Opponent Modeling in Board Games Using the Estimation Exploration Algorithm

Julian Jocque

March 19, 2015

Abstract

In this paper I will detail my senior thesis work done on modeling opponents in board games using the estimation exploration algorithm. The paper first explains background knowledge to explain concepts relevant to the project. From there I build an outline of a system that could be used to model opponents in konane using the estimation exploration algorithm. This system is implemented and then finally tested to get data. The results found from this system were quite promising, showing that in some cases this system is capable of producing the same moves as an opponent in over 80% of scenarios. Furthermore, it was found that for every opponent that I tested against, the resultant models were significantly better than an untrained random player. These results indicate that the estimation exploration algorithm can be used to effectively model an opponent in a board game, albeit with varying degrees of success.

I would like to give thanks to my advisor John Rieffel for guiding me through this research project as well as his guidance on the tensegrity project and with life.

I'd also like to thank Cara Peterhansel for her constant love and willingness to listen to endless rambling about game AI and EEAs and everything else.

Lastly I'd like to thank my parents. They have been enormously supportive of me throughout my entire life and I cannot thank them thoroughly enough for that.

Contents

List of Figures

1 Introduction

Game AI has been an evolving research area for years. The advances in game AI have come in the form of theoretical advances as well as pragmatic challenges and competitions like the famous Deep Blue vs. Garry Kasparov chess match[8]. These advances have massively improved upon processing capabilities and evaluation efficiency. Game AI has borrowed from and pushed forward other aspects of artificial intelligence.

However, one aspect of Game AI which has only been explored in a very narrow range of games, like Poker[3], is opponent modeling. The term opponent modeling refers to building a model of an opponent to try to predict the next move the opponent will make. This could be used to more intelligently choose a move to counteract the opponent's play.

Opponent modeling also has much wider implications, particularly in the arena of business and warfare strategy. One extremely effective and widely used tactic in both business and warfare is hiding information from an opponent. It is of vital importance that one's opponent is not aware of one's strategy in business and warfare, otherwise the opponent could work to counteract one's strategy. This is why governments and businesses quietly invest into espionage. Effective opponent modeling is one way to overcome a lack of information. If one is able to see through their opponent's attempt to disguise their information, then one is able to use this information to make more sound decisions to combat the opponent.

Another use for opponent modeling is the historical significance of having an accurate model of human players. If we were able to model Garry Kasparov's chess play to a high level of accuracy, then mankind would be able to experience his playstyle for years to come.

The estimation exploration algorithm could prove to be an effective means of modeling an opponent. Because of the promising results that the estimation exploration algorithm has shown in problems similar in structure to the problem of modeling an opponent, I believe the estimation exploration algorithm will prove to be an effective way of modeling an opponent in board games.

2 Background and Related Work

This section will go over past work and concepts related to my work that the reader will need to understand in order to understand my project.

2.1 Opponent Modeling

A lot of work related to modeling an opponent in a game is related to poker. The reason for this is that in poker it is absolutely essential to understand the play of opponents in order to play optimally [3][11]. The need for opponent modeling is largely due to the fact that in poker, there is hidden information. Because of the fact that there is hidden information, it is impossible to use game theory alone to arrive at optimal play[7]. In poker there is also the unique problem that opponents will intentionally rapidly change their strategy in order to be less predictable. One's opponents do not know what is in one's hand but the opponents can try to guess based on what one's play style is like, and subsequently play more efficiently by abusing the fact that they have a good guess of what is in one's hand.

There has been other work in opponent modeling across a wide range of games. Schadd et al.[25] discussed opponent modeling in a real time strategy game. This is a very different problem from poker because the models must be done while the game is actively being played. Real time strategy games are not played turn by turn, they are instead played in real time. Games also last much less time than poker and much less information can be derived from the opponents. However, Schadd et al. were able to overcome these problems by greatly simplifying the models they used for their opponents.

The papers relating to poker have many variables at play. In particular, "A Demonstration of the Polaris Poker System", explained how the Polaris Poker System has many inputs which are then fed into a neural network to make decisions about which move to make[7]. The main simplification that was made for opponent modeling in real time strategy games was to use 'hard coded' classifiers for players. The proposed system made the assertion that each opponent was either aggressive or defensive and of those two types, they made one of four units. This essentially broke the problem space down to only having to classify which of eight potential groups an opponent belongs to and counteract the opponent's strategy. This was effective because of the more strategically simplified nature of the real time strategy game they chose. Board games

play much more like poker, where the strategies can be more easily represented by how much an opponent favours one move over another rather than classifying an opponent into a limited set of play styles.

Thagard discussed adversarial problem solving for games in "Adversarial Problem solving: Modeling an Opponent Using Explanatory Coherence" and mentioned both chess and poker[29]. Interestingly, the paper specifically mentions how much research has been put into opponent modeling in poker but how chess research hadn't yet caught on.

