. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY : 1– January 2012

Effect of noisy fitness in RTS Player Behaviour Optimisation Using Evolutionary Algorithms

Abstract This paper describes an Evolutionary Algorithm (EA) for evolving the decision engine of a program (which, in this context, is called *bot*) designed to play the Planet Wars game. This game, which was chosen for the Google Artificial Intelligence Challenge in 2010, requires the bot to deal with multiple target planets, while achieving a certain degree of adaptability in order to defeat different opponents in different scenarios. The decision engine of the bot is based on a set of rules that have been defined after an empirical study. A Genetic Algorithm (GA) is then used for tuning the set of constants, weights and probabilities that define the rules, and, therefore, the general behaviour of the bot. This paper describes the GA and analyses in depth the results obtained by the bot using the evolved decision engine when competing with other bots, which include a bot offered by Google as a sparring partner, and a scripted bot (it has a pre-established behaviour, independent of inputs). The evaluation of the individuals is based on the result of some non-deterministic combats, whose outcome depends on random draws as well as the enemy action, and is thus noisy. This noisy fitness is addressed in the GA and its effects are then analysed in depth in the experimental section. From these experiments we conclude that tackling randomness via repeated combats and reevaluations reduces the effect of the noisy fitness and makes the GA a highly valuable approach for solving this problem. The described bot defeated the baseline bot in most game environments and obtained a ranking position in top-30% of the Google Artificial Intelligence competition.

Keywords Real-Time Strategy Games, Evolutionary Algorithms, Genetic Algorithms, Noisy Fitness, Player Behaviour Optimisation, Parameter Evolving, Artificial Intelligence, Game Bots

1 Introduction and Problem Description

Bots [33] are autonomous agents that interact with a human user or with other bots within a computer. Amongst other applications, bots are one of the most important elements in modern video games; within these environments they run automated tasks for competing or cooperating with the human player in order to increase the challenge of the game, thus making their *intelligence* one of the fundamental parameters in the video game design [17].

Bots apply Artificial Intelligence (AI) tools and techniques and are used as enemies or teammates in several types of videogames, such as strategy, fight, racing or platform games, but they are more commonly within the First Person Shooter (FPS) games scope [9, 25, 29], where they are usually designed as opponents of the human player. However, this paper deals with Real-Time Strategy (RTS) games, which are a sub-genre of strategy-based video games in which the contenders control a set of 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: After eliminating all enemy units, obviously, but also when certain points or game objectives have been reached.

A typical RTS gives the player the possibility to create additional units and structures during the course of the game, usually at a cost

in resources that must be gathered via the creation or exploitation of those structures. For instance a game will feature *mines* from where *gold* can be extracted and then used to create *barracks* from where new units are *built*. Another usual feature is their real time nature, 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 some examples of RTS games.

RTS games often employ two levels of AI [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; 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.

Such difficulties are probably one of the reasons why Google chose this kind of games for their AI Challenge 2010 [11]. In this contest, real time is sliced in one second *turns*, with players receiving the chance to play sequentially. However, *actions* happen at the *simulated* same time, thus becoming a trait of the particular implementation and not a feature of the game itself.

This paper describes an evolutionary approach for generating the decision engine of a bot that plays *Planet Wars* (or Galcon [34]), the RTS game that was chosen for the above referred competition. The decision engine was implemented in two steps: first, a set of parametrised rules that model the behaviour of the bot was defined by a human player (after playing and analysing several matches); the second step of the process applied a Genetic Algorithm (GA) [8] for evolving these parameters offline (i.e., not during the match, but prior to the game battles).

Evolutionary Algorithms (EAs) are a class of probabilistic search and optimisation algorithms gleaned from the model of darwinistic evolution [3]. EAs include several subtypes, depending on the data structure that is preferently used for representing solutions - GAs, Evolution Strategies and Genetic Programming to name a few - but the main features are common to all of them: a population of possible solutions (individuals) of the target problem, a selection method that favours better solutions and a set of operators that act upon the selected solutions. After an initial population is created (usually randomly), the selection and operators are successively applied to the individuals in order to create new populations that replace the older one. This process guarantees that the average quality of the individuals tends to increase with the number of generations. Eventually, depending on the type of problem and on the efficiency of the EA, the optimal solution may be found.

Given the characteristics and capabilities of EAs, we propose a real-number GA for optimising the variables that tune rules which model the bot's behaviour.

The evaluation of the quality (fitness) of each set of rules in the population is made by playing the bot against predefined opponents, being a pseudo-stochastic or noisy function, since the results for the same individual evaluation may change from time to time, yielding good or bad values depending on the battle events and on the opponent's actions (their behaviour, and even our bot's is non-deterministic).

In the experiments, we show first that the set of rules evolve towards better bots; finally, an efficient player is returned by the GA. Several experiments have been conducted to analyse the issue of the cited *noisy fitness* in this

problem. The experiments show its presence, but also the good behaviour of the implemented mechanisms to deal with it: reevaluation of all the individuals (even those who remain between generations), and evaluation considering five matches (instead of just one) in five different and representative maps (defined to model a huge range of possible combats). So the algorithm yields good individuals even in these conditions.

The paper is structured as follows: The following section reviews related approaches to behavioural engine design in similar game-based problems. Section 3 addresses the problem by describing the Planet Wars game. A brief introduction to GAs is presented in Section 4. Section 5 presents the method, termed GeneBot, starting from the initial approach, detailing the finite state machine and the parameters which model its behaviour, and then the GA used to evolve the strategies. The experiments and results are described and discussed in Section 6. Finally, the conclusions and future lines of research are presented in Section 7.

2 State of the Art

Video games have become one of the biggest sectors in the leisure industry; some games cost more to develop than blockbuster movies. Most of the companies concentrate their investments in increasing the graphical quality of the games, as a measure to ensure a best-seller product, instead of innovating in challenging the player skills. Nowadays, computers have a higher processing power and the user is hardly astonished by the graphical component of a game. So, the players have turned their attention to other aspects of the game. In particular, they mostly request opponents exhibiting intelligent behaviour, or just better human-like behaviours [18].

During these years, the games AI research has followed a different path, mainly starting with the improvement of FPS Bot's AI with Doom™ or Quake™ [17] by the beginning of the 90s; and the most famous environment inside this kind of games, Unreal Tournament ™ [29, 9, 25].

Most of the research has been done on relatively simple games such as Super Mario [32], Pac-Man [21] or Car Racing Games [26], being many competitions devoted to choose the best bot playing in each of them, but also for more complex games such as the included in the scope of the present work, the RTS competitions such as the Starcraft AI Competition [35].

RTS games show an emergent component [31] as a consequence of the cited two level AI, since the units behave in many different (and sometimes unpredictable) ways. This feature can make a RTS game more entertaining for a player, and maybe more interesting for a researcher. There are many research problems with regard to the AI for RTSs, including planning in an uncertain world with incomplete information; learning; opponent modelling and spatial and temporal reasoning [6, 14].

However, the reality in the industry is that in most of the RTS games, the NPC (bot) is basically controlled by a fixed script (pre-established behaviour independent of inputs) that has been previously programmed (following a finite state machines or a decision tree, for instance). Once the user has learnt how such a game will react, the game quickly loses its appeal. In order to improve the users' gaming experience, some authors such as Falke et al. [10] proposed a learning classifier system that can be used to equip the computer with dynamically-changing strategies that respond to the user's strategies, thus greatly extending the games playability.

In addition, in many RTS games, traditional artificial intelligence techniques fail to play at a human level because of the vast search

spaces that they entail. In this sense, Ontano et at. [27] proposed to extract behavioural knowledge from expert demonstrations in form of individual cases. This knowledge could be reused via a case based behaviour generator that proposed advanced behaviours to achieve specific goals.

So, recently a number of soft-computing techniques and algorithms, such as co-evolutionary algorithms [24, 5, 16] or multi-agent based methods [13], just to cite a few, have already been applied to handle these problems in the implementation of RTS games. For instance, there are many benefits attempting to build adaptive learning AI systems which may exist at multiple levels of the game hierarchy, and which co-evolve over time. In these cases, co-evolving strategies might be not only opponents but also partners operating at different levels [19]. Other authors propose using co-evolution for evolving team tactics [2]. However, the problem is how tactics are constrained and parametrised and how the overall score is computed.

Evolutionary algorithms have also been used in this field [28, 15], but they involve considerable computational cost and thus are not frequently used in on-line games. In fact, the most successful proposals for using EAs in games correspond to off-line applications [30], that is, the EA works (for instance, to improve the operational rules that guide the bot's actions) while the game is not being played, and the results or improvements can be used later during the game. Through offline evolutionary learning, the quality of bots' intelligence in commercial games can be improved, and this has been proven to be more effective than opponent-based scripts.

This way, in this work, an offline GA is applied to a parametrised tactic (set of behaviour model rules) inside the Planet Wars game (a "simple" RTS game), in order to build the decision engine of a bot for that game, which will be considered later in the online matches.

