386.64	42.07	-1451.82	-402.79	-84.79
Player 4		6534.92	-413.87	-9.93	-386.64	42.07	-1451.82	-402.79	-84.79
Player 5		2626.51	323.88	-233.98	-497.36	3.80	5966.30	-153.10	-231.34
Player 6		21186.12	-94.47	-67.46	-151.08	4.50	8101.55	955.23	-101.64
Player 7		11302.87	473.48	-314.01	-358.23	-35.24	10320.29	176.37	-157.93
Player 8		11302.87	473.48	-314.01	-358.23	-35.24	10320.29	176.37	-157.93
Player 9		11600.52	-1038.47	-84.00	-154.00	-86.21	7489.63	1292.02	-156.60
Player 10		11600.52	-1038.47	-84.00	-154.00	-86.21	7489.63	1292.02	-156.60
Player 11		11600.52	-1038.47	-84.00	-154.00	-86.21	7489.63	1292.02	-156.60
Player 12		-5643.27	751.13	-214.63	-995.20	13.90	10107.16	376.65	-4.91
Player 13		11302.87	473.48	-314.01	-358.23	-35.24	10320.29	176.37	-157.93
Player 14		11600.52	-1038.47	-84.00	-154.00	-86.21	7489.63	1292.02	-156.60
Player 15		741.79	-128.63	-59.06	-615.40	-6.27	79.54	332.96	-29.04
Player 16		-3364.98	167.38	-136.28	-462.44	13.36	7676.24	-299.91	-205.68
Player 17		-3364.98	167.38	-136.28	-462.44	13.36	7676.24	-299.91	-205.68
Player 18		11600.52	-1038.47	-84.00	-154.00	-86.21	7489.63	1292.02	-156.60
Player 19		11600.52	-1038.47	-84.00	-154.00	-86.21	7489.63	1292.02	-156.60
Player 20		7868.27	177.67	-83.66	-501.89	-34.27	651.80	-220.77	-156.88
Player 21		11302.87	473.48	-314.01	-358.23	-35.24	10320.29	176.37	-157.93
Player 22		5601.75	-3000.67	61.56	-476.52	-11.91	1583.78	-440.78	-70.08
Player 23		11302.87	473.48	-314.01	-358.23	-35.24	10320.29	176.37	-157.93
Player 24		11302.87	473.48	-314.01	-358.23	-35.24	10320.29	176.37	-157.93
Player 25		11600.52	-1038.47	-84.00	-154.00	-86.21	7489.63	1292.02	-156.60
Player 26		11600.52	-1038.47	-84.00	-154.00	-86.21	7489.63	1292.02	-156.60
Player 27		10587.12	-125.80	-360.55	-261.83	14.89	13448.57	162.67	-166.90
Player 28		-30.40	-279.70	-23.05	-662.23	74.58	6237.41	455.03	-246.57
Player 29		9807.98	754.86	-32.40	-545.95	-84.11	7392.81	417.38	-114.20
Player 30		10587.12	-125.80	-360.55	-261.83	14.89	13448.57	162.67	-166.90
Player 31		15130.09	-1195.02	-29.59	-473.87	23.57	8439.18	722.05	-61.21
Player 32		7429.25	30.19	-376.25	-481.93	36.89	14987.85	-163.42	-125.04
Player 33		4790.79	114.64	-82.55	-732.12	80.81	8862.05	106.77	-148.82
Player 34		7429.25	30.19	-376.25	-481.93	36.89	14987.85	-163.42	-125.04
Player 35		7429.25	30.19	-376.25	-481.93	36.89	14987.85	-163.42	-125.04
Player 36		7272.72	-230.37	-432.99	-381.47	73.05	15565.44	-456.20	-162.17
Player 37		7272.72	-6078.43	122.46	-1226.22	-88.87	7465.96	149.17	-4.41
Player 38		15130.09	-1195.02	-29.59	-473.87	23.57	8439.18	722.05	-61.21
Player 39		6658.45	1077.47	-227.13	-557.75	84.85	1395.78	126.43	74.17
Player 40		15130.09	-1195.02	-29.59	-473.87	23.57	8439.18	722.05	-61.21
Player 41		16255.52	-5994.23	28.88	-1259.47	-127.51	7446.18	853.11	-54.99
Player 42		15130.09	-1195.02	-29.59	-473.87	23.57	8439.18	722.05	-61.21
Player 43		16255.52	-5994.23	28.88	-1259.47	-127.51	7446.18	853.11	-54.99
Player 44		11291.52	-1712.38	-185.10	-730.16	-69.72	3773.95	614.89	-108.97
Player 45		16255.52	-5994.23	28.88	-1259.47	-127.51	7446.18	853.11	-54.99
Player 46		16255.52	-5994.23	28.88	-1259.47	-127.51	7446.18	853.11	-54.99
Player 47		16255.52	-5994.23	28.88	-1259.47	-127.51	7446.18	853.11	-54.99
Player 48		16255.52	-5994.23	28.88	-1259.47	-127.51	7446.18	853.1	-54.99
Player 49		12888.34	-759.39	-145.63	-646.52	-58.23	5120.29	313.08	-104.77
Player 50		16255.52	-5994.23	28.88	-1259.47	-127.51	7446.18	853.11	-54.99

6. Appendix B: Code

```
#/usr/bin/env python

#
# ChessBoard - a Python program to find the next move in chess
#
# Copyright (c)            John Eriksson - http://arainyday.se
# Copyright (c) 2010       Doug Blank <doug.blank@gmail.com>
# Copyright (c) 2011       Meena Seralathan <senlorandir@gmail.com>
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
    USA
#

# $Id: $

from copy import deepcopy
import random, math

def printReason(game_result):
        # Reason values
        if game_result == 0:
                print("Running...")
        elif game_result == 1:
                print("INVALID_MOVE")
        elif game_result == 2:
                print("INVALID_COLOR")
        elif game_result == 3:
                print("INVALID_FROM_LOCATION")
        elif game_result == 4:
```

```
                print("INVALID_TO_LOCATION")
        elif game_result == 5:
                print("MUST_SET_PROMOTION")
        elif game_result == 6:
                print("GAME_IS_OVER")
        elif game_result == 7:
                print("AMBIGUOUS_MOVE")
        # Result values
        elif game_result == 8:
                print("WHITE_WIN")
        elif game_result == 9:
                print("BLACK_WIN")
        elif game_result == 10:
                print("STALEMATE")
        elif game_result == 11:
                print("STASIS_COUNT_LIMIT_RULE")
        elif game_result == 12:
                print("THREE_REPETITION_RULE")


def makeRepr(state, board):
        b = ""
        for l in board:
                b += "%s%s%s%s%s%s%s%s" % (l[0],l[1],l[2],l[3],l[4],l
                    [5],l[6],l[7])

        d = (b,
        state.player,
        state.white_king_castle,
        state.white_queen_castle,
        state.black_king_castle,
        state.black_queen_castle,
        state.ep[0],
        state.ep[1],
        state.game_result,
        state.stasis_count)
        #turn,wkc,wqc,bkc,bqc,epx,epy,game_result,stasis_count
        s = "%s%s%d%d%d%d%d%d%d:%d" % d
        return s

class Chromosome(object):
        #Knight, Bishop, Rook, Queen, Legal Move
        def __init__(self, values):
```

```
        self.genes = values

    def __repr__(self):
        return "Knight: " + str(self.genes[0]) + ", Bishop: " +
            str(self.genes[1]) + ", Rook: " + str(self.genes
            [2]) + ", Queen: " + str(self.genes[3]) + ", Legal 
            Moves: " + str(self.genes[4]) + ", Control of Center
            : " + str(self.genes[5]) + ", Promotion: " + str(
            self.genes[6]) + ", Castling: " + str(self.genes[7])

    def getKnightParameter(self):
        return self.genes[0]

    def getBishopParameter(self):
        return self.genes[1]

    def getRookParameter(self):
        return self.genes[2]

    def getQueenParameter(self):
        return self.genes[3]

    def getLegalMovesParameter(self):
        return self.genes[4]

    def getControlOfCenterParameter(self):
        return self.genes[5]

    def getPromotionParameter(self):
        return self.genes[6]

    def getCastleParameter(self):
        return self.genes[7]

    def crossover(self, gene2):
        temp = deepcopy(self.genes)
        crossing_point = random.randint(0, 6)
        for i in range(crossing_point, len(self.genes)):
            self.genes[i] = gene2[i]
            gene2[i] = temp[i]

        return gene2
```

```
class State(object):
    """
    Container for move state.
