Evolving Bot AI in Unreal™*

Antonio Miguel Mora[1], Ramón Montoya[2], Juan Julián Merelo[1], Pablo García Sánchez[1], Pedro Ángel Castillo[1], Juan Luís Jiménez Laredo[1], Ana Isabel Martínez[3], and Anna Espacia[3]

[1] Departamento de Arquitectura y Tecnología de Computadores, Universidad de Granada, Spain
{amorag,jmerelo,pgarcia,pedro,juanlu,}@geneura.ugr.es
[2] Consejeria de Justicia y Administración Pública, Junta de Andalucíam, Spain
rangel.montoya@juntadeandalucia.es
[3] Instituto Tecnológico de Informática, Universidad Politécnica de Valencia, Spain
{amartinez,aesparcia}@iti.upv.es

Abstract. This paper describes the design, implementation and results of an evolutionary bot inside the PC game Unreal™, that is, an autonomous enemy which tries to beat the human player and/or some other bots. The default artificial intelligence (AI) of this bot has been improved using two different evolutionary methods: genetic algorithms (GAs) and genetic programming (GP). The first one has been applied for tuning the parameters of the hard-coded values inside the bot AI code. The second method has been used to change the default set of rules (or states) that defines its behaviour. Both techniques yield very good results, evolving bots which are capable to beat the default ones. The best results are yielded for the GA approach, since it just does a refinement following the default behaviour rules, while the GP method has to redefine the whole set of rules, so it is harder to get good results.

1 Introduction and Problem to Solve

Unreal™ [9] is a First Person Shooter (FPS), an action game in which the player can only see the hands, and the current weapon of his character, and has to fight against enemies by shooting to them. It was developed by Epic Games and launched for PCs in 1998, with great success since it incorporates one of the best multiplayer modes to date. In that mode, up to eight players (in the same PC or connected through a network) fight among themselves, trying to defeat as much of the others (enemies) as possible, getting the so-called *frag* for each defeat, moving in a limited scenario (or map) where some weapons and useful items appear frequently. The players can be human or automatic and autonomous ones, known as *bots*. Each player has a life level which is decreased every time he receives a weapon impact, this decrement depends on the weapon power, the distance, and damaged zone in the character. In addition there are some items that can be used to increase this level or to protect the player.

* Supported in part by the MICYT projects NoHNES (TIN2007-68083) and TIN2008-06491-C04-01, and the Junta de Andalucía (P06-TIC-02025 and P07-TIC-03044).

C. Di Chio et al. (Eds.): EvoApplications 2010, Part I, LNCS 6024, pp. 171–180, 2010.

There are many games which include multiplayer modes against human or bots, but there are some features, which made Unreal™ the best framework for us, such as the fact that its *native* bots already have a high level AI, and it offers an own programming language, *UnrealScript*, which is easy, useful and quite powerful. This does not mean that UnrealScript is without drawbacks, including the fact that arrays must be one-dimensional and the limitation in the number of iterations in a loop. These will have to be taken into account when programming a native bot, but in spite of these lacks, it is still the best environment we have found to develop our work.

We should also emphasize that the Unreal™ bot's AI was (and still is) considered as one of the best in all the FPS bots ever designed. It is mainly based on the definition and handling of *states*, each of them modelling the behaviour of the bot when it has a specific status, location in the map, or relationship with the other players (enemies). Many possible states, and many different flow lines between them are defined in a finite state machine [1].

The bot, during a game, changes its current state depending on some factors present in its surroundings and depending on its own status and situation. So, the state change depends most of the times on some specific parameters. They are usually compared with some values, which are most of times hard-coded. This way, the state change (and the power of the Bot's AI) strictly depends on some constant values.

Therefore, the first issue addressed in this work has been the improvement of these constants using a Genetic Algorithm [3], since it is possible to define an array including those parameters and change (evolve) their values to get a better behaviour in the game.

In addition, the finite state machine which determines the bot's AI flow, could be improved too, by avoiding superfluous states or changing the flow between them, in order to arise new transitions between different states. With this objective, a Genetic Programming Algorithm [4] has been applied, since it is an excellent metaheuristic to improve graphs or trees, as is this case.