2.2 System Identification

System identification is a thoroughly used and researched problem in both science and engineering. System identification works by having an experimenter be given a hidden system and test it to either learn about the structure of the system or to make the system act in a particular desired way[5]. For example, a chemist may be given an unknown substance and be tasked with coming up with an accurate breakdown of what is in the substance by performing tests on the substance. We will be looking at treating an opponent in a game as if they contain a hidden system which they use to decide what move they will make next. If one performs system identification on this hidden system that the opponent holds, one can learn how it is that they will make future moves and use that knowledge to make moves which one can be certain will be better.

The literature on system identification is almost exclusively high-level theoretical work that is relevant either to pure math or upper level mechanical engineering concepts. However, the literature will be relevant to using the estimation exploration algorithm to perform system identification.

One paper, "Nonlinear Black-box Modeling in System Identification: a Unified Overview" by Sjöberg et al. goes over black-box modeling[28]. Black-box modeling uses system identification to model a completely unknown system. This paper also describes grey-box modeling, which is a type of modeling in which some information about the hidden system is known but some parameters must be tuned and white-box modeling, where the physical state of a system is completely known.

The type of modeling that the estimation exploration algorithm does is grey-box modeling. It requires that the representations for models and tests be closely tied to the system under test, so the system must

already be fairly well known but a lot of the specifics can be tuned by the algorithm.

2.3 Game Concepts

This section will explain the concepts related to game playing and game AI which are necessary for understanding my work. I will use chess for illustrative purposes because it is a very well known board game. However, these concepts are general and apply to all board games.

2.3.1 Game State

Game state is the way of describing the position of a game. In the case of chess, the game state is described by a chess board with chess pieces on it. Game states can be either valid or invalid. Invalid game states are ones which cannot be reached through legal play of the game and valid states are those which can exist within a legally played match of that game.

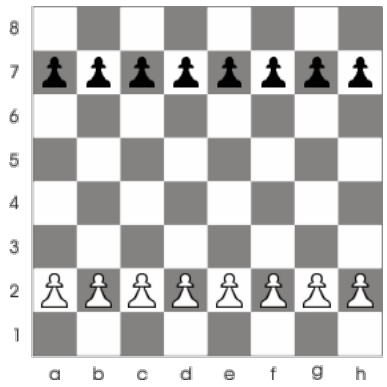

Figure 1: Example of an invalid state in chess[17].

The above figure is an example of an invalid state, because there is no set of moves that can legally result in the above board. At the very least, whichever player lost their king first would lose the game immediately, making it impossible to remove the other king.

Figure 2: One valid game state of a chess game in progress[23].

The above figure is an example of a valid game state because it is possible to arrive at this state through legal play of chess.

Whenever I refer to a game state for the rest of the paper, it can be assumed I am referring to a valid game state.

2.3.2 Game Tree

A game tree consists of the initial state of a game followed by the legal moves that can follow from that initial state. The initial state is simply whatever position the game begins in and all board games have one. A complete game tree would be one that has the initial state and every single possible move following from that state and all subsequent states. For a simple game like tic tac toe, it is feasible to explore the entirety of the game tree. However, for more complicated games like chess it is not possible to explore the entire game tree by today's computers, as it would take many millions of years to do so.

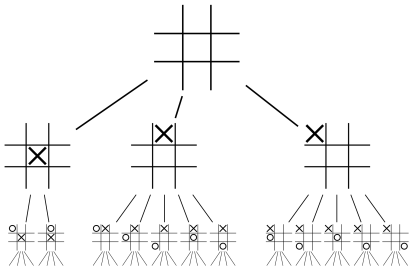

Figure 3: A graphical representation of a tic tac toe game tree[14].

The above figure shows an example of a tic tac toe game tree to depth three. That means it shows the initial position as well as possible states that result from that initial state with two moves, one from each player. This figure is a bit misleading though, as it only shows three possible moves from the initial state, when it should show all nine. This was just to make the graphic an appropriate size to get the idea behind game trees.

2.3.3 Minimax

Minimax is an algorithm for making correct move choices in a board game. Given a complete game tree and knowledge of what states result in a win or a loss, minimax would be able to make the best possible move for either player at each game state along the game tree[24].

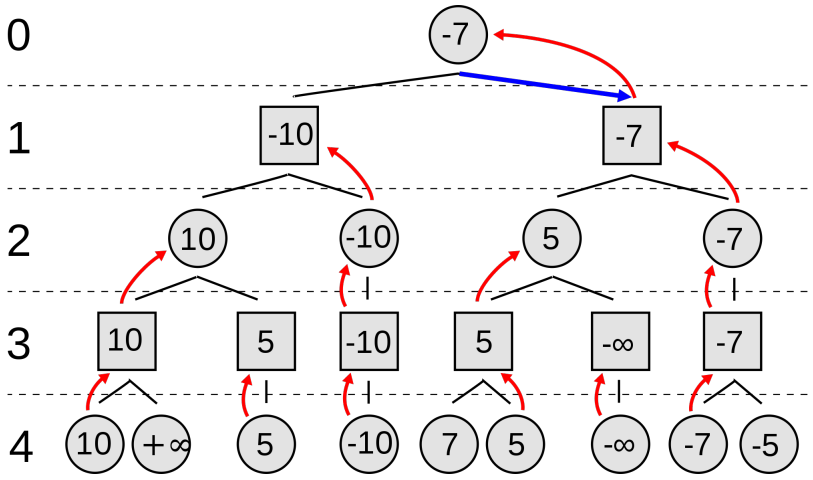

Figure 4: Image of a game tree with minimax values and choices overlaid[15].