    """
    def __init__(self, player):
        self.game_result = 0
        self.reason = 0

        # States
        self.player = player
        self.null_ok = True
        self.verify = False
        self.fail_high = False
        self.white_chromosome = Chromosome([random.randint(0,
            12), random.randint(0, 12), random.randint(0, 12),
            random.randint(0, 12), random.randint(0, 12)])
        self.black_chromosome = Chromosome([random.randint(0,
            12), random.randint(0, 12), random.randint(0, 12),
            random.randint(0, 12), random.randint(0, 12)])
        self.white_king_castle = True
        self.white_queen_castle = True
        self.black_king_castle = True
        self.black_queen_castle = True
        #none or the location of the current en pessant pawn:
        self.ep = [0, 0]
        self.stasis_count = 0
        self.move_count = 0

        self.black_king_location = (0, 0)
        self.white_king_location = (0, 0)

        # three rep stack
        self.three_rep_stack = []

        # full state stack
        self.state_stack = []
        self.state_stack_pointer = 0

        # all moves, stored to make it easier to build
            textmoves
        #[piece,from,to,takes,promotion,check/checkmate,
            specialmove]
```

```
        #["KQRNBP",(fx,fy),(tx,ty),True/False,"QRNB"/None,"+#"/
            None,0-5]
        self.cur_move = [None,None,None,False,None,None,0]
        self.moves = []

        self.promotion_value = 1

def setEP(self,epPos):
        self.ep[0], self.ep[1] = epPos

def clearEP(self):
        self.ep[0] = 0
        self.ep[1] = 0

def threeRepetitions(self):

        ts = self.three_rep_stack[:self.state_stack_pointer]

        if not len(ts):
                return False

        last = ts[len(ts)-1]
        if(ts.count(last) == 3):
                return True
        return False

def endGame(self, reason):
        self.game_result = reason

def pushState(self, board):

        if self.state_stack_pointer != len(self.state_stack):
                self.state_stack = self.state_stack[:self.
                    state_stack_pointer]
                self.three_rep_stack =  self.three_rep_stack[:
                    self.state_stack_pointer]
                self.moves = self.moves[:self.
                    state_stack_pointer-1]

        three_state = [self.white_king_castle,
                self.white_queen_castle,
                self.black_king_castle,
                self.black_queen_castle,
```

```
                    deepcopy(board),
                    deepcopy(self.ep)]
            self.three_rep_stack.append(three_state)

            state_str = makeRepr(self, board)
            self.state_stack.append(state_str)

            self.state_stack_pointer = len(self.state_stack)

    def pushMove(self):
            self.moves.append(deepcopy(self.cur_move))

    def getMoveCount(self):
            """
            Returns the number of halfmoves in the stack.
            Zero (0) means no moves has been made.
            """
            return len(self.state_stack)-1

    def getCurrentMove(self):
            """
            Returns the current halfmove number. Zero (0) means
                before
            first move.
            """
            return self.state_stack-1

    def setPromotion(self, promotion):
            """
            Tell the chessboard how to promote a pawn.
            1=QUEEN,2=ROOK,3=KNIGHT,4=BISHOP You can also set
                promotion to
            0 (zero) to reset the promotion value.
            """
            self.promotion_value = promotion

    def getPromotion(self):
            """
            Returns the current promotion value.
            1=QUEEN,2=ROOK,3=KNIGHT,4=BISHOP
            """
            return self.promotion_value
```

```
def getLastMoveType(self, board):
        """
        Returns a value that indicates if the last move was a "
            special
        move". Returns -1 if no move has been done. Return
            value can
        be: 0=NORMAL_MOVE 1=EP_MOVE (Pawn is moved two steps
            and is
        valid for en passant strike) 2=EP_CAPTURE_MOVE (A pawn
            has
        captured another pawn by using the en passant rule)
        3=PROMOTION_MOVE (A pawn has been promoted. Use
            getPromotion()
        to see the promotion piece.) 4=KING_CASTLE_MOVE (
            Castling on
        the king side.) 5=QUEEN_CASTLE_MOVE (Castling on the
            queen
        side.)
        """
        if self.state_stack_pointer<=1: # No move has been done
             at thos pointer
                return -1

        self.undo(board)
        move = self.moves[self.state_stack_pointer-1]
        res = move[6]
        self.redo(board)

        return res

def getLastMove(self):
        """
        Returns a tupple containing two tupples describing the
            move
        just made using the internal coordinates.

        In the format ((from_x, from_y), (to_x, to_y))
        Ex. ((4, 6), (4, 4))
        Returns None if no moves has been made.
        """
        if self.state_stack_pointer<=1: # No move has been done
             at thos pointer
                return None
```

```
            self.undo(board)
            move = self.moves[self.state_stack_pointer-1]
            res = (move[1], move[2])
            self.redo(board)

            return res

    def getAllMoves(self, board, format=1):
            """
            Returns a list of all moves done so far in Algebraic
                chess notation.
            Returns None if no moves has been made.
            """
            if self.state_stack_pointer<=1: # No move has been done
                 at this pointer
                    return None

            res = []

            point = self.state_stack_pointer

            self.gotoFirst(board)
            while True:
                    move = self.moves[self.state_stack_pointer-1]
                    res.append(self.formatTextMove(move, format))
                    if self.state_stack_pointer >= len(self.
                        state_stack)-1:
                            break
                    self.redo(board)

            self.state_stack_pointer = point
            self.loadCurState(board)

            return res

    def getLastMove(self, board, format=1):
            """
            Returns the latest move as Algebraic chess notation.
            Returns None if no moves has been made.
            """
            if self.state_stack_pointer<=1: # No move has been done
                 at that pointer
```

```
                return None

            self.undo(board)
            move = self.moves[self.state_stack_pointer-1]
            res = self.formatTextMove(move, format)
            self.redo(board)
            return res

    def gotoMove(self, board, move):
            """
            Goto the specified halfmove. Zero (0) is before the
                first move.
            Return False if move is out of range.
