Attribute Learning System

[Applying Genetic Algorithms to Improve RPG Combat Mechanics]

Austin Cory Bart
Virginia Tech
acbart@vt.edu

K. Alnajar
Virginia Tech
kar@vt.edu

ABSTRACT

Having a good set of moves for players to choose from in role-playing games (RPG) is essential for the game to succeed. Often times in an RPG, the players have various attributes which these moves can effect and coming up with good formulas for this is not easy. The process of creating an effective set of moves can take time and can be a difficult challenge to overcome in the design process. This paper propeses an implementation to effectively create these moves using a genetic algorithm implementation. Two seperate implementation styles of genetic algorithms are used, a tree style and a vector style. The results show that the vector styled approach for the genetic algorithm shows promising results in move set creation.

Categories and Subject Descriptors

1.2.1 [**ARTIFICIAL INTELLIGENCE**]: Applications and Expert Systems — *games*

General Terms

Genetic Programming

Keywords

Genetic, Programming, Game, Development

1. PROBLEM

Turn-based combat is an extremely common video game mechanic in which players fight computer controlled enemies and other players using a pre-defined set of moves called a *movelist*. One of the most challenging aspects of developing this mechanic is finding a functional and balanced movelist. Moves should take advantage of the capabilities of the system, while still being simple and enjoyable for players. At the same time, the moves must be just complicated enough to ensure that players have a good understanding of the game and their own efficacy in order to develop mastery. Creating a movelist that falls within this criteria is a time consuming task for game developers, as the addition and subtraction of moves requires the entire system to be created and tested iteratively.

A move is an operation that advances a combat from one state to another. A combat's state is given by a set of *attributes*, numbers that indicate the performance and statistics of the players. Although the names of the attributes vary among game engines, common examples include "Health", "Attack", and "defense". For clarity, we differentiate beteween *primary* attributes — which determine the outcome of a player's battle — and *secondary* attributes — which have no direct impact on the outcome of the battle. "Health" is often treated as a primary attribute, as when one player runs out of health then the game is over. "Attack" and "Defense" are used to calculate how much certain moves affect "Health", but their final value has no impact on the outcome of the game, so they are secondary attributes. Moves can take on a vast number of forms, even in a system with a small number of attributes. This model can accurately describe a wide range of popular games with turn-based combat, such as Pokemon [1]. Although many games follow the model of primary and secondary attributes, imbalanced moves lead to many scenarios where secondary attributes can be completely ignored. This "greedy" approach to combat has players simply spamming their primary-affecting moves until they win combat, requiring little strategy.

1.1 Prior Work

The problem of generating movelists is largely novel, with little prior work. However, there is a significant, growing interest in formal game development, a largely hitherto adhoc process. Many of these tools have a prescriptive nature, rather than explicitly generating code. McNaughton, et al. [3], for instance, describe how Generative Design Patterns can be used by novice programmers to create components of video games.

On the other hand, Artificial Intelligence research has led to tools that can automate the game development process and enable emergent gameplay. Amato and Shani[**?**] applied Reinforcement Learning to develop adapative enemies that operate on a strategic – as opposed to tactical – level. Zook and Riedl [6] examine dynamic difficulty adjustment for players with varying degrees of game skill. Their paper focuses on challenge tailoring in a role-playing style game using a temporal player model to assist in predicting future strength. This concept proposes a way for games to self-monitor and adapt based on player action. This concept is

interesting in that it, broadly speaking, creates a playable environment in an automated process. Our work builds on these successful studies where AI methods were used to simplify and empower the game development process.

1.2 Approach

To simplify the development of turn-based combat engines, we have created a system to automatically generate balanced movelists. The system is targetted to game developers who can tailor the framework to their individual game engine. When designing our system, we studied the problem space very carefully. Because the state space of movelists is extremely large and has many local maxima, our framework uses a *Genetic Algorithm*. A Genetic Algorithm mimics the natural process of evolution by repeatedly manipulating a population of *phenotypes*. These phenotypes are composed of mutable properties called *genes*. Each new iteration of a population is called a *generation*. A phenotype can be changed via *mutation* - where genes are changed at random - and via *cross-over* - where genes are combined with the genes of another phenotype. A *fitness function* maps the phenotypes to a *utility* – a real number that indicates that phenotype's value in relation to an "ideal" move.