3 The Planet Wars Game

The Google AI Challenge (GAIC) [11] is an AI competition in which the participants create bots to compete against each other. The game chosen for last year's competition, Planet Wars, is the object of the study presented in this paper. For this game, we propose to design the behavioural engine of a bot and GAs to optimise its efficiency. Planet Wars is a simplified version of the game Galcon [34], aimed at performing bot's fights. The Google's contest version of the game involves two players.

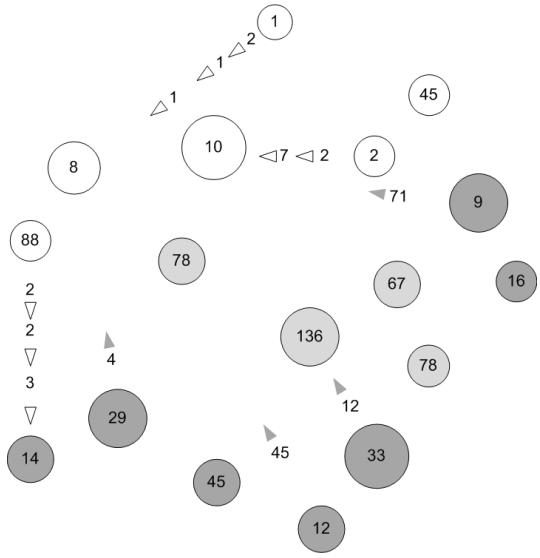


Fig. 1. Simulated screen shot of an early stage of a run in Planet Wars. White planets belong to the player (blue colour in the game), dark grey belong to the opponent (red in the game), and light grey planets belong to no player. The triangles are fleets, and the numbers (in planets and triangles) represent the starships.

A Planet Wars match takes place on a map that contains several planets, each one of them with a number assigned to it that represents the quantity of starships that the planet is currently hosting (see Figure 1). At a given

time step, each planet hosts a specific quantity of starships that may belong to the player, the opponent or may be neutral (i.e., they belong to no player). Ownership is represented by a colour, being blue the colour assigned to the player, red to the enemy and grey to neutral starships. In addition, each planet has a growth rate that indicates how many starships are generated during each round action and then added to the fleet of the player that owns the planet.

The objective of the game is to defeat all the opponent's planets. Although Planet Wars is a RTS game, this implementation has transformed it 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 (after 200 actions, in Google's Challenge), the winner is the player owning more starships; if a player is completely eliminated before that, the other wins, obviously.

Each planet has some properties: *X and Y Coordinates*, *Owner's PlayerID*, *Number of Starships* and *Growth Rate*. Players send fleets to conquer other planets (or to reinforce its own), and every *fleet* also has a set of properties: *Owner's PlayerID*, *Number of Starships*, *Source PlanetID*, *Destination PlanetID*, *Total Trip Length*, and *Number of turns remaining until arrival*. A simulated turn is one-second long. This is the maximum time for a bot to perform its actions.

Another singularity of the problem is that 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) to use them in the next turn. This constraint is defined by the problem features and the competition rules. Following them, each bot is invoked in every turn independently, so the only way of preserving knowledge would be writing it in a file. However it is strictly forbidden by the competition rules. In short, every one-second time step, the bot must deal with an unknown map, as if it was a new game. This constraint makes the development of the bot an interesting challenge, since the impossibility of knowing in advance which actions have been successful or not in the past, or learning from the other player actions, makes it impossible to apply complex method that need a training period and learning from examples.

In fact, each autonomous bot is implemented as a function that takes as input the list of planets and fleets (the current status of the game), each one with its features' values, and outputs a text file with actions to perform. In each simulated turn, the player must choose where to send fleets of starships, departing from one of the player's planets, towards other planet in the map. This is the only action that the bot is allowed to perform. The fleets may need more than one time step to reach destination. When a fleet reaches a planet, it fights against the enemy's (or neutral) forces assigned to that planet (losing one starship for each one of opponent's starships on the planet) and, in the case its fleet outnumbers the enemy's units, the player takes control of the planet with the remaining ships. If the planet already belongs to the player, then the incoming units are added to the current fleet. In each time-step, the forces in the planets owned by the players (i.e., every planet except the neutral ones) are increased according to each planet's growth rate.

Therefore, the goal is to initially design and then evolve a function that, according to the state of the map in each simulated turn (input) returns a set of actions to perform in order to fight the enemy, conquer its resources, and, ultimately, win the game.

The above-referred constraints of the Google Challenge (a bot is not allowed to store information and each time-step is limited to one second) make it difficult to implement an on-line approach, that is, an evolutionary algorithm that is started and yields a result at

each turn. Therefore, the described EA (a GA, in this case), is executed before the game and, once a satisfactory bot is found, the solution (the bot's set of rules and parameter values) is considered to play the game.

4 Evolutionary Algorithms: Genetic Algorithms

A Genetic Algorithm is a type of Evolutionary Algorithm which evolves a population of candidate solutions to a problem towards optimal (local or global) points of the search space by recombining parts of the solutions to generate a new population. The decision variables of the problem are encoded in strings or vectors with a certain length and cardinality. In GAs' terminology, these strings are referred to as *chromosomes*, each string position is a *gene* and its values are the *alleles*. The alleles may be binary, integer, real-valued, etc, depending on the codification (which in turn may depend on the type of problem).

The "best" parts of the chromosomes (or building-blocks) are guaranteed to spread across the population by a selection mechanism that favours better (or fitter) solutions. The quality of the solutions is evaluated by computing the fitness values of the chromosomes; this fitness function is usually the only information given to the GA about the problem.

A standard GA's procedure goes as follows: First, a population of chromosomes is randomly generated. All the chromosomes in the population are then evaluated according to the fitness function. A pool of parents (or mating pool) is selected by a method that guarantees that fitter individuals have more chances of being in the pool - tournament selection [8], fitness proportionate selection, also known as roulette-wheel selection [7] and stochastic universal sampling [3] are just some of the possible selection methods. Then a new population is generated by recombining the genes in the parents' population. This is usually done with a *crossover operator* (1-point crossover, uniform crossover, or BLX-α (for real-coded chromosomes), amongst many proposals that can be found in Evolutionary Computation literature) that recombines the genes of two parents and generates two offspring according to a crossover probability p_c that is typically set to values between 0.6 and 1.0 (if the parents are not recombined, they are copied to the offspring population).

After the offspring population is complete, the new chromosomes are mutated before being evaluated by the fitness function. *Mutation* operates at gene level, randomly changing the allele with a very low probability p_m (for instance, p_m is usually set to $1/l$ in many binary GAs, being l the chromosome length).

Once the evaluation of the newly generated population has been performed, the algorithm starts the *replacement of the old population*. There are several techniques for replacement, that is, for combining the offspring population with the old population in order to create the new population. Generational replacement, for instance, replaces the old population by the offspring. A steady-state strategy will replace a fraction (typically, two individuals) of the old population by the best individuals in the offspring population. Sometimes, an $e-elitism$ strategy is used, i.e., the best e chromosomes from the old population are copied without mutation to the new population. The remaining individuals are selected according to any method.

This process goes on until a stop criterion is met. Then, the best individual in the population is retrieved as a possible solution to the problem. Algorithm 1 shows the pseudo-code of a standard GA.

Algorithm 1 sGA()
Standard Genetic Algorithm pseudo-code

```
Initialize Population(P)
while not termination condition do
  Select P' individuals from P
  Recombine individuals P' to generate offspring population O
  Mutate individuals in O
  Evaluate population O
  Replace all (or some) individual in P by those in O
end while
```

Population size is the main bottleneck for a GA. One the most important issues that must be addressed when starting to design or tune a GA is to ensure an adequate supply of raw building-blocks. That is, if one supplies it with an initial population that is too small, the algorithm will converge very often to local optima; too big and the computational effort increases beyond indispensable. Besides population, the GA practitioner must tune a few parameters more: mutation probability, crossover probability, selective pressure, and others, depending on the type of the GA. However, a simple GA with these parameters is very efficient on a wide range of problems. Another important feature to set is the evaluation (or fitness) function, which is usually determined depending on the problem properties, but which should be analysed and well-defined in order to get a good indicator of the individuals' quality. It is usually easy to define if the problem has a deterministic nature, but sometimes it could be difficult when there is any stochastic component in the evaluation.

In this paper, we propose a GA with real-coded individuals, and crossover and mutation operators adapted to this codification. The population size has been set after systematic experimentation; moreover, the fitness function have been defined having in mind the noisy nature of the problem evaluation function, which consists in perform battles between the individual to evaluate and a sparring enemy, since it depends on the non-deterministic opponent's behaviour. This function tries to avoid this effect in order to search find good individuals in any case.

5 GeneBot, A Genetic Approach for Winning the Planet Wars Game

In Section 3 we explained that the main constraint in the environment is the limited processing time available to perform the correspondent actions (1 second). In addition, another important constraint stated that no memory is allowed, that is, the bot cannot maintain a register of the results or efficiency of previous actions. These restrictions strongly limit the design and implementation possibilities for a bot, since many metaheuristics are based on a memory of solutions or on the assignment of payoffs to previous actions in order to improve future behaviour, and most of them are quite expensive in running time; running an EA in each time-step of 1 second, for instance, or a Monte Carlo method [20], is almost impossible. Besides, only the overall result of the strategy can be evaluated. It is not possible to optimise individual actions due to the lack of feedback from one turn to the next, as we mentioned before.