            """
            move+=1
            if move > len(self.state_stack):
                    return False
            if move < 1:
                    return False

            self.state_stack_pointer = move
            self.loadCurState(board)

    def loadCurState(self, board):
            s = self.state_stack[self.state_stack_pointer-1]
            b= s[:64]
            v = s[64:72]
            f =  int(s[73:])

            idx = 0
            for r in range(8):
                    for c in range(8):
                            board[r][c]=b[idx]
                            idx+=1

            self.player                             = v[0]
            self.white_king_castle    = int(v[1])
            self.white_queen_castle = int(v[2])
            self.black_king_castle    = int(v[3])
            self.black_queen_castle = int(v[4])
            self.ep[0]                                = int(v[5])
            self.ep[1]                                = int(v[6])
            self.game_result                    = int(v[7])
```

```
            self.stasis_count = f

    def gotoFirst(self, board):
            """
            Goto before the first known move.
            """
            self.state_stack_pointer = 1
            self.loadCurState(board)

    def gotoLast(self, board):
            """
            Goto after the last knwon move.
            """
            self.state_stack_pointer = len(self.state_stack)
            self.loadCurState(board)

    def undo(self, board):
            """
            Undo the last move. Can be used to step back until the
                initial board setup.
            Returns True or False if no more moves can be undone.
            """
            if self.state_stack_pointer <= 1:
                    return False
            self.state_stack_pointer -= 1
            self.loadCurState(board)
            return True

    def redo(self, board):
            """
            If you used the undo method to step backwards you can
                use this
            method to step forward until the last move i reached.
                Returns
            True or False if no more moves can be redone.
            """
            if self.state_stack_pointer == len(self.state_stack):
                    return False
            self.state_stack_pointer += 1
            self.loadCurState(board)
            return True
```

```
class ChessBoard(object):
        """
        """
        # Promotion values
        QUEEN = 1
        ROOK = 2
        KNIGHT = 3
        BISHOP = 4

        # Reason values
        INVALID_MOVE = 1
        INVALID_COLOR = 2
        INVALID_FROM_LOCATION = 3
        INVALID_TO_LOCATION = 4
        MUST_SET_PROMOTION = 5
        GAME_IS_OVER = 6
        AMBIGUOUS_MOVE = 7

        # Result values
        WHITE_WIN = 8
        BLACK_WIN = 9
        STALEMATE = 10
        STASIS_COUNT_LIMIT_RULE = 11
        THREE_REPETITION_RULE = 12

        # Special moves
        NORMAL_MOVE = 0
        EP_MOVE = 1
        EP_CAPTURE_MOVE = 2
        PROMOTION_MOVE = 3
        KING_CASTLE_MOVE = 4
        QUEEN_CASTLE_MOVE = 5

        # Text move output type
        AN = 0      # g4-e3
        SAN = 1   # Bxe3
        LAN = 2   # Bg4xe3

        def __init__(self):
                self.resetBoard()
                self.num_Queen_B = 1
                self.num_Queen_W = 1
```

```
        self.num_Bishop_B = 2
        self.num_Bishop_W = 2

        self.num_Knight_B = 2
        self.num_Knight_W = 2

        self.num_Rook_B = 2
        self.num_Rook_W = 2

        self.num_Pawn_B = 8
        self.num_Pawn_W = 8

        self.can_promote_b = False
        self.can_promote_w = False

        self.valid_moves_b = 0
        self.valid_moves_w = 0

    def __repr__(self):
        """
        Return the current board layout.
        """
        s = "  lower = b upper = W\n"
        s += "  +-----------------+\n"
        rank = 8
        i = 0
        for l in self.board:
            s += "%d | %s %s %s %s %s %s %s %s | %d\n" % (
                rank, l[0], l[1], l[2], l[3],
                l[4], l[5], l[6], l[7], i)
            rank-=1
            i += 1
        s += "  +-----------------+\n"
        s += "  A B C D E F G H\n"
        s += "  0 1 2 3 4 5 6 7\n"
        return s


    def getOtherPlayer(self, state):
        if state.player == 'w':
            return 'b'
        elif state.player == 'b':
            return 'w'
```

```
        else:
                raise AttributeException("invalid player: '%s'"
                    % state.player)

def canCastle(self, state):
        if state.white_king_castle and state.white_queen_castle
            and state.black_king_castle  and state.
           black_queen_castle:
                return True
        return False

def controlOfCenter(self):
        black = 0
        white = 0

        if self.getColor(3, 3) == "w":
                white += 1
        elif self.getColor(3, 3) == "b":
                black += 1

        if self.getColor(4, 4) == "w":
                white += 1
        elif self.getColor(4, 4) == "b":
                black += 1

        if self.getColor(3, 4) == "w":
                white += 1
        elif self.getColor(3, 4) == "b":
                black += 1

        if self.getColor(4, 3) == "w":
                white += 1
        elif self.getColor(4, 3) == "b":
                black += 1

        if black > white:
                return "b"
        elif white > black:
                return "w"
        else:
                return ""
```

```
def checkKingGuard(self, state, fromPos, moves, specialMoves
    ={}):
        result = []

        kx, ky = self.getKingLocation(state)
        fx, fy = fromPos

        done = False
        fp = self.board[fy][fx]
        self.board[fy][fx] = " "
        if not self.isThreatened(state, kx, ky):
                done = True
        self.board[fy][fx] = fp

        if done:
                return moves

        for m in moves:
                tx, ty = m
                sp = None
                fp = self.board[fy][fx]
                tp = self.board[ty][tx]

                self.board[fy][fx] = " "
                self.board[ty][tx] = fp

                if ((m in specialMoves) and
                        specialMoves[m] == self.EP_CAPTURE_MOVE
                            ):
                        sp = self.board[state.ep[1]][state.ep
                            [0]]
                        self.board[state.ep[1]][state.ep[0]] =
                            " "

                if not self.isThreatened(state, kx, ky):
                        result.append(m)

                if sp:
                        self.board[state.ep[1]][state.ep[0]] =
                            sp

                self.board[fy][fx] = fp
                self.board[ty][tx] = tp
```

```
        return result

    def isFree(self, x, y):
        return self.board[y][x] == ' '

    def getColor(self, x, y):
        if self.board[y][x] == ' ':
            return ' '
        elif self.board[y][x].isupper():
            return 'w'
        elif self.board[y][x].islower():
            return 'b'

    def isThreatened(self, state, lx, ly):
        #pawns
        if state.player == 'w':
            if lx<7 and ly>0 and self.board[ly-1][lx+1] ==
                'p':
                    return True
            elif lx>0 and ly>0 and self.board[ly-1][lx-1]
                == 'p':
                    return True
        else:
            if lx<7 and ly<7 and self.board[ly+1][lx+1] ==
                'P':
                    return True
            elif lx>0 and ly<7 and self.board[ly+1][lx-1]
                == 'P':
                    return True

        #knights
        m =[(lx+1, ly+2), (lx+2, ly+1), (lx+2, ly-1), (lx+1, ly
            -2),
                (lx-1, ly+2), (lx-2, ly+1), (lx-1, ly-2), (lx
                    -2, ly-1)]
        for p in m:
            if p[0] >= 0 and p[0] <= 7 and p[1] >= 0 and p
