Senior Project (draft)
Lafayette College
Department of Mathematics

Wah Loon Keng[*]

November 3, 2015

Abstract

We conduct a survey study on machine learning, its mathematical foundations, general techniques, and the latest progress in both the academia and the industry. However, most machine learners excel only at the task they are trained for - this is a huge limitation. Motivated by this, we study a new paradigm called the Never-Ending Language Learning (NELL)[1] and its potentials to overcome the singular nature of machine learners. Lastly, we design and implement a system based on NELL to predict the stock market performance. This is driven by the abundance of data, and the complex, interacting, non-singular nature of the task that a machine learner has yet to match a human on.

[*]Lafayette College, Easton, PA 18042, USA. kengw@lafayette.edu.

Contents

1 Introduction

We have just entered the golden age of machine learning. The field has in fact been around for decades, born from artificial intelligence and pattern recognition. It wasn't until recently that we have the level of computational power and the abundance of data to apply it.

The machine learners have accomplished some truly astonishing feats. They are systems that are trained with a copious amount of data, which allows them to surpass the performance level of human experts. Today, they are actively being researched and deployed by the biggest tech companies such as Google, IBM, Microsoft, Facebook. In fact, their major products are powered by machine learning - Google Adsense, IBM Watson, and Facebook's M - doing tasks such as ads suggestion, image recognition, medical diagnosis, and natural language processing.

Typically, machine learning is best suited for the tasks that cannot be solved precisely or efficiently by algorithms. Many of these are what humans excel at - image recognition, language processing, pattern deduction. These capabilities are so trivial to us, yet they are so difficult to mimic by traditional algorithmic approaches. The prime example of *"a child can easily recognize a cat in a picture, but a super computer can't"* best illustrates the failure of the algorithmic methods. Machine learning comes in to save the day.

In a nutshell, one can think of a it as a machine that "learns" from the data. On the implementation level, it is an automated regression software that can construct an impressively accurate model. The machine takes in a training set - input data where each entry is labeled with the intended output, then trains on it and learns to recognize the pattern. After that, when fed with new unlabeled input, it gives some predicted output based on its training. However, just like a typical regression model, its scope is strictly bounded to its training set. For example, a machine that recognizes cats cannot recognize a fish without extra training. Moreover, an image recognizer cannot diagnose diseases. In this sense a machine learner's scope is **singular**. We will address this issue later.

The mathematical foundations of machine learning is built atop regression theory and linear algebra. Next, we present the formalization of a problem in machine learning as well as the terminologies.

1.1 Formalization

The following formalization is standard in the literature, we reference Tom M. Mitchell's popular textbook *Machine Learning*[2].

Definition 1. *A computer program is said to* ***learn*** *from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*

We will use the terms *training* and *learning* interchangeably. Next, we define a **problem**.

Definition 2. *A* ***problem*** *is a triple* $\{T, P, E\}$*. The goal of a machine learner solving the problem is to output a learned target function* $\hat{V}$ *that approximates an ideal target function* V*. The target function maps from the vector of input features* b *to the set of output* O*.*

The feature vector b, output O, and the function representation for $\hat{V}$ are defined by the programmer as deemed suitable. In a simple regression model, b is a list of chosen input features, O is the set of predicted values. Typically the target function $\hat{V}$ will include a set of weights $W = \{w_i \mid i \in \{1, 2, \cdots, n\}\}$ which are tuned during the learning. They serve as the "memory" of the machine to compute predicted output once it is trained.

Upon having an explicit representation, and given the training data, we can devise the learning mechanism. First, define an error term E between the training data and the actual machine output. Then, use the error term to devise an algorithm that tunes the weights iteratively to minimize the error. The algorithm terminates on meeting a minimum threshold error, that is when the machine can perform sufficiently good.