Those reasons led us to the definition of a set of rules which models the on-line (during the game) bot's AI. The rules have been formulated by a human player (after playing several matches in Galcon and analysing them), and are strongly dependent on some key parameters, which ultimately determine the behaviour of the bot.

Anyway, there is only one type of action: move starships from one planet to another. The nature of this movement, however, will be different depending on whether the target planet belongs to oneself or the enemy. As the action itself is very simple, the difficulty lies in choosing which planet creates a fleet to send forth, how many starships will be included in it and what will the target planet be. The main ex-

ample of this type of behaviour is the Google-supplied baseline example, which we will call GoogleBot; the behaviour of this bot will be explained next, followed by our first attempt at defeating this bot, which we called AresBot (described in Subsection 5.2). Finally, we will explain the main mechanisms governing the bot described in this paper, called GeneBot 5.3, and previously presented in [36].

5.1 GoogleBot, a basic bot for playing Planet Wars

The development kit of GAIC includes an example of bot, the so-called GoogleBot. This bot is quite simple, but it is designed to work well independently of the map configuration, so it may be able to defeat bots that are optimised for a particular kind of map, for instance, those configured to work well for situations in which enemy bases are far away (or the other way round).

GoogleBot works as follows: for a specific state of the map, the bot seeks for the planet it owns that hosts most of the ships and uses it as the base for the attack. The target will be chosen by calculating the ratio between the growth-rate and the number of ships for all enemy and neutral planets. It waits until the expeditionary attack fleet has reached its target. When it lands, it goes back to *attack mode*, selecting another planet as base for a new expedition.

In spite of its simplicity, the Google bot manages to win enough maps if its opponent is not good enough or is geared towards a particular situation or configuration. In fact the Google AI Contest recommends that any candidate bot should be able to win the GoogleBot every time in order to have any chance to get in the hall of fame; this is the baseline to consider the bot as a challenger, and the number of turns it needs to win is an indicator of its quality.

Next we will show our first initial design of a competent challenger for GoogleBot.

5.2 AresBot, the first approach

As previously said, GoogleBot has a simple behaviour (for instance there is no movement of troops from one planet to another which might need some more to defend it). Moreover, attacks start in a single planet at one time; even if the player owns two planets with the same amount of troops, it will attack just one target. Besides that, at any particular moment, the options for it are just attacking or waiting, wasting the time for making other possible actions.

The first step in this research was to design a new hand-coded strategy for the Google AI Challenge that was better than the one scripted in the GoogleBot.

This approach, which was named *AresBot*, works as follows: at the beginning of a turn, the bot tries to find its own *base planet*, decided on the basis of a score function. The rest of the planets are designated *colonies*. Then, it determines which *target planet* to attack (or to reinforce, if it already belongs to it) in the next turns (since it can take some turns to get to that planet). If the planet to attack is neutral, the action is known as *expansion*; however, if the planet is occupied by the enemy, the action is called *conquest*. The base planet is also reinforced with starships coming from colonies; this action is called *tithe*, a kind of tax that is levied from the colonies to the imperial see. The rationale for this behaviour is first to keep a stronghold that is difficult to conquer by the enemy, and at the same time to easily create a staging base for attacking the enemy. Furthermore, colonies that are closer to the target than to the base also send fleets to attack the target instead of reinforcing the base. This allows starships to travel directly to where they are required instead of accumulating at the base and then be sent. Besides, once a planet is being

attacked it is marked so that it is not targeted for another attack until the current one has finished; this can be done straightforwardly since each attack fleet includes its target planet in its data structure.

The internal flow of AresBot's behaviour with these states is shown in Figure 2.

A set of parameters (weights, probabilities and amounts to add or subtract) has been included in the rules that model the bot's behaviour (shown in the diagram in Figure 2). These parameters have been adjusted by hand, and they totally determine the behaviour of the bot. Their values and meaning are:

- $tithe_{perc}$: percentage of starships which the bot sends (regarding the number of starships in the planet).
- $tithe_{prob}$: probability that a colony sends a tithe to the base planet.
- ω_{NS-DIS}: weight of the number of starships hosted at the planet and the distance from the base planet to the target planet; it is used in the score function of target planet. It weights both the number of starships and the distance instead two different parameters since they would be multiplied and will act as just one as it is.
- ω_{GR}: weight of the planet growth rate in the target planet score function.
- $pool_{perc}$: proportion of extra starships that the bot sends from the base planet to the target planet.
- $support_{perc}$: percentage of extra starships that the bot sends from the colonies to the target planet.
- $support_{prob}$: probability of sending extra fleets from the colonies to the target planet.

Each parameter takes values in a different range, depending on its meaning, magnitude and significance in the game. These values are considered in expressions used by the bot to take decisions. For instance, the function that assign a score/cost to a *target planet* p is defined as (following a structure-based notation for p):

$$Score(p) = \frac{p.NumStarships \cdot \omega_{NS-DIS} \cdot Dist(base, p)}{1 + p.GrowthRate \cdot \omega_{GR}} \quad (1)$$

where ω_{NS-DIS} and, ω_{GR} are weights related to the number of starships, the growth rate and the distance to the target planet. *base*, as explained above, is the planet with the maximum number of starships, and p is the planet to evaluate. The divisor is added 1 to avoid a zero division.

This score is considered as a cost, so the chosen planet should be the one with the minimum value for this function. Once the target enemy planet is identified, a particular colony can provide a part of its starships to the base planet. Moreover, if the distance between the colony and the target planet is less than the distance between the base and target planet, there is a likelihood that the colony also sent a number of troops to the target planet.

Once tithe and attack fleets from the colonies are scheduled, the remaining starships from the base planet are sent to attack the target. If it is in an expansion mode, the planet will not generate starships. Therefore, and since the neutral planets do not increase its number of starships, enough troops will be sent to conquer the target planet with a certain number of extra units, since the neutral planets do not increase its number of starships. On the other hand, if it is trying to conquer a planet, the bot estimates the number of starships (grouped in fleets) required for the task.

This bot already had a behaviour more complex than GoogleBot, and was able to beat

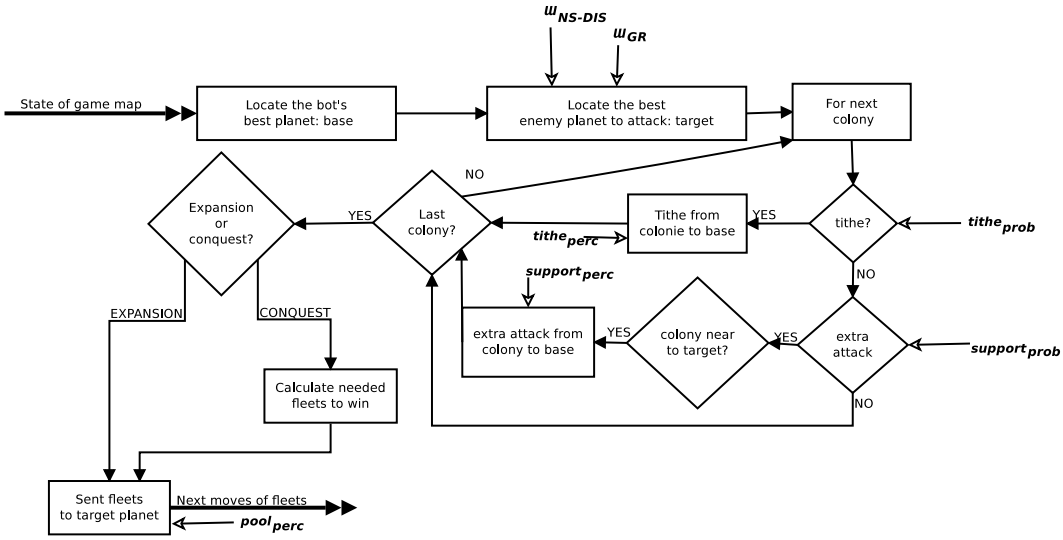


Fig. 2. Diagram of states governing the behaviour of AresBot and GeneBot, with the parameters that will be evolved highlighted. These parameters are set by hand in AresBot, and evolved for GeneBot.

it in 99 out of 100 maps; however, it needed lots of turns to beat it; this means that faster bots or those that developed a strategy quite fast would be able to beat it quite easily. That is why we decided to perform a systematic exploration of the values for the parameters shown above, in order to find a bot that is able to compete successfully (to a certain point) in the google AI challenge.

5.3 GeneBot, a genetically optimised AresBot

The main idea in this work is to perform an offline parameter optimisation, by applying a Genetic Algorithm (GA) to the set of parameters that rule AresBot (explained in Subsection 5.2), so that the resulting bot, called *GeneBot*, is the result of an optimisation process. The evolutionary algorithm is not part of the competing bot; once the parameters have been found, they are fixed and the bot uploaded to the Google AI Challenge site behaved just like other hand-coded bot.

Therefore, the objective is to find the parameter values that maximise the efficiency of the bot's behaviour, and study the relative importance of each of them in the bot's AI modelling.