We performed a number of experiments with our genetic algorithm in order to develop balanced moves. Many of these experiments were staged as we developed the subroutines of our algorithm, in order to ensure that our implementation was well-designed. Afterwards, we varied a multitude of our system's parameters to measure their effect on the rate of utility growth and maximal utility achieved by our system. As we gathered data, we did additional experiments to evaluate the validity of the results achieved.

2. IMPLEMENTATION OF THE GENETIC ALGORITHM

In our system, the phenotype is a movelist and the genes are the moves that make up the movelist. While designing our system, we explored two different implementations of moves. Additionally, we experimented with a number of fitness functions.

2.1 Fitness Function

We based our fitness function on the results of battle simulations between two players. The two players were computer controlled, basing decisions on the Minimax algorithm (limited to depth five). Each player was given a distinct movelist from which to select actions. The utility of the selected moves in the Minimax algorithm was calculated from how they impacted the primary attributes of each player.

When a battle simulation completes, we use various statistics from the simulation to calculate the fitness. A number of approaches were tested in determining the overall fitness for a set of moves:

Move Usage In an ideally balanced game, all moves should be used approximately equally. Over the course of a battle, we keep track of all the moves used as a ratio of the total moves used. Then, the standard deviation of these percentages is calculated. A well balanced game - where moves are used evenly - has a low standard deviation, but an unbalanced game will have a high deviation.

Battle Victory In a typical game, there should be final victory or outcome between the players. This metric awarded a positive utility for battles that involved a victor and a negative reward for battles that ended in a stalemate.

Battle length We believed that a good game should last with in a certain range of turns. This way the battles are neither too short (where one player has an overt advantage) or too long (where neither player is able to gain an advantage).

Linearity In order for a battle to progress smoothly, each player should show a steady decrease in primary attributes, rather than a sudden loss at a single point in the battle. This function looked at the progression of player statistics and gave a high utility for battles where both the playersâĂŹ progress was a steady decrease in attributes, and a negative utility for sudden or extreme changes in the player attributes.

Preliminary testing revealed the most success with a fitness function solely based on Move Usage and Battle Victory. After analyzing the results from the Linearity metric, we determined that this led to battles where the secondary attributes were underrepresented, which was the antithesis of one of our primary goals. Experiments with Battle Length did not seem to have a serious impact on utility, for reasons unclear. Since we desired consistency in our utility function across all results, during our experiments we did not include Battle Length and Linearity in our utility calculations.

2.2 Move

As previously described, a move maps how battle states (represented as vectors of values matching attributes) advance to their next states, via mathematical operations. Figure 1 visually shows this transformation.

Turn i					
Player 1			Player 2		
Health	Attack	Defense	Health	Attack	Defense
100	10	10	100	10	10

↓ Player 1 uses a move to affect Player 2's Health ↓

Turn i+1					
Player 1			Player 2		
Health	Attack	Defense	Health	Attack	Defense
100	10	10	**70**	10	10

Figure 1: Representation of a move transforming the state of a battle, directly affecting the vector of attribute values.

2.3 Function Tree

We first attempted to represent moves as Abstract Syntax Trees of operators over attributes. Every node in the AST is an *operator*, with zero-arity operators (which yield attributes from the input vector, unmodified) being terminal nodes. Early experiments with *ephemeral nodes* - coded as

zero-arity operators that returned a constant value, as opposed to an attribute - led to their inclusion. Figure 2 gives an example of a Function Tree with a binary operator (addition), unary operator (doubling), and two nullary operators (Player 1's Attack attribute and an ephemeral constant). At the top of the tree, the output attribute is shown (Player 2's Health attribute).

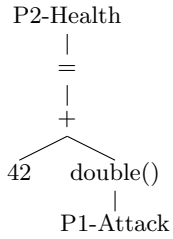

Figure 2: Example of a Function Tree

2.3.1 *Mutation Algorithm*

There are many well-known methods for mutating an AST, so we chose to implement a core subset of them:[4]

Node Replacement Mutation A node is selected and replaced with another operator of equal arity.

Subtree Mutation A subtree is selected, and is given a new parent. This new, taller subtree replaces the original, increasing the depth of the tree.

Shrink Mutation A node is selected and its arity is reduced, decreasing the number of subtrees.

Preliminary tests of the mutation algorithms revealed that mutation had an undesirably dramatic effect on the structure of the Function Tree. In order to keep mutational changes to a minimum, we refactored the algorithms to only affect terminal and pre-terminal nodes. The result of this change is demonstrated in Figure 3, which shows the lessened logarithmic relationship between the number of times a tree is mutated and the Edit Distance from its parent.

