Towards Automatic StarCraft Strategy Generation
Using Genetic Programming

Pablo García-Sánchez*, Alberto Tonda†, Antonio M. Mora*, Giovanni Squillero‡ and J.J. Merelo*
*Department of Computer Architecture and Computer Technology,
University of Granada, Granada, Spain
Email: pablogarcia@ugr.es,jjmerelo@geneura.ugr.es,amorag@geneura.ugr.es
†UMR 782 GMPA, INRA
Thiverval-Grignon, France
Email: alberto.tonda@grignon.inra.fr
‡CAD Group, DAUIN
Politecnico di Torino, Torino, Italy
Email: giovanni.squillero@polito.it

Abstract—Among Real-Time Strategy games few titles have enjoyed the continued success of StarCraft. Many research lines aimed at developing Artificial Intelligences, or “bots”, capable of challenging human players, use StarCraft as a platform. Several characteristics make this game particularly appealing for researchers, such as: asymmetric balanced factions, considerable complexity of the technology trees, large number of units with unique features, and potential for optimization both at the strategical and tactical level. In literature, various works exploit evolutionary computation to optimize particular aspects of the game, from squad formation to map exploration; but so far, no evolutionary approach has been applied to the development of a complete strategy from scratch. In this paper, we present the preliminary results of StarCraftGP, a framework able to evolve a complete strategy for StarCraft, from the building plan, to the composition of squads, up to the set of rules that define the bot’s behavior during the game. The proposed approach generates strategies as C++ classes, that are then compiled and executed inside the OpprimoBot open-source framework. In a first set of runs, we demonstrate that StarCraftGP ultimately generates a competitive strategy for a Zerg bot, able to defeat several human-designed bots.

I. INTRODUCTION

Real Time Strategy (RTS) games are a sub-genre of tactical videogames where the action takes place in real time, i.e., there are no turns. Players must manage units, production structures and resources in order to, normally, win a battle against a rival. Some famous and successful RTSs are Age of Empires™, Warcraft™ and StarCraft™. The latter has also become very popular among the scientific community, as it is considered the unified test-bed for several research lines [1], such as Machine Learning, content generation, and optimization.

StarCraft [2] was developed by Blizzard Entertainment in 1998. It is a space strategy game in which three different races fight to dominate the galaxy: Terran, humans with futuristic weaponry; Protoss, aliens with highly advanced technology and powerful but expensive units; and Zerg, insect-like monsters that usually aim at overrunning the opponent with swarms of small units. A good part of StarCraft’s success is due to an excellent balance between the three species, the complexity of units and buildings, and the existence of many different viable strategies.

A huge research community is centered on the creation of agents that play this game [1], also called “bots” (short for robots), with several periodic StarCraft AI competitions [3] held at annual conferences such as CIG or AIIDE, that encourages researchers to improve their methods and create stronger bots.

There are normally two levels of Artificial Intelligence (AI) in RTS games [4]: the first one takes decisions over the whole set of units (for instance, workers, soldiers, machines or vehicles), plans a build order, and generally defines a high-level direction of the match. The second level concerns the behavior of each unit, or small subsets of units. These two levels can be considered strategic and tactical, respectively. This paper focuses on the strategic part of the AI engine.

Although most of the development of StarCraft bots is focused on manually writing an AI, or improving the parameters on which its behavior depends on using Machine Learning techniques [5], the automatic creation of complete strategies from scratch has not been addressed so far.

This paper presents a framework for the automatic generation of strategies for StarCraft, using Genetic Programming (GP) [6]. GP belongs to the category of Evolutionary Algorithms [7], optimization techniques inspired by natural evolution. This method is commonly used to create solutions internally encoded as trees or linear graphs [8], but it has also been used for the generation of Turing-complete programs. These algorithms are normally used to generate computer programs to perform an objective task, optimizing a metric or objective functions called fitness. This technique can produce combinations of conditions and actions that are potentially very different from what a human programmer could design, making it possible to obtain competitive bots from scratch, i.e. without adding human knowledge. This introduces a high difference with respect to the usual improvement of behavioral models by mean of EAs [9], [10], [11], which is based on the optimization of parameters that guide the bots behavior, and consequently constrained to a human-designed model and its possible limitations.

