A Hyper-Heuristic Genetic Algorithm To Evolve a Commander For a Capture The Flag Game

Victor de Cia Costa Amadeus Torezam Seilert
Léo Françoso dal Piccol Sotto Vinícius Veloso de Melo

Federal University of São Paulo - UNIFESP, Institute of Science and Technology - ICT
São José dos Campos, São Paulo, Brazil

***Abstract*—This paper presents results from using a hyper-heuristic genetic algorithm to evolve a Commander for a Capture The Flag (CTF) game. The CTF game employed in this study runs on The AI Sandbox framework that was used in the competition promoted by AiGameDev. The Commander issues high-level orders to a team between 4 and 15 bots that must capture the flag of the enemy team and return it to its flag scoring location. The orders are for each bot and include requests to charge, defend, attack, or move to a location. Before issuing an order, the Commander has to evaluate the environment and then make the decision. These evaluations are usually organized in modules or blocks of codes that are executed sequentially. Thus, the developer uses his creativity to implement these modules and to organize the code. This paper is focused on the second part of the problem: code organization. The idea proposed in this paper is the employment of a genetic algorithm to evolve a Commander built of blocks of codes extracted from other Commanders. The algorithm selects the blocks from a set, join them to result in a complete Commander, and evaluates it against other Commanders in a set of maps to calculate a score, which represents the quality of the Commander. Results of battles in maps that were not used in the evolution phase show that the proposed approach presents a great potential to generate high-quality Commanders without human intervention.**

***Keywords*—Hyper-Heuristic, Genetic Algorithm, Game, Bot.**
Authors' contact: victor.decia@gmail.com, amadeus.seilert@gmail.com, leo.sotto352@gmail.com, vinicius.melo@unifesp.br

I. Introduction

Modern digital games in computers, consoles and, more recently, smartphones and tablets, have become very advanced in recent years. However, despite the improved graphics, physics, sounds, interactivity, among other aspects, players usually complain against the Artificial Intelligence (AI) of NPCs (non-player characters) arguing that they are far from being intelligent. As games become more complex, with more players, more items and guns, larger scenarios, the development of interesting AIs also becomes a very hard task.

In the development of AI for agents in games, it is very common the use of hard-coded scripts with Finite States Machines [1], [2] and, more recently, Behavior Trees [3], [4], [5], [6], which are executed when some specific condition is reached by the player. When these conditions, or the way the logic is organized, make the actions to be executed by the agent very predictable, it may contribute to non-intelligent behavior of NPCs.

On the other hand, in the last years some researches have been made with evolutionary [7] and machine learning [8] algorithms to improve the AI in games [9], [10], [11], [12]. Because the automatic creation of a complete code from scratch in not possible with current techniques, one of the main ideas is to automate the creation of *parts* of the code, instead of letting the developer do the whole work. This automation allows a game to be parallely run in a cluster, for instance, where several thousands or millions of games can be executed in a feasible amount of time and the collected information be employed to improve the game's AI. Thus, new strategies and behaviors may be automatically discovered by the algorithms, using almost no human intervention. The objective is that the generated AI outperforms the hard-coded AI in efficiency and adaptiveness.

A. Motivation and Contribution

The AI *GameDev* group promoted an AI competition in 2013 for their Capture The Flag (CTF) game that runs on their AI Sandbox framework. With the SDK, the developers have access to the API and to some example AIs (the Commander of a team of bots) that can battle against other Commanders in several different maps. Participants were allowed to manually code their Commanders and to use machine learning and other optimization algorithms.

Instead of developing a new Commander from scratch, the approach in this work uses an algorithm to create a Commander. The algorithm selects random blocks of codes from known Commanders, arranges them in a sequence to generate a complete source-code, and runs the battle to evaluate the new Commander. An iterative process is repeated, trying to find even better Commanders. This iterative process is performed by an evolutionary optimization algorithm, the Genetic Algorithm (GA, [13]), acting as a Hyper-Heuristic [14], [15] to evolve the code.

Therefore, the investigation in this paper is to evaluate if a competitive Commander can be automatically generated by arranging strategies from human developed Commanders. As a result, a tool was developed to automate the design of the Commander by using a template that is filled with blocks of code from a dataset. The tool can be used by developers to test their ideas, which are automatically arranged with other

ones,trying to select the best arrangement of blocks. Using the tool a Commander may be evolved for a specific map or for a set of maps. In the end several Commanders are returned, letting the developer evaluate them, check the blocks that were selected, and analyze the performance of the Commander.

