Designing Competitive Bots for a Real Time Strategy Game using Genetic Programming

A. Fernández-Ares, P. García-Sánchez,
A.M. Mora, P.A. Castillo, and J.J. Merelo

Dept. of Computer Architecture and Computer Technology,
CITIC-UGR, University of Granada, Spain
antares@ugr.es,pablogarcia@ugr.es

Abstract. The design of the Artificial Intelligence (AI) engine for an autonomous agent (bot) in a game is always a difficult task mainly done by an expert human player, who has to transform his/her knowledge into a behavioural engine. This paper presents an approach for conducting this task by means of Genetic Programming (GP) application. This algorithm is applied to design decision trees to be used as bot's AI in 1 vs 1 battles inside the RTS game Planet Wars. Using this method it is possible to create rule-based systems defining decisions and actions, in an automatic way, completely different from a human designer doing them from scratch. These rules will be optimised along the algorithm run, considering the bot's performance during evaluation matches. As GP can generate and evolve behavioural rules not taken into account by an expert, the obtained bots could perform better than human-defined ones. Due to the difficulties when applying Computational Intelligence techniques in the videogames scope, such as noise factor in the evaluation functions, three different fitness approaches have been implemented and tested in this work. Two of them try to minimize this factor by considering additional dynamic information about the evaluation matches, rather than just the final result (the winner), as the other function does. In order to prove them, the best obtained agents have been compared with a previous bot, created by an expert player (from scratch) and then optimised by means of Genetic Algorithms. The experiments show that the three used fitness functions generate bots that outperform the optimized human-defined one, being the area-based fitness function the one that produces better results.

1 Introduction

Real-Time Strategy (RTS) games are a sub-genre of strategy-based videogames in which the contenders struggle to control a set of resources, units and structures that are distributed in a playing arena. A proper control and a sound strategy and tactics for handling these units is essential for winning the game, which happens after the game objective has been fulfilled, normally eliminating all enemy units, but sometimes also when certain points or game objectives have been reached.

Their main feature is their real-time nature (which is explicit in its denomination, real time strategy games), i.e. the player is not required to wait for the results of other players' moves as in turn-based games. Command and Conquer™, Starcraft™, Warcraft™ and Age of Empires™ are examples of RTS games.

Two levels of AI are usually considered in RTS games [1]: the first one, interpreted by a Non-Playing Character (NPC), which is also a bot, makes decisions over the whole set of units (workers, soldiers, machines, vehicles or even buildings); the second level is devoted implement the behaviour of every one of these small units. These two level of actions, which can be considered *strategic* and *tactical*, make them inherently difficult to be designed by a human; but this difficulty is increased by their real-time nature (usually addressed by constraining the time that each bot can use to make a decision) and also for the huge search space that is implicit in its action.

For these reasons, in this work a Genetic Programming (GP) approach is proposed as an automatic method to create the Artificial Intelligence (AI) engine of autonomous agents in a RTS. The objective of GP is to create functions or programs to solve determined problems, moreover the individual representation is usually in form of a tree, formed by operators (or *primitives*) and variables (*terminals*). The aim of using GP in this scope is the creation of behavioural rule-based engines following an heuristic, algorithmic and automatic process. Thus, instead of implementing them from scratch by a human (expert or not), this method will define a set of rules that could be more complex (or simpler) than those defined by the humans. In addition, this algorithm is able to evaluate every possible set of rules, assigning to that set a value according to the corresponding bot's performance (during battles). Thus, these sets will be improved (evolved) along the algorithm run in order to increase that bot's performance.

In order to implement and test this proposal, we have considered the game *Planet Wars*, a RTS which was presented under the Google AI Challenge 2010[1]. It has been used by several authors for the study of computational intelligence techniques in RTS games [2–4], due to it is a simplification (just one type of resource and one type of unit) of the elements that those commercial RTSs present. Thus, the players in this game just can manage planets and starships (or just ships). The aim is conquering the whole galaxy in the current map, against an enemy trying to do the same. The planets can produce new ships and the ships are destroyed one by one for both players when they crash.