In both cases we have implemented bots with a *Genetic AI*, or **Genetic Bots (G-Bots)**. We have used evolutionary algorithms to improve the Unreal™ AI given their well-known capacity for optimization.

These way, each G-Bot improves its AI by playing a game; getting a better global behaviour in time, that is, defeating as much enemies as possible (getting frags) and being defeated as less as possible.

2 State of the Art

At the very beginning, the FPS games just included a single player mode (e.g. Wolfenstein in 1987), after this, most of them offered multiplayer possibilities but always against other human players, such as Doom in 1988. The first known game in including autonomous bots (with a simple AI) was Quake in 1992. It presented not only the option of playing against machine-controlled bots, but also the possibility to modify them (just in appearance or in other few aspects) or

create new ones. In order to do this, the programmers could use a programming language called QuakeC, widely used in those years, but which presented some troubles, since it was strictly limited and hard-constrained. So, the bots created using it showed a simple AI (based in fuzzy logic in the best case), and it was not possible to implement more complex techniques as evolutionary algorithms. Unreal™ appeared some years later, being the first game that included an easily programming environment and a more powerful language, so plenty of bots where created. But just a few applied metaheuristics or complex AI techniques, and most of them are based in predefined hard-coded scripts.

Nowadays, there are many games that offer similar possibilities, but almost all of them are devoted to the creation of new maps (sceneries) or characters, being mainly related to the graphical aspect or modifications in their appearance.

In the last few years there have been some studies related to the application of metaheuristics to the behaviour improvement of bots in computer games, as [2,8] where the authors apply Artificial Neural Networks, [6] where evolutionary techniques are used, or [7] in which an evolutionary rule-based system (every individual is a set of conditions, which depend on the bot status and a specific action) has been applied, and which has also been developed under the Unreal™ framework (Unreal Tournament™ 2004).

In the present work we have chosen the original Unreal™ instead of the newer game, since it has a simpler environment but which is enough to perform the proposed study.

3 Genetic Bots

As previously stated in Sect. 1, the objective is to improve the behaviour of an Unreal standard bot, by changing its AI algorithm. Specifically there are two approaches: the first one consist in modifying the default AI by improving the values considered in the conditions assessed to change the current state (and go to another one); the second one is related to the improvement of the finite state machine which the bot's AI follows.

This way, the **Genetic Algorithm-based Bot** (GA-Bot from now on), tries to optimise the values of a set of parameters which represent each one of the hard-coded constants that are in the bot's AI code. These parameters determine the final behaviour of the bot, since most of them are thresholds depending on which, the bot state changes (for instance the distance to an enemy or the bot's health level).

So firstly, it was necessary to determine the parameters to optimise. This way and after a deep analysis of the bot's AI code, 82 parameters were identified. These were too many parameters, since UnrealScript considered as the maximum length for an array 60 floats. In addition it is difficult to evolve such a big array inside a game, since the evaluation function depends on the results of the game, and it would need many individuals and generations to do it.

The number must be reduced, so some parameters were redefined as function of others, and some of the less relevant were unconsidered in the GA individual.

At the end, the array includes just 43 parameters. This set corresponded to an individual in the GA. Thus, each chromosome in the population is composed by 43 genes represented by normalised floating point values (it is a real-coded GA). This way, each parameter moves in a different range, depending on its meaning, magnitude and significance in the game, but all of them are normalised to the [0,1] range. The limits of the range have been estimated, conforming a width interval when the parameters are just modifiers (they are added or subtracted), and a short one if they are considered as the main factor in the bot's decision taking, to avoid an extremely bad behaviour.

This models, in such a way, one approach to the bot's AI. So, the GA evolves 'the behaviour of the bot' (it evolves the trigger values to change between states).

The evaluation of one individual is performed by setting the correspondent values in the chromosome as the parameters for a bot's AI, and placing the bot inside a scenario to fight against other bots (and/or human players). This bot is fighting until a number of global frags is reached, since it is not possible to set a time play for a bot in Unreal, so once the frags threshold has been reached (and the current bot is defeated), the next bot's AI will be the correspondent to the next chromosome in the GA.