The organization of this work is as follows: Section II presents related works. Section IV briefly presents the algorithms to evolve a Commander. Section V presents the methodology employed in this work, Section VI presents the test case and simulations to validate the architecture, discusses and analyzes the tests. Finally, Section VII presents the conclusions of this work.

II. Related Work

A bot AI is commonly composed by several hard-coded conditions and possible responses for different cases - an scripted AI. In these cases, however, the AI can only cover or present reactions for situations previously imagined by the developer. Recently, new approaches using adaptive AI and evolutionary computation have been researched and yielded satisfactory results in defeating random and human-made bots.

An evolutionary approach to create a bot AI consists of, by means of evolutionary algorithms such as Genetic Algorithms, evolving aspects of an existing bot AI, or randomly generating an initial population of bots and evolving it to a goal, measured by an evaluation function.

Examples of researches that obtained good results in evolving bot AIs can be seen in [9], [10], [11], [12]. Bakkes et al. [9] evolved team behavior for the Capture The Flag game mode of Quake III. Their algorithm evolved each state of a bot's finite-state machine and then used them for a final one. Cole et al. [10] used also a genetic algorithm to tune parameters in a bot for Counter-Strike. In [12], an evolutionary algorithm was applied to evolve rules for a bot AI in Quake III. Mora et al. [11] used a Genetic Algorithm to optimize the parameters of a bot AI for the Unreal game, and Genetic Programming to evolve states for the behavior of the bot.

III. The CTF game using AI Sandbox

The game for which we have evolved the AI is called Capture The Flag (CTF), and runs on the AI Sandbox [16] platform. In each game, two teams composed of 4 to 15 bots compete against each other to score the most points. Points are scored whenever a team returns an enemy flag (see Figure 1) to its flag scoring location. The Commanders, the "brains" of the game, are built to control all bots in the battlefield by sending them orders when called by the game, at specific time intervals (ticks). The orders defined by the API are: attack, charge, defend, and move to a location. The Commander has information about the visibility, flag situation, combat and status of each bot (see Figure 2), thus the user has a lot of possibilities to create modules to control the bots.

A brief description of the basic idea of each Commander investigated in this work is shown below. The descriptions for the example commanders (Random, Greedy, Defender and Balanced) were taken from the AI Sandbox documentation

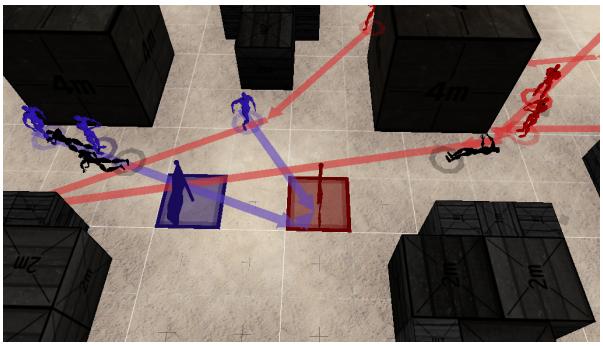

Figure 1. Game image [16].

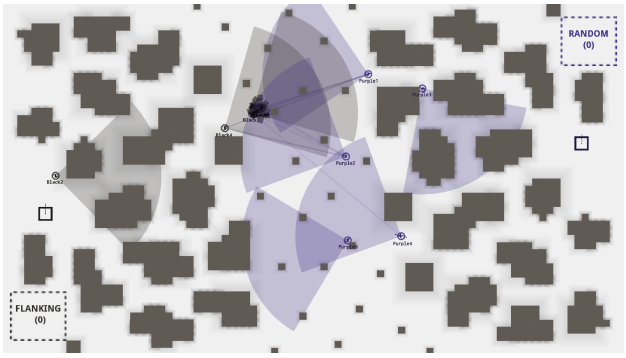

Figure 2. Game simulation image [16].

[16], while Kilroy [17] and BTSample [18] are commanders released by other developers.