The figure above illustrates the way that minimax makes decisions. The assumptions that minimax makes are that the first player to make a move is attempting to maximize their score at the end of the game and the other player is attempting to minimize the first player's score at the end of the game.

With these assumptions, minimax will recursively go to the leaf nodes of the game tree and ascertain values for the game state at those leaf nodes. Because at the end of all board games it is unambiguous who has won, these values completely accurately depict which player won the game. Then, minimax goes up

the game tree and at each state it chooses the appropriate move based on whether the current player is attempting to maximize or minimize final utility.

This can lead to somewhat counter-intuitive choices. For example, in the figure above the first move that the maximizing player chooses is the -7 move. However, the move with -10 has a possibility for resulting in a leaf node of $+\infty$. A greedy player would be more likely to try to get this very positive outcome, but that play would actually result in a utility of -10, rather than $+\infty$. This is because the minimizing player would avoid letting the $+\infty$ be played.

```
function MINIMAX-DECISION(state) returns an action
  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s))
  return v
```

Figure 5: Pseudocode for the minimax algorithm[24].

Above is the pseudocode for the minimax algorithm. It defines a function $Minimax\text{-}Decision$ which takes in a game state and returns an action. For chess, the game state is a chess board with pieces arranged in a legal configuration and the action is a resulting move from that game state. This move will be either for white or for black, so it is important to specify one of the players of the game as the maximizer and one as the minimizer.

2.3.4 Static Evaluators

Static evaluators are functions which are given a game state and return the utility of the state for a given player.

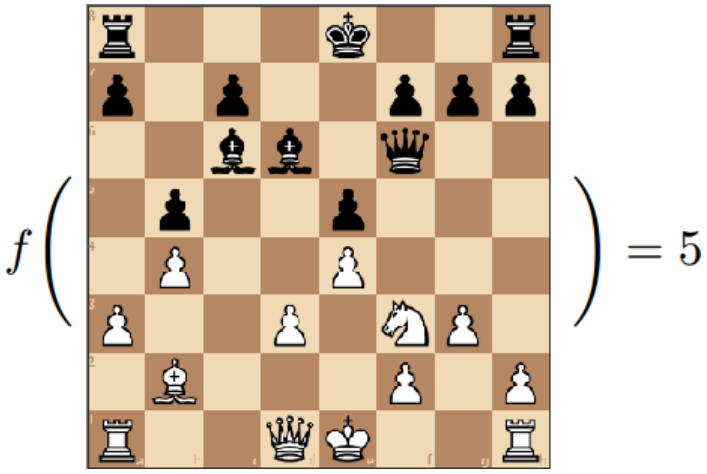

Figure 6: Static evaluators are functions taking in game states and returning numbers[10].

The above figure shows the idea that a static evaluator takes in a game state and returns a number value for how strong that state is for a given player.

The purpose of a static evaluator is to make it so that, rather than needing a complete game tree to function, minimax can work to any depth into a game tree. This is because, rather than having minimax recurse to the leaf nodes of a game tree it can be modified slightly to return the utility of the static evaluator once the desired depth has been reached by minimax.

This modification makes it so that the static evaluator is the most important factor in determining the strength of a game engine. This is because, if an engine is using a static evaluator which does not return accurate utility values, it will tend towards making moves which are more likely to result in the engine's loss than if they had a more accurate depiction of the strength of each state.

Static evaluators can work in any way, as long as they take in a game state and which player to evaluate for as input and return a number as output. However, there are common ways to structure static evaluators and the one which I will be looking at and using is a feature vector with weight values. This means that the

static evaluator will have a number of features that it looks at. In chess these features would be things like "number of queens I have" or "number of pawns my opponent has" or "number of rooks I have that are on the same vertical line". There can be as many or as few features as wanted, depending on how complicated one wants their static evaluator to be. These features are then associated with weight values. For example, we may weigh each queen that the opponent has as a -5.3 utility to the game state or we may weigh each pawn we have as being worth $+1.3$ utility to the game state.

2.4 Artificial Evolution

Artificial evolution is a way of searching through a space of possible solutions to a given problem which mimics the biological process of evolution. It is meant to gradually find more and more optimal solutions to a problem as it runs.

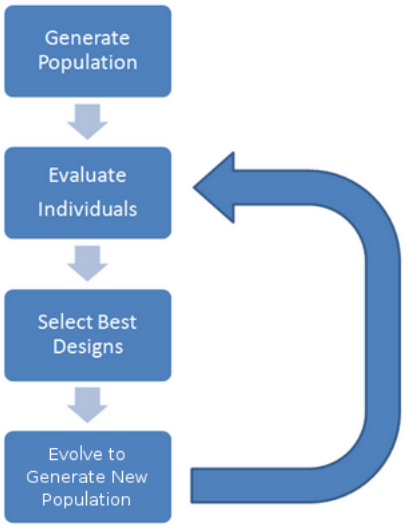

Figure 7: Overview of the evolutionary process[2].