The described GA uses a floating point array to codify all parameters shown in the previous versions, and follows a *generational* [23] scheme with *elitism* (the best solution always survives). The genetic operators include a *BLX-*α crossover [12] (with α equal to 0.5), very common in this kind of chromosome codification to maintain the diversity, and a *gene mutator* which mutates the value of a random gene by adding or subtracting a random quantity in the $[0, 1]$ interval. Each operator have an application rate or probability (0.6 for crossover and 0.02 for mutation). These values were set according to what is usual in the literature and tuned up by systematic experimentation.

The *selection mechanism* implements a *2-tournament* [4], where two randomly chosen individuals compete for being chosen as one of the parents of the next population. Some other mechanisms were considered (such as roulette wheel), but eventually the best results were obtained for this one, which represents the lowest selective pressure. The elitism has been im-

plemented 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 *evaluation* of one individual is performed by setting the correspondent values in the chromosome as the parameters for GeneBot's behaviour, and placing the bot inside *five different maps* to fight against a GoogleBot. These maps were chosen for its significance and are as follows:

- Bases towards the middle of the map, and the best targets between them. This map is shown in Figure 3.
- Very few and widely spread planets, bases away from each other.
- Bases apart from each other, planets away from bases in the *corners* of the map.
- Bases as far apart as possible, with planets crowding the center of the map.
- The last map is similar to the first, but planets are in the corners of the map instead of the center.

These maps represent a wide range of situations, and it was considered that if a bot was able to beat GoogleBot in all five, and also in a minimum amount of turns, it would have a high probability of succeeding in the majority of 'real' battles.

The bots then fight *five matches (one in each map)*. The result of every match is non-deterministic, since it depends on the opponent's actions and the map configuration, conforming a *noisy fitness* function, so the main objective of using these different maps is dealing with it, i.e. we try to test the bot in several situations, searching for a good behaviour in all of them, but including the possibility of yielding bad results in any map (by chance). In addition, there is a *reevaluation* of all the individuals every generation, including those who remain from the previous one, i.e. the elite. They are mechanisms implemented in order to avoid in part the noisy nature of the fitness function, trying to obtain a real (or reliable) valuation of every individual.

The performance of the bot is reflected in two values: the first one is *the number of turns* that the bot has needed to win in each arena (WT), and the second is *the number of games that the bot has lost* (LT). In every generation the bots are ranked considering the LT value (being better the lower this number is); in case of coincidence, then the WT value is also considered, so the best bot is the one that has won every single game; if two bots have the same LT value, the best is the one that needs less turns to win. Thus the *fitness* associated to an individual (or bot in this case) could be considered as the minimum aggregated number of turns needed for winning the five battles.

A multi-objective approach would, in principle, be possible here; however, it is clear that the most important thing is to win the most games, or all in fact, and then minimise the number of turns; this way of ranking the population can be seen as a strategy of implementing a constrained optimisation problem [22]: minimise the number of turns needed to win *provided that* the individual is able to win every single game.

Thus the *fitness* associated to an individual (or bot in this case) could be considered as the minimum aggregated number of turns needed for winning the five battles.

Finally, in case of a complete draw (same value for LT and WT), zero is returned, meaning that no one has won.

The source code of all these bots can be found at:
`https://forja.rediris.es/svn/geneura/Google-Ai2010`.

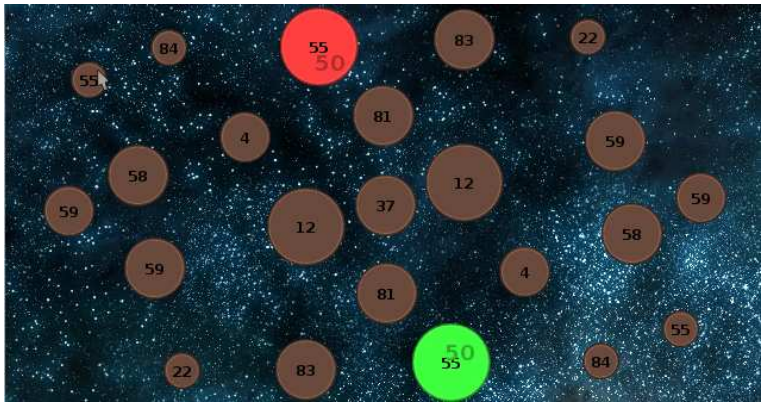


Fig. 3. One of the maps used to *train* the bots. The bot planet is shown in green, bottom center, with the number 55; the enemy planet is shown in red, has the same number as a label and is placed at the top center. The rest of the brown planets are neutral: they do not attack or grow more starships, but of course the number of present ships is offset against the number of attacking ones.

6 Experiments and Results

In order to test the GA described in Section 5.3, several experiments and studies have been performed. First of all, the experimental setting includes the parameters used in the algorithm (obtained through systematic experimentation), which can be seen in Table 1.

Number of generations	20
Number of individuals	200
Crossover probability	0.6
α	0.5
Mutation probability	0.02
Replacement policy	2-individuals elitism

Table 1. Parameter setting considered in the Genetic Algorithm.

GeneBot approach have been used to evolve the initial set of bots' behaviour parameters (those of AresBot), 10 runs have been performed, in order to calculate average results with a certain statistical confidence. Due to the high computational cost of the evaluation of one individual (around 40 seconds), a single run of the GA takes around two days with this configuration. The commented evaluation is performed, as previously stated, by playing in 5 representative maps. Besides, Google provides 100 example/test maps to check the bots, so they will be used to evaluate the value of the bots once they (their parameters) have been defined. The following sections describe each one of the studies developed for proving the value of the presented method, including a complete fitness study showing its noisy nature and the effectiveness of the implemented mechanisms for dealing with this feature of the problem.

6.1 Parameter optimisation

In the first experiment, the parameters which determine the bot's behaviour have been evolved (or improved) by means of a GA, obtaining the so-called GeneBot. The algorithm yields the evolved values shown in Table 2.

Looking at Table 2 the evolution of the parameters can be noted. If we analyse the new values for the best bot of all the 10 executions, it can be seen that the best results are obtained by strategies where colonies have a low probability of sending tithe, $tithe_{prob}$, to the base planet (only 0.28 or 0.07 in average value). In addition, those tithes send ($tithe_{perc}$) just a few

	$tithe_{perc}$	$tithe_{prob}$	ω_{NS-DIS}	ω_{GR}	$pool_{perc}$	$support_{perc}$	$support_{prob}$
AresBot	0.1	0.5	1	1	0.25	0.5	0.9
GeneBot Best	0.034	0.289	0.662	0.079	0.711	0.451	0.476
GeneBot Average	0.333 ±0.24	0.071 ±0.11	0.592 ±0.16	0.460 ±0.27	0.759 ±0.05	0.492 ±0.16	0.504 ±0.24

Table 2. Initial behaviour parameter values of the original bot (AresBot), and the optimised values (evolved by a GA) for the best bot and the average (of the 10 best bots) obtained using the evolutionary algorithm (GeneBot).

of the hosted starships, which probably implies that colonies should be left on its own to defend themselves, instead of supplying the base planet. Initially in Aresbot tithe parameters were designed to take low values, because a big tithe percentage with a high probability would mean a high movement of fleets, which would imply a big decreasing of the base planet defence against enemies. These values (percentage and probability) have been evolved independently in GeneBot and both have been set as even smaller for getting a better behaviour.

On the other hand, the probability for a planet to send starships to attack another planet, $support_{prob}$, is quite high (0.47 or 0.5 in average), and the proportion of units sent, $support_{perc}$, is also elevated, showing that it is more important to attack with all the available starships than wait for reinforcements. Related to this property is the fact that, when attacking a target planet, the base also sends ($pool_{perc}$) a large number of extra starships (71.1% or 75.9% in average of the hosted ships). Finally, to define the target planet to attack, the number of starships hosted in the planet, ω_{NS-DIS}, is much more important than the growth range ω_{GR}, but also considering the distance as an important value to take into account.

One fact to take into account in this work is that even as this solution looks like a simple GA (since it just evolves seven parameters) for a simple problem, it becomes more complicated due to the noisiness and complex fitness landscape; that is, small variations in parameter values may imply completely different behaviours, and thus, big changes in the battle outcome. This fitness nature is studied in the next section.

6.2 Noisy Fitness study

In a pseudo-stochastic environment as is this, it is important to test the stability of the evaluation function, i.e. check if the fitness value is representative of the individual quality, or if it has been yielded by chance. This factor is based in the variance of a single match which mainly depends on the opponent's behaviour (cannot be predicted). Thus, if the enemy bot performs good actions (as expected), our own bot could be evaluated with a reliable criteria and evolved to fight against good enemies; however if the enemy presents a strange or unexpected behaviour (a bad behaviour), for instance choosing a target planet which cannot be conquered by itself, sending more starships than it owns, etc, the bot being evaluated could win by chance. In this case one considered as a good bot might not perform well when it fights against a challenging enemy.

In order to avoid this random factor a reevaluation of the fittest individuals has been implemented, even if they survive for the next generation, testing continuously them in combat. In addition (and as previously commented), the fitness function performs 5 matches in 5 representative maps for calculating an aggregated number of turns, which ensures (in part) strongly penalising an individual if it gets a bad result.