- **RandomCommander**: sends everyone to randomized positions or a random choice of flag location. The behavior of returning the flag to the home base after capturing it is purely emergent!
- **GreedyCommander:** always sends everyone to the flag of the enemy and the guy carrying the flag back again.
- **DefenderCommander:** leaves everyone to defend the flag except for one lone guy to grab the other team's flag.
- **BalancedCommander:** an example commander that has one bot attacking, one defending and the rest randomly searching the level for enemies.
- **Kilroy14:** this bot "started with Defender commander and added some adaptations to help it track visible enemies and look towards killed teammates to help make it a better contender". Then, Balanced flanking method was added to the current single attacker in the team. Also, a dictionary is associated with each bot for associated roles, state and properties. Other strategies are used, like adding a simple 'evade' behavior, sorting distances, and defense functions that help a bot only defend angles of interest when near outer map walls and corners.
- **btSampleCommander:** this bot uses a behavior tree (BT). It assigns and run a BT for each bot. The function-

Table I. CHARACTERISTICS OF THE AVAILABLE MAPS, WHERE R=RESPAWN DEAD BOTS AT EVERY R SECONDS, O=NUMBER OF OBSTACLES, B=NUMBER OF BOTS PER TEAM, BFD=BASE-FLAG DISTANCE, AND FDS=FLAG-SCORE DISTANCE. ONLY ONE MAP WAS USED TO TRAIN THE COMMANDER, BUT ALL OTHERS WERE USED TO TEST IT.

Map	R	O	B	BFD	FSD
map00	45	medium	5	close	close
map01	45	several	15	close	close
map02	20	medium	8	far	far
map03	2	medium	14	medium	far
map10	20	few	8	medium	medium
map11	30	few	6	far	far
map12	30	few	10	far	medium
map13	5	medium	7	close	close
map20	45	medium	11	close	close
map21	45	medium	9	medium	close
map22	10	few	7	far	medium
map23	5	several	9	far	close
map30	30	several	12	far	close
map31	30	several	13	far	far
map32	5	several	10	far	medium
map33	5	several	12	far	close
map40	30	few	8	medium	far
map51	30	none	5	far	far
map52	30	several	10	far	far
map53	30	several	12	far	far

ing of the BT used is rather simple: it checks a condition and chooses between returning flag, take flag, or attack enemy.

There were 20 maps available in the SDK. Some characteristics that we could identify are presented in Table I. The base-flag distance is the distance between the respawn location of a team and the flag location of the opponent team, which interferes in search of the flag from the base. The flag-score distance is the distance between the score's location of a team and the flag location of the opponent team, which interferes in the bringing of the flag to the score zone.

IV. THE ALGORITHMS TO EVOLVE A COMMANDER

In this work, a Genetic Algorithm was employed as a Hyper-heuristic to evolve a Commander. The efficiency of these methods are well known and applicable to a wide range of problems [13], [19], [20].

A. Genetic Algorithms

Genetic Algorithms (GAs, [13]) are one of the most employed evolutionary algorithms for global optimization. It has been widely used to solve continuous and combinatorial problems over the years [21], [22], [23].

The GA tries to mimic the process of natural evolution. Usually, solutions for a problem are represented as chromosomes of individuals. A population of individuals is evolved over the generations, generating better individuals in the process. Each individual has a fitness value, that corresponds to its quality. Some individuals from the population are selected, usually based on their fitness, to generate one or more children that will contain part of the genetic material from the parents. Children can suffer from genetic mutation, making them more different and possibly with better characteristics. The new individuals

Let ***P*** be a random, initial population
Evaluate ***P*** using the fitness function
While some convergence criteria is not satisfied
  Let ***P1*** be the individuals selected from ***P*** to be parents
  Let ***P2*** be the offspring after applying the crossover operator to ***P1***
  Mutate individuals in offspring population ***P2***
  Evaluate ***P2*** using the fitness function
  Replace individuals from ***P*** with individuals from ***P2*** according to some criteria
Return ***P*** and corresponding fitness values

Figure 3. Basic pseudo-code of a Genetic Algorithm.

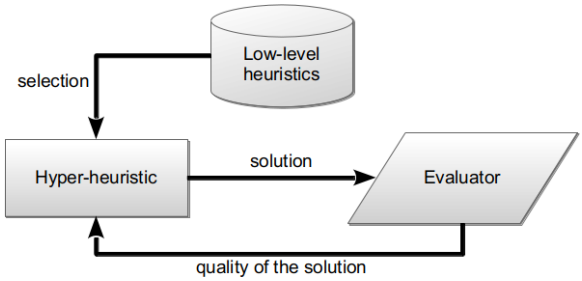

Figure 4. Basic example of a hyper-heuristic flowchart.

are evaluated by the fitness function and may be inserted into the population, replacing older individuals. This process is repeated until a stop condition is achieved, for instance, a maximum number of generations or a desired value for the best fitness. A basic pseudo-code of a GA is presented in Figure 3.