The above figure shows the idea behind artificial evolution. When we are evolving models, the population is made up of static evaluators and the way that the individuals are evaluated and the best designs are chosen is through a fitness function. The fitness function of any evolutionary algorithm takes in a proposed design and returns a number value related to how good that design is. Our fitness function will be

determining how close a given static evaluator is to modeling an opponent by representing how accurately it can predict moves from the opponent which we have already seen.

The final step, evolving a new population, can be done in any way which produces a new set of static evaluators to be evaluated. However, there are a few common ways to evolve. Uniform crossover takes the best designs and then "breeds" them with each other by taking a random number of weight values of the feature vector from each parent. This produces offspring which have some weights from each parent and therefore, should act similarly but not completely the same as either parent.

Another part of this form of artificial evolution can be including mutation after crossing over. This would be that, for each of the weight values of the new offspring there is a certain, usually small, percentage chance that the given weight value would have a randomly chosen small value either added to it or subtracted from it. This can be useful for encouraging diversity among a population so that many different solutions are found rather than a few, very similar solutions.

The evolutionary algorithms can be any level of sophistication. More complex fitness functions and more complex crossover functions, like multi-point crossover could prove to be more effective methods of evolution than simple ones.

2.5 Estimation Exploration Algorithm

The systems identification method I will be employing to perform opponent modeling is called the estimation exploration algorithm. It is an algorithm developed in "Nonlinear System Identification using Coevolution of Models and Tests"[5]. The basic outline of the algorithm is as follows[5].

1. **Characterization of the target system** - In this step, a representation of the target hidden system will be defined. A representation for the inputs to the system and the outputs from the system will also be defined.

2. **Initialization** - Either an existing initialization set for the representations mentioned in step 1 will be used followed by going to step 3 or a random set of representations will be created and tested once, storing the results then going to step 4.

3. **Exploration** - First, a random set of tests to run will be generated. That is to say, a set of the representations for tests from step 1 will be created randomly. Then these representations will be evolved based on the best model(s) so far and tested for fitness, storing the results. Fitness will be determined by how much a particular test creates disagreement among the given set of models.

4. **Estimation** - The existing models of the system will be evolved, encouraging diversity among them. The fitness of a particular model is defined by its ability to accurately model all previous sets of inputs and outputs from the system. The best model out of the set of models will be sent to step 6, and N of the best models will be sent to step 3 to continue evolution.

5. **Termination** - The algorithm will continue until a sufficiently accurate model of the hidden system is found. If no good models or tests are found, the representation may be flawed, the hidden system may not return sufficient information in its output or the search space may be too large.

6. **Validation** - The given model will be tested using previously unused input to determine its accuracy.

This algorithm is relatively simple. From a beginning set of models of the hidden system, the algorithm will send an input to the system and observe the output. Based on this output, the algorithm will determine how accurate each model is. Then the algorithm will choose the best of the current models and evolve it so that the next input we send to the hidden system will cause the highest level of disagreement among the models. This disagreement is crucial, as it maximizes the amount of information that we get out of one test. This makes the models evolve much more rapidly than without tests that maximize disagreement among them.

The estimation exploration algorithm has been shown to be a very powerful algorithm for systems identification. One of the works that used this algorithm was able to make a robot that was resilient to damage[6]. The robot maintained a model of itself and learned to move forward from that model. Then, if the robot became damaged, all that it had to do was update its internal model of itself to relearn how to move forward. Another example of this algorithm in use is in "Distilling Free-form Natural Laws from Experimental Data"[26]. In this paper, Schmidt et al. showed that using only data from a pendulum swinging and the estimation exploration algorithm, they could derive Lagrangian and Hamiltonian laws of physics

as well as other laws about conservation of energy.

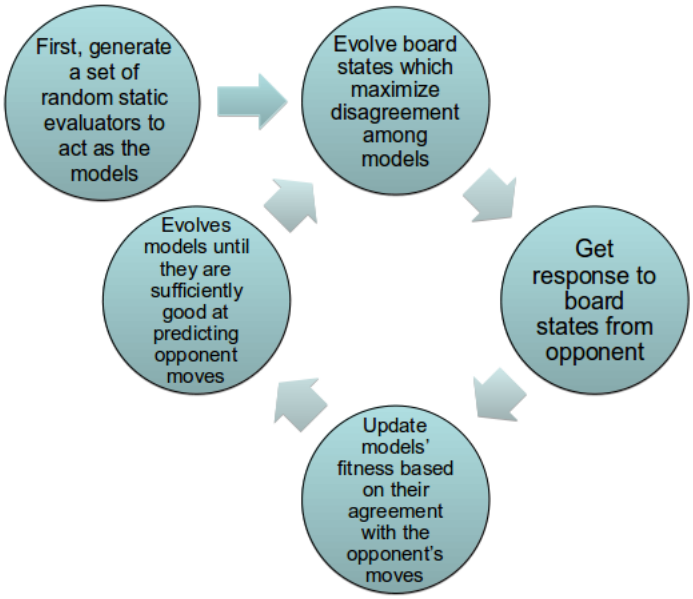

Figure 8: Figure to show how I will use the EEA to evolve a model of an opponent in a board game.