The proposed approach directly writes the candidate strategies as C++ classes, that are then compiled in the open-source
bot OpprimoBot [12]. Since the focus of this contribution is on high-level strategies, the tactics of units and squads are left to be managed by OpprimoBot’s default AI. The fitness function used to drive the evolutionary process is based on a series of actual matches against properly selected opponent. The resulting statistics are then analyzed and parsed to obtain a numerical metric. Even if a military victory is always considered the most important result, two different functions are tested:

1) Victory: the default final score returned by StarCraft at the end of one match against each of 12 different opponents;
2) Report: a more complex metric, aiming at separating military success from in-game economy development, using different rewards, computed after 3 matches against each of 4 different opponents.

The aim of these trials is to obtain indications on which kind of fitness metric could prove more beneficial: whether the result of a match can be generalized to multiple matches against the same opponent; and whether evaluating a candidate against a smaller variety of tactics could still return a strategy able to play well against new opponents.

The best strategies obtained after the evolutionary runs are then validated against different human-designed bots, also based on OpprimoBot: since the tactical low-level aspects of these bots are the same, theoretically the comparison should mainly concern the efficiency of the respective high-level strategies. Preliminary results show that the proposed approach can generate competitive Zerg bots, able to effectively tackle new opponents not used during training or never seen before, opening promising perspectives for future works.

The rest of the paper is structured as follows: after a background in computational intelligence in RTS in Section II, the proposed methodology is shown in Section III. Then, the experimental results are explained in Section IV. Finally, the conclusions and future work are presented.

II. BACKGROUND

A. Genetic Programming for bot generation

GP [6] belongs to a wide class of probabilistic optimization techniques, loosely inspired by the neo-Darwinian paradigm of natural selection. The algorithm is defined by a set (population) of candidate solutions (individuals) for a target problem, a selection method that favors better solutions, and a set of operators (crossover, mutation) that act upon the selected solutions. First, an initial population is created (usually randomly). Then, the selection method and operators are successively applied to produce new solutions (offspring) that might replace the less fitted in the population. In fact, at the end of each iteration (generation), the candidate solutions are compared on their goodness (fitness), and the worst ones are removed. Through this process, the quality of the individuals tends to increase with the number of generations.

A candidate solution in GP is usually internally modeled as a tree or as a linear graph [8]. In µGP [13], the framework used in this contribution, each node in the internal representation (genome) corresponds to a user-defined macro, eventually with a set of parameters. The operators can modify a macro’s parameters or even the structure of the genome, adding or removing nodes and subtrees. As the genome of an individual can be constrained in several ways, the resulting solutions can feature a considerable complexity. When applied to AI in games, GP solutions have been encoded as sets of rules or finite-state machines. Being non-human is, in fact, one of the main advantages of GP: the candidate solutions proposed by this algorithm can be very far from human intuition, and still be competitive.

Thus, GP has been used to generate autonomous playing agents (bots) in different types of games, such as famous case studies in Game Theory [14], simple board games [15], strategy games involving Assembly programs [16] and even First Person Shooters [17]. Bots created using GP have sometimes obtained higher rankings than solvers produced by other techniques, defeating even high-ranking human players (or human-designed solutions) [18]. In the case of RTS, some works are focused on the Planet Wars game, as it is a simplification of an RTS. In this case, genetic programming was proved as a valid way to generate bots that can outperform optimized handmade bots [19]. In other works different fitness functions were compared, and the results showed that a fitness that compare individuals using victories produces more aggressive bots [20]. These promising results are a good motivation to apply GP to a more complex commercial game, such as StarCraft.