Firstly it is important to note that the

number of turns required by a bot to win in a map is the most important factor of the two measured by the fitness, since there is a 200 turns restriction for winning the game, thus making the second factor rather a restriction, as said above. The bot must also beat GoogleBot in every map for having any kind of chance in the challenge.

In the first turns the two opponents (bots) handle the same number of starships, so getting an advantage in a few turns implies the bot knows what to do, and it is able to accrue many more ships (by conquering ship-growing planets) fast. If it takes plenty of turns, the actions of the bot have some room for improvement, and it would even be possible (if the enemy is slightly better than the one issued by Google as a baseline) to be defeated. That is why the number of turns is considered when assigning the fitness to the bot; the faster it is able to beat the test bot, the better chance it will have to defeat any enemy bot. It should be also remembered that this number of turns is an aggregated of the number of turns needed to beat GoogleBot in 5 different (and representative) maps, that is, every bot is run 5 times in every evaluation, as previously stated. The first study in this line is the evolution of fitness along the generations. Since the algorithm is a GA, it would be expected that the fitness is improved every generation. To prove it, the evolution vs the number of aggregated turns needed to win in the already mentioned 5 maps is shown in Figure 4.

This graph shows how fitness tends to improve, that is, how the number of turns required for winning decreases with the number of generations. Such behaviour is even clearer when looking at the line that shows the average values (the light grey one). Even so, there are decrements in all the functions, since the outcome of games has a random component which is reflected in the fitness value. This evolution is expected when dealing with a GA.

A second experiment has been conducted in this line. 10 runs of the algorithm have been performed, but in this case the mechanisms applied for dealing with the noisy fitness nature have not been considered, that is, the evaluation of every individual is performed just playing one game against an opponent (instead of five), and this battle is placed in just one map (instead of five different), randomly chosen among the five representative ones to avoid specialised bots in that map.

The aim of this study is to prove the high noisy fitness landscape, which should be dealt with for a better performance (as we have done in our algorithm by means of (elitist) individuals reevaluation and an evaluation function considering five matches in five different maps).

The evolution of the fitness values in this case can be seen in Figure 5, where the same graphs as in the previous figure are shown.

As it can be seen in the figure, there are marked fitness variations between generations. If one compare the graphs with those shown in Figure 4, it can be noticed the good performance of the noise avoiding mechanisms (reevaluation, five matches, and five different maps per evaluation) included in the new GA approach, drastically reducing this effect. Obviously the fitness values are lower when noise in evaluation is not addressed, because it corresponds to the turns needed for winning one combat (not an aggregation of five as in the new algorithm).

Once it has been proved (at least in part) the value of the noise treatment mechanisms, we conduct some other experiments considering this approach for the GA.

The next study in this scope tries to show the fitness tendency or stability, that is, if a bot is considered as a good one (low aggregated number of turns), it would be desirable that its associated fitness remains being good in most battles, and the other way round: if the bot is considered as a bad one (high aggregated num-

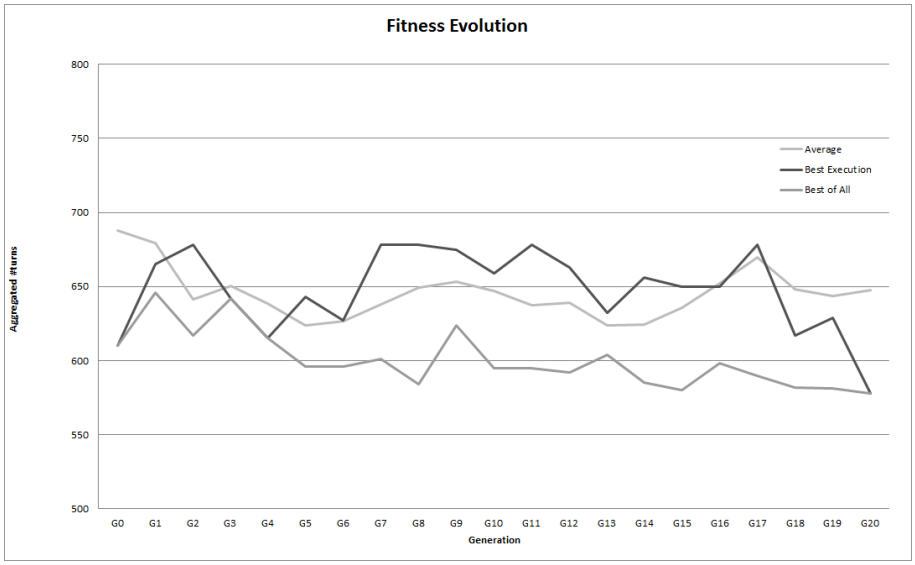


Fig. 4. The graphs show the complete execution of the best bot (best execution), the distribution of the best individuals (fitness) in every run, and the average of the best in 10 runs: as the evolution progresses (number of generations increases), the aggregate number of turns needed to win on five maps decreases (on average) on the three cases; however, since the result of combat, and thus the fitness, is not totally deterministic, it can increase from one generation to the next.

ber of turns). We are interested in knowing whether the fitness we are using actually reflects the ability of the bot in beating other bots. It could be considered as a measure of the robustness of the algorithm.

Figure 6 shows the fitness associated to two different GeneBots when fighting against the GoogleBot 100 times (battles) in the 5 representative maps. Both of them have been chosen randomly among all the bots in the 10 runs, selecting one with a good fitness value (around 550 turns), named *WinnerBot*, and another bot with a bad fitness value (around 2600 turns), called *LoserBot*.

As it can be seen, both bots maintain their level of fitness in almost every battle, winning most of them in the first case, and losing the majority in the second case. In addition, both of them win and lose battles in the expected frequency, appearing some outlier results due to the stochastic nature of these fights (they strongly depends on the opponent decisions).

6.3 Comparison of generated bots

The next study is devoted to prove the good behaviour of the evolved bots (the parameters, in fact), and thus the algorithm good performance in this task. Three bot will be considered: the initial and hand-coded AresBot, the best GeneBot among those evolved, and a bot whose parameters are composed by the average values of the best individuals (see the parameter values in Table 2).

This way, they have been tested considering 100 different battles, one in each of the example maps provided by Google, where they have fought against the standard GoogleBot. The results are shown in Table 3.

These preliminary results show that the Best GeneBot attains a good performance, winning all the battles but one, which is a draw according to the Max value for the number of turns (1001). AresBot and the Average GeneBot performs in a similar way, showing the latter a smaller number of turns, but a worse

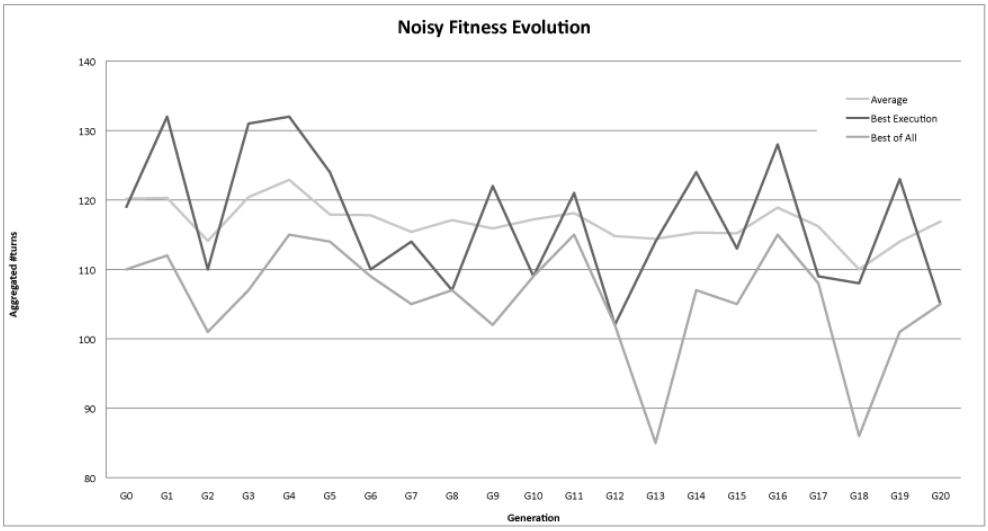


Fig. 5. No noisy fitness treatment. The graphs show the whole run of the best bot (best execution), the distribution of the best individuals (fitness) in every run, and the average of the best in 10 runs. The trend is to reduce the number of turns needed to win in the three graphs, but there are big oscillations due to the non-deterministic results of every combat (just one per evaluation).

	Number of Turns			Victories
	Average and Std. Dev	Min	Max	
AresBot	217 ± 157	49	1001	93
GeneBot Best	203 ± 131	43	741	99
GeneBot Average	251 ± 202	38	1001	91

Table 3. Results after 100 battles between each one of the Bots and the Google Standard Bot. The number 1001 arises if the battle exhausts the maximum number of turns without winner.

average in this value.

Considering again that the number of turns is the main term of the fitness function, an analysis of the turns required by the bots to beat Googlebot in the 100 test arenas has been performed, using the same bots as in the previous study. It can be seen in Figure 7.