In the context of board games, the estimation exploration algorithm will be used to model an opponent. The models will be static evaluators with varying weights for a constant feature vector. In other words, the weight values that the static evaluators use will be evolving.

The tests will be sets of game states that are presented to the opponent to get a resulting move. These tests will be evolved to maximize disagreement among the models.

2.6 Konane

The game I've chosen to work with is konane. It is a game similar to checkers and is often called Hawaiian Checkers. The rules of konane are as follows[27][1]:

1. The game begins with all the pieces on the board (or table, ground, etc.) arranged in an alternating pattern.

2. Players decide which colors to play (black or white).

3. Black traditionally starts first, and must remove one of its pieces from the middle of the board, or from one of the four corners of the board. There are 4 pieces (2 black and 2 white diagonally opposite each other) that form a $2x2$ square array in the middle of the board. Black can either remove one of those two black pieces, or remove a black piece from one of the four corners of the board. The four corners of the board will also consist of two black pieces and two white pieces that are diagonally opposite from each other.

4. White then removes one of its pieces orthogonally adjacent to the empty space created by Black. There are now two orthogonally adjacent empty spaces on the board.

5. From here on, players take turns capturing each other's pieces. All moves must be capturing moves. A player captures an enemy piece by hopping over it with their own piece similar to draughts. However, unlike draughts, captures can be done only orthogonally and not diagonally. The player's piece hops over the orthogonally adjacent enemy piece, and lands on a vacant space immediately beyond. The player's piece can continue to hop over enemy pieces but only in the same orthogonal direction. The player can stop hopping enemy pieces at any time, but must at least capture one enemy piece in a turn. After the piece has stopped hopping, the player's turn ends. Only one piece may be used in a turn to capture enemy pieces.

6. The player unable to make a capture is the loser; his opponent is the winner.

Figure 9: Konane board of a game in progress to help illustrate the game[18].

The major advantage to using this game is that it is easy to implement so it will be used as a solid proof of concept that the estimation exploration algorithm can be used to model opponents in a board game. These concepts will be able to be generalized to any board game.

3 Implementation

I implemented the entire system that I had planned out early in the project. In addition, I wrote a number of programs that help with interpreting the data that the system generates. All source code is available for viewing and modification at `https://github.com/okiyama/EEA-Opponent-Modeling`.

Below is a timeline overview of what I implemented for my thesis in rough chronological order.

I began with Professor Rieffel's konane starter code, I built a working game of konane that has minimax and alpha beta pruning. I didn't mention alpha beta pruning previously because it is essentially just an implementation detail and all it does is make minimax work more quickly, without making it any less accurate.

After the engine was in place, I built the models portion of the estimation exploration algorithm by having static evaluators which can be substituted in easily and which can also be evolved. I previously discussed how I represent static evaluators. The evolutionary techniques I chose to employ were uniform crossover and mutation, as previously discussed.

With models and a working engine in place, I tried to familiarize myself with evolutionary algorithms a little bit by writing code to make a static evaluator hill climb to try to maximize its win percentage versus an unchanging opponent. Hill climbing is not technically an evolutionary algorithm because it does not use crossover or mutation. However, it did prove useful as the structure of hill climbing is extremely similar to evolutionary algorithms. Hill climbing is essentially the same as an evolutionary algorithm except rather than generating a new population through crossover, a hill climbing algorithm simply chooses the most fit individual and generates a new population by mutating that individual to make children.

I knew that I was going to need to be able to create tests for an opponent before I could do much more in the EEA so I built a system for generating random board states. This was important because I was going to need to be able to rapidly retrieve random board states for when I was generating tests for the models

to work against. The solution I chose was to have random players play against each other and record each board state that occurred during their gameplay to disk, for later use. This meant I could undergo a very time intensive step of generating thousands of random legal board states one time only then retrieve states very quickly by doing a simple list lookup.

Because I had board states at my disposal, I was able to start testing opponents for their responses. This ended up being much less straightforward than I thought initially because I knew that in the near future, I would need to make these tests be able to evolve. Because of this, I built a somewhat complicated class structure to differentiate between an individual test, which is a board state with a response from the opponent, a set of tests which is what we evolve to maximize disagreement and a suite of sets of tests which will be used to evolve the sets against each other. The end result was a system that could later be extended to allow for sets of tests that could evolve. At the time it was only capable of creating purely random tests and presenting those to an opponent though.

Then I changed my hill climbing so that it evolved to match moves that an opponent made, rather than trying to evolve to maximize win percentage.

After that I moved back to the tests and made them evolve to maximize disagreement among the models. This proved somewhat difficult as the disagreement metric was not very obvious to come up with. The metric I ended up going with was for every board state in the given test, give it a +1 disagreement score for each time two models did not agree on what move to make for the current board state. After that score was found, I normalized by the number of tests * the number of models, to make disagreement vary between 0 and 1.