B. AI in StarCraft

Thanks to BWAPI, an open-source API developed by fans of the game, StarCraft has become a challenging test-bed for research in RTS, used in dozens of publications on the subject [12]. Different research lines targeted different aspects of this game, ranging from building order definition, micro-management optimization, strategy prediction, efficient map navigation, or maximizing resource production.

In [21] Togelius et al. used a multi-objective evolutionary algorithm to automatically generate maps taking into account aspects such as playability, skill differentiation and fairness. But it is in the creation of bots where most of efforts are focused. Ontaño et al. recently presented a survey on research in this topic [1]. They also proposed a classification of the bot’s AI taking into account the problems to address: strategy, tactics and reactive control. Different techniques have been used to generate (or optimize) AIs for this game: Hagelbäck [12] combined potential field navigation with the A* algorithm for the behavior of the squads, while Churchill and Buro [22] used a Branch & Bound algorithm to optimize the building order of the structures and units in real time.

EAs have been used to optimize StarCraft bots at different levels: for example, Othman et al. evolved the parameters of hand-coded high level strategies [5], while Liu et al. [23] optimized the 14 parameters of the micro-management tactics for units. However, these methods require a hand-coded base to optimize their numerical values, and human expertise to design the bots in the first place. To the best of our knowledge, no other authors have used GP to automatically generate strategies for StarCraft from scratch.

[1] https://github.com/bwapi/bwapi
III. PROPOSED APPROACH

In the proposed evolutionary framework (depicted in Figure 1), the evolutionary core (the algorithm) receives a set of code constraints used to automatically generate the code of the bots by means of GP. For each individual in the population, a .cpp file is generated, compiled by the external evaluator and compared against a set of enemies in several StarCraft matches, to ultimately obtain its fitness. The fitness value associated to an individual will later influence its probability to reproduce, creating new derived solutions, and to survive in future generations of the algorithm.

The obtained files are then sent to an external evaluator, that will return a fitness value. The fitness values will be then exploited by the evolutionary core to create better and better solutions.

The next subsections will detail the encoding of the individuals in the population, and the two fitness functions used.

A. AI encoding

Each individual in the proposed framework represents the AI that manages the strategy of a StarCraft bot, and is encoded as a C++ class inside such a bot. The class features two methods: a constructor, defining the initial building plan (which units/structures will be built in which order) and the initial squads (groups of units with a specific role, such as attacker or defender); and a method called every frame to compute new actions (such as adding units to a squad, creating new squads, modifying the building plan) on the basis of the current situation. It is important to notice that only the strategy is evolved, while the tactics (e.g. the behavior of a single unit) are managed by other classes inside the bot, remaining unchanged during the process.

Producing compilable C++ code through a GP-like evolutionary toolkit is not a trivial task: the structure of an individual is considerably complex, and relies upon the concepts of Section, Subsection and Macro. In the following, a Section is defined as a part of the individual that contains several Subsections and appears exactly once in the genome; a Subsection can appear multiple times and contain several Macros, but Subsections strictly follow the order in which they are declared, and all their instances appear before or after the instances of other Subsections; finally, a Macro represents a single instructions, or block of code, and Macro instances can appear in any order inside the Subsection they belong to. Using this block hierarchy, a properly defined structure guarantees that every individual produced by the algorithm will compile without errors.

As the class features two methods, the structure of an individual is divided into two Sections, here called initialization and computeActions. A scheme for the individual’s structure is reported in Figure 2.

![Individual structure](image)

**Fig. 2.** Individual structure: Sections appear only once; SubSections can appear multiple times, always in the same order; Macros can appear multiple times, in any order.