Example 1. *We provide an explicit example with the checkers game:*

The checkers learning problem:

1. *Task* T*: playing checkers*
2. *Performance measure* P*: percent of games won*
3. *Training experience* E*: games played against another computer*

Next we need to determine the representations of the available knowledge. For the input feature vector b*, we can choose a few board states, for instance,*

$$b =< x_1, x_2, x_3, x_4 >$$

where the x_i*'s are respectively the numbers of black pieces, black kings, red pieces and red kings on the board. We can choose a simple linear combination as our target function,*

$$\hat{V}(b) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4$$

where w_i*'s are the weights to be tuned during the training. Our output set* O *is the list of moves at each game step given the features* x_i*'s at the time. We can define a bijection between the output range of* $\hat{V}(b)$ *and the* O*, but we will exclude it here.*

Once we have a representation, devise a learning mechanism for training. Suppose now we have a set of positive (game won) training examples $\{< b, V_{train}(b) >\}$*. Define the error term using simple sum of squares*

$$E = \sum_{<b, V_{train}(b)> \in \text{ training examples}} (V_{train}(b) - \hat{V}_{train}(b))^2$$

Then, using this, devise an algorithm for training / weight-tuning. The error term above, when differentiated, yields gradient term of $\sim (V_{train}(b) - \hat{V}_{train}(b))$. *One can imagine training as nodging a point closer to its true value, and the gradient gives the direction to nodge towards. This gives us a simple LMS algorithm for minimizing E using that gradient term:*

Algorithm **LMS weight update (learning) algorithm**

Initialize the weights w_i*'s to random values. For each training example* $< b, V_{train}(b) >$*:*

1. *Use the current weights to compute* $\hat{V}(b)$
2. *For each weight* w_i*, update it by*

$$w_i \leftarrow w_i + \eta \ (V_{train}(b) - \hat{V}_{train}(b)) \ x_i$$

where η *is the parameter that moderates the stepsize. The algorithm terminates when* w_i*'s stop changing, i.e. when the error becomes sufficiently small (zero).*

One can quickly see that the example machine learner above is just a simple regressor. Obviously this toy model will not be a good checker player. Nevertheless, given a huge training data, if we choose the right knowledge representation, feature vector b, target function $\hat{V}$, error term E, and the learning algorithm, the resultant machine can perform very well.

To suit different types of problems, we can employ different **representations**:

T	P	E	b	O
image recognition	correct labeling	labeled images	array of pixels	label of objects
driving a car	successful trip	test drives	traffic condition	wheel, pedals
text translation	correct translations	pretranslated texts	semantics	translated text
ads suggestion	ads click rate	ads metadata	user profile	ads displayed

Table 1: Different representations for various problems.

Moreover, we can employ different **learning mechanisms** - special data structures, target functions and learning algorithms. This gives rise to the different techniques, or types, of machine learners such as the Neural Nets, Support Vector Machines, Deep Learning and etc. We will investigate the popular types in the next section.

A huge variety of machine learners today are being used for different purposes - most of the leading tech companies tweak and design their own. They may have very different

representations and **learning mechanisms**, but the overall idea is still the same - gather labeled data, define the representations and learning algorithms, and train. The entire process is very modular, and the resultant machines are **stagnant** - the machine no longer improve once training is done; and **singular** - they can only function very narrowly only on the tasks they are trained for. For our discussions, we call this the **stagnant & singular paradigm**. In fact, this paradigm can readily be observed across most of the popular machine learners today.

1.2 Popular techniques

Damn son.

1.2.1 Decision Tree

1.2.2 Knowledge Base

1.2.3 ANN with Perceptron

1.2.4 ANN with Gradient Descent

1.2.5 convnet

1.2.6 SVM

1.2.7 Deep Learning

1.3 Applications

Google translate google ad sense google deep dream, deep mind Google deep learning (play game!) IBM watson

1.4 Paradigm: Powers & Limitations

2 NELL as a New Paradigm

2.1 Inspiration & Features

2.2 Architecture

2.3 Applications

2.4 Powers & Limitations

3 NELL for Stock Market Prediction

3.1 Problem statement

3.2 Potential of NELL

3.3 Design

3.4 Implementation

3.5 Assessment

4 Future Work

5 Formalization

The game is inherently dependent on the board, which is a world map of 42 countries, interconnected in specific ways. Decisions to attack or defend are based on the distribution of the armies, the surroundings of a location, and connectivity of countries. These motivate our formalizing the game board and algorithms based on an undirected graph. From now we shall refer to the graph representation as *map*:

Definition 3. *A **map** is a connected, undirected planar graph, with 42 nodes, each representing a country. The nodes are named with indices* $0-41$*, and are connected the same way as are countries on the game board by undirected edge of weight 1.*

We assign data fields to each node, namely its country name, the continent it is in, its player owner, the number of armies of the owner in it, its worth and pressure as determined by some metric described below.