Finally, after having tests which evolve to maximize the disagreement of a set of models and having models which evolve to correctly predict those tests, I had a fully working EEA. This luckily happened relatively early in the project which left me a fair bit of time to improve on certain aspects as well as spend time collecting and working with the data.

The extra programs I wrote were a fairly robust figure generator as well as a program written to analyze the strength of the models which I discovered. The figure generator was built slowly throughout the entire project and had to be modified frequently to match up with the changing formats of the data files I was

using. As the project grew, I was storing more and more data per line in my data files which required changing the graphing code. The model analyzer program is fairly straightforward. It loads up a data file of models and finds the most appropriate ones then tests them out against the same opponent as before to see how accurate they are on a large scale. It is worth noting that this requires getting thousands of moves from the opponent, so this program is only feasible for use with a computer opponent.

All throughout the process of implementation I was continually testing and collecting data. I have multiple gigabytes of raw CSV data that I've generated throughout the project. Most of this data was generated thanks to the Union College supercomputing cluster, which I was graciously given access to. This saved me from leaving my laptop on 24/7 and it also made it much easier to get large amounts of data.

In addition to this, I have spent a very large amount of time learning the related concepts that I've explained in this paper as well as reading many related papers. A lot of the hours I put into this thesis didn't end up resulting in direct, measurable output but instead in teaching me about the subject material so that I have the ability to output code that will get me closer to my goals.

4 Results

My results were quite promising. The majority of my testing was playing against my own game engine. This was largely because it was custom built from the ground up to be easy to test against, so any other system would necessarily be more difficult to work with. That being said, I did try having my system try to evolve playing against myself as well as a program that I found online. The interesting thing about the program I found online is that it happened to use the exact same API as my program, so it was easy to test against.

My system got so much data out largely due to being able to run many instances of the program on the Union College super computer. While I was running my tests, I filled up many of the nodes on the cluster which made the process monitor of the cluster look like this:

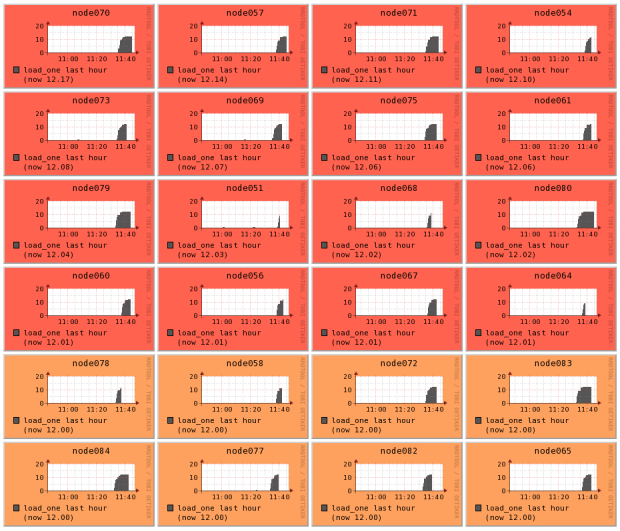

Figure 10: Maxing out the processing power of nodes on the cluster

That's over 100 cores all being used simultaneously for data generation. The only other machine I had to use for data generation was my laptop, with only 4 cores. I would have never been able to do some of the data generation and interpretation that I did without the resource of the cluster at my fingertips.

The following figures are generated by running my system with varying depths for the models and the opponents. I found that with the depths were the same, the system almost always succeeded at finding a good model. However, even minor differences could result in not arriving at good models.

For all of these figures, fitness for each model is defined as:

fitness = (number of tests this model correctly predicts * 100) + (diversity score of this model * 0.0001)

The reason for these weight values was to guarantee that a more accurate model will always win over a less accurate model while having diversity act as a tie breaker between equally accurate models. The jumps in fitness that you see are when a new round of the EEA begins. The models continuously evolve until there are enough individuals in the population that are fully accurate at predicting the current set of tests to evolve a new population. These jumps are because new rounds introduce more tests for the models to accurately predict, which makes their fitness jump. This jump makes sense intuitively because it means that the models are accurately solving a more difficult challenge in being accurate against more

tests. Additionally, each figure represents exactly two hours of evolution.

Figure 11: A typical “good” result

The above figure shows what the fitness of the models looks like in a ”good“ run of the evolution. Fitness raises gradually and by the end, 16 tests are accurately predicted.