Two-point crossover and *simple gene mutation* (change the value of a random gene (or none) by adding or subtracting a random quantity in [0,1]) have been applied as genetic operators (see [5] for a survey). The GA follows the *generational + elitism* scheme, considering as the *selection probability (SP)* for one individual a value calculated using its rank in the population depending on the fitness value, instead of calculating it directly considering the fitness function. This way, it is avoided that superindividuals (those with a very high fitness value) dominate the next population, and a premature convergence occurs. This method is known as *lineal order selection*. Once the SP has been calculated, a probability roulette wheel is used to choose the parents in each cross operation. The elitism has been implemented by replacing a random individual in the next population with the global best at the moment. The worst is not replaced in order to preserve diversity in the population.

The *fitness function* was defined considering the main factors to score a game (through an evaluation function), after some experimentation, they are:

- *frags*, the number of defeated enemies by the bot
- *W*, the number of weapons the bot has picked up
- *P*, the associated power to these weapons
- *I*, the number of items the bot has collected
- *d*, the number of times the bot has been defeated
- *t*, game time the bot has been playing

So, the fitness function equation for the chromosome i is:

$$F_i = \frac{frags_i + [\frac{P_i}{d_i} + (\frac{W_i \cdot 10}{d_i})^{-1}] + \frac{I_i}{10} - \frac{d_i}{10}}{t_i} \quad (1)$$

Where the constant values are used to decrease the relative importance of each term. So, as can be seen, $frags$ is the most important term in the formula.

The following factor (inside square brackets) is related to the weapons; it is composed by two terms: the first one considers the importance of the associated power of the picked up weapons, in average, since a player loses all the weapons once it is defeated. The second term weighs the number of weapons collected by the bot, but it is again an average. In addition this term is inverted since it should take low values when the bot has collected lots of weapons in a life. The objective of this whole factor is to assign a higher weight to a bot which has picked up less but powerful weapons, since searching for them is a risky task, and takes some extra time. The other two factors are devoted to weigh the collected items and the number of times the bot has been defeated (with a negative weight). All the terms are divided by the time the bot has been playing to normalise the fitness of all the chromosomes (they play for a different time).

Since it is the first approach, we have decided to consider an aggregative function instead of a multi-objective one. It is easier to implement and test, and it requires less iterations. The multi-objective approach will be addressed in future works.

The **Genetic Programming-based Bot** (GP-Bot from now on) works in a different way, since it is based in a genetic programming algorithm [4], which can be used to evolve a set of rules. The idea is to redefine the initial set of rules (flows between states) which determines the behaviour of the bot.

The first approach was to work with all the possible inputs and outputs which can be considered, looking at the whole set of states that a default bot can manage, but due to the big amount of them and their complexity, this would mean a huge set of rules to evolve (represented as trees), which can be unapproachable by an algorithm defined inside UnrealScript (due to its strong array size constraints, and limited resources).

So, we decided just to consider the two most important states:

- *Attacking*, in which the bot decides between some possible actions as: search, escape, move in a strategic way, attack and how to do it (from distance, nearby, close, from a flank).
- *Roaming*, in which the bot searches for weapons and items

The flow diagrams of both states are respectively shown in Fig. 1 and Fig. 2.

Then, only the functions applied in the decision taking (since there are some others just used to show an animation in the game, for instance, such as `HitWall`) were studied, in order to get the inputs and the outputs. These functions are devoted to determine the next state to pass, from the one in which the bot currently is, so each one of the functions is divided into a set of sub-functions (*inputs*), and being the *outputs* the correspondent 'jumps' to the next states/substates. For example, there is a function to check if the bot has found an item (`ItemFound`), which returns a 'TRUE' or a 'FALSE' value depending on what the bot has found. It can be considered as an input. One possible output could be `GotoState('Roaming', 'Camp')`, for instance.

This way, all the possible inputs and outputs for these states are used to define rules in the form:

```
IF <INPUT> [AND <INPUT>] THEN <OUTPUT>
```

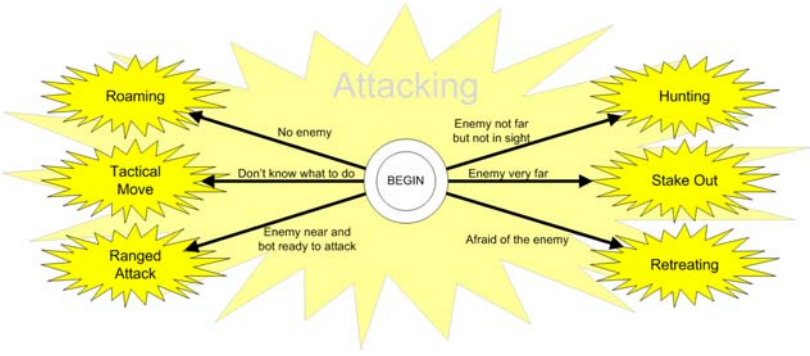

Fig. 1. Flow diagram of Bot's Attacking state. The states are represented by stars.