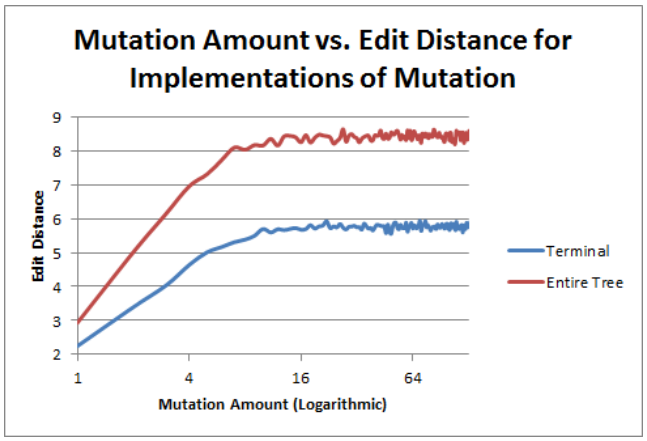

Figure 3: Mutation Amount vs. Edit Distance for Implementations of Mutation

2.3.2 *Cross-over Algorithm*

The problem of crossing over Abstract Syntax Trees is well-established in the literature. We investigated several possible implementations before choosing one based on work by Poli and Langdon [5]. Their uniform cross-over algorithm matches common nodes in the two parents trees, starting from the root. When dissimilar parallel nodes (named *Boundary Nodes*) are encountered, a subtree is chosen randomly from the two options.

Unfortunately, although this implementation was described favorably in the literature, we found that it rarely caused a true combination of the parents, but instead led to one parent being copied wholesale in favor of the other. Figure 4 graphs the result of a 1000 trials where two parents were randomly generated and then crossed with each other. The Percentage Difference from Parent was calculated as the change in edit distance from one parent to the child over the edit distance of the two parents. Ideally, this distributed would be heavily banded around 50%; instead, more than 3/5 of the data is at 0% or 100% percent difference (indicating a parent was copied completely).

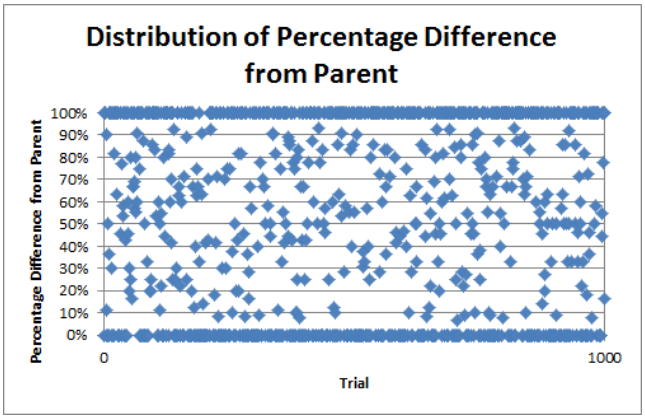

Figure 4: Distribution of Percentage Difference from Parent in Function Tree Cross-over

2.3.3 *Validation of Trees*

While we performed structural analysis of the our Function Tree implementation, we were doubtful that these structural modifications would have a corresponding impact on the numerical results of the functions. For example, changing a function of the form "Player 1's 1^{st} Health + player 2's 1^{st} Attack" to use division instead of addition would yield a largely different set of outputs. In order to quantitatively measure the similarity of two functions, we used a two-sample KolmogorovâĂŞSmirnov test. This variant of the KS-test is a non-parametric method for determining if two sets of samples come from the same distribution.

Because the KS-test expects one-dimensional data, the domain of the inputs (normally multivariate) was restricted from the real numbers to [-100,100], and their cross-product treated as a single input. For example, in a system with one primary attribute and one secondary, the Function Tree would have

$$|(100) - (-100)|^4 = 1600000000$$

integer inputs and outputs, following the pattern demonstrated in Figure 5.

P1 Primary	P1 Secondary	P2 Primary	P2 Secondary
-100	-100	-100	-100
-100	-100	-100	-99
...			
-100	-100	-99	-100
-100	-100	-99	-99
...			
100	100	100	99
100	100	100	100

Figure 5: Input Pattern for the KS-Test

To calculate our samples, we strode over the inputs at a moderate pace in order to further reduce the domain. A value of 1 from the test indicates evidence of two completely different functions, whereas a value of 0 indicates evidence of two identical functions.