Definition 4. *A **region** is a connected subgraph consisting of nodes all owned by the same player. Each player can own many regions, which together partition the map.*

Definition 5. *A **border** node of an AI is its node that is adjacent to at least an enemy node.*

Definition 6. *A **attackable** node for an AI an enemy node adjacent to its border.*

Definition 7. *The **shape** of a region is the measure of its shape/roundess. To compute the shape, find the maximum and minimum distances between the border nodes in the region, and shape=(max-min)/max. If a region is round, shape = 0; if it is a line, shape = 1.*

Definition 8. ***Radius** is the measure of shortest distance from an origin node. We identify the neighbors of node* $\mathcal{O}$ *at radius* k *to be the nodes whose shortest distance from* $\mathcal{O}$ *is* k*.*

Furthermore, we define the fields that will be useful in our algorithms:

Definition 9. *The **worth** of a node is the measure of its importance to an AI, as calculated by its internal metric algorithm, and is used by the AI to prioritize its decisions: which node should it defend/attack first.*

Definition 10. *The **pressure** of a node as perceived by an AI is the measure of the average army distribution around the node, up to 5 unit radii away. It is calculated by the AI's internal metric and used to prioritize decisions.*

Note that the worth and pressure of a node are not the same when calculated by opposing AIs due to different perceptions, metric and AI personalities. Each AI will be calculating these values for all 42 nodes at each turn.

Finally, we introduce a data structure as the raw representation of overall army distribution on the map for various calculations:

Definition 11. *The **Radius Matrix (RM)** from an origin node* $\mathcal{O}$ *is the matrix that better represents the connectivity of neighbor nodes of the origin within some radius. It is enumerated by the Radius Matrix Algorithm below, and each entry is the name of some node.*

*Its corresponding **Army Matrix (AM)** is a different representation of the RM, with each entry now being* $z \in \mathbb{Z}$*, where* $|z|$ *is the number of armies at the node, and* z *is positive if the node is owned by the calculating AI, and negative otherwise.*

6 Algorithms

We now enumerate the algorithms for each step of the game, which will collectively form the final algorithm used by the AI to play the game.

6.1 The Matrix Algorithms

Algorithm **Radius Matrix (RM) for an origin node $\mathcal{O}$**

Starting from an origin node $\mathcal{O}$, initialize an empty matrix for its RM,

1. Add the index of each adjacent node (at radius 1) of $\mathcal{O}$ to a new row in RM.
2. Repeat for $i \in \{2, 3, ..., n\}$, where n is the maximum radius covered:

   For each entry p at column i, get all n_p of its adjacent nodes at radius $i + 1$ from $\mathcal{O}$.
3. Duplicate the row of entry p while appending to it each of the n_p adjacent nodes at column $i + 1$. If $n_p = 0$, append "*empty*" instead. The process is akin to a Cartesian product.
4. Return the RM for $\mathcal{O}$.

Note that the column number will coincide with the radius from $\mathcal{O}$. The RM with n columns is a representation of the connectivity from the origin up to radius n, where each row is the shortest path from the origin to a point at radius n, and there may exist many such paths.

Algorithm **Army Matrix (AM) for an origin node $\mathcal{O}$**

We can convert an RM into AM, a representation using the number of armies,

1. Find the RM for node $\mathcal{O}$ using the RM algorithm.
2. For each entry p in RM, if node p has the same owner as $\mathcal{O}$, replace the entry with the number of army at p; else, replace with the negative of the number of army at p. If an entry p is "*empty*", append 0 instead.
3. return the AM for $\mathcal{O}$.

This transforms an RM into its alternate form AM, which gives a representation of the army distribution and connectivity around the origin node $\mathcal{O}$. This matrix can be used for calculating the **pressure** from definition 10. For our project we calculate the matrices up to radius 5, which we think is sufficient given that per game turn a player can only move adjacently among nodes.