B. Hyper-heuristics

According to Ross [14], "The key idea is to devise new algorithms for solving problems by combining known heuristics in ways that allow each to compensate, to some extent, for the weaknesses of others. They might be thought of as heuristics to choose heuristics.". It is important to notice, as shown in Figure 4), that the solution generated by a hyper-heuristic is not a solution for a problem, as occurs in common optimization algorithms. Actually, the solution is the codification of an algorithm to find solutions for a problem. As a result, the evaluator has to run the coded solution on the problem, several times if necessary, to assess its quality.

Therefore, a hyper-heuristic is an optimization algorithm employed to create another algorithm by, for instance, selecting and grouping pre-coded components in a template code. Other, more complex possibility, is the creation of components by the evolution of the source code [15]. In this paper, the template approach is employed to evolve the Commanders.

Hyper-heuristics have been successfully applied to solve several problems, e.g., improvement of optimization algorithms [24], the development of heuristics [25], [26], decision tree classifiers [27], solving scheduling [28], and timetabling [29].

The methodology applied in this work is presented in the next Section.

V. METHODOLOGY

To evolve a Commander, an adaptive length chromosome hyper-heuristic genetic algorithm (ALChyper-GA), similar to the one proposed in [19], was employed to work on a previously prepared set of low-level heuristics (blocks of codes from Commanders) and evolve the arrangement of blocks guided by an evaluation function. The heuristics were taken from the Commanders presented in Section III.

A. The Template

A Commander is basically composed by the following parts, which were used as the template to be filled:

Head: it is the top of the source-code's file. The head of all Commanders are grouped in the template's head, which will contain all imports, functions, global variables, etc, taking care of repeated names. Moreover, one must identify blocks of code inside of the Tick function that may be necessary for a specific Commander to work and include them in the template's head. For instance, if one Commander has a function or procedure implemented in the Tick, then this function or procedure must be inserted in the template's head to be accessible for any new Commander, even if it will be unnecessary.

Tick function: is called by the game, at every interval of time, to ask the Commander the new orders to submit to the bots. The Commander must evaluate the environment and decide what each bot has to do next. To perform this task it has access to several information (via the API) of the map, the flag, and each bot. This is the *brain* of the game, the part that must be evolved. Because blocks from different Commander may be inserted into the same Tick function, it must have all local variables and initialization procedures required by the Commanders.

Shutdown function: it is responsible for, when the battle ends, calculate the score/quality of the Commander to be used by the GA as fitness value. It also must have all local variables and codes required by the Commanders. Also, this function is the responsible for calculating the score/quality of the Commander using the approach employed in this work (see Section V-D).

B. The Blocks of Code

The hyper-heuristic will create a sequence of codes using blocks available in the dataset, which can be seen as low-level heuristics. To create this dataset, one must manually extract logically independent blocks of code from the Tick function in the Commanders. Since the source code of the Commanders are in Python, we have considered that a block is independent if it is indented/aligned with the first command inside the Tick function (see Figure 5). Thus, an inner-loop is not independent, neither an *if* inside of another block (loop, *if*), neither an *else* that depends on the *if*.

However, a loop below another loop may be considered independent even if the second one depends on a value updated by the first one. If only one of the loops is inserted in the template, the code will probably result in a low score and then this Commander may be replaced. As one can see, there is no need to verify this kind of coherence in the code because the hyper-heuristic can deal with it.

```
def tick(self):
    # INITIAL ALIGNMENT
    init = 0

    # BEGIN BLOCK
    if condition:
        commands
    # END BLOCK

    # BEGIN BLOCK
    if condition:
        commands
        if condition2:
            commands
        else:
            commands
    # END BLOCK

    # BEGIN BLOCK
    for variable in list:
        if condition:
            for variable2 in list2:
                commands
    # END BLOCK
```

Figure 5. Examples of blocks of code.

Each extracted block received an integer ID. The initial dataset had 14 blocks, where a large part had numerical parameters, for instance, $if\ distance < 1$. These parameters need a high level of experience in the game to be optimized [10]. Taking this into consideration, variants of the blocks were created with different, empirically chosen, values ($if\ distance < 2$), extending to a total of 44 blocks of code.

C. The Genetic Algorithm

The hyper-heuristic was implemented in the Python programming language, using the PyEvolve library[1], because it was the official programming language of the competition, very flexible, and easy to code.