Three different fitness functions will be presented and tested in this paper: the first one is a variation of the previously used *victory-based fitness* [2], which evaluates all the individuals in the population by playing five different matches (in five different maps) against a sparring bot. The aim of the repetitions is to avoid the noisy factor present in these dynamic environments [5, 6]. Due to this, the fitness value for an individual could dramatically vary between different matches, since it depends on the pseudo-stochastic opponent's actions, and also on its own non-deterministic decisions. The other two presented fitness functions also point to reduce the influence of noise in the evolution, but using additional

[1] `http://planetwars.aichallenge.org/`

data obtained during the execution of the bot. Thus, they consider the number of ships generated by each bot rather than the number of turns and compute, respectively, a *linear regression* (Slope) based on the percentage of ships with respect to the total, and *the integral* (Area) of the function which represents these numbers. All of them consider the final results of every individual (bot) after the aforementioned five matches (on average).

The work is then focused on proving the value of these evolved rule-based control systems for the agents. To this end, several experiments have been conducted, considering the aforementioned fitness functions, and an evolutionary bot as rival. This bot, called GeneBot, was presented in a previous work [2].

2 Background and problem description

2.1 Genetic Programming

Genetic Programming (GP) [7] is a kind of Evolutionary Algorithm (EA), that is, a probabilistic search and optimization algorithms gleaned from the model of darwinistic evolution, based on the idea that in nature structures undergo adaptation. EAs work on a population of possible solutions (individuals) for the target problem and use a selection method that favours better solutions and a set of operators that act upon the selected solutions. However, GP is a structural optimisation technique where the individuals are represented as hierarchical structures (typically tree) and the size and shape of the solutions are not defined a priori as in other methods from the field of evolutionary computation, but they evolve along the generations. So, the main difference with respect to GAs is the individual representation and the genetic operators to apply, which are mainly focused on the management (and improvement) of this kind of structure. The flow of a GP algorithm is the same as any other EA: a population is created at random, each individual in the population is evaluated using a fitness function, the individuals that performed better in the evaluation process have a higher probability of being selected as parents for the new population than the rest and a new population is created once the individuals are subjected to the genetic operators of crossover and mutation with a certain probability. The loop is run until a predefined termination criterion is met.

2.2 Planet Wars Game

In this paper we work with a version of the game Galcon, aimed at performing bot's fights which was used as base for the Google AI Challenge 2010 (GAIC)[2].

A Planet Wars match takes place on a map (see Fig. 1) that contains several planets (neutral, enemies or owned), each one of them with a number assigned to it that represents the quantity of ships that the planet is currently hosting.

The aim of the game is to defeat all the ships in the opponent's planets. Although Planet Wars is a RTS game, this implementation has transformed it

[2] http://ai-contest.com

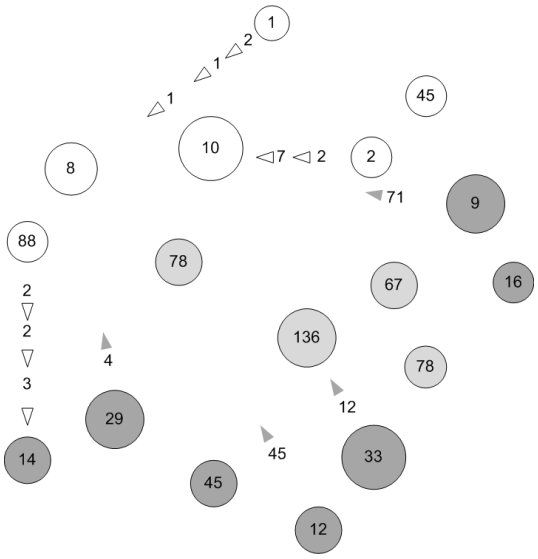

Fig. 1. Simulated screenshot of an early stage of a run in Planet Wars. White planets belong to the player (blue colour in the game), dark grey belong to the opponent (red in the game), and light grey planets belong to no player. The triangles are fleets, and the numbers (in planets and triangles) represent the ships. The planet size means growth rate of the amount of ships in it (the bigger, the higher).

into a turn-based game, in which each player has a maximum number of turns to accomplish the objective. At the end of the match, the winner is the player that remains alive, or that which owns more ships if more than one survives.