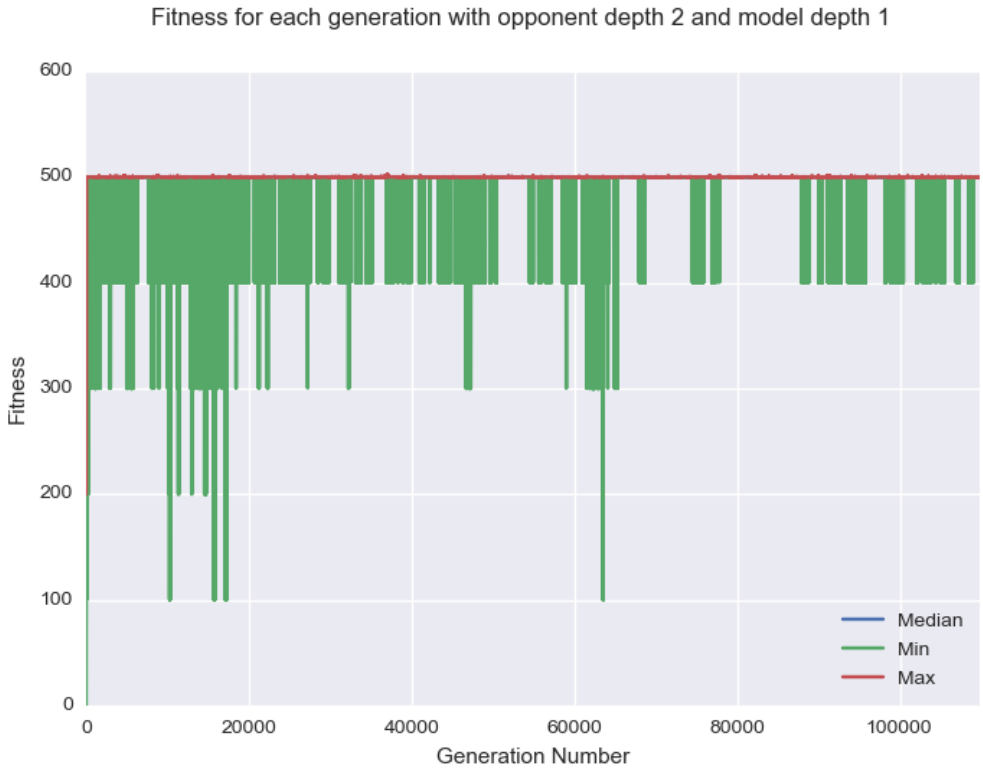

Figure 12: A typical "bad" result

The above figure shows what a "bad" run of the evolution looks like. The evolution gets stuck and doesn't ever accurately predict more than a few tests. This sort of pathology was one that I found throughout all of my testing. I was able to dramatically decrease how often this happened by adjusting parameters of the evolution like how often mutation would occur and how much a mutation would change the model. This was useful for breaking out of being stuck because it forced diversity on the models. This made them search in previously unexplored parts of the problem space.

		Opponent Depth			
		1	2	3	4
Model Depth	1	63.8%	43.4%	44.6%	46.2%
	2	41.1%	76.2%	66.5%	49.2%
	3	56.9%	58.5%	43.8%	60.8%
	4	34.9%	40.8%	42.3%	83.1%

The above table shows the percentage accuracy of the best model that was found through two hours of evolution. The model depth on the left shows the depth that the minimax player was using when trying to model the opponent and the opponent depth indicates the depth the opponent was using for their minimax player. This percent accuracy was found by choosing thousands of random board positions and seeing if the moves that the opponent makes agree with the moves that the models make. It's worth noting that because this method requires providing moves for thousands of board positions, it is not a very realistic test to use with a human player and only works well with opponents that are programs willing to play through thousands of board states.

The data shows that no best model found was worse than 34.9% accuracy. We also find that the average accuracy of the data is 53%. These numbers sound fairly low, but the accuracy of a model making purely random moves is **10.7%**. I found the percentage for a random player empirically and it is directly related to how my program works, so it is fair to compare the results above versus this benchmark number for a random model. We can see then that even our worst models perform significantly better than random, which is a good start for any evolutionary algorithm but also that we sometimes reach quite high percentage accuracies. This shows that in all cases, at the very least there is some level of modeling that is successfully occurring. In cases where the settings for the opponent and the model are similar, the modeling can be quite accurate. This points to two possibilities, either that my particular implementation is flawed or the approach has limits in and of itself. My implementation is admittedly simplistic in many regards, particularly the evolutionary processes which could potentially cap the modeling capabilities of the system. However, it is also possible that there is simply limits to how well a static evaluator model can model an arbitrary opponent's play.

When evolving versus the opponent that I found online, which I previously mentioned, my system found a model with a 42.8% accuracy rating. I have no knowledge of how this program chose to play, which is quite promising as it shows that even using significantly different programs, it is still possible to get a reasonably accurate model of an opponent.

I did some preliminary testing with modeling a human player, but unfortunately I was the only person that I tested. From this testing my system did not find any promising models. I suspect this is because my system is based entirely upon building a model which is trying to win the game. My system explicitly cannot model an opponent who is actively trying to lose or who has an arbitrary strategy like "make the first legal move you see". I feel that my knowledge of konane was so weak that I did not have an internal winning strategy that I was employing in any sort of meaningful way. It would be very interesting to test my system against a konane veteran, but I did not have any on hand while working on this project.

5 Conclusion

The goal of the project was to model opponents in board games using the estimation exploration algorithm. I had set out a plan to implement a program which used konane as an example board game to demonstrate the strength of the estimation exploration algorithm for modeling opponents. From there, I implemented the system that I had planned on making in its entirety. After the system was in place, I began gathering data and interpreting it. The results I found were quite promising to indicate that the EEA works to model opponents in some capacity.