Crossover: as previously commented, hyper-heuristic uses adaptive length chromosomes instead of fixed length ones. It is simply the one-point crossover where for each parent a different point is selected.

Mutation: two operators were employed at each call to this function, in a random choice with probability of 50% each. The first one is the Swap mutation and the second one is the New Value (in range 0 to 43) mutation.

No corrections are made in the individuals, so repetitions are allowed.

D. The Evaluation Function

The fitness of and individual (Commander) is calculated in the *Shutdown function* using Eq. 2:

[1] www.pyevolve.com

$$flagsPerSec = \frac{myScore}{timePassed}, \quad (1)$$

$$score = myScore - theirScore + flagsPerSec, \quad (2)$$

where *myScore* is how many flags our team captured and *theirScore* how many flags we lost, or, in other words, the enemy team captured, and *flagsPerSec* is a metric to standardize the number of flags when battles have different duration.

VI. Simulation Results and Discussion

For the simulation we used AI Sandbox version 20.7, CTF SDK version 1.7.1, Python 2.7.3, Ubuntu Linux 12.04 64bit in a desktop Intel core(TM) i5-2400 CPU 3.10GHz with 4 Gb RAM. In order to evolve our Commander, named *BRAINIAC,* using the hyper-heuristic, we have chosen to put it to battle against the best Commander available, *Kilroy,* in *map00*. By doing this, one can have an idea of how good it performs in the map it was trained in as well as in unseen maps, allowing for an analysis of its effectivity and adaptivity. The configuration used in hyper-heuristic is presented in Table II.

Table II. Configuration of the hyper-heuristic.

Parameter	Value
Population size	50
Initial chromosome size	12
Maximum chromosome size	Unlimited
Generations	130
Crossover rate	0.9
Mutation rate	0.5
Elitism	1
Selection	Ranking

The final best evolved Commander, named *BRAINIAC*, presents the following chromosome with 14 blocks:

$$chromo = [18, 7, 24, 3, 35, 12, 27, 6, 11, 9, 32, 34, 38, 24].$$

A total of 13 different blocks were selected (block number 24 is repeated) from the 44 available (see Section V-B). The descriptions in Table III, taken from the comments in the original source codes, are from some of these blocks. The blocks are sequentially executed in the code, thus a posterior block may directly interfere in the orders issued by a previous block. This also means that the resulting Commander may not present a logic that a human would use and that some blocks may be useless.

To investigate the effectiveness of *BRAINIAC* in different situations, it was tested against all Commanders it evolved from, playing in all available maps (20). A battle in a map has a duration of 300 seconds, with respawn of dead bots at every R seconds, as shown in Table I. For each map, *BRAINIAC* played 30 times against each Commander (a total of $20*6*30 = 3600$ battles), and the result of a battle (Won, Drew, or Lost) is based on the final number of flags captured. The percentage of the results are presented in Table IV.

Table III. Descriptions of some blocks of code.

ID	Description
18	check combatEvents for latest activity; remove dead enemies from tracking list; check for flag drop
7	move scramble bots
24	Return the flag home relatively quickly!; Find the enemy team's flag position and run to that.; defend the flag!
3	Second process bots that are in a holding attack pattern.
35	spawncamp - not currently using this role because of spawncamp protections, but could be altered to camp further from spawn to catch bots exiting spawn
12	periodically reset back to facing enemy flag
27	In this example we loop through all living bots without orders (self.game.bots_available); All other bots will wander randomly
6	The same as ID 27, but with different configuration in some parameters

A. Discussion

The first consideration is that *BRAINIAC* was evolved to beat *Kilroy,* which was supposed to be the best opponent, in *map00*. Therefore, *BRAINIAC* is being evolved to be specialized. Thus it is expected that it outperforms *Kilroy* in this map but draw or lose in other maps. The same is valid for the other opponents in the simulation. It is important to notice that all percentages are rounded, thus the sums may not result in 100%.

The first analysis is *BRAINIAC* versus *Kilroy* (see Table IV). In *map00*, *BRAINIAC* lost 15% of the battles, whereas we expected it to win all battles. *BRAINIAC* lost some battles in *map11*, *map12, map21*, *map30, map31, map52*, and *map53*, but only in the last one the number of defeats was lower than the number of wins. These results are substantially relevant because one may argue that the hyper-heuristic evolved a Commander that discovered and exploited weaknesses of *Kilroy*, no matter the map. Therefore, the first objective of this investigation - evolve a commander to beat Kilroy in map00 and maybe in other maps - was achieved. Actually, Kilroy was outperformed int the majority of the maps.

The second objective is the comparison with other Commanders in *map00*. *BRAINIAC* lost 27% of the battles for *Balanced*, 12% for *Random*, and 15% for *Greedy*. *Balanced* was good at killing *BRAINIAC*'s defenders. The other important aspect to notice is the result of the *Defender* Commander. For this map, and for various other maps, the percentage of Draws is either 100% or close to it. It was possible to observe that the number of attackers that *BRAINIAC* sent to capture the flag was very small, making them get killed. A different strategy should be evolved for this opponent. Finally, from a total of 180 battles (6 opponents, 30 runs), *BRAINIAC* Won 118 (66%), Drew 41 (22%) and Lost 21 (12%). These results were above expectations.

The third analysis is by map. The maps where *BRAINIAC* achieved more than or equal to 70% of wins were: *map02*, *map13*, *map30*, and *map52*. The worse results, more than or equal to 20% of losses, were in maps: *map11*, *map21*, *map30, map31*, *map40*, *map52*, and *map53*.

We tried to discover patterns to justify *BRAINIAC's* performance in those maps using the characteristics in Table I. However, based only on the results of this simulation, it can be said that the performance of the evolved bot in a specific map cannot be easy generalized. Other map details, not included in

Table IV. SIMULATION RESULTS IN PERCENTAGE (ROUNDED), 20 MAPS, 30 RUNS BY MAP. ZEROS ARE REPRESENTED AS '-'.

	BRAINIAC			*Kilroy*			*Balanced*			*Btsample*			*Defender*			*Random*			*Greedy*		
Map	***Won***	***Drew***	***Lost***	***Won***	***Drew***	***Lost***	***Won***	***Drew***	***Lost***	***Won***	***Drew***	***Lost***	***Won***	***Drew***	***Lost***	***Won***	***Drew***	***Lost***	***Won***	***Drew***	***Lost***
map00	0,66	0,23	0,12	0,15	0,09	0,76	0,27	0,15	0,58	-	0,03	0,97	-	0,91	0,09	0,12	0,12	0,76	0,15	0,06	0,79
map01	0,25	0,70	0,05	-	1,00	-	-	0,97	0,03	-	0,57	0,43	-	1,00	-	0,13	0,33	0,53	0,17	0,33	0,50
map02	0,81	0,15	0,04	-	-	1,00	0,07	0,13	0,80	-	0,07	0,93	0,13	0,33	0,53	0,03	0,17	0,80	0,03	0,20	0,77
map03	0,19	0,68	0,13	-	0,43	0,57	0,03	0,87	0,10	-	0,77	0,23	-	1,00	-	0,03	0,87	0,10	0,73	0,13	0,13
map10	0,56	0,35	0,09	-	0,10	0,90	0,07	0,57	0,37	-	0,03	0,97	-	1,00	-	0,23	0,17	0,60	0,27	0,23	0,50
map11	0,63	0,15	0,22	0,03	0,03	0,93	0,20	0,37	0,43	-	0,03	0,97	0,33	0,07	0,60	0,23	0,20	0,57	0,50	0,20	0,30
map12	0,65	0,19	0,16	0,17	0,27	0,57	0,13	0,30	0,57	0,03	0,17	0,80	0,37	0,13	0,50	0,03	0,17	0,80	0,23	0,10	0,67
map13	0,75	0,24	0,01	-	0,20	0,80	0,03	0,30	0,67	-	0,03	0,97	0,03	0,90	0,07	-	-	1,00	-	-	1,00
map20	0,32	0,60	0,08	-	0,67	0,33	0,03	0,83	0,13	0,07	0,60	0,33	-	1,00	-	0,13	0,40	0,47	0,27	0,10	0,63
map21	0,22	0,50	0,28	0,23	0,50	0,27	0,50	0,40	0,10	0,03	0,47	0,50	-	1,00	-	0,23	0,50	0,27	0,67	0,13	0,20
map22	0,63	0,26	0,11	-	0,10	0,90	0,27	0,57	0,17	-	0,07	0,93	0,13	0,53	0,33	-	0,20	0,80	0,27	0,10	0,63
map23	0,22	0,78	-	-	0,60	0,40	-	1,00	-	-	0,70	0,30	-	1,00	-	-	0,83	0,17	-	0,53	0,47
map30	0,72	0,05	0,23	0,11	0,09	0,80	0,09	-	0,91	-	0,06	0,94	0,97	0,03	-	-	-	1,00	0,20	0,14	0,66
map31	0,37	0,33	0,30	0,09	0,31	0,60	0,69	0,09	0,23	-	0,49	0,51	0,29	0,23	0,49	0,11	0,71	0,17	0,63	0,17	0,20
map32	0,37	0,59	0,04	-	0,26	0,74	0,03	0,86	0,11	-	0,46	0,54	-	1,00	-	0,14	0,66	0,20	0,06	0,31	0,63
map33	0,19	0,81	-	-	1,00	-	-	1,00	-	-	0,91	0,09	-	1,00	-	-	0,94	0,06	-	0,03	0,97
map40	0,55	0,22	0,23	-	0,51	0,49	0,06	0,14	0,80	-	0,11	0,89	0,46	0,20	0,34	0,06	0,23	0,71	0,83	0,11	0,06
map51	0,60	0,22	0,18	-	-	1,00	0,34	0,23	0,43	0,20	0,37	0,43	0,11	0,63	0,26	0,06	-	0,94	0,34	0,11	0,54
map52	0,70	0,05	0,24	0,14	0,09	0,77	0,80	0,09	0,11	-	-	1,00	0,03	0,03	0,94	0,17	0,09	0,74	0,31	0,03	0,66
map53	0,56	0,24	0,20	0,29	0,46	0,26	0,17	0,43	0,40	-	0,06	0,94	0,37	0,20	0,43	0,11	0,09	0,80	0,26	0,20	0,54
Average	0,50	0,37	0,13	0,06	0,34	0,60	0,19	0,46	0,35	0,02	0,30	0,68	0,16	0,61	0,23	0,09	0,33	0,57	0,30	0,16	0,54