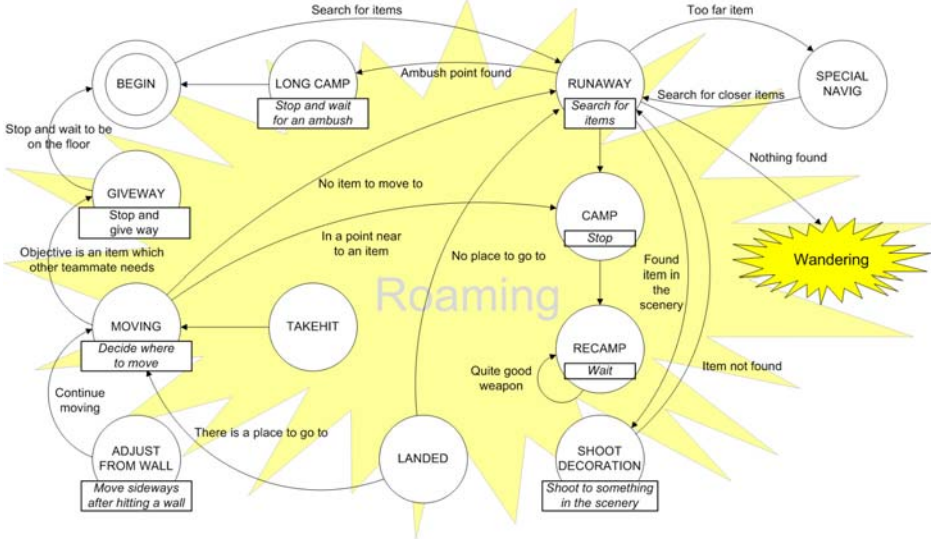

Fig. 2. Flow diagram of Bot's Roaming state. The states are represented by stars, and the sub-states by circles.

Where just two inputs have been considered as a maximum, since after several experimental runs, rules having three (or more) inputs were never triggered.

These rules are modelled as trees, considering as parent nodes the `IF` and `AND` expressions, connected through `RL` nodes (lists); and having as final nodes, the considered input and outputs.

An example tree can be seen in Fig. 3, where a Bot's AI is modelled with four rules:

`IF X1 THEN Y7`, `IF X3 AND X5 THEN Y2`, `IF X8 THEN Y1`, and `IF X9 THEN Y9`

So, every GP-Bot would have an AI structure based in the main set of states, but instead of the two previously mentioned (Attacking and Roaming), they

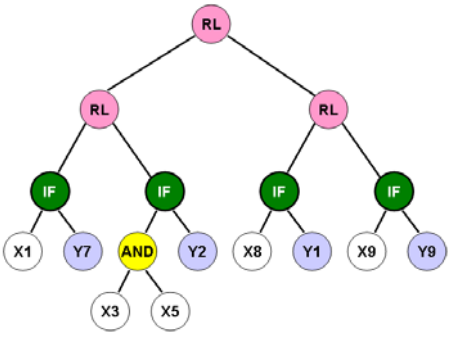

Fig. 3. Example Tree of rules for a Bot's AI. The rules are defined by IF nodes, Xs are inputs and Ys are outputs.

consider one tree of rules, which should be obtained by the evolution of the individuals in the GP algorithm. Thus, every individual in the algorithm is composed by the tree of rules, and also by one chromosome like those considered in the GA-Bot. This way, the evolution is performed over two different AI aspects: the rules, and the parameters, which means it is a *GPGA-Bot*.

