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

RPG combat mechanics - attributes (primary vs. secondary)

Need to define moves movelist attributes primary secondary

balancing moves - naive strategies and poor gameplay as a result of move options time consuming - creating moves properly takes time and testing large space - huge number of ways to calculate and create moves talk about some games where the moves were not ideal mention lack of literature on this specific topic?

2. APPROACH

2.1 Prior Work

Dynamic Difficulty Adjustment

2.2 Target Audience

Game developers,

2.3 Approach

Because the state space is extremely large with many local maxima, we used 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 properties are changed at random - and via cross-over - where properties are combined with the properties of another phenotype. A *fitness function* maps the phenotypes to a real number that indicates that phenotypeâĂŹs value. We performed a number of experiments with our genetic algorithm in order to develop balanced moves. In particular, we investigated the effect that varying parameters had on our rate of utility growth and maximal utility achieved. As we gathered data, we did additional experiments to evaluate the validity of the results achieved.

3. IMPLEMENTATION OF THE GENETIC ALGORITHM

In our system, the phenotype is a Movelist, a union of two sets of moves (one set for each player). A move is abstractly defined as a function that maps a set of input attributes to a single output attribute. While designing our system, we explored two different implementations of moves. Additionally, we experimented with a number of fitness functions.

3.1 Fitness Function

We based our fitness function on the results of battle simulations between two players. The two player types used were based on Minimax algorithms with a depth limit of four. Each player was given a moveset from which to select in their action state. The utility of the selected moves in the Minimax algorithm were based on maximizing the primary attributes of each player. Once a single battle simulation completes, we use various statistics from the simulation to calculate the fitness. A number of approaches were taken 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 in an ideal battle, 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 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 antithesis to our goals. Since we desired consistency in our utility function across all results, during our experiments we did not include Battle Length and Linearity in our calculations.

3.2 Move

As previously mentioned, our system defines a Move as an abstract function to map one state of the battle to the next. A state in the battle can be seen as a vector of the two playersâĂŹ attributes, demonstrated in **??**.

3.3 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. **??** gives an example of a Function Tree with a binary operator (addition), unary operator (doubling), and two nullary operators (an attribute and an ephemeral constant).

3.3.1 Mutation Algorithm

There are many well-known methods for mutating an AST, so we chose to implement an interesting subset of them.

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.[**?**]

Node Replacement Mutation A node is selected and replaced

Preliminary tests of the mutation algorithms revealed that they had an undesirably dramatic effect on the structure of the Function Tree. **??** demonstrates how repeatedly mutating the entire tree has almost random impact on the Levenshtein Edit Distance between the mutant and the original. In order to keep mutational changes to a minimum, we refactored the algorithms to only affect terminal and preterminal nodes. The result of this change is demonstrated in **??**, which shows a logarithmic relationship between the number of times a tree is mutated and the Edit Distance from its parent.

3.3.2 Cross-over Algorithm

3.3.3 Validation of Trees

3.4 Function Vector

The function vectors were modeled as vectors of coefficients. In this model, each attribute was assigned a coefficient. In addition, the vector had a constant that could be used for additional adjustments to the vector equation. Due to the fundamental difference between the vector and tree scheme a different approach was taken for the mutation and cross-over algorithms.

3.4.1 Mutation Algorithm

The mutation of the vector functions was performed using a randomly generated coefficient within a specified range. This allowed for manipulation of the vectors in a significantly reduced space. This was verified in a [KS-test] validation, which showed that the mutations acting upon the vectors had a subtle influence.

3.4.2 Cross-over Algorithm

The cross-over algorithm of the vector functions utilized the coefficients to find an average between coefficients of similar attribute effectors within the vector equations. Testing showed this to be significantly effective in producing new moves with high utility.

3.4.3 Validation of Vectors

To validate the vector mutation function we used the Kolmogorov-Smirnov (K-S) test to look at the difference in changes made by mutation. The results of the K-S test showed only a marginal amount of change (R2=0.14), which was expected given the limitation imposed on the range of possible coefficients allowed. Validation of the cross-over function for vectors was done through 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.

4. EXPERIMENTS

4.1 Move Generation Experiments

4.1.1 Tree vs. Vector

4.1.2 *Mutation Rate vs. Cross-over Rate vs. Parents Retained*

4.1.3 *Attribute Type Frequencies*

4.2 Validation of Moves

Ran simulations of Minimax/Random to determine whether Minimax dominated. To look at the effectiveness of the moves generated by our solution, we chose validate the movesets by using a strategic and nonstrategic player to simulate a battle with the moves. If the moves produced are viable, then the expected outcome of this testing should show that a strategic Minimax player should outperform and defeat a player who uses moves at random. The results of this

5. CONCLUSION

6. FUTURE WORK

6.1 Genetic Operators

7. REFERENCES