1) initialization: This part of the individual defines the initial building plan and the starting groups of units. The Subsection buildingPlan, that appears only once, lists between 5 and 50 instances of the Macro addBuilding, the instruction that adds a building to the queue of structures to be built. The Subsection squadInitialization defines a squad: between 2 and 10 squads can be created by the strategy. The Macro addUnitToSquad adds 1 to 5 units of a certain type to the squad, so a squad can initially feature between 1 and 50 units. Figure 3 shows an example of a building plan generated by the proposed approach.

2) computeActions: This Section represents a method encoding a set of rules that will change the behavior of the bot during the execution, depending on the current situation. Figure 4 shows an example of a generated computeActions method.

B. Genetic Operators

Different genetic programming operators are used to cross and mutate the structure of the individuals. OnePointImpreciseCrossover and TwoPointImpreciseCrossover can combine two individuals, cutting their structures in one or two points, respectively. subGraphInsertionMutation, subGraphRemovalMutation and subGraphReplacementMutation can add, remove or replace an instance of a Subsection, respectively. insertionMutation, removalMutation and replacementMutation can add, remove or replace an instance of a Macro, respectively. singleParameterAlterationMutation and alterationMutation act on a Macro instance, randomizing one or all its parameters, respectively. These operators are part of the μGP framework, and are illustrated in more detail in [13].

C. Evaluation

The efficiency of a StarCraft strategy can be easily measured by matching it against different opponents. This fitness
Fig. 3. Example of a generated initialization method.

evaluation, taking into account victories as primary target to improve, has been successfully used in previous works to generate agents for RTS games using GP [19]. Obtaining a reliable global assessment of a strategy’s strength, however, requires a considerable number of evaluations [20]: not only the fact that some game plans are more effective on specific maps, due to their shape or size; but it is well-known that some strategies can be used as a counter to specific plans (for example, an early attack could be efficiently blocked by a defensive deployment); and finally, the game uses some random elements (e.g. amount of damage inflicted) that, while reasonably constrained between thresholds, can still influence the development of a match, so two games between the two same strategies on the same map could lead to two different outcomes [20]. However, extra information may also help to guide the evolution, for example differencing military and economic victories to generate more aggressive bots (focused in win killing the enemy and not collecting materials).

1) Victory-based fitness: This fitness compares, in lexicographical order, a vector of the number of victories against 4 different tiers of opponents in different tiers of strength. The tiers have been created through a preliminary tournament between the 12 considered strategies. In order to save computational time, a candidate bot is evaluated against a tier if it won at least once against an opponent from the lower one. The final position of the vector is a ratio between the average score obtained by the bot over the score obtained by the opponents.

2) Report-based fitness: There are several in-game metrics that can be exploited to provide a smoother slope towards good solutions, such as the number of enemy units destroyed, the amount of resources gathered, the number of units produced, etc. In particular, the following have been chosen as fitness values, to be maximized and evaluated in lexicographical order:

1) Military victory: each time the candidate strategy is able to defeat an opponent before the alloted time runs out, this score is incremented by 1.
2) Economic victory: if the alloted time runs out, but the in-game score for units and structures of the candidate strategy is higher than the opponent’s, this score is incremented by 1.
3) Relative destruction: the average score for units and structures destroyed, over the same score for the opposing strategy.
4) Time to loss: the average time the candidate strategy resisted before losing against the opponents it was not able to defeat.
5) Relative economy: the average score for units and structures built, over the same score for the opposing strategy.

IV. EXPERIMENTAL EVALUATION

Given the consideration in Subsection III-C it is easy to see how the number of evaluations needed for the assessment of a
candidate strategy quickly explodes: considering a minimum of 3 matches in the same conditions to obtain a reliable average, evaluating a candidate strategy against 10 opponents on 10 different maps would require $3 \cdot 10 \cdot 10 = 300$ matches; and a match long enough to be significant would last at least 10 wall-clock minutes, even at maximum in-game speed. Thus, a self-evident disadvantage of the proposed approach is the sheer quantity of computational power necessary to evaluate the individuals. For this reason, in this proof-of-concept experimental evaluation, the focus is on a limited number of strategies, evaluated on a single map: overfitting the final strategy on these conditions is almost a given, but the objective is not to obtain a tournament-competitive player, but rather to assess whether the approach is viable.