There are two strong constraints which determine the possible methods to apply to design a bot: a simulated turn takes *just one second*, and the bot is *not allowed to store any kind of information* about its former actions, about the opponent's actions or about the state of the game (i.e., the game's map).

Therefore, the aim in this paper is to study the improvement of a bot according to the state of the map in each simulated turn (input), returning a set of actions to perform in order to fight the enemy, conquering its resources, and, ultimately, wining the game.

3 State of the art

RTS games have been used extensively in the computational intelligence area (see [8] for a survey).

Among other techniques, Evolutionary Algorithms have been widely used as a Computational Intelligence method in RTS games [8]. For example, for parameter optimization [9], learning [10] or content generation [11].

One of these types, Genetic Programming, has been proved as a good tool for developing strategies in games, achieving results comparable to human, or human-based competitors [12]. They also have obtained higher ranking than solvers produced by other techniques or even beating high-ranking humans [13]. GP has also been used in different kind of games, such as board-games [14], or (in principle) simpler games such as Ms. Pac-Man [15] and Spoof [16] and even in modern video-games such as First Person Shothers (FPS) (for example, Unreal^{TM} [17]). With respect to RTS games, there are just a few applications on pathfinding [18] and definition of tactics in an abstract tactical game [19]. In this paper, the aim is to apply GP inside a modern RTS, in order to define the whole behavioural engine for an autonomous player (non-player character), trying to improve a rule-based system previously defined by a human expert.

Planet Wars, the game used in this work, has also been used in other researches as an experimental framework for agent testing. We have considered this game in some previous studies [20, 2, 21, 3, 22], mainly applying Genetic Algorithms for evolving (the parameters of) a behavioural engine previously defined by a human expert from scratch. Those works have respectively defined a first approach, compared different implementations, analysed the noise influence, defined expert bots, and implemented co-evolutionary approaches.

The present work means a new step in this research line, which tries to avoid the strict limitations that the initial bot had, i.e. since it was defined by a human expert, it had a fixed structure (a Finite State Machine) which just offers a few degrees of improvement, namely a set of eight parameters. The use of GP here will provide us with a new tool for completely redefine the bot's AI engine, which could also get better results than previous bots. Thus GP has been applied to create the Decision Tree that the bot will use to make decisions during the game. In order to prove the method value, the resulting agents will be compared with a competitive bot previously presented: GeneBot [2], our initial bot improved by means of Genetic Algorithms. This bot also proved (in that work) to be better than a human-defined bot (AresBot).

4 GPBot

The Genetic Programming-based bot or *GPBot* evolves a set of rules which, in turn, models a Decision Tree. During the evolution, every individual in the population (a tree) must be evaluated. To do so, the tree is set as the behavioural engine of an agent, which is then placed in a map against a rival in a Planet Wars match. Depending on the obtained results, the agent (i.e. the individual) gets a fitness value, that will be considered in the evolutionary process as a measure of its validity.

Thus, during the match the tree will be used (by the bot) in order to select the best strategy at every moment, i.e. for every planet a target will be selected along with the number of ships to send from one the other.
The used Decision Trees are binary trees of expressions composed by two different *types of nodes*:

- *Decision*: a logical expression formed by a variable, a less than operator (<), and a number between 0 and 1. It is the equivalent to a "primitive" in the field of GP.
- *Action*: a leave of the tree (therefore, a "terminal"). Each decision is the name of the method to call from the planet that executes the tree. This method indicates to which planet send a percentage of available ships (from 0 to 1).