                [1] <= 7:
                    if self.board[p[1]][p[0]] == "n" and
                        state.player=='w':
                            return True
```

```
                    elif self.board[p[1]][p[0]] == "N" and
                        state.player=='b':
                            return True

        dirs = [(1, 0), (-1, 0), (0, 1), (0, -1),
                    (1, 1), (-1, 1), (1, -1), (-1, -1)]
        for d in dirs:
            x = lx
            y = ly
            dx, dy = d
            steps = 0
            while True:
                steps+=1
                x+=dx
                y+=dy
                if x<0 or x>7 or y<0 or y>7:
                    break
                if self.isFree(x, y):
                    continue
                elif self.getColor(x, y)==state.player:
                    break
                else:
                    p = self.board[y][x].upper()
                    if p == 'K' and steps == 1:
                        return True
                    elif p == 'Q':
                        return True
                    elif p == 'R' and abs(dx) !=
                        abs(dy):
                        return True
                    elif p == 'B' and abs(dx) ==
                        abs(dy):
                        return True
                    break
        return False


    def hasAnyValidMoves(self, state):
        for y in range(0, 8):
            for x in range(0, 8):
                if self.getColor(x, y) == state.player:
                    if len(self.getValidMoves(state
                        , (x, y))):
```

```
                                        return True
        return False

#    ------------------------------------------------------------------------

def traceValidMoves(self, state, fromPos, dirs, maxSteps=8):
        moves = []
        for d in dirs:
                x, y = fromPos
                dx, dy = d
                steps = 0
                while True:
                        x+=dx
                        y+=dy
                        if x<0 or x>7 or y<0 or y>7:
                                break
                        if self.isFree(x, y):
                                moves.append((x, y))
                        elif self.getColor(x, y)!=state.player:
                                moves.append((x, y))
                                break
                        else:
                                break
                        steps+=1
                        if steps == maxSteps:
                                break
        return moves

def getValidQueenMoves(self, state, fromPos):
        moves = []
        dirs = [(1, 0), (-1, 0), (0, 1), (0, -1),
                        (1, 1), (-1, 1), (1, -1), (-1, -1)]
        moves = self.traceValidMoves(state, fromPos, dirs)
        moves = self.checkKingGuard(state, fromPos, moves)
        return moves

def getValidRookMoves(self, state, fromPos):
        moves = []
        dirs = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        moves = self.traceValidMoves(state, fromPos, dirs)
        moves = self.checkKingGuard(state, fromPos, moves)
        return moves
```

```
def getValidBishopMoves(self, state, fromPos):
        moves = []
        dirs = [(1, 1), (-1, 1), (1, -1), (-1, -1) ]
        moves = self.traceValidMoves(state, fromPos, dirs)
        moves = self.checkKingGuard(state, fromPos, moves)
        return moves

def getValidPawnMoves(self, state, fromPos):
        moves = []
        specialMoves = {}
        fx, fy = fromPos
        if state.player == 'w':
                movedir = -1
                startrow = 6
                ocol = 'b'
                eprow = 3
                if fy + movedir == 0:
                        can_promote_w = True
        else:
                movedir = 1
                startrow = 1
                ocol = 'w'
                eprow = 4
                if fy + movedir == 0:
                        can_promote_b = True

        if self.isFree(fx, fy+movedir):
                moves.append((fx, fy+movedir))

        if fy == startrow:
                if self.isFree(fx, fy+movedir) and self.isFree(
                    fx, fy+(movedir*2)):
                        moves.append((fx, fy+(movedir*2)))
                        specialMoves[(fx, fy+(movedir*2))] =
                            self.EP_MOVE
        if fx < 7 and self.getColor(fx+1, fy+movedir) == ocol:
                moves.append((fx+1, fy+movedir))
        if fx > 0 and self.getColor(fx-1, fy+movedir) == ocol:
                moves.append((fx-1, fy+movedir))

        if fy == eprow and state.ep[1] != 0:
                if state.ep[0] == fx+1:
```

```
                moves.append((fx+1, fy+movedir))
                specialMoves[(fx+1, fy+movedir)] = self.
                    EP_CAPTURE_MOVE
            if state.ep[0] == fx-1:
                moves.append((fx-1, fy+movedir))
                specialMoves[(fx-1, fy+movedir)] = self.
                    EP_CAPTURE_MOVE

        moves = self.checkKingGuard(state, fromPos, moves,
            specialMoves)

        return (moves, specialMoves)

    def getValidKnightMoves(self, state, fromPos):
        moves = []
        fx, fy = fromPos
        m =[(fx+1, fy+2), (fx+2, fy+1), (fx+2, fy-1), (fx+1, fy
            -2),
                (fx-1, fy+2), (fx-2, fy+1), (fx-1, fy-2), (fx
                    -2, fy-1)]
        for p in m:
            if p[0] >= 0 and p[0] <= 7 and p[1] >= 0 and p
                [1] <= 7:
                    if self.getColor(p[0], p[1])!=state.
                        player:
                            moves.append(p)

        moves = self.checkKingGuard(state, fromPos, moves)

        return moves

    def getValidKingMoves(self, state, fromPos):
        moves = []
        specialMoves={}

        if state.player == 'w':
            c_row = 7
            c_king = state.white_king_castle
            c_queen = state.white_queen_castle
            k = "K"
        else:
            c_row = 0
            c_king = state.black_king_castle
```

```
            c_queen = state.black_queen_castle
            k = "k"

    dirs = [(1, 0), (-1, 0), (0, 1), (0, -1),
                    (1, 1), (-1, 1), (1, -1), (-1, -1) ]

    t_moves = self.traceValidMoves(state, fromPos, dirs, 1)
    moves = []

    self.board[fromPos[1]][fromPos[0]] = '␣'

    for m in t_moves:
            if not self.isThreatened(state, m[0], m[1]):
                    moves.append(m)

    if c_king:
            if (self.isFree(5, c_row) and self.isFree(6,
               c_row) and
                    self.board[c_row][7].upper() == 'R'):
                    if (not self.isThreatened(state, 4,
                       c_row) and
                            not self.isThreatened(state, 5,
                                c_row) and
                            not self.isThreatened(state, 6,
                                c_row)):
                            moves.append((6, c_row))
                            specialMoves[(6, c_row)] = self
                                .KING_CASTLE_MOVE
    if c_queen:
            if (self.isFree(3, c_row) and self.isFree(2,
               c_row) and
                    self.isFree(1, c_row) and self.board[
                       c_row][0].upper() == 'R'):
                    if (not self.isThreatened(state, 4,
                       c_row) and
                            not self.isThreatened(state, 3,
                                c_row) and
                            not self.isThreatened(state, 2,
                                c_row)):
                            moves.append((2, c_row))
                            specialMoves[(2, c_row)] = self
                                .QUEEN_CASTLE_MOVE
```

```
        self.board[fromPos[1]][fromPos[0]] = k

        return (moves, specialMoves)

#------------------------------------------------------------
def movePawn(self, state, fromPos, toPos):
        moves, specialMoves = self.getValidPawnMoves(state,