During the evolution some genetic operators are used, firstly on the parameters (cross and mutation) which are those presented in the GA-Bot. In addition two specific operators are considered to perform the evolution of the rule trees. The *tree-crossover operator* has been implemented by choosing two different nodes in the parents and interchanging all the sub-tree, below each one of these nodes, taking into account some restrictions to preserve the tree coherence (i.e. not to have an IF node as a final one, or two AND nodes as parent and child). The *tree-mutation operator* just chooses a random node and substitutes the whole sub-tree below it by a randomly generated (but coherent) one. The *selection mechanism* is the same as in the GA-Bot (the lineal order selection).

Relating to the *fitness function*, in this case it should be valued both, the tree of rules and the parameter configuration, so the function has been lightly updated, considering two new factors:

- S, the number of shoots the bot has fired. Included to reward the bots which pick up weapons and use them (some bots do not use them, or do not do it correctly).
- rR, the repetition of rules. Tries to avoid the excessive repetition of rules in the behaviour of a bot, so every 5 repetitions for a rule, it is increased.

So, the fitness function equation for the chromosome i is:

$$F_i = \frac{frags_i + [\frac{P_i}{d_i} + (\frac{W_i \cdot 10}{d_i})^{-1}] + \frac{I_i}{10} - \frac{d_i}{10} + \frac{S_i}{50} - \frac{rR_i}{500}}{t_i} \quad (2)$$

Where all the factors are the same as in Equation 1, excepting the last two, which have been included to evaluate the behaviour in a more accurate way. But neither of these factors has a high relevance.

4 Experiments and Results

We have performed some experiments to test the algorithms. Each of them consists in launching a game match for eight players, being all of them bots[1], and being one of them (its AI) the GA-Bot or the GPGA-Bot.

Each run takes plenty of time (around one hour per generation in average) since every individual in the algorithms (an AI approach), is playing until a number of defeats is reached, and the match is played in real-time. It also depends very much on the map where the bots are fighting so, if it is a big map, it takes longer to reach this number (and change to the next individual). We have considered the parameters showed in Table 1, which have been defined starting from the 'standard' values, and tuning them through systematic experimentation.

Table 1. Parameters of GA and GP-GA algorithms

Number of individuals	30
Mutation Probability	0.01
Crossing Probability	0.6
Number of defeats per chromosome	40

In this work, the experiments have been devoted to test the good behaviour of the algorithms (and also of the bots), since they cannot be compared for the moment with the results yielded by other algorithms.

Four maps have been considered, and five bots of each type (GA and GP-GA) have been tested in each one of these maps. A classical run takes in average around 20 hours for 15 generations, but it depends on the map size.

The algorithms behaviour (related to the fitness) has resulted to be the expected, with some fluctuations in the average due to the classical diversification in the evolutionary algorithms, as can be seen in the Fig. 4 for some examples.

In this figure, a clear evolution in the average fitness is showed for all the cases. This evolution is more marked in the GPGA-Bots, since there are stronger changes in the AI in this algorithm (it evolves the bot's state transition rules), so the improvements in the behaviour are much more obvious, and follow a clear progression. In the GA-Bots this change is less marked since the behaviour is quite similar, but a bit optimised in each generation. These bots show a very coherent behaviour just from the first generation, because their behaviour rules remain the same, and just the decision parameters are changed (evolved). On the other hand, the GPGA-Bots have most of times an incoherent behaviour at the very beginning, since the rules belonging to the main states can be almost random, but the improvements can be more easily noticed after some generations, as has been previously commented.

Relating to the game score, the GA-Bots always beat their rivals, getting a high number of frags in some generations. The GPGA ones cannot get the first position, since the bad behaviour presented in the first generations means low frags, and also being an easy target for their enemies. Two screenshots are

[1] Although it is possible to include also human players.