The decisions are based in the values of different *variables* which are computed considering some other variables in the game. They are defined by a human expert, and are:

- *myShipsEnemyRatio*: Ratio between the player's ships and enemy's ships.
- *myShipsLandedFlyingRatio*: Ratio between the player's landed and flying ships.
- *myPlanetsEnemyRatio*: Ratio between the number of player's planets and the enemy's ones.
- *myPlanetsTotalRatio*: Ratio between the number of player's planet and total planets (neutrals and enemy included).
- *actualMyShipsRatio*: Ratio between the number of ships in the specific planet that evaluates the tree and player's total ships.
- *actualLandedFlyingRatio*: Ratio between the number of ships landed and flying from the specific planet that evaluates the tree and player's total ships.

Finally, the possible *decisions* are:

- *Attack Nearest (Neutral—Enemy—NotMy) Planet*: The objective is the nearest planet.
- *Attack Weakest (Neutral—Enemy—NotMy) Planet*: The objective is the planet with less ships.
- *Attack Wealthiest (Neutral—Enemy—NotMy) Planet*: The objective is the planet with higher lower rate.
- *Attack Beneficial (Neutral—Enemy—NotMy) Planet*: The objective is the more beneficial planet, that is, the one with highest growth rate divided by the number of ships.
- *Attack Quickest (Neutral—Enemy—NotMy) Planet*: The objective is the planet easier to be conquered: the lowest product between the distance from the planet that executes the tree and the number of ships in the objective planet.
- *Attack (Neutral—Enemy—NotMy) Base*: The objective is the planet with more ships (that is, the base).
- *Attack Random Planet.*
- *Reinforce Nearest Planet*: Reinforce the nearest player's planet to the planet that executes the tree.
- *Reinforce Base*: Reinforce the player's planet with higher number of ships.

– *Reinforce Wealthiest Planet*: Reinforce the player's planet with higher grown rate.
– *Do nothing*.

An example of a possible decision tree is shown below. This example tree has a total of 5 nodes, with 2 decisions and 3 actions, and a depth of 3 levels.

```
if(myShipsLandedFlyingRatio < 0.796)
   if(actualMyShipsRatio < 0.201)
      attackWeakestNeutralPlanet(0.481);
   else
      attackNearestEnemyPlanet(0.913);
else
   attackNearestEnemyPlanet(0.819);
```

The bot's behaviour is explained in Algorithm 1.

Algorithm 1 Pseudocode of the proposed agent. The same tree is used during all the agent's execution

```
// At the beginning of the execution the agent receives the tree
tree ← readTree()
while game not finished do
  // starts the turn
  calculateGlobalPlanets() // e.g. Base or Enemy Base
  calculateGlobalRatios() // e.g. myPlanetsEnemyRatio
  for Each p in PlayerPlanets do
    calculateLocalPlanets(p) // e.g. NearestNeutralPlanet to p
    calculateLocalRatios(p) //e.g actualMyShipsRatio
    executeTree(p,tree) // Send a percentage of ships to destination
  end for
end while
```

Next section explains one of the main components of the evolutionary process, i.e. the fitness function. As previously stated, three different functions have been implemented, which are used to evaluate the agent's performance during the matches.

5 Fitness Functions

5.1 Fitness based in Victories

This a variation of the hierarchical fitness considered in [2]. In this approach, an individual is better than another if it wins in a higher number of maps. In case of equality of victories, then the individual with more turns to be defeated (i.e.

the stronger one) is considered as better. The maximum fitness in this work is, therefore, 5 victories and 0 turns. For two bots, A and B, the fitness comparison (and therefore, their order inside the population) is defined as Algorithm 2 shows.