6.2 The Pressure Algorithm

The pressure of each node from an AI's point of view is the average number of army surrounding the node. More positive pressure indicates the node is a better stronghold of the AI; more negative pressure indicates is surrounded by more enemies.

The calculation of pressure depends on the AI's perception of threat, which can be represented using a metric that varies based on its personality.

Definition 12. *The **threat perception** of an AI is the way it sees the threat of army distribution up to some radius away poses on an origin node. E.g. 10 enemy armies further away poses less threat than 5 enemy armies nearby. The **threat perception** is quantified by defining a metric: a normalized vector or length = max radius of AM, where the individual value of the vector is the weight multiplied to the army number at that radius. The procedure is describe below.*

Algorithm **The Metric Algorithm**

To enumerate the metric for an AI's threat perception, with scope radius = 5,

1. Choose a weight function, for example, constant, Gaussian,
2. Evaluate function values for with the input distance vector $\{1, 2, 3, 4, 5\}$
3. Renormalize the output vector and return it as the metric vector.

This metric vector $\mathbf{w}$ is then dotted with a row $\mathbf{r}$ in the AM, which is a list of number of armies at incremental distance away from an origin node $\mathcal{O}$, and the partial pressure PP for it is:

$$PP(\mathbf{r}) = \mathbf{w} \cdot \mathbf{r}$$

Algorithm **The Pressure Algorithm**

The AI calculates the pressure for each node using its personality trait **threat-perception**, or the metric vector $\mathbf{w}$:

1. Update the data fields of the map and call the AM algorithm to compute the AMs for all 42 nodes.
2. For each node $\mathcal{O}$, compute the dot product between $\mathbf{w}$ and each row of the node's AM; the result is a column vector $\mathbf{c}$.
3. The first column of the original RM is a repeated list of m adjacent nodes of $\mathcal{O}$, suppose each node i repeats q_i times in the column, so in total the column has length

$q_1 + q_2 + \cdots + q_m$. Renormalize this sequence into $nq_1 + nq_2 + \cdots + nq_m$ For each batch q_i of the column vector $\mathbf{c}$ from above, take its mean, then multiply by the renormalized weight nq_i. Then sum all m of the results, call this scalar $s(\mathcal{O})$.

4. Now that the column $\mathbf{c}$ has been reduced to a scalar representing the average army distribution around the origin $\mathcal{O}$, account for the number of armies (sign-sensitive, negative for enemy) here $a(\mathcal{O})$ by adding the scalar, and return the pressure of node $\mathcal{O}$, $P(\mathcal{O}) = s(\mathcal{O}) + a(\mathcal{O})$.

Thus at each turn, the AI updates the data fields and calculates the pressure, i.e. the average number of surrounding armies, for each node, using its threat perception metric.

6.3 The Worth Algorithm

Algorithm **The Worth Algorithm**

At each turn, the AI evaluates the worth of each node to prioritize its attacks and defenses. Suppose it considers m factors, each of which assumes a real positive value, with more positive being more worthy. To compute the final worth scalar, simply order the m factors from the most vital, and dot it with a factor vector $\{10^{m-1}, ..., 100, 10, 1\}$.

For our AI, we consider the following factors (ordered from the most important). For each node $\mathcal{O}$ calculate and append to the list of factors:

1. continent-fraction $= \frac{\text{(number of nodes with the same owner in the same continent } \mathcal{O}\text{)}}{\text{(total number of nodes in the continent)}}$
2. If $\mathcal{O}$ is own node, the region-index: Enumerate for each player its nodes, and group them by regions, then order them from the biggest to the smallest regions. The region-index of $\mathcal{O}$ is its index in this list. Or if $\mathcal{O}$ is enemy, the attackable index: of the region list enumerated above, extract the sublist with nodes that are attackable, i.e. is an enemy adjacent to one of your nodes. The attackable index of $\mathcal{O}$ is its index in this sublist; -1 otherwise.
3. shape: find the region $\mathcal{O}$ is in and compute the shape as in definition 7.
4. degree: the degree of $\mathcal{O}$, i.e. the number of adjacent nodes it has.
5. pressure: as calculated from the pressure algorithm.
6. Finally, return the dot product between this factor list and $\{10^4, 1000, 100, 10, 1\}$.