            fromPos)

        if not toPos in moves:
                return False

        if toPos in specialMoves:
                t = specialMoves[toPos]
        else:
                t = 0

        if t == self.EP_CAPTURE_MOVE:
                self.board[state.ep[1]][state.ep[0]] = ' '
                state.cur_move[3]=True
                state.cur_move[6]=self.EP_CAPTURE_MOVE

        pv = state.promotion_value
        if state.player == 'w' and toPos[1] == 0:
                if pv == 0:
                        state.reason = self.MUST_SET_PROMOTION
                        return False
                pc = ['Q', 'R', 'N', 'B']
                p = pc[pv-1]
                state.cur_move[4]=p
                state.cur_move[6]=self.PROMOTION_MOVE
                #state.promotion_value = 0
        elif state.player == 'b' and toPos[1] == 7:
                if pv == 0:
                        state.reason = self.MUST_SET_PROMOTION
                        return False
                pc = ['q', 'r', 'n', 'b']
                p = pc[pv-1]
                state.cur_move[4]=p
                state.cur_move[6]=self.PROMOTION_MOVE
                #state.promotion_value = 0
        else:
                p = self.board[fromPos[1]][fromPos[0]]
```

```
        if t == self.EP_MOVE:
                state.setEP(toPos)
                state.cur_move[6]= self.EP_MOVE
        else:
                state.clearEP()

        if self.board[toPos[1]][toPos[0]] != ' ':
                state.cur_move[3]=True

        self.board[toPos[1]][toPos[0]] = p
        self.board[fromPos[1]][fromPos[0]] = " "

        state.stasis_count = 0
        return True


def moveKnight(self, state, fromPos, toPos):
        moves = self.getValidKnightMoves(state, fromPos)

        if not toPos in moves:
                return False

        state.clearEP()

        if self.board[toPos[1]][toPos[0]] == " ":
                state.stasis_count+=1
        else:
                state.stasis_count=0
                state.cur_move[3]=True

        self.board[toPos[1]][toPos[0]] = self.board[fromPos
            [1]][fromPos[0]]
        self.board[fromPos[1]][fromPos[0]] = " "
        return True


def moveKing(self, state, fromPos, toPos):
        if state.player == 'w':
                c_row = 7
                k = "K"
                r = "R"
        else:
```

```
            c_row = 0
            k = "k"
            r = "r"

    moves, specialMoves = self.getValidKingMoves(state,
        fromPos)

    if toPos in specialMoves:
            t = specialMoves[toPos]
    else:
            t = 0

    if not toPos in moves:
            return False

    state.clearEP()

    if state.player == 'w':
            state.white_king_castle = False
            state.white_queen_castle = False
    else:
            state.black_king_castle = False
            state.black_queen_castle = False

    if t == self.KING_CASTLE_MOVE:
            state.stasis_count+=1
            self.board[c_row][4] = " "
            self.board[c_row][6] = k
            self.board[c_row][7] = " "
            self.board[c_row][5] = r
            state.cur_move[6] = self.KING_CASTLE_MOVE
    elif t == self.QUEEN_CASTLE_MOVE:
            state.stasis_count+=1
            self.board[c_row][4] = " "
            self.board[c_row][2] = k
            self.board[c_row][0] = " "
            self.board[c_row][3] = r
            state.cur_move[6] = self.QUEEN_CASTLE_MOVE
    else:
            if self.board[toPos[1]][toPos[0]] == " ":
                    state.stasis_count+=1
            else:
                    state.stasis_count=0
```

```
                                state.cur_move[3]=True

                        self.board[toPos[1]][toPos[0]] = self.board[
                            fromPos[1]][fromPos[0]]
                        self.board[fromPos[1]][fromPos[0]] = " "

                return True

        def moveQueen(self, state, fromPos, toPos):

                moves = self.getValidQueenMoves(state, fromPos)

                if not toPos in moves:
                        return False

                state.clearEP()

                if self.board[toPos[1]][toPos[0]] == " ":
                        state.stasis_count+=1
                else:
                        state.stasis_count=0
                        state.cur_move[3]=True

                self.board[toPos[1]][toPos[0]] = self.board[fromPos
                    [1]][fromPos[0]]
                self.board[fromPos[1]][fromPos[0]] = " "
                return True

        def moveBishop(self, state, fromPos, toPos):
                moves = self.getValidBishopMoves(state, fromPos)

                if not toPos in moves:
                        return False

                state.clearEP()

                if self.board[toPos[1]][toPos[0]] == " ":
                        state.stasis_count+=1
                else:
                        state.stasis_count=0
                        state.cur_move[3]=True
```

```
            self.board[toPos[1]][toPos[0]] = self.board[fromPos
                [1]][fromPos[0]]
            self.board[fromPos[1]][fromPos[0]] = " "
            return True

    def moveRook(self, state, fromPos, toPos):

            moves = self.getValidRookMoves(state, fromPos)

            if not toPos in moves:
                    return False

            fx, fy = fromPos
            if state.player == 'w':
                    if fx == 0:
                            state.white_queen_castle = False
                    if fx == 7:
                            state.white_king_castle = False
            elif state.player == 'b':
                    if fx == 0:
                            state.black_queen_castle = False
                    if fx == 7:
                            state.black_king_castle = False

            state.clearEP()

            if self.board[toPos[1]][toPos[0]] == " ":
                    state.stasis_count+=1
            else:
                    state.stasis_count=0
                    state.cur_move[3]=True

            self.board[toPos[1]][toPos[0]] = self.board[fromPos
                [1]][fromPos[0]]
            self.board[fromPos[1]][fromPos[0]] = " "
            return True

    def parseTextMove(self, state, txt):

            txt = txt.strip()
            promotion = None
            dest_x = 0
            dest_y = 0
```

```
h_piece = "P"
h_rank = -1
h_file = -1

# handle the special
if txt == "O-O":
        if state.player == 'w':
                return (None, 4, 7, 6, 7, None)
        if state.player == 'b':
                return (None, 4, 0, 6, 0, None)
if txt == "O-O-O":
        if state.player == 'w':
                return (None, 4, 7, 2, 7, None)
        if state.player == 'b':
                return (None, 4, 0, 2, 0, None)

files = {"a":0, "b":1, "c":2, "d":3, "e":4, "f":5, "g"
    :6, "h":7}
ranks = {"8":0, "7":1, "6":2, "5":3, "4":4, "3":5, "2"
    :6, "1":7}

# Clean up the textmove
"".join(txt.split("e.p."))
t = []
for ch in txt:
        if ch not in "KQRNBabcdefgh12345678":
                continue
        t.append(ch)

if len(t)<2:
        return None

# Get promotion if any
if t[-1] in ('Q', 'R', 'N', 'B'):
        promotion = {'Q':1, 'R':2, 'N':3, 'B':4}[t.pop
            ()]

if len(t)<2:
        return None

# Get the destination
if not (t[-2] in files) or not (t[-1] in ranks):
        return None
```

```
        dest_x = files[t[-2]]
        dest_y = ranks[t[-1]]

        # Pick out the hints
        t = t[:-2]
        for h in t:
                if h in ('K', 'Q', 'R', 'N', 'B', 'P'):
                        h_piece = h
                elif h in ('a', 'b', 'c', 'd', 'e', 'f', 'g', '
                    h'):
                        h_file = files[h]
                elif h in ('1', '2', '3', '4', '5', '6', '7', '
                    8'):
                        h_rank = ranks[h]

        # If we have both a source and destination we don't
            need the piece hint.