Algorithm 2 Comparison between two individuals using hierarchical fitness.

```
A, B ∈ Population
if A.victories = B.victories then
  if A.turns >= B.turns then
    A is better than B
  else
    B is better than A
  end if
else
  if A.victories > B.victories then
    A is better than B
  else
    B is better than A
  end if
end if
```

In this fitness, we are only interested in the final result (position and number of turns). We do not include in the analysis how the bot has reached them. The problem of this function is that the consideration of two different terms makes it difficult the comparison between different evaluations.

Thus, in this work two additional evaluation functions have been proposed in order to let easier and fairer comparison methods between bots, trying to add another factor in order to reduce the influence of noise [5]. Both of them are based in the percentage of ships belonging to each player in every turn. They are normalized considering the total amount of ships in the game for that turn (including neutrals ships in neutral planets), so for each player, there is a different *cloud* of ships. Below, are described the two alternatives to deal with this cloud of points for the fitness function: the use of slopes and areas.

5.2 Fitness based in Slope

In this case, a square regression analysis is computed in order to transform the cloud of points into a simple line. The line is represented as $y = \alpha \times x + \beta$, where α and β are calculated as shown in Equations 1 and 2, computing a least squares regression. For every bot in the simulation we calculate α and (*slope*). This *slope* is the fitness of every bot for that simulation.

$$\alpha = \frac{\sum_{i=1}^{n}(X_i - \bar{X}_i)(Y_i - \bar{Y}_i)}{\sum_{i=1}^{n}(X_i - \bar{X}_i)^2} \tag{1}$$

$$\beta = \bar{Y} - \alpha\bar{X} \tag{2}$$

Theoretical maximum and minimum values are set for this fitness. An optimum bot that wins in the first turn, has an ideal slope of 1, so this is the maximum value of our fitness. On the other hand, a bot that loses in the first turn, has a slope of -1. Thus, if we calculate the *slope*, we know if the bot $WINs$ ($slope > 0$) or $LOSEs$ $slope < 0$. The values of the different battles are summed to compute the global *slope*. Then, the bot with the highest value will be the best is each turn or battle.

5.3 Fitness based in Area

In this function, the integral of the curve of the bot's live-line is used for calculating the area that is 'covered' by the fitness cloud of points (see Equation 3). This *area* is normalized considering the number of turns, and thus it represents the average percentage of ships during the battle for each player.

$$area = \frac{\int_0^t \%ships(x)dx}{t} \tag{3}$$

As in previous case, maximum and minimum values has been set for this fitness. If an optimal bot wins in the first turn, the area of each live-line is close to 1, so this is the maximum value of the fitness. Otherwise, if a bot loses in the first turn, its live-line area is close to 0. In this case, we do not extract additional about which bot wins the battle, because the area of the live-line is not related with the winner of the battle. Thus, we are losing some information.

6 Experimental Setup

Sub-tree crossover and 1-node mutation evolutionary operators have been used, following other researchers' proposals that have used these operators obtaining good results [17]. In this case, the mutation randomly changes the decision of a node or mutate the value with a step-size of 0.25 (an adequate value empirically tested). Each configuration is executed 30 times, with a population of 32 individuals and a 2-tournament selector for a pool of 16 parents.

To test each individual during the evolution, a battle with a previously created bot is performed in 5 different (but representative) maps provided by Google is played. Also, as proposed by [2], and due to the noisy fitness effect, all individuals are re-evaluated in every generation.

A publicly available bot has been chosen for our experiments[3]. The bot to confront is *GeneBot*, proposed in [2]. As stated, this bot was trained using a GA to optimize the 8 parameters that conforms a set of hand-made rules, obtained from an expert human player experience. Table 1 summarizes all the parameters used.

After all the executions we have evaluated the obtained best individuals in all runs confronting to the other bots in a larger set of maps to study the behaviour

[3] It can be downloaded from `https://github.com/deantares/genebot`

Parameter Name	*Value*
Population size	32
Crossover type	Sub-tree crossover
Crossover rate	0.5
Mutation	1-node mutation
Mutation step-size	0.25
Selection	2-tournament
Replacement	Steady-state
Stop criterion	50 generations
Maximum Tree Depth	7
Runs per configuration	30
Evaluation	Playing versus GeneBot [2]
Maps used in each evaluation	map76, map69, map7, map11, map26

Table 1. Parameters used in the experiments.

of the algorithm and how good are the obtained bots versus enemies and maps that have not been used for training.