In our validation trials, we generated two trees randomly, measured their edit distance, and then graphed that metric against the results of a KS-test. Figure 6 shows the surprisingly strong correlation between Edit Distance and the KS-Test. Unfortunately, it also demonstrates a weakness of the Function Tree implementation: the values quickly get very high (reaching .4), which means that we expand through the search space extremely quickly even when performing a single mutation.

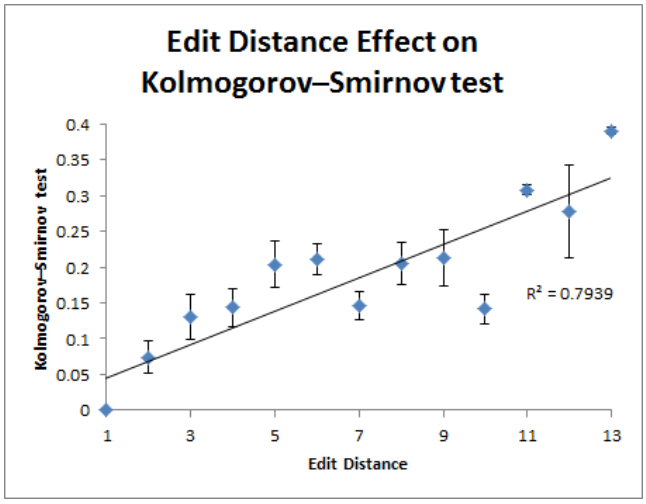

Figure 6: Edit-Distance vs. Kolmogorov-Smirnov Test for Function Trees

2.4 Function Vector

Function Trees suffer from a lack of fine-grained state exploration. A single mutation can cause relatively drastic changes to the function. For this reason, we decided to implement and compare an alternative model - functions described as vectors, dubbed Function Vectors. The function vectors map a coefficient to each possible input attribute, with an additional additive constant. When the Function Vector is being evaluated, the current state's values are multiplied by their matching coeffecients, and then summated with the additive constant; Figure 7 gives an example of the equation used for evaluating Function Vectors. Due to the fundamental difference between the vector and tree scheme, a different approach was taken for the mutation and cross-over algorithms.

$$\text{P2 Health} = (0.1\times \text{ P1 Attack }) + (.5\times \text{ P2 Attack }) + 5$$

Figure 7: Example Function Vector

2.4.1 *Mutation Algorithm*

Function Vectors were mutated by adding a randomly-generated value (limited to a tight range) to a randomly-chosen coeffecient. Because the structure of Function Vector does not change with a mutation, a small percentage of the time the entire FV would be randomly replaced with a completely new move. This ensured that the entire possible state-space for the Vectors was explored.

2.4.2 *Cross-over Algorithm*

Once again, the constant structure of Vectors made it easy to implement a Cross-over Algorithm. When the parent Vectors had the same output attribute, their coeffecients were individually averaged together. Otherwise, one parent was randomly chosen to be copied wholesale, similar to the way that Function Trees resolved different parents in the extreme case.

2.4.3 *Validation of Vectors*

To validate the vector mutation function we used the K-S test to look at the difference in changes made by mutation. The results of the K-S test (Figure 8) showed only a marginal amount of change, which was expected given the limitation imposed on the range of possible coefficients allowed. Although there was no real correlation between the number of times a function was mutated and the results of the K-S test ($R^2 = 0.1394$), this was acceptable given that we were intentionally exploring a smaller state space.

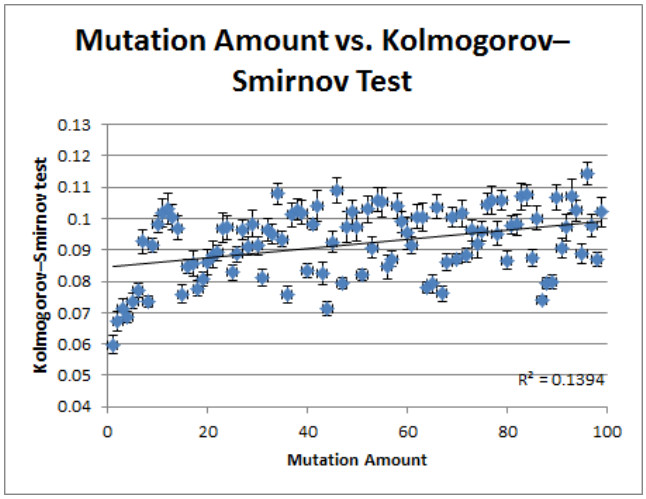

Figure 8: Mutation Amount vs. Kolmogorov-Smirnov Test for Function Vectors

Validation of the cross-over function for vectors was done through preliminary comparison and analysis of initial battle

simulations and iterations. Looking at the utility results when running initial tests showed a correlation between a high cross-over rate and improved utility. These results are demonstrated more explicitly in the next section.