        # This will make us make the move directly.
        if h_rank > -1 and h_file > -1:
                h_piece = None

        return (h_piece, h_file, h_rank, dest_x, dest_y,
            promotion)

def formatTextMove(self, move, format):
        #piece, from, to, take, promotion, check

        piece = move[0]
        fpos = tuple(move[1])
        tpos = tuple(move[2])
        take = move[3]
        promo = move[4]
        check = move[5]
        special = move[6]

        files = "abcdefgh"
        ranks = "87654321"
        if format == self.AN:
                res = "%s%s%s%s" % (files[fpos[0]], ranks[fpos
                    [1]],
```

```
                                                                files[
                                                                    tpos
                                                                    [0]],

                                                                    ranks
                                                                    [
                                                                    tpos
                                                                    [1]])

        elif format == self.LAN:

            if special == self.KING_CASTLE_MOVE:
                return "O-O"
            elif special == self.QUEEN_CASTLE_MOVE:
                return "O-O-O"

            tc = "-"
            if take:
                tc = "x"
            pt = ""
            if promo:
                pt = "=%s" % promo
            if piece == "P":
                piece = ""
            if not check:
                check = ""
            res = "%s%s%s%s%s%s%s%s" % (piece, files[fpos
                [0]], ranks[fpos[1]],
                                                                        tc
                                                                        ,

                                                                        files
                                                                        [
                                                                        tpos
                                                                        [0]],

                                                                        ranks
                                                                        [
                                                                        tpos
                                                                        [1]],
```

```
pt
,
c
)
```

```
elif format == self.SAN:

        if special == self.KING_CASTLE_MOVE:
                return "O-O"
        elif special == self.QUEEN_CASTLE_MOVE:
                return "O-O-O"

        tc = ""
        if take:
                tc = "x"
        pt = ""
        if promo:
                pt = "=%s" % promo.upper()
        p = piece
        if state.player == 'b':
                p = p.lower()
        if piece == "P":
                piece = ""
        if not check:
                check = ""
        fx, fy = fpos
        hint_f = ""
        hint_r = ""
        for y in range(8):
                for x in range(8):
                        if self.board[y][x] == p:
                                if x == fx and y == fy:
                                        continue
                                vm = self.getValidMoves
                                    (state, (x, y))
                                if tpos in vm:
                                        if fx == x:
                                                hint_r
                                                    =
                                                    ranks
                                                    [fy]
                                        else:
```

```
                                                                  hint_f
                                                                     =
                                                                    files
                                                                    [fx]
                if piece == "" and take:
                        hint_f = files[fx]
                res = "%s%s%s%s%s%s%s%s" % (piece, hint_f,
                   hint_r, tc,
                                                                                  files
                                                                                     [
                                                                                     tpos
                                                                                     [0]],

                                                                                     ranks
                                                                                     [
                                                                                     tpos
                                                                                     [1]],

                                                                                  pt
                                                                                     ,

                                                                                     check
                                                                                     )

        return res

def getValidMoves(self, state, location):
        """
        Returns a list of valid moves. (ex [ [3, 4], [3, 5],
           [3, 6]
        ... ] ) If there isn't a valid piece on that location
           or the
        piece on the selected location hasn't got any valid
           moves an
        empty list is returned. The location argument must be
           a tuple
        containing an x, y value Ex. (3, 3)
        """
        if state.game_result:
                return []

        x, y = location
```

```
        if x < 0 or x > 7 or y < 0 or y > 7:
            return False

        if self.getColor(x, y) != state.player:
            return []

        p = self.board[y][x].upper()
        if p == 'P':
            m, s = self.getValidPawnMoves(state, location)
            return m
        elif p == 'R':
            return self.getValidRookMoves(state, location)
        elif p == 'B':
            return self.getValidBishopMoves(state, location
                )
        elif p == 'Q':
            return self.getValidQueenMoves(state, location)
        elif p == 'K':
            m, s = self.getValidKingMoves(state, location)
            return m
        elif p == 'N':
            return self.getValidKnightMoves(state, location
                )
        else:
            return []


    #
        ------------------------------------------------------------------------

    # PUBLIC METHODS
    #
        ------------------------------------------------------------------------


    def getMoves(self, state):
        retval = []
        for y in range(0, 8):
            for x in range(0, 8):
                if self.getColor(x, y) == state.player:
                    moves = self.getMoveFrom(state,
                        (x, y))
                    if moves:
```

```
                                    retval.append(((x,y),
                                        self.board[y][x],
                                        moves))
        if state.player == "b":
            self.valid_moves_b = len(retval)
        elif state.player == "w":
            self.valid_moves_w = len(retval)
        return retval

    def getMoveFrom(self, state, location):
        """
        Returns a list of valid moves. (ex [ [3, 4], [3, 5],
            [3, 6]
        ... ] ) If there isn't a valid piece on that location
            or the
        piece on the selected location hasn't got any valid
            moves an
        empty list is returned. The location argument must be
            a tuple
        containing an x, y value Ex. (3, 3)
        """
        x, y = location
        if x < 0 or x > 7 or y < 0 or y > 7:
                return False
        p = self.board[y][x]
        p = p.upper()
        if p == 'P':
                m, s = self.getValidPawnMoves(state, location)
                return m
        elif p == 'R':
                return self.getValidRookMoves(state, location)
        elif p == 'B':
                return self.getValidBishopMoves(state, location
                    )
        elif p == 'Q':
                return self.getValidQueenMoves(state, location)
        elif p == 'K':
                m, s = self.getValidKingMoves(state, location)
                return m
        elif p == 'N':
                return self.getValidKnightMoves(state, location
                    )
        else:
```

```
                return []

    def resetBoard(self):
        """
        Resets the chess board and all states.
        """
        self.board = [
                ['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r'],
                ['p']*8,
                [' ']*8,
                [' ']*8,
                [' ']*8,
                [' ']*8,
                ['P']*8,
                ['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
                ]

    def getKingLocation(self, state):
        for y in range(0,8):
            for x in range(0,8):
                if self.board[y][x] == "K" and state.player
                    == 'w':
                    return (x,y)
                if self.board[y][x] == "k" and state.player
                    == 'b':
                    return (x,y)
        print "can't find king?!"
        print self.board

    def isCheck(self, state):
        """
        Returns True if the current players king is checked.
        """
        kx, ky = self.getKingLocation(state)
        return self.isThreatened(state, kx, ky)

    def getBoard(self):
        """
        Returns a copy of the current board layout. Uppercase
            letters
        for white, lowercase for black. K=King, Q=Queen, B=
            Bishop,
```

```
        N=Night, R=Rook, P=Pawn.  Empty squares are markt with
            a
        period (.)
        """
        return deepcopy(self.board)

def makeMove(self, state, fromPos, toPos):
        """
        Tries to move the piece located om fromPos to toPos.
            Returns
        True if that was a valid move.  The position arguments
            must be
        tuples containing x, y value Ex. (4, 6).  This method
            also
        detects game over.

        If this method returns False. You can use the getReason
             method
        to determin why.