After obtaining an ordered list of worth nodes, we can partition it while preserving the order into lists of border nodes and attackable nodes, and reorder them based on strategies.

Furthermore, the AI makes it context-sensitive by remembering the pressures from the past turn, and reorder the list based on pressure-drop between turns.

To justify our factors above, conquering a whole continent gives a player extra armies per turn while strengthening the region. Furthermore, a larger region is harder to attack than a smaller region, thus we place more importance on the nodes there.

The shape of a region is vital for defense and army mobility during fortification. This is the classic problem of minimizing the surface/volume ratio, or in this lower dimensional case, the perimeter/surface ratio. Intuitively, a thin region is vulnerable, and has bad army mobility. A thick, concentric shape is stronger. The average distance between nodes is also shorter and thus aids mobility. Moreover, a node with higher degree improves mobility since it can reach many other nodes.

The pressure measures the ease of attacking or defending a node; the less negative a node is, the less enemy presence it has. It is wise to not attack the enemy's stronghold, but to seek its weak point of entry, which can be detected from a less-negative pressure.

6.4 The Priority Algorithm

At each game turn, the AI updates the priority nodes to attack/defend. The list of priority nodes depends on the AI's personality trait **priority**, whether it is agressive (attack-then-defend) or defensive (defend-then-attack).

Algorithm **The Priority Algorithm**

1. Update the data fields for the AI.
2. Call the pressure algorithm on the map.
3. Call the worth algorithm on the map.
4. Repartition the worth nodes and reorder by attackables/borders first based on the AI's personality, whether agressive or defensive. Furthermore, for the attackable, choose the best origin of attack by the highest pressure.

6.5 The Placement Algorithm

This describes how the AI places the armies into the its priority nodes based on its personality trait **placement**.

Algorithm **The Placement Algorithm**

1. If the trait is **cautious**, place armies along its priority nodes (if is enemy, use the best origin of attack) until all pressures are > 0, then with the extra armies, place 4 each down the same list; repeat until none left.
2. If the trait is **tactical**, place armies down the list until node pressure is > 4, then with extra, place 4 each down the list; repeat until none left.

6.6 The Attack Algorithm

The AI decides to launch attacks from the best attack origin (calculate with in priority list) based on its personality trait **attack**.

Algorithm **The Attack Algorithm**

For all attackables down the priority list,

1. If the trait is **rusher**, the AI harassess constantly, i.e. while the best attack origin has 2 more armies than the enemy target, keep attacking before moving to next target.
2. If the trait is **carry**, same as above, but the difference threshold is 4 (higher). Furthermore, the AI will accumulate the cards to reserve more armies for late game.

6.7 The Fortifying Algorithm

All AIs use the same fortifying algorithm.

Algorithm **The Fortifying Algorithm**

1. Find the border node $\mathcal{O}$ with the lowest pressure, and find a non-border ally node with higher pressure, transfer all but 1 troop to the border node if possible. This is to always push the unused central forces out to the borders where armies are mostly needed.
2. If no fortification done above, find a border node with the highest pressure, and transfer any neighboring armies (all but 1) to it. This is for the accumulation of armies during late game by making strong node even stronger.

This algorithm aims to create a center-weak border-strong army distribution within a region with, that is to utilize the maximum number of armies by putting them to the border nodes. Such distribution is efficient for both offense and defense. The second part of the algorithm kicks in when there is less border nodes during the late game, when one player controls a larger region. This will accumulate all armies toward a border node to overwhelm the enemy, hopefully ending the game quicker.

7 AI Algorithms and Personalities

We now put everything together to form the AI, which has four personality traits parametrized in its algorithms:

1. The **threat perception** trait /metric in the Pressure Algorithm; function variations: {Constant, Survival}.
2. The **priority** trait in the Priority Algorithm; variations: {agressive, defensive}.
3. The **placement** trait in the Placement Algorithm; variations: {cautious, tactical}.
4. The **attack** trait in the Attack Algorithm; variations: {rusher, carry}.

Thus there are $2^4 = 16$ AI personalities, and more if we allow richer variations.