3. EXPERIMENTS

3.1 Move Generation Experiments

In order to collect data on the effectiveness of the system and validate the moves, 25 iterations were run over a variety of conditions. The base conditions of the algorithm were set as follows, unless manipulated for the purpose of a particular experiment: a constant population of 500 movesets, both players were Minimax players with a look-ahead depth of four, mutation was set to 50

3.1.1 Tree vs. Vector

Both the tree and vector implementations were tested evenly in the experiments under the various conditions. The results of the highest utilities of each type of implementation across the iterations can be seen in the graph. The vector implementation significantly outperformed the tree implementation across nearly every experiment, with the highest tree utility failing to break a value of 825-utility, whereas the vector implementation reached a 1000-utility score multiple times across various experiments.

3.1.2 Mutation Rate vs. Cross-over Rate vs. Parents Retained

To see the effects of various conditions on the genetic algorithm, these variables were manipulated in multiple experiments, and the effects of these changes can be seen in the utility over iteration graphs. The results show that for the tree implementation, an equal 20

3.1.3 Attribute Type Frequencies

In these tests, we wanted to look at what effect the number of primary and secondary attributes may have on the overall utility of movesets. To this, attribute type frequencies were tested with combinations of 1 to 2 primary attributes and 1 to 3 secondary attributes. Results of these tests can be seen in the given graphs [graph reference]. For the tree implementation, the tests vary little effect on the outcome utilities, with the average consistently ranging in the 750 utility range. The results of the vector tests showed a greater effect of attribute frequency on the utility. The utility of tests conducted with a single primary attribute were yielded a higher maximum utility, with a 1000 utility reached in the tests with a single primary attribute, and either two or three secondary attributes. This testing validated the use of the default single primary attribute with two secondary attributes that was used for the attribute type frequencies in the other experiments.

3.2 Validation of Moves

To look at the effectiveness of the moves generated by our solution, we chose to validate the movesets by using a strategic Minimax player, and a nonstrategic (random) or lower level Minimax player to simulate a battle with the moves. If the moves produced were viable, then the expected outcome of this testing would show that a strategic player would outperform and defeat a random or lower level player. The results of this testing did not show what was expected. In many battle simulations the lower level Minimax was able to defeat the higher level Minimax player, and in some cases the random player was able to defeat the Minimax player. After looking at the movesets on a case by case basis, it was found that the test results which did not follow the prediction was due to an issue in the way moves that affected the primary status of the opponent had been formed. In these cases. the losing player often had no move that would be able to reduce a primary attribute of the other player in order to win.

4. CONCLUSION

The overall results of our move validation experiments showed that the movesets were not applicable in a true RPG setting. The poor results of the validation is most likely due to the utility function that determines the value of the movesets not properly taking into account an equal chance for each player to win the match through strategy. It became apparent that the utility function used for the movesets did not take into account a practical usage of the moves to defeat the opponent in all circumstances. The failure to take this feature of battles into account meant that though the move usage was strategic and distributed evenly, and there was a victor associated, it provided no opportunity for the player with the moveset that lacked a strong primary reducing move to win the battle. This makes the movesets generated by the system unusable in a practical application. However, we did find that in cases where both players were given a set of moves where each had an effective primary reducing move, the results of the validation experiments were

5. FUTURE WORK

As indicated by the results of our failed validity tests on the movesets, the most immediate work that needs to be done is a redesign of the utility function calculation. It is possible that given a utility function that takes into account overlooked aspects of the usage in these simulations, movesets which may be applied more practically in a game design could be generated by the system. It would be interesting to look at abstracting the design of the individual moves to provide flexibility for other game engines. In our original design, we proposed a system where every move has a unique function behind it. However, some games use a single formula for every move, and then assign unique attributes on a per-move basis (such as âĂIJBase powerâĂİ or âĂIJAccuracyâĂİ, in addition to player attributes. Supporting game engines such as these might dramatically alter the search space, but could lead to more tenable results. Finally, we will explore a trending alternative to traditional genetic algorithms called *Novelty Search*, which has been used in research for generating game content dynamically [2]. The Novelty Search algorithm rewards new behavioral changes in a phenotype, rather than applying a fitness function. By removing the focus on moving towards an objective, which can make it difficult to break out of deceptive local maxima, this algorithm has been shown to be effective for certain problem domains. Given this still inchoate research problem, applying Novelty Search or another variant search algorithm could be a fruitful strategy.