        """
        fx, fy = fromPos
        tx, ty = toPos

        state.cur_move[1]=fromPos
        state.cur_move[2]=toPos

        #check invalid coordinates
        if fx < 0 or fx > 7 or fy < 0 or fy > 7:
                state.reason = self.INVALID_FROM_LOCATION
                return False

        #check invalid coordinates
        if tx < 0 or tx > 7 or ty < 0 or ty > 7:
                state.reason = self.INVALID_TO_LOCATION
                return False

        #check if any move at all
        if fx==tx and fy==ty:
                state.reason = self.INVALID_TO_LOCATION
                return False

        #check if piece on location
        if self.isFree(fx, fy):
```

```
            state.reason = self.INVALID_FROM_LOCATION
            return False

    #check color of piece
    if self.getColor(fx, fy) != state.player:
            state.reason = self.INVALID_COLOR
            return False

    # Call the correct handler
    p = self.board[fy][fx].upper()
    state.cur_move[0]=p
    if p == 'P':
            if not self.movePawn(state, (fx, fy), (tx, ty))
                :
                    if not state.reason:
                            state.reason = self.
                                INVALID_MOVE
                    return False
    elif p == 'R':
            if not self.moveRook(state, (fx, fy), (tx, ty))
                :
                    state.reason = self.INVALID_MOVE
                    return False
    elif p == 'B':
            if not self.moveBishop(state, (fx, fy), (tx, ty
                )):
                    state.reason = self.INVALID_MOVE
                    return False
    elif p == 'Q':
            if not self.moveQueen(state, (fx, fy), (tx, ty)
                ):
                    state.reason = self.INVALID_MOVE
                    return False
    elif p == 'K':
            if not self.moveKing(state, (fx, fy), (tx, ty))
                :
                    state.reason = self.INVALID_MOVE
                    return False
    elif p == 'N':
            if not self.moveKnight(state, (fx, fy), (tx, ty
                )):
                    state.reason = self.INVALID_MOVE
                    return False
```

```
            else:
                return False

            state.pushState(self.board)
            state.pushMove()
            other = self.getOtherPlayerState(state)
            state.move_count += 1
            return True

    def getOtherPlayerState(self, state):
            newState = deepcopy(state)
            if state.player == 'w':
                newState.player = 'b'
            else:
                newState.player = 'w'
            return newState

    def checkStatus(self, state):
            print("Move: %s" % state.move_count)
            print(self)
            if self.isCheck(state):
                state.cur_move[5]="+"

            if not self.hasAnyValidMoves(state):
                if self.isCheck(state):
                    state.cur_move[5]="#"
                    if state.player == 'w':
                        state.endGame(self.BLACK_WIN)
                    else:
                        state.endGame(self.WHITE_WIN)
                else:
                    state.endGame(self.STALEMATE)
            else:
                if state.stasis_count == 100:
                    state.endGame(self.
                        STASIS_COUNT_LIMIT_RULE)
                elif state.threeRepetitions():
                    state.endGame(self.
                        THREE_REPETITION_RULE)

            printReason(state.game_result)

def randomPlayer(board, state, moves):
```

```
        # moves is a list of [(from, piece, moves), ...]
        # [((0, 1), 'P', [(0, 2), (0, 3)]), ...]
        selection = random.choice(moves)
        fromPos = selection[0]
        toPos = random.choice(selection[2])
        return fromPos, toPos

def evaluate_heuristic(board, state):
        if state.player == "w":
                res = (state.white_chromosome.getKnightParameter()*(
                    board.num_Knight_W-board.num_Knight_B)) +  (state.
                    white_chromosome.getBishopParameter()*(board.
                    num_Bishop_W-board.num_Bishop_B)) + (state.
                    white_chromosome.getRookParameter()*(board.
                    num_Rook_W-board.num_Rook_B)) + (state.
                    white_chromosome.getQueenParameter()*(board.
                    num_Queen_W-board.num_Queen_B)) + (state.
                    white_chromosome.getLegalMovesParameter()*(board.
                    valid_moves_w-board.valid_moves_b))
                if board.controlOfCenter() == "w":
                        res = res + state.white_chromosome.
                            getControlOfCenterParameter()
                if board.can_promote_w:
                        res = res + state.white_chromosome.
                            getPromotionParameter()
                        board.can_promote_w = False
                if board.canCastle(state):
                        res = res + state.white_chromosome.
                            getCastleParameter()
                return res
        else:
                res = (state.black_chromosome.getKnightParameter()*(
                    board.num_Knight_W-board.num_Knight_B)) +  (state.
                    black_chromosome.getBishopParameter()*(board.
                    num_Bishop_W-board.num_Bishop_B)) + (state.
                    black_chromosome.getRookParameter()*(board.
                    num_Rook_W-board.num_Rook_B)) + (state.
                    black_chromosome.getQueenParameter()*(board.
                    num_Queen_W-board.num_Queen_B)) + (state.
                    black_chromosome.getLegalMovesParameter()*(board.
                    valid_moves_w-board.valid_moves_b))
                if board.controlOfCenter() == "b":
```

```
                        res = res + state.black_chromosome.
                            getControlOfCenterParameter()
                if board.can_promote_b:
                        res = res + state.black_chromosome.
                            getPromotionParameter()
                        board.can_promote_b = False
                if board.canCastle(state):
                        res = res + state.black_chromosome.
                            getCastleParameter()
                return res

def null_move_search(board, state, player, bound, fromPos, toPos, alpha
    , beta, quiescence):
        state.null_ok = False
        n_board = deepcopy(board)
        next_state = n_board.getOtherPlayerState(state)
        pos, val = alpha_beta_search(n_board, next_state, player, bound
            +2+1, fromPos, toPos, alpha, beta, quiescence)
        fromPos, toPos = pos
        if val >= alpha or val <= beta:
                if state.verify:
                        bound = bound + 2
                        state.verify = False
                        state.fail_high = True
                else:
                        return pos, val, bound

        if state.fail_high:
                next_board = deepcopy(n_board)
                next_board.makeMove(state, fromPos, toPos)
                next_state = n_board.getOtherPlayerState(state)
                pos, val = null_move_re_search(next_board, next_state,
                    player, bound+1, fromPos, toPos, alpha, beta,
                    quiescence)
        return pos, val, bound

def null_move_re_search(board, state, player, bound, fromPos, toPos,
    alpha, beta, quiescence):
        pos, val = alpha_beta_search(board, state, player, bound,
            fromPos, toPos, alpha, beta, quiescence)
        fromPos, toPos = pos

        if state.fail_high and (val < beta or val > alpha):
```

```
                new_bound = new_bound - 2
                state.fail_high = False
                state.verify = True
                n_board = deepcopy(board)
                n_board.makeMove(state, fromPos, toPos)
                next_state = n_board.getOtherPlayerState(state)
                return null_move_re_search(n_board, next_state, player,
                    new_bound+1, fromPos, toPos, alpha, beta,
                    quiescence)
        return pos, val

def alpha_beta(board, state, player, bound, fromPos, toPos, alpha, beta
    ):
        t_board = deepcopy(board)
        t_state = deepcopy(state)
        return alpha_beta_search(t_board, t_state, player, bound,
            fromPos, toPos, alpha, beta, 0)

def alpha_beta_search(t_board, state, player, bound, fromPos, toPos,
    alpha, beta, quiescence):
        new_bound = bound
        possible_list = t_board.getMoves(state)
        # [fromPos, pieceType, possible_moves[]]
        if bound >= 2 or len(possible_list) < 1:
##              if t_board.isCheck(state) and quiescence < 2:
##                      for moves in possible_list:
##                              f_pos = moves[0]
##                              for next_move in moves[2]:
##                                      n_board = deepcopy(t_board)
##                                      n_board.makeMove(state, f_pos,
    next_move)
##                                      next_state = n_board.
    getOtherPlayerState(state)
##                                      pos, val = alpha_beta_search(
    n_board, next_state, player, new_bound-2, f_pos, next_move, alpha,
    beta, quiescence + 1)
##                                      return pos, val

                res = evaluate_heuristic(t_board, state)
                return (fromPos, toPos), res

##          if ((not t_board.isCheck(state)) and state.null_ok and ((not
    state.verify) or depth < 5)):
```

```
##                  pos, val, new_bound = null_move_search(t_board, state
    , player, new_bound-2, fromPos, toPos, alpha, beta, quiescence)
##          elif not state.null_ok:
##                  state.null_ok = True
        if state.player == player:
                best = None
                for moves in possible_list:
                        f_pos = moves[0]
                        for next_move in moves[2]:
                                n_board = deepcopy(t_board)
                                n_board.makeMove(state, f_pos,
                                    next_move)
                                next_state = n_board.