Future work would most notably include making the evolutionary process of the system more sophisticated. This could allow for the system to find even more accurate models.

Another direction to go from here would be to find and create more opponents to test against. My test suite was largely made up of models and opponents using mostly the same code. It would be good to test against a wider suite of opponents as well as human opponents.

It would also be worthwhile to implement this work in other games like chess to see how much the results compare from one game to another.

References

[1] Lilinoe Andrews et al. *The Hawaiians of Old*. Bess Press, 2002.

[2] Ben Berger. "Evolution Overview". Image made by Ben Berger, modified and used with permission.

[3] Darse Billings et al. "Opponent modeling in poker". In: *AAAI/IAAI*. 1998, pp. 493–499.

[4] Marco Block et al. "Using reinforcement learning in chess engines". In: *Research in Computing Science* 35 (2008), pp. 31–40.

[5] Josh C Bongard and Hod Lipson. "Nonlinear system identification using coevolution of models and tests". In: *Evolutionary Computation, IEEE Transactions on* 9.4 (2005), pp. 361–384.

[6] Josh Bongard, Victor Zykov, and Hod Lipson. "Resilient machines through continuous self-modeling". In: *Science* 314.5802 (2006), pp. 1118–1121.

[7] Michael Bowling et al. "A demonstration of the Polaris poker system". In: *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. International Foundation for Autonomous Agents and Multiagent Systems. 2009, pp. 1391–1392.

[8] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. "Deep blue". In: *Artificial intelligence* 134.1 (2002), pp. 57–83.

[9] Stanford CS221. *Deep Blue*. URL: http://garabedyan.files.wordpress.com/2011/04/chess-shannon-type-a.png.

[10] Stanford CS221. *Deep Blue*. URL: http://stanford.edu/~cpiech/cs221/img/evaluation.png.

[11] Aaron Davidson et al. "Improved opponent modeling in poker". In: *International Conference on Artificial Intelligence*. 2000, pp. 1467–1473.

[12] David B Fogel et al. "A self-learning evolutionary chess program". In: *Proceedings of the IEEE* 92.12 (2004), pp. 1947–1954.

[13] Wikimedia Foundation. *Chess theory*. URL: http://upload.wikimedia.org/wikipedia/commons/b/b5/Starting_position_in_a_chess_game.jpg.

[14] Wikimedia Foundation. *Game Tree*. URL: http://upload.wikimedia.org/wikipedia/commons/thumb/d/da/Tic-tac-toe-game-tree.svg/2000px-Tic-tac-toe-game-tree.svg.png.

[15] Wikimedia Foundation. *Minimax*. URL: http://upload.wikimedia.org/wikipedia/commons/thumb/6/6f/Minimax.svg/2000px-Minimax.svg.png.

[16] Michael Genesereth, Nathaniel Love, and Barney Pell. "General game playing: Overview of the AAAI competition". In: *AI magazine* 26.2 (2005), p. 62.

[17] Chess Guru. *The pawn*. URL: http://www.chessguru.net/chess_rules/chess_piece/pawn/position.png.

[18] Ian Henry. *Konane*. URL: http://tabtop.blogspot.com/2009/04/konane.html.

[19] Andreas Junghanns. "Are There Practical Alternatives To Alpha-Beta in Computer Chess?" In: *ICCA J* 21 (1998), pp. 14–32.

[20] Graham Kendall and Glenn Whitwell. "An evolutionary approach for the tuning of a chess evaluation function using population dynamics". In: *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*. Vol. 2. IEEE. 2001, pp. 995–1002.

[21] Lennart Ljung. *System identification*. Springer, 1998.

[22] *Minimax Principle*. http://www.encyclopediaofmath.org/index.php?title=Minimax_principle&oldid=28249. Accessed May 29th 2014.

[23] Hikaru Nakamura. *HUSKIES,USCL,UPDATES,ETC*. 2011. URL: http://hikarunakamura.com/wp-content/uploads/2011/12/Barcenilla1.jpg.

[24] Stuart Russell, Peter Norvig, and Artificial Intelligence. "A modern approach". In: *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* 25 (1995).

[25] Frederik Schadd, Sander Bakkes, and Pieter Spronck. "Opponent Modeling in Real-Time Strategy Games." In: *GAMEON*. 2007, pp. 61–70.

[26] Michael Schmidt and Hod Lipson. "Distilling free-form natural laws from experimental data". In: *science* 324.5923 (2009), pp. 81–85.

[27] Helaine Selin and Ubiratan D'Ambrosio. *Mathematics across cultures: The history of non-Western mathematics*. Vol. 2. Springer, 2001.

[28] Jonas Sjöberg et al. "Nonlinear black-box modeling in system identification: a unified overview". In: *Automatica* 31.12 (1995), pp. 1691–1724.

[29] Paul Thagard. "Adversarial problem solving: Modeling an opponent using explanatory coherence". In: *Cognitive Science* 16.1 (1992), pp. 123–149.

[30] Andrew Tridgell. "KnightCap-a parallel chess program on the AP1000". In: *Proceedings of the Seventh Fujitsu Parallel Computing Workshop*. 1997.