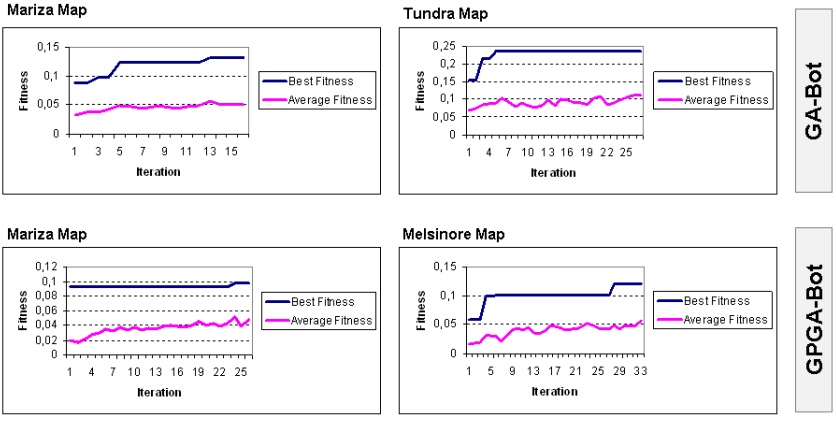

Fig. 4. Example results for two GA-Bots and two GPGA-Bots in two different maps

Fig. 5. Screenshots of the final score (after some generations) for one GA-Bot (GeneticBOT, on the left) and one GPGA-Bot (PGeneticBOT, on the right)

showed in Fig. 5, where it can be seen the classification for both types of bots after some generations.

In both cases we have implemented the final AI configuration (yielded by each one of the algorithms) into two definitive bots, being both of them better than the standard ones presented in Unreal™, winning all the matches.

5 Conclusions and Future Work

In this work, two different evolutionary algorithms have been implemented to improve the AI of the default bots in a PC game named Unreal™. The approaches have been a genetic algorithm, to optimise the decision parameters in the bots; and a genetic programming method, to optimise the set of rules which the bots consider in their AI. Looking at the results, both algorithms work as expected, reaching a clear improvement, and yielding final bot's AI configurations which get the best scores in the matches against the standard bots.

This is our first approach to this problem so there are many future lines of work starting from this point. The first one is the implementation of some

different methods to evolve the AI bots, in order to compare the results with those yielded by the presented algorithms, and also perform some studies to find the best parameter setting for the current algorithms. Another task to address is the implementation of these algorithms inside a newer engine (as Unreal Tournament™), in order to avoid the constraints which obstruct a better problem definition and solving (such as limited arrays and number of iterations in loops).

The third line of improvement is related to the fitness function which is currently an aggregative function, so it could (or should) be separated into different functions, transforming the problem into a multi-objective one, closer to the real problem to address for getting a good bot's AI.

We also want to remark that this is a rather 'noisy' problem, were each of the individuals has a different value for the fitness function at every time, since it depends on many factors which continuously change in time and can be different between two evaluations for the same bot (i.e. the position of the bots while it has been playing, their weapons, the situation of the new weapons, or the position of our bot when it appears in the map), which complicate sometimes the evolution in the algorithms. So maybe a dynamical approach could yield better results.

The last ideas are related to the performance study of a co-evolutionary approach, since it is possible to put in action more than one G-Bot in an scenario. Following the same line, we would also like to implement a cooperative method, where the bots would be grouped into teams which fight between them to get the best results as a whole.

References

1. Booth, T.L.: Sequential Machines and Automata Theory, 1st edn. John Wiley and Sons, Inc., New York (1967)
2. Cho, B.H., Jung, S.H., Seong, Y.R., Oh, H.R.: Exploiting intelligence in fighting action games using neural networks. IEICE - Trans. Inf. Syst. E89-D(3), 1249–1256 (2006)
3. Goldberg, D.E.: Genetic Algorithms in search, optimization and machine learning. Addison-Wesley, Reading (1989)
4. Koza, J.R.: Genetic Programming: On the programming of computers by means of natural selection. MIT Press, Cambridge (1992)
5. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs, 3rd edn. Springer, Heidelberg (1996)
6. Priesterjahn, S., Kramer, O., Weimer, A., Goebels, A.: Evolution of human-competitive agents in modern computer games. In: IEEE Congress on Computational Intelligence, CEC 2006, pp. 777–784 (2006)
7. Small, R., Bates-Congdon, C.: Agent Smith: Towards an evolutionary rule-based agent for interactive dynamic games. In: IEEE Congress on Evolutionary Computation, CEC 2009, May 2009, pp. 660–666 (2009)
8. Soni, B., Hingston, P.: Bots trained to play like a human are more fun. In: IEEE International Joint Conference on Neural Networks, IJCNN 2008, IEEE World Congress on Computational Intelligence, pp. 363–369 (June 2008)
9. Wikipedia: Unreal — wikipedia, the free encyclopedia (2009), `http://en.wikipedia.org/wiki/Unreal`