This figure shows that most battles (around 60%) end in about 200 turns, with all bots achieving similar results, but with Best GeneBot beating the others. AresBot, in general, is better than Average GeneBot, not exceeding the 600 turns in its worst games, while Average GeneBot and even BestGeneBot turned out to need many turns in some cases, sometimes going over 800 turns and even 1000; those are usually games failed with a draw and finished by exhaustion of the maximum number of allowed turns.

Finally, we have tested the value of the three bots by making them fight in pairs, considering again the same 100 battle maps (one match per battle). Through this experiment we could decide which is the best approach and prove the utility of the GA application to optimise the initial bot, getting a better opponent than a hand-coded and expert knowledge-based one.

The results of each of these fights are shown in Figure 8.

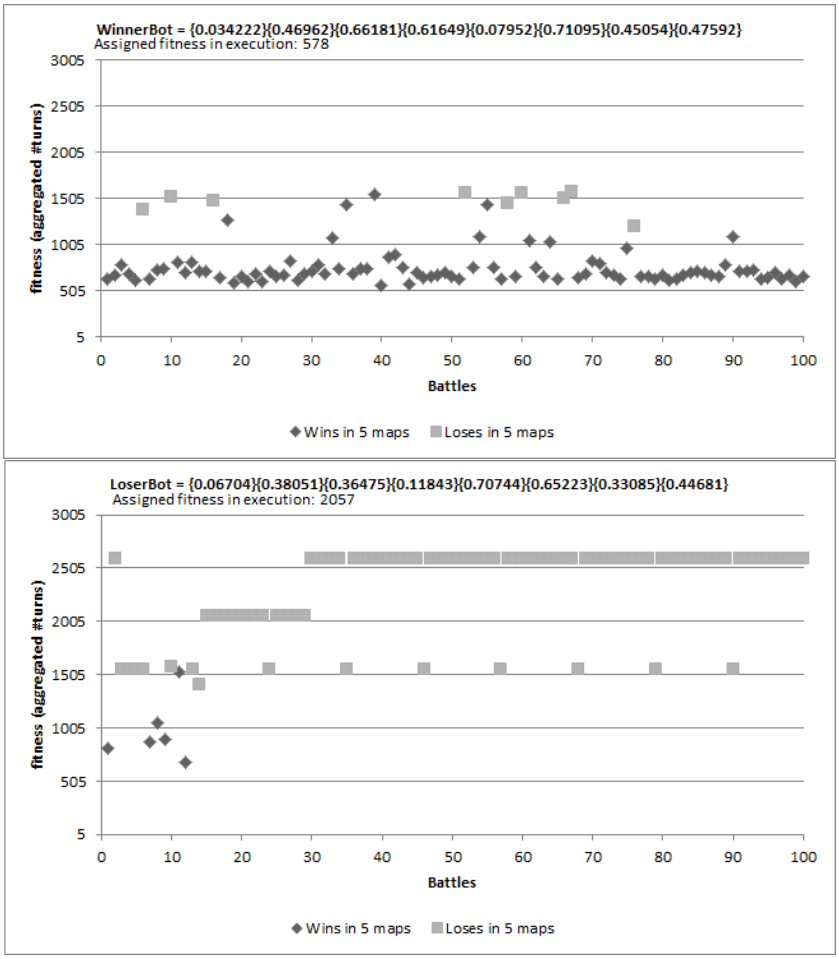


Fig. 6. Fitness tendency of two different and random individuals (bots) in 100 different battles, everyone composed by 5 matches in the representative maps, against the GoogleBot. The upper one is a very *good* bot, *WinnerBot* , according to the main term of its fitness function (the aggregated number of turns). The lower graph corresponds to a bot considered as *bad*, *LoserBot*, considering its high aggregated number of turns consumed in the battles. A victory is considered if the bot wins in the 5 matches.

It can be seen that the Best GeneBot outperforms the other two, being a much better fighter than AresBot, and quite better than the Average approach. This bot performs well regardless of the stage or the enemy. Average GeneBot beats AresBot, clarifying its value in the comparison with the hand-coded approach.

One could think that the best evolved bot should win all the battles, but the maps provided from the Google competition are balanced, in the sense that some of them have been implemented for being advantageous for just one of the contestants (and the other way round), so in some of these situations, the bot in the worse position is not able to win. Moreover, some of the maps are better-suited for one specific strategy, which could not be the one adopted by the evolved bot.

The inclusion of the Average GeneBot tries to show that the good behaviour yielded by the evolved bots has not been arisen by chance in the best individual, but it is also present in the average results, which means that the algorithm works in evolving the bots, since the

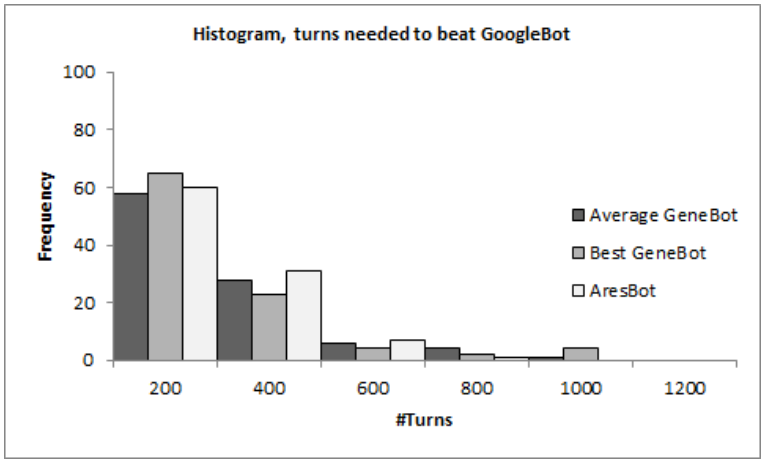


Fig. 7. Histogram of the number of turns needed to win, measured for the 100 example maps in the Google AI Challenge kit.

obtained individuals (in average) are much better than the initial one and also the baseline GoogleBot.

As an additional study 100 battles, consisting of five matches every one, have been performed, bringing face to face the best bots yielded in each run of the present algorithm against the best bots obtained by the algorithm without the noisy fitness dealing mechanisms. This way, a figure similar to the previous ones have been created, showing in each cell the colour of the winning bot is every battle (it wins in 3 out of 5 matches). Each of the matches is held in one of the commented representative maps.

The results of these fights are shown in Figure 9.

As it can be seen in the figure, the bots obtained by the new algorithm (which deals with the noisy nature of fitness) performs much better than those obtained with the simpler version, winning almost twice as many combats. This means that, effectively (five evaluations and five representative maps) the problem of noisy evaluations is being correctly addressed.

6.4 Bot vs Bot analysis

Finally, a study concerning the behaviour of several GeneBots has been performed to establish the validity of the function we have used to evaluate fitness. The participants in this experiment have been divided in four categories (A,B,C,D) as can be seen in the right side of Figure 10.

Ten individuals in each category were selected, one per execution, and battles including all the bots were performed (every one in the 5 representative maps). Figure 10 shows the battle results.

As it can be seen in the figure, the graph compounds a very symmetric matrix of victories, where better individuals win over the worst ones, at least in general. Moreover, battles in the same category show that individuals have almost the same chance to win.

The results prove that the set of obtained parameter values for the individuals, and also the number of turns to win as (main) fitness measure, have a strong influence in the performance of the bot. This way, individuals with lower fitness can hardly win to the fittest ones and the other way round. However, it also proves the *noisy* nature of fitness, with the

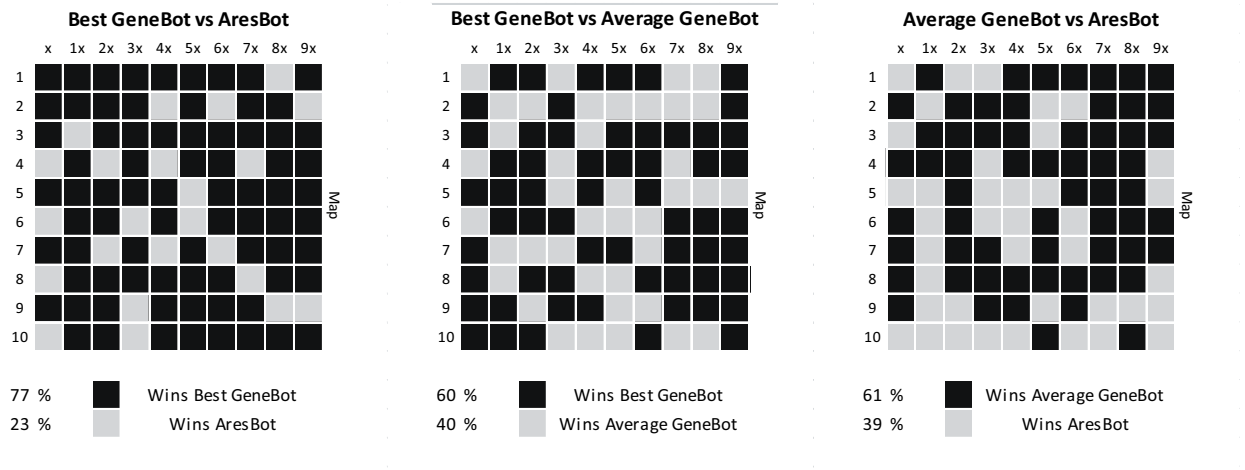


Fig. 8. Results of 100 battles (of just one match) in all the provided maps for three different bots. Every cell corresponds to a battle numbered as the correspondent column + row. Best GeneBot is clearly the winner, beating both AresBot and Average GeneBot. The latter seems to be a better fighter than the hand-coded one.

chance of the worst bot beating the best bot non-zero.