The used framework is OSGiLiath, a service-oriented evolutionary framework. The generated tree is compiled in real-time and injected in the agent's code using Javassist [4] library. All the source code used in this work is available under a LGPL V3 License in `http://www.osgiliath.org`.

7 Results

Table 2 shows the obtained results after executing 20 times each approach, i.e. GP algorithm using every fitness implementation. Although these fitness are not comparable, as they obviously apply different metrics, the *Victory-based* fitness achieves values near to the optimum (5) at the end of the run (look at the best individual and average population values). The *Slope* and *Area* fitness yield results under their theoretical optimum, as they depend on more information ranges (variation in the number of ships). *Area* obtains slightly better values than *Slope*.

	Average best fitness	Average population fitness
Victory	4,761 ± 0,624	4,345 ± 0,78
Slope	2,296 ± 0,486	2,103 ± 0,486
Area	2,838 ± 1,198	2,347 ± 0,949

Table 2. Average results obtained for each approach at the end of the runs. Everyone has been tested 20 times.

[4] `www.javassist.org`

Even though the *Victory-based* fitness yields better results (near optimal), to do a fair comparison, we have confronted the 20 best bots obtained with each configuration (one per run) against GeneBot. However these matches have been performed in 9 different maps than those where the bots were trained (evolved). These maps, provided by Google, are considered as representative, because they have different features to promote a wide set of strategies, i.e. different distributions of planets, sizes and number of initial ships.

This experiment has been conducted in order to validate if the bots obtained by the proposed approaches can be competitive in terms of quality in maps not used for evolution/evaluation. Results are shown in Figure 2. As it can be seen, again the *Victory-based* fitness achieves significantly better results than the other methods.

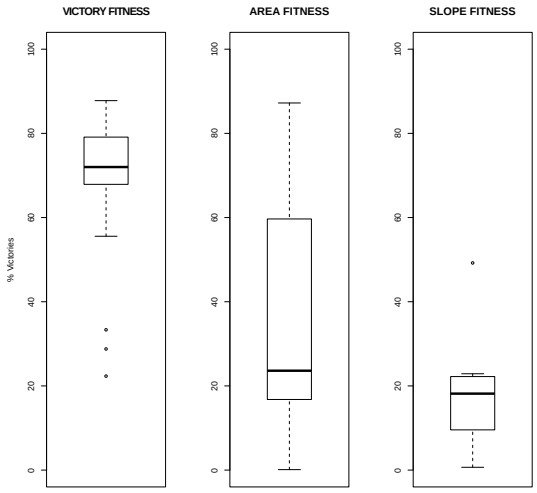

Fig. 2. Boxplot of average percentage of victories of the bots obtained by each approach vs. GeneBot. 9 different matches have been performed per bot in different maps.

Finally, an additional experiment has been conducted, proposing a direct comparison between the three methods. To this end each of the best individuals obtained per approach has been tested competing against all the rest (in a vs 1 battles) in 9 matches per pair of bots, one per representative map.

This allows a comparison with a wider number of bots, and also, allows the analysis of their behavior against rivals not previously used during training (as in the experiment above). The boxplots of the average percentage of victories from the best bots obtained by each method are shown in Figure 3. It is clear from the image that the *Slope* fitness does not get good results with respect to the other methods. This can be explained because the this fitness loses information during the run, in comparison with the others, obtaining bots less aggressive.

The *Victory-based* fitness achieves better results in average, being also more robust (small standard deviation). However, looking at the *Area* fitness results, they outperform several times the *Victory* results, obtaining more victories.