Corresponding to the game moves per turn:

1. Getting and placing new armies;
2. Attacking, if you choose to, by rolling the dice;
3. Fortifying your position (moving troops between an adjacent pair of your nodes),

the AI has these primary methods:

1. Update: Call the Priority and Worth algorithms.
2. Get-and-Place-Armies: Call the Placement Algorithm.
3. Attack: Call the Attack Algorithm as many times as wanted.
4. Fortify: Call the Fortify Algorithm.

A game can have as many participating AIs as permitted by the rules. During the initial game setup, countries are randomly assigned to the AIs. Then, the AIs take turn to call their Update and Get-and-Place-Armies methods to complete the setup. Then the game begins and the AIs take turn to call all their primary methods in sequence, until the game terminates, or ties at the maximum number of rounds.

As opposed to physical game, the virtual game has no limit on the number of army pieces − it just keeps creating more as needed; nor it has the limit on the cards − it keeps reshuffling a new deck once an old one runs out.

Next we present the results from our implementation.

8 Implementation Results

8.1 Formalization of Problem

Our initial approach to the problem is one that focuses on the gross number of countries and armies an AI possesses throughout the game. As the game progresses, we monitor how the AIs interact in their attempt to win both armies and territories based on their personalities. From this, conclusions were drawn regarding personality types, player order effect, as well as game starting conditions.

8.2 Algorithms and Decisions

8.2.1 Initial Concerns

An initial issue was normalization of the time axis across several games. While games were initially capped at 100 rounds, where the game was declared a draw, the games that did not end in a draw had variable game length. The majority of the data analysis done in this project was done where there was a definitive winner. From this, the time axis was normalized by considering games in terms of percentage of game length. To the AI the length of the game is unknown during gameplay. However, the analysis can show where pivotal points occur across the game. While initially, players will be unable to use this information an understanding of both territorial and army control will influence their decisions.

8.2.2 The Difference Plots

In this analysis, the percentage of both armies and countries in total will be compared to the percentage of the game in where these points occur. These points for players *p1, p2* will be found using the following

$$diff\_army = \frac{p1\_army \text{ - } p2\_army}{p1\_army \text{ - } p2\_army}$$

$$diff\_country = \frac{p1\_country \text{ - } p2\_country}{p1\_country \text{ - } p2\_country}$$

This data is stored in a matrix that holds both army numbers and round numbers. In this matrix, data from games resulting in draws is discounted. The equation will hold for any winner. From this we expect a divergent trend from zero as the game proceeds to an end. From this matrix the difference in both armies and countries are calculated and plotted in R. From this, R will produce a trend line. This trend line will provide information for both the initial game play configuration and the development of the game.

8.2.3 The Fun Function

In this analysis a fun function was built to hold and later display some basic analysis of the games played. The fun function includes, total wins of both AIs, probability of winning, odds of winning, the average total armies, the average total armies during a win, the maximum number of armies needed to win , the minimum number of armies needed to win, average turns, and average turns without ties. When a game play data set is passed from the JSON through the rjson package, the function returns this summary of game play.

Imbedded in this function is the survival plot of two AIs. The survival package in R allows from the plotting of the survival of both armies over the course of the game. As before, the data is filtered to view only games that have a definitive winner. These graphs show the deterioration of the two AIs over the course of the game. From this, conclusions regarding how the game develops can be made.

8.3 Results, Performance, and Analysis

We immediately see several trends given by the fun function. A clear player order effect was found between nearly every AI. While some AIs were simply more powerful and able to win regardless of characteristics, in nearly every case we see that more wins are generated from the AI if they play first compared to when they play second. For some more closely related AIs, the odds of winning changes in its favor given the player order effect. From this we can conclude that for certain comparable AIs, player order may greatly affect the outcome of the game.

Furthermore, in every case, the trend line from the total army plot simply shifted its intercept value but held consistent slope. This intercept can be credited to first round play. After the first round (when the first point is calculated) the army advantage can be a one player advantage. This first round however rarely changes the results of the game. Consider figure 1.

Figure 1.

Here the green and red dots represent AI0s and AI4s percent total armies respectively over time. Again, the dots only represent games in which a definitive winner was found. Here we see that AI0 has a clear advantage but when the player order is changed, AI4 wins more games. Notice, however, while the trend line shifts down, that the slope stays constant. This implies that while the game is certainly affected by player order, on average, the game develops the same regardless of player order.