6. REFERENCES

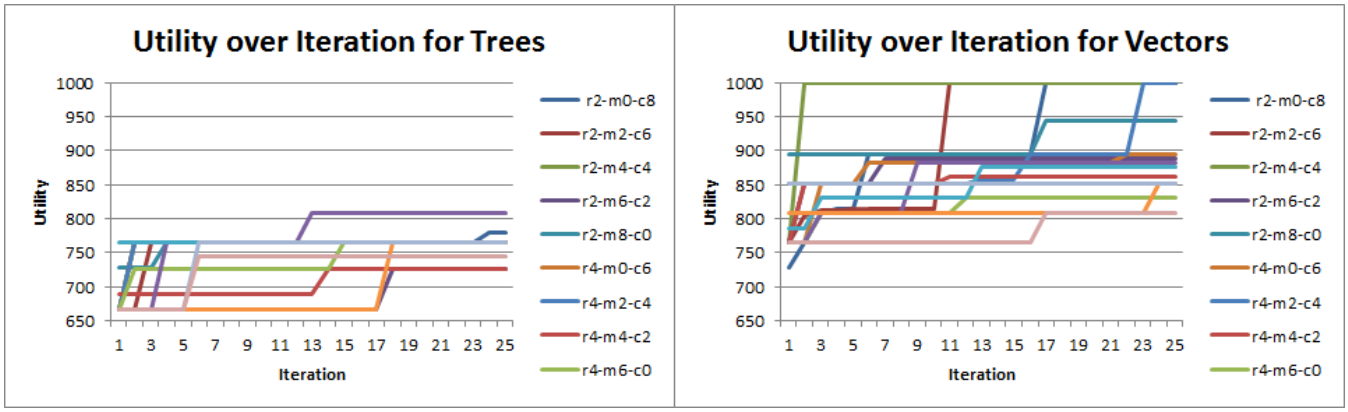

Figure 9: Maximal Utility over Iterations for Varying Retention, Mutation, and Cross-over Rates

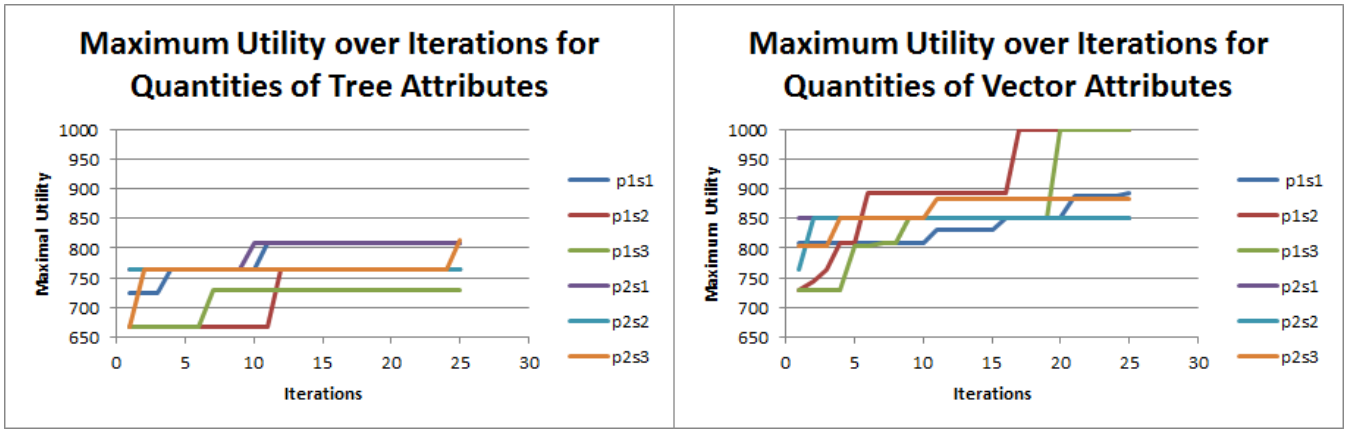

Figure 10: Maximal Utility over Iterations for Varying Quantities of Attributes

[1] GameFAQs. Pokemon red advanced battling, July 2010.
[2] A. Liapis, G. Yannakakis, and J. Togelius.