The most competitive bots in current championships usually implement Protoss and Terran [1] strategies, meaning that they are probably easier to manage for hand-coded AIs. For this first approach, we choose to evolve a Zerg strategy, a weaker strategy in principle, to assess whether a GP engine can successfully manage this faction, and also because it is based in quick and cheap units.

### A. Setup

The experiments have been performed on a group of 8 VirtualBox[1] virtual machines (VMs), one Master and 7 Clients, running Windows XP (see Figure 5). The Master VM runs the evolutionary algorithm $\mu$GP[4][24], that creates the strategy classes following the constraints specified in Subsection III-A. The algorithm is configured with the parameters reported in Table I. These parameters have been chosen because they have been used successfully in previous works that use GP for RTS bot generation [19]. It is important to notice that the 30 generations set as a stop condition correspond to about 5 days (120 hours) of computation, on the hardware used for the experiments.

Candidate strategies are compiled inside the OpprimoBot[5] framework [12], v14.12, obtaining a DLL with the complete bot. Finally, the DLL is sent to the TournamentManager[6].

---

software, that distributes the matches to be played among the Client VMs and collects the results. A Python 2.7 script directs the execution of all the involved programs, and parses the TournamentManager files to obtain the fitness values.

For the Report-based fitness, the Dummy and three human-designed players from the difficult tiers have been used: OBTerranDummy, OBProtossReaverDrop, OBTerranWraith-Harass, OBZergLurkerRush (repeating 3 times each strategy).

### B. Results

As a StarCraft game requires real time to be executed (usually 10 minutes per match), even using 8 VMs in parallel, each execution of the algorithm requires several days. However, as this is a proof-of-concept, we analyze the only run we have performed for each fitness compared. The best fitness obtained for is shown in Tables II and III.

#### TABLE II. Fitness of the best individual obtained and average fitness of the population using Victory-based evaluation.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Best individual</th>
<th>Average of population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tier 3</td>
<td>3</td>
<td>1.73</td>
</tr>
<tr>
<td>Tier 2</td>
<td>3</td>
<td>1.85</td>
</tr>
<tr>
<td>Tier 1</td>
<td>2</td>
<td>1.93</td>
</tr>
</tbody>
</table>

#### TABLE III. Fitness of the best individual obtained and average fitness of the population fitness using Report-based evaluation.

<table>
<thead>
<tr>
<th>Military Victories</th>
<th>Best individual</th>
<th>Average of population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Military victories</td>
<td>3</td>
<td>1.76</td>
</tr>
<tr>
<td>Economic victories</td>
<td>1</td>
<td>1.93</td>
</tr>
<tr>
<td>Relative destruction</td>
<td>400245</td>
<td>358172</td>
</tr>
<tr>
<td>Time to loss</td>
<td>1120</td>
<td>1380.7</td>
</tr>
<tr>
<td>Relative Economy</td>
<td>0.309</td>
<td>0.501</td>
</tr>
</tbody>
</table>

Evolution of the average individuals for both fitness approaches is shown in Figure 5 and Figure 7 (only the number of victories of the fitness vector is plotted). It is interesting to remark, that both obtained bots have been able to beat more than one strategy at the end of the evolution. As it can be seen, in both methods, the average fitness of the population during all the run has lower values for the “difficult” type of victory (first value of the fitness vector), while the obtained best individuals (in Tables) have reached higher values for this number (Tier 3 victories and military victories, respectively). Therefore, the population is not stagnated for 30 generations, so more generations may lead to even better individuals.