                                    getOtherPlayerState(state)
                                pos, result = alpha_beta_search(n_board
                                    , next_state, player, new_bound+1,
                                    f_pos, next_move, alpha, beta,
                                    quiescence)
                                if result > alpha:
                                        alpha = result
                                        best = (f_pos, next_move)
                                if alpha >=  beta:
                                        return best, beta
                return best, alpha

        else:
                worst = None
                for moves in possible_list:
                        f_pos = moves[0]
                        for next_move in moves[2]:
                                n_board = deepcopy(t_board)
                                n_board.makeMove(state, f_pos,
                                    next_move)
                                next_state = n_board.
                                    getOtherPlayerState(state)
                                pos, result = alpha_beta_search(n_board
                                    , next_state, player, new_bound+1,
                                    f_pos, next_move, alpha, beta,
                                    quiescence)
                                if result < beta:
                                        beta = result
                                        worst = (f_pos, next_move)
                                if beta <= alpha:
```

```
                                        return worst, alpha
                return worst, beta

def play(player1, player2, p1_genes, p2_genes):
        state = State('w')
        state.white_chromosome = p1_genes
        state.black_chromosome = p2_genes
        board = ChessBoard()
        while state.game_result == 0:
                moves = board.getMoves(state)
                if moves:
                    if state.player == 'w':
                            f, t = player1(board, state, moves)
                            pos, res = alpha_beta(board, state, 'w', 0,
                                f, t, float(-1e3000), float(1e3000))
                    else:
                            pos, res = alpha_beta(board, state, 'b', 0,
                                f, t, float(-1e3000), float(1e3000))
                    fromPos = pos[0]
                    toPos = pos[1]
                    print("%s moves %s from %s to %s" %
                              (state.player, board.board[fromPos[1]][
                                 fromPos[0]],
                               fromPos, toPos))

                    board.makeMove(state, fromPos, toPos)
                    if state.game_result == 0:
                            state.player = board.getOtherPlayer(state)
                            board.checkStatus(state)
                else:
                        break
        if state.game_result == 8:
                print "returning 2, 0"
                return 2, 0
        elif state.game_result == 9:
                print "returning 0, 2"
                return 0, 2
        else:
                print "returning tie"
                return 1, 1

def pop_dynamics(size):
        population = [None]*(size+1)
```

```
population[0] = Chromosome([3, 3, 5, 9, 1, 8, 1, 1])
for y in range(size):
        population[y+1] = Chromosome([random.randint(0, 12),
            random.randint(0, 12), random.randint(0, 12), random
            .randint(0, 12), random.randint(0, 12), random.
            randint(0, 12), random.randint(0, 12), random.
            randint(0, 12)])

for x in range(len(population)):
                print "chromosome: " +str(population[x])

c = 0
training = True
p1 = 0
p2 = 0
average = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
std_dev = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
total = 0.0
mutation_rate = 0.5
crossover = 0.1
generation = 0
print "Generation 0"

while training:
        for x in range(8):
                average[x] = 0.0
                std_dev[x] = 0.0

        i = random.randint(c, len(population)-1)
        print "Member " + str(c) + " is playing member " + str(
            i) +"."
        t_p1, t_p2 = play(randomPlayer, randomPlayer,
            population[c], population[i])
        p1 = p1 + t_p1
        p2 = p2 + t_p2

        t_p2, t_p1 = play(randomPlayer, randomPlayer,
            population[i], population[c])
        p1 = p1 + t_p1
        p2 = p2 + t_p2

        for i in range(len(population)):
```

```
		for j in range(len(average)):
			average[j] = average[j] + population[i
				].genes[j]
	print "average␣*␣size:␣" +str(average)

	for k in range(len(population)):
		for l in range(len(std_dev)):
			std_dev[l] = std_dev[l] + math.pow(
				population[k].genes[l] - (average[l
				]/len(population)), 2)

	for m in range(len(std_dev)):
		std_dev[m] = std_dev[m] / (len(population[0].
			genes) - 1)
		std_dev[m] = math.pow(std_dev[m], 0.5)
		total = total + std_dev[m]
	print "std.␣deviation:␣" +str(std_dev)
	total = total / len(std_dev)
	print "total:␣" + str(total)

	if mutation_rate < random.random():
		if p1 > p2 and p1 == 4:
			for n in range(len(population[c].genes)
				):
				population[i].genes[n] =
					population[c].genes[n] + ((
					random.random()-0.5)*2*
					std_dev[n])
		elif p1 > p2 and p1 == 3:
			for n in range(len(population[c].genes)
				):
				population[i].genes[n] =
					population[c].genes[n] + ((
					random.random()-0.5)*0.5*
					std_dev[n])
		elif p2 > p1 and p2 == 4:
			for n in range(len(population[c].genes)
				):
				population[c].genes[n] =
					population[i].genes[n] + ((
					random.random()-0.5)*2*
					std_dev[n])
		elif p2 > p1 and p2 == 3:
```

```
                                for n in range(len(population[c].genes)
                                    ):
                                        population[c].genes[n] =
                                            population[i].genes[n] + ((
                                            random.random()-0.5)*0.5*
                                            std_dev[n])
                        else:
                                for n in range(len(population[c].genes)
                                    ):
                                        population[c].genes[n] =
                                            population[c].genes[n] + ((
                                            random.random()-0.5)*0.5*
                                            std_dev[n])
                                        population[i].genes[n] =
                                            population[i].genes[n] + ((
                                            random.random()-0.5)*0.5*
                                            std_dev[n])

                else:
                        if p1 > p2:
                                population[i].genes = population[c].
                                    genes
                        elif p2 > p1:
                                population[c].genes = population[i].
                                    genes

##              t_cross = random.random()
##              if (t_cross < crossover):
##                      t_a, t_b = population[c].crossover(random.
    randint(1, 6), population[i])
##                      population[c].genes = t_a
##                      population[i].genes = t_b

                for i in range(len(population)):
                        print "chromosome: " +str(population[i])

                c = c + 1
                if c > len(population)-1:
                        c = 0
                        generation = generation + 1
                        population.reverse()
                        print "Generation: " + str(generation)
```

```
                if(total <= 1.0 or generation >= 72):
                        training = False

        print "System converged to: " + str(population[0])

if __name__ == "__main__":
        pop_dynamics(50)
```