The resulting Genebot was presented to the AI Challenge 2010 finishing in the 1454th position (won 9, lost 7), around the top-30% bots. It won the combats in which it performed many fleet movements to the opponent's base planet. In addition it was successful if the bot attacks the enemies from several origin planets. The target planet selection method seemed to be hard to predict by the enemies. It lost the matches where the chosen target planet was wrong (they change its status during the movement of fleets if it takes more than one turn). The original AresBot started in the 2000 position during all the initial phase (we just can send one of the bots), in which several matches were performed, meaning a very good improvement due to the GA application.

7 Conclusions and Future Work

The Google AI Challenge 2010 was an international programming contest where game-playing programs (bots) fought against others in a RTS game called Planet Wars. This paper shows how Evolutionary Algorithms (EAs), can be applied to the design of this type of bots, and how designed bots can obtain good results in a real-world challenge by submitting them to the competition. Within the constraints that we placed on the evolution of the bot, we have proved that Genetic Algorithms (GeneBot) can improve the efficiency of a hand-coded bot (AresBot), winning more battles in a lower number of turns. Besides, from looking at the parameters that have been evolved, we can draw some conclusions to improve overall strategy of hand-designed bots; results show that it is important to attack planets with almost all available starships, instead of keeping them for future attacks, or that the number of ships in a planet and its distance to it, are two criteria to decide the next target planet, much more important than the growing rate.

In addition, the presence of noisy fitness, i.e. the evaluation of one individual may strongly vary from one generation to the next due to the non-deterministic opponents' behaviour, has been studied in several experiments. In order to deal with it, the described GA has implemented some mechanisms, such

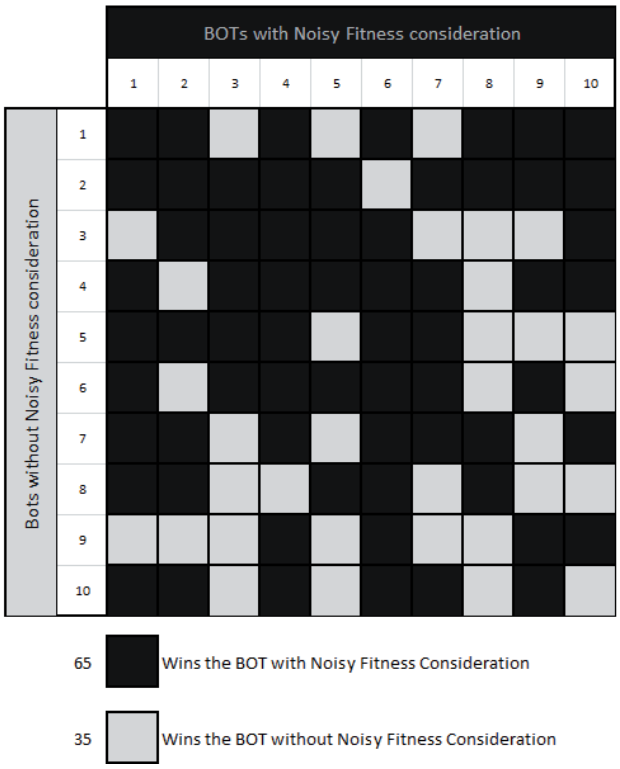
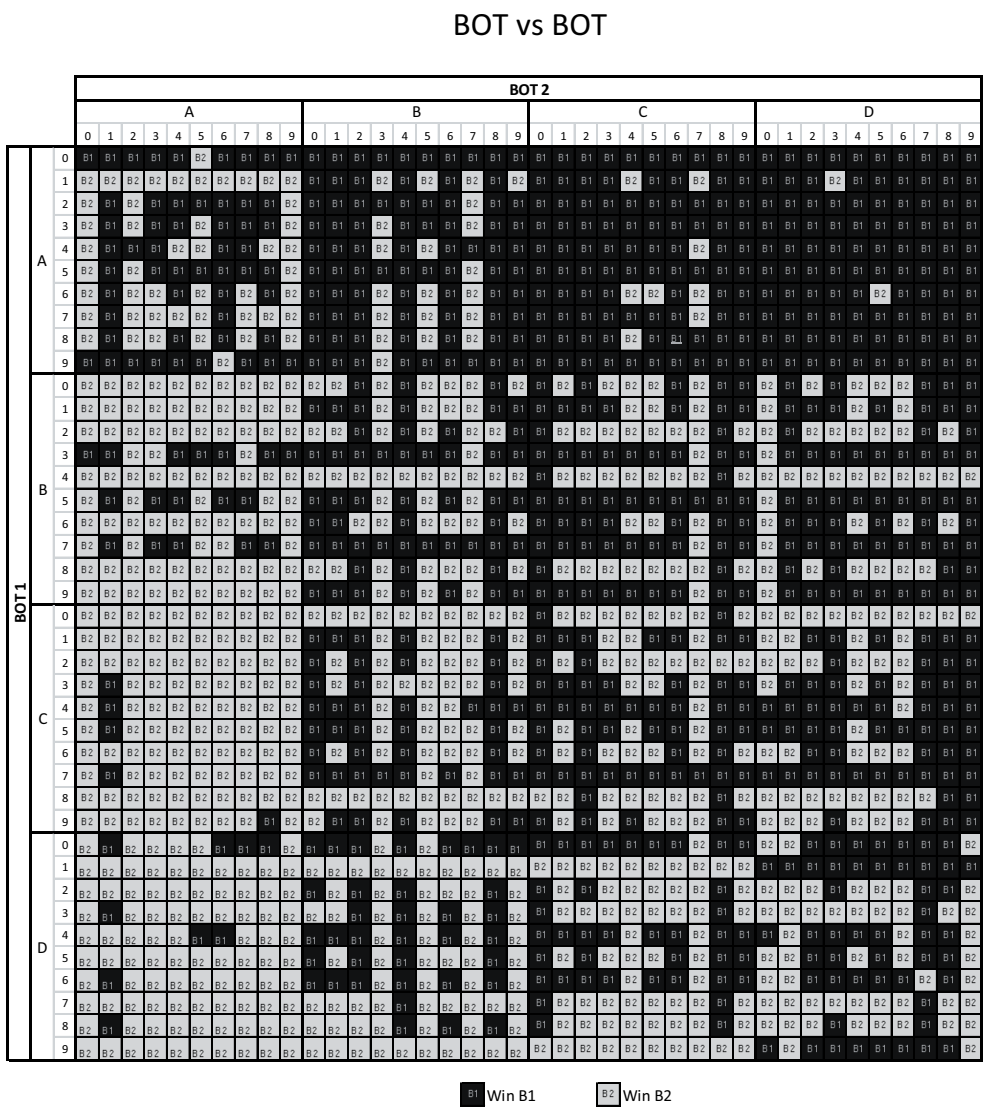


Fig. 9. Results of 100 battles involving the best 10 bots obtained by the algorithm which deals with noisy fitness (columns), and the best 10 bots yielded by the algorithm without this treatment (rows). Each battle consists in 5 matches (in the 5 representative maps). The colour of the cell means the victory of the correspondent bot in 3 out of 5 matches.

as an evaluation function consisting in repeated matches in different (and representative) maps, and a reevaluation of all the population per generation (even the elite). The experiments compare the algorithm with and without these mechanisms, concluding that the algorithm which applies them performs better, and yields results which correctly address this disadvantage, being quite robust.

In general, the improvement to the original AresBot offered by the algorithm could seem small from the purely numeric point of view; the best GeneBot is able to win some more maps where AresBot was beaten, and the aggregate number of turns is just a 10% better. However, it has been proved in the experimental section that this small advantage confers some leverage to win more battles, which in turn, will increase its ranking in the Google AI challenge. This indicates that an evolutionary algorithm holds a lot of promise in optimising any kind of behaviour, even a parametrised behaviour such as the one programmed in GeneBot; at the same time, it also shows that when the search space is constrained by restricting it to a single strategy, no big improvements should be expected.

The described bot (GeneBot) finished 1454 in the contest, winning nine matches and losing seven, which placed it amongst the top-30% bots, meaning that the evolutionary approach for fine-tuning the parameters is at least promising for these types of problems. Although the approach is limited by the strategy itself, it is a good improvement over the original AresBot which started in a position below



CATEGORIES:	
A	The best of every execution
B	Better than the 60% of population
C	Better than the 30% of population
D	The worst of every execution

Victories B1

	A	B	C	D
A	53%	80%	92%	98%
B	17%	53%	62%	61%
C	5%	41%	55%	56%
D	9%	36%	39%	50%

Victories B2

	A	B	C	D
A	47%	20%	8%	2%
B	83%	47%	38%	39%
C	95%	59%	45%	44%
D	91%	64%	61%	50%

Fig. 10. Massive GeneBot battle, every category includes 10 bots which fight against the rest (including a copy of himself) in 5 representative maps. A bot wins a battle if it beats the opponent in 3 out of 5 matches. Bottom tables summarise the category descriptions, and the percentage of wins of each category against the others for a better visualisation.