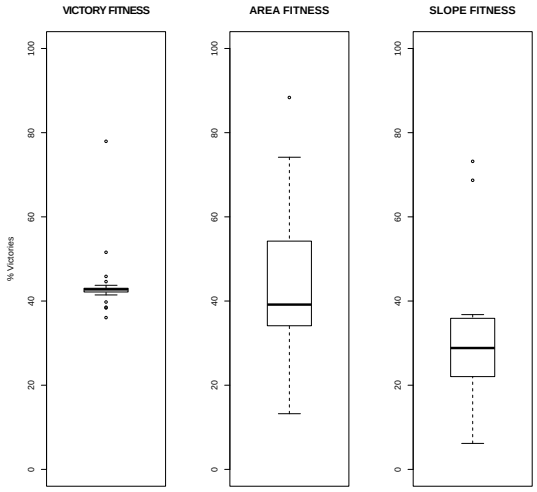

Fig. 3. Boxplot of average percentage of victories of the best bots obtained by each method vs. the rest.

There is still a final remark, which concerns to the percentage of draw matches. As it can be seen in Table 3. The *Victory-based* fitness achieves more draws against bots of the same type than the other approaches. This can be explained because they use less information to perform the evolution, obtaining quite similar behaviours. This is also reinforced by previous results in which the bots of this fitness seemed to perform similarly, obtaining close number of victories and thus, getting small standard deviation values.

	Victory	Area	Slope
Victory	62.06%	23.15 %	12.47 %
Slope	-	19.07 %	12.50 %
Area	-	-	11.87 %

Table 3. Percentage of draw matches between bots per fitness approach.

8 Conclusions

The objective of this work is to validate if using Genetic Programming can create competitive bots for RTS games, and study the behaviour of different fitness functions, as they can affect directly to the creation of these bots. Three different fitness functions have been compared to generate bots for the Planet Wars game. A competitive bot available in the literature (GeneBot) has been used to evaluate the generated individuals (fighting against it). This bot was the best one obtained in an evolutionary process which optimized different parameters inside a human-designed behavioural engine.

Different information of the run of the game is taken into account in these functions to obtain a metric to guide the evolution. The results show differences depending on the fitness used: a victory-based fitness that prioritizes the number of victories generate better bots in average than fitness that take into account the number of spaceships during all the run of the battle. This can be explained because this fitness exploits the individuals to generate more aggressive bots. However, their performance decreases confronting versus different types of bots.

In future work, other rules will be added to the proposed algorithm (for example, ones analysing the map) and more competitive enemies will be used. In addition, the approach could be implemented and tested in more complex RTS games, such as Starcraft, or even in different videogames like Unreal^{TM} or Super Mario^{TM} .

Acknowledgements

This work has been supported in part by FPU research grant AP2009-2942 and projects SIPESCA (G-GI3000/IDIF, under Programa Operativo FEDER de Andalucía 2007-2013), CANUBE (CEI2013-P-14), ANYSELF (TIN2011-28627-C04-02) and PYR-2014-17 included in GENIL - CEI BIOTIC (Granada).

References

1. Ahlquist, J.B., Novak, J.: Game Artificial Intelligence. Game Development Essentials. Thompson Learning, Canada (2008)
2. Fernández-Ares, A., Mora, A.M., Merelo, J.J., García-Sánchez, P., Fernandes, C.: Optimizing player behavior in a real-time strategy game using evolutionary algorithms. In: IEEE Congress on Evolutionary Computation (CEC 2011). (2011) 2017–2024
3. Fernández-Ares, A., García-Sánchez, P., Mora, A.M., Merelo, J.J.: Adaptive bots for real-time strategy games via map characterization. In: 2012 IEEE Conference on Computational Intelligence and Games, CIG 2012, IEEE (2012) 417–721
4. Lara-Cabrera, R., Cotta, C., Leiva, A.J.F.: On balance and dynamism in procedural content generation with self-adaptive evolutionary algorithms. Natural Computing **13**(2) (2014) 157–168

5. Mora, A.M., Fernández-Ares, A., Merelo, J.J., García-Sánchez, P., Fernandes, C.M.: Effect of noisy fitness in real-time strategy games player behaviour optimisation using evolutionary algorithms. J. Comput. Sci. Technol. **27**(5) (2012) 1007–1023
6. Guervós;, J.J.M.: Using a wilcoxon-test based partial order for selection in evolutionary algorithms with noisy fitness. Technical report, GeNeura group, university of Granada (2014)
7. Koza, J.R.: Genetic Programming: On the programming of computers by means of natural selection. MIT Press, Cambridge, MA (1992)
8. Lara-Cabrera, R., Cotta, C., Fernández-Leiva, A.J.: A review of computational intelligence in rts games. In: FOCI, IEEE (2013) 114–121
9. Esparcia-Alcázar, A.I., Martínez-García, A.I., Mora, A.M., Merelo, J.J., García-Sánchez, P.: Controlling bots in a first person shooter game using genetic algorithms. In: IEEE Congress on Evolutionary Computation, IEEE (2010) 1–8
10. Stanley, K.O., Bryant, B.D., Miikkulainen, R.: Real-time neuroevolution in the nero video game. IEEE Transactions on Evolutionary Computation (2005) 653–668
11. Mahlmann, T., Togelius, J., Yannakakis, G.N.: Spicing up map generation. In: Proceedings of the 2012t European conference on Applications of Evolutionary Computation. EvoApplications'12, Berlin, Heidelberg, Springer-Verlag (2012) 224–233
12. Sipper, M., Azaria, Y., Hauptman, A., Shichel, Y.: Designing an evolutionary strategizing machine for game playing and beyond. IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews **37**(4) (2007) 583–593
13. Elyasaf, A., Hauptman, A., Sipper, M.: Evolutionary design of freecell solvers. IEEE Transactions on Computational Intelligence and AI in Games **4**(4) (2012) 270–281
14. Benbassat, A., Sipper, M.: Evolving both search and strategy for reversi players using genetic programming. (2012) 47–54
15. Brandstetter, M., Ahmadi, S.: Reactive control of ms. pac man using information retrieval based on genetic programming. (2012) 250–256
16. Wittkamp, M., Barone, L., While, L.: A comparison of genetic programming and look-up table learning for the game of spoof. (2007) 63–71
17. Esparcia-Alcázar, A.I., Moravec, J.: Fitness approximation for bot evolution in genetic programming. Soft Computing **17**(8) (2013) 1479–1487
18. Strom, R.: Evolving pathfinding algorithms using genetic programming. Gamasutra Webpage (Accessed on 25/05/2014) URL=http://www.gamasutra.com/view/feature/131147/evolving_pathfinding_algorithms_.php?print=1.
19. Keaveney, D., O'Riordan, C.: Evolving robust strategies for an abstract real-time strategy game. In Lanzi, P.L., ed.: CIG, IEEE (2009) 371–378
20. Fernández-Ares, A., Mora, A.M., Merelo, J.J., García-Sánchez, P., Fernandes, C.M.: Optimizing strategy parameters in a game bot. In: Proc. 11th International Work-Conference on Artificial Neural Networks, IWANN 2011, Springer, LNCS, vol. 6692 (2011) 325–332
21. Mora, A.M., Fernández-Ares, A., Merelo, J.J., García-Sánchez, P.: Dealing with noisy fitness in the design of a RTS game bot. In: Proc. Applications of Evolutionary Computing: EvoApplications 2012, Springer, LNCS, vol. 7248 (2012) 234–244
22. Fernández-Ares, A.J., Mora, A.M., Arenas, M.G., García-Sánchez, P., Merelo, J.J., Castillo, P.A.: Co-evolutionary optimization of autonomous agents in a real-time strategy game. In: Proc. Applications of Evolutionary Computing: EvoApplications 2014, Springer (2014) In Press