the study, may be more adequate. If this hypothesis is correct, then it may be possible to generate a Commander that chooses a strategy based on the map.

Surprisingly, the Commander that defeated *BRAINIAC* most (in average) was the *Greedy* Commander (30%) and not *Kilroy*. However, as previously stated, it can suggest that *BRAINIAC* exploited *Kilroy*'s Achille's heel. To also outperform the Greedy Commander, *BRAINIAC* should battle against it during the evolution phase, what requires only a small change in the API's competition tool to add that Commander as a new opponent in the list.

In general, as can be seen by the average values, *BRAINIAC* has won approximatelly half of the battles (50%), drew 37% and lost only 13%, making it the clear winner of the simulated competition.

VII. CONCLUSIONS AND FUTURE WORK

The development of high-quality intelligent behavior for bots in games is a hard task to be performed manually, and thus the use of optimization and machine learning algorithms has been emerging as a trend. However, using this kind of algorithms to develop a bot AI from scratch is not achievable yet.

In this paper, we have investigated the use of a Hyper-heuristic Genetic Algorithm (ALChyper-GA) to evolve the AI of a Commander to control bots in a Capture The Flag game. Using blocks of codes from open source Commanders, the hyper-heuristic evolved the Commander, named *BRAINIAC*, by finding an arrangement of available blocks to outperform a specific opponent (*Kilroy*, which seemed to be the best available Commander) at a specific map (*map00*).

To evaluate *BRAINIAC* we performed 3600 battles against several opponents in several maps. *BRAINIAC* lost 6% of the battles for *Kilroy* and 13% of all battles. It is very important to remember that *BRAINIAC* was not expected to perform well in any map different from *map00*.

In this paper we have shown that: 1) it is possible to evolve a Commander using blocks of code from other Commanders; 2) it is possible to evolve a specialized Commander able to exploit the weaknesses of an opponent; 3) besides being specialized, the Commander is capable of defeating new opponents in unseen scenarios, showing a high degree of adaptability; 4) a Commander specialized at defeating an advanced opponent may not guarantee that all other opponents will be easily defeated; 5) there are very strong indicatives that the automated design of good Commanders is an achievable task and that high-quality Commanders may be generated from high-quality opponents, instead of using the simple examples from the API.

This paper worked as a proof-of-concept, and several studies can be made in future works. Different ALChyper-GA configurations can be tested, more Commanders can be used in the evolution, more maps, longer matches, a different evaluation function, new blocks of code, etc.

Other important aspect is that several blocks of codes present numerical