2000 (to be later replaced by this version, since only one submission was admitted).

However, a lot of work remains to be done, either to compete in next year's challenge, or to explore all the possibilities the genetic evolution of bot's behaviour can offer. As future work, and following the present work, some other mechanisms for dealing with the noisy nature of the fitness function (such as presampling or the consideration of a noise threshold) will be studied. On another line we will intend to develop a dynamic algorithm for modifying the parameters on-line (for instance, to improve the planet's defences when enemies are more aggressive, or vice-versa). In addition, a deeper study with different types of AI bots and maps will be performed, evolving for instance complex rule-based bots, or using other available bots for training and testing our's, obtaining a higher improvement. The baseline strategy will also have to be reassessed. Since it is based on a certain sequence of events, it can only go as far as that strategy. Even with the best parameters available, it could easily be defeated by other strategies. A more open approach to strategy design, even including genetic programming as mentioned by the other participants in the forum, is a promising approach.

In the evolutionary algorithm front, several improvements might be attempted. For the time being, the bot is optimised against a single opponent; instead, several opponents might be tried (the three described in these papers, for instance), or even other individuals from the same population, in a coevolutionary approach. Another option will be to change the bot from a single optimised strategy to a set of strategies and rules that can be chosen also using an evolutionary algorithm. Finally, a multi-objective EA will be able to explore the search space more efficiently, although in fact the most important factor is the overall number of turns needed to win.

References

[1] J. B. Ahlquist and J. Novak, *Game Artificial Intelligence*, series Game Development Essentials. Canada: Thompson Learning, 2008.

[2] P. Avery and S. Louis, "Coevolving team tactics for a real-time strategy game," in *Proceedings of the 2010 IEEE Congress on Evolutionary Computation*, 2010.

[3] T. Bäck, *Evolutionary algorithms in theory and practice*, Oxford University Press, New York, 1996.

[4] T. Bäck, D. B. Fogel, and Z. Michalewicz, Eds., *Evolutionary Computation 1: Basic Algorithms and Operators*, 1st ed. Taylor and Francis; 1st edition, 2000.

[5] N. Beume *et al.*, "Intelligent anti-grouping in real-time strategy games," in *International Symposium on Computational Intelligence in Games*, I. Press, Ed., Perth, Australia, 2008, pp. 63–70.

[6] M. Buro, "Call for AI research in RTS games", in *AAAI workshop on Challenges in Game AI*, D. Fu and J. Orkin, Eds., San Jose, July 2004, pp. 139–141.

[7] D.E. Goldberg, *Genetic Algorithms in search, optimisation and machine learning*, Addison Wesley, 1989.

[8] D.E. Goldberg, B. Korb, K. Deb, *Messy genetic algorithms: motivation, analysis, and first results*,Complex Systems, Vol. 3(5), pp. 493–530, 1989.

[9] A. I. Esparcia-Alcázar, A. I. M. García, A. Mora, J. J. M. Guervós, and P. García-Sánchez, "Controlling bots in a first person shooter game using genetic algorithms," in *IEEE Congress on Evolutionary Computation*. IEEE, 2010, pp. 1–8.

[10] W. Falke-II and P. Ross, "Dynamic Strategies in a Real-Time Strategy Game," *E. Cantu-Paz et al. (Eds.): GECCO 2003, LNCS 2724, pp. 1920-1921, Springer-Verlag Berlin Heidelberg*, 2003.

[11] Google, "Google AI Challenge 2010", *http://ai-contest.com*, 2010.

[12] F. Herrera, M. Lozano, and A. M. Sánchez, "A taxonomy for the crossover operator for real-coded genetic algorithms: An experimental study", *International Journal of Intelligent Systems*, vol. 18, pp. 309–338, 2003.

[13] J. Hagelbäck and S. J. Johansson, "A multi-agent potential field-based bot for a full RTS game scenario", in *AIIDE*, C. Darken and G. M. Youngblood, Eds. The AAAI Press, 2009.

[14] J.-H. Hong and S.-B. Cho, "Evolving reactive NPCs for the real-time simulation game", in *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05)*, 4-6 April 2005.

[15] S.-H. Jang, J.-W. Yoon, and S.-B. Cho, "Optimal strategy selection of non-player character on real time strategy game using a speciated evolutionary algorithm", in *Proceedings of the 5th IEEE Symposium on Computational Intelligence and Games (CIG'09)*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 75–79.

[16] D. Keaveney and C. O´Riordan, "Evolving robust strategies for an abstract real-time strategy game", in *International Symposium on Computational Intelligence in Games*, I. Press, Ed., Milano. Italy, 2009, pp. 371–378.

[17] J. E. Laird, "Using a computer game to develop advanced AI", *Computer*, pp. 70–75, 2001.

[18] L. Lidén, "Artificial stupidity: The art of intentional mistakes", in *AI Game Programming Wisdom 2*. Charles River Media, INC., 2004, pp. 41–48.

[19] D. Livingstone, "Coevolution in hierarchical ai for strategy games", in *IEEE Symposium on Computational Intelligence and Games (CIG05)*. Essex University, Colchester, Essex, UK: IEEE, 2005.

[20] S. Lucas, "Computational intelligence and games: Challenges and opportunities", *International Journal of Automation and Computing*, vol. 5, no. 1, pp. 45–57, 2008.

[21] E. Martín, M. Martínez, G. Recio, and Y. Saez, "Pac-mAnt: Optimization based on ant colonies applied to developing an agent for Ms. Pac-Man", in *Computational Intelligence and Games, 2010. CIG 2010. IEEE Symposium On*, G. N. Yannakakis and J. Togelius, Eds., August 2010, pp. 458–464.

[22] J. J. Merelo, A. M. Mora and C. Cotta, "Optimizing worst-case scenario in evolutionary solutions to the MasterMind puzzle", in *Evolutionary Computation, 2011. CEC '11. IEEE Congress on*, June 2011, pp. 2669–2676.

[23] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd ed. Springer, 1996.

[24] C. Miles and S. J. Louis, "Co-evolving real-time strategy game playing influence map trees with genetic algorithms", in *International Congress on Evolutionary Computation*, I. Press, Ed., Portland, Oregon, 2006.

[25] A. M. Mora, M. Ángel Moreno, J. J. Merelo, P. A. Castillo, M. I. G. Arenas, and J. L. J. Laredo, "Evolving the cooperative behaviour in Unreal™ bots",

in *Computational Intelligence and Games, 2010. CIG 2010. IEEE Symposium On*, G. N. Yannakakis and J. Togelius, Eds., August 2010, pp. 241–248.

[26] E. Onieva, D. A. Pelta, J. Alonso, V. Milanés, and J. Pérez, "A modular parametric architecture for the TORCS racing engine", in *Proceedings of the 5th IEEE Symposium on Computational Intelligence and Games (CIG'09)*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 256–262.

[27] S. Ontanon, K. Mishra, N. Sugandh, and A. Ram, "Case-based planning and execution for real-time strategy games", in *Case-Based Reasoning Research and Development*, ser. Lecture Notes in Computer Science, R. Weber and M. Richter, Eds. Springer Berlin / Heidelberg, 2007, vol. 4626, pp. 164–178.

[28] M. Ponsen, H. Munoz-Avila, P. Spronck, and D. W. Aha, "Automatically generating game tactics through evolutionary learning", *AI Magazine*, vol. 27, no. 3, pp. 75–84, 2006.

[29] R. Small and C. Bates-Congdon, "Agent Smith: Towards an evolutionary rule-based agent for interactive dynamic games", in *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, May 2009, pp. 660–666.

[30] P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma, "Improving opponent intelligence through offline evolutionary learning", *International Journal of Intelligent Games & Simulation*, vol. 2, no. 1, pp. 20–27, February 2003.

[31] P. Sweetser, *Emergence in Games*, ser. Game development. Boston, Massachussetts: Charles River Media, 2008.

[32] J. Togelius, S. Karakovskiy, J. Koutnik, and J. Schmidhuber, "Super mario evolution", in *Proceedings of the 5th IEEE Symposium on Computational Intelligence and Games (CIG'09)*. Piscataway, NJ, USA: IEEE Press, 2009.

[33] Wikipedia, "Computer Game Bot — Wikipedia, The Free Encyclopedia", `http://en.wikipedia.org/wiki/Computer_game_bot`

[34] Wikipedia, "Galcon — Wikipedia, The Free Encyclopedia", `http://en.wikipedia.org/w/index.php?title=Galcon&oldid=399245028`

[35] "Starcraft AI Competition", `http://eis.ucsc.edu/StarCraftAICompetition`

[36] Authors, "Our algorithm", in *International conference*, Year, Pages.