For the same AI match, the fun function provides two survival charts. Now consider figure 2.

Figure 2.

In the first plot we see that player two holds several armies until near round 50. Suddenly over the next two rounds however, we see an incredible drop off. AI0 has mostly defensive characteristics. From this it appears that AI0 condenses its portion of the graph and builds armies. Somewhere near halfway through the game, it branches out and gains several territories in a very small period of time. When the order of the game is reversed we see a much more even trend. Both AIs seem to lose armies at a consistent rate. This effect could be accredited to player order effect or the characteristics of the players in regards to their section of the sub graph.

While these trends, in general, hold true, one AI in particular defied the general trends. We will call this AI, AI3. AI3, in general, tended to produce more wins when playing in the second player position. This trend was tested more rigorously against 15 new AIs. From this data, the fun function produced the win records for both AI3 and the other AIs for both player order. The difference between wins was then calculated. That is $\Delta win = \mathit{AI3\_wins} - \mathit{AIOpponent\_wins}$. This figure was calculated for both player orders. This data is represented in figure 3.

Figure 3.

As in figure 3, on average AI3 wins 3.625 games while playing second in comparison to playing first where the AI won only 0.5 games on average. This in turn would seem to offer a solution to the player order effect, mentioned above. By simply playing the AI3 characteristics (Survival, Defensive, Cautious, Carry), on average the advantage would go to the second player, thus making player order obsolete.

Another nuanced observation shows that this conclusion may not be entirely true. Under further observation, another AI which we will call AI4 (Survival, Aggressive, Tactical, Rusher) seemed to be superior to AI3 in both player orders. A more rigorous study of these two AIs revealed that the assumption was correct. The fun function showed that on average, AI4 wins more often than AI3. Consider figure 4 and figure 5.

Figure 4.

Figure 5.

Notice that these two figures show very low correlation when player order changes. When AI3 goes first, it grosses more wins than AI4. When AI3 goes second, however, AI4 wins the majority of the games. While by a narrow margin, when AI4 goes first, AI4 wins 56 more games than AI3 in a 500 game trial. Thus if AI3 plays second they will on average lose to AI4.

8.4 AI behaviors and Surprises

In this project several AIs were posed against each other in a game of Risk. Initially, it was predicted that a large player order effect would affect the game. This hypothesis held true for the majority of AIs that were tested in this experiment. AI3 however, proved that by using the correct strategy, a player would actually hold an advantage by going second in the game. From this however, AI4 proved that their still exist AIs that can beat AI3 while being the first player to move.

From this project, we advise future Risk players to abstain from declaring their player strategy until the player order is determined. From this, we advise that second players play a cautious, defensive game, while first players play with aggressive tendencies. A consistent analysis of strategy will reveal the best strategy for either player to take throughout the duration of the game.

9 Conclusion

We have reformalized the board game *Risk* as a graph optimization, and implemented an AI to solve the problem, i.e. to play the game. Several decision algorithms were devised based on graph properties, and they in turn form the personalities of the AI. Our study of 4 binary traits, or a total of 16 personality variations of the AI, is merely the beginning. We have discovered some interesting AI behaviours and performance at a level beyond any human players. One can potentially investigate even more variations of the traits.

For future studies, an analysis of subgraphs would provide deep insight into the value of individual territories. The AIs are built in such a way that the priority function is based off of current demographic of troops and shape of their graph. Evaluating the graph as a whole would provide insight to the value of different subgraphs that would proactively affect the priority function. This conditional thinking is much more like the ideas of a human player and thus could inform future strategies.

Furthermore, game shifts could be analyzed using the fun function and the aforementioned subgraph analysis. This would allow for defensive players an understanding of which territories that are worth defending and worth conceding. This shift parameter would allow for insight to game length, army distribution, and other key factors.

10 References

1. Carnegie Mellon University, *NELL: Never-Ending Language Learning.* `http://rtw.ml.cmu.edu/rtw/`

2. Tom M. Mitchell, *Machine Learning.* McGraw-Hill, 1997.