### C. Validation

After the evolution, and in order to validate the generated bots of the two different fitness approaches, a championship has been performed. The two generated bots have been confronted versus all the hand-coded strategies of OpprimoBot (including those not considered for evolution, in the case of the Report-based fitness) and also against the complete OpprimoBot. Each bot has been confronted 10 times against each enemy. Table IV shows the number of victories (from 0 to 10) of our generated bots.

Results of this championship show that the first runs of our method have generated bots able to defeat human-coded strategies, and even a complete complex bot (OpprimoBot). A comparison of the two generated bots shows a prevalence of the Victory-based fitness, that wins over more strategies, and more times. This can be explained because a greater number of bots has been used for the fitness value computation for...
rules triggered almost the same amount of times.

The activated Victory-based rules basically generate an exploratory squad with zerglings, hydralisks and mutalisks, and an offensive one with scourges, overlords and queens. Sometimes one of these types of squads are duplicated (rules 2 and 8) in the unit construction queue, and updated with scourges (rule 5), lurkers (rule 9) or hydralisks (rule 3) or evolving units to devourers and lurkers (rule 6) or guardians (rule 3). Rule 7 only adds buildings to the building plan to be constructed when possible (defiler, extractor and dark swarm). In the case of the Report-based, 6 different types of squads (offensive, rush and exploratory) are queued at first, and the two rules executed add hydralisks to existent squads, but no more squads are generated. Percentages of the rules are shown in Table V.

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Percentage when winning</th>
<th>Percentage when losing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Victory-based</td>
<td>Report-based</td>
</tr>
<tr>
<td>2</td>
<td>10.20</td>
<td>7.662</td>
</tr>
<tr>
<td>3</td>
<td>21.58</td>
<td>23.05</td>
</tr>
<tr>
<td>6</td>
<td>13.22</td>
<td>13.22</td>
</tr>
<tr>
<td>7</td>
<td>21.86</td>
<td>23.49</td>
</tr>
<tr>
<td>8</td>
<td>20.17</td>
<td>19.76</td>
</tr>
<tr>
<td>9</td>
<td>12.34</td>
<td>12.79</td>
</tr>
<tr>
<td>1</td>
<td>51.53</td>
<td>50.98</td>
</tr>
<tr>
<td>3</td>
<td>49.01</td>
<td>49.01</td>
</tr>
</tbody>
</table>

V. Conclusions

StarCraft has become a de facto test-bed for research on RTS games, as it provides different strategic levels for agent generation and optimization, well balanced types of races, and a huge community of players and researchers.

This paper proposes a preliminary study on the usage of Genetic Programming to automatically generate high-level strategies for StarCraft, using the StarcraftGP framework. Two different methods for evaluation of the bots during the evolution have been compared: a victory-based fitness and a report-based fitness, both using different number of enemies for evaluation. The first run for each method has been able to automatically generate strategies that can defeat bots hand-coded by human experts. Results show that the victory-based fitness can generate better bots than the report-based method,
winning more than half of the battles against hard-coded strategies, and even considering a complete bot.

Future work will address several improvements of the proposed methodology, for example using more fitness functions, more runs per method to validate their results, or the usage of different maps in evaluation and testing. Also new experiments will be performed using more game information, or other methods, such as co-evolution. The optimization of the other two races of the game (Protoss and Terran) will be also studied. Other aspects of the game will be also optimized using our framework, such as the low-level squad behavior, or even the micro-management of each unit. Finally, other techniques can be used in conjunction with our method, for example, optimizing the learning and map analysis modules available in the literature.

ACKNOWLEDGMENT

The authors would like to thank Johan Hagelbäck and Dave Churchill for their help and the insight they provided. This work has been supported in part by SIPESCA (Programa Operativo FEDER de Andalucía 2007-2013), TIN2011-28627-C04-02 and TIN2014-65494-C4-3-P (Spanish Ministry of Economy and Competitiveness) and SPIP2014-01437 (Direcció General de Tráfico) and GENIL PYR-2014-17, awarded by the CEIBioTIC UGR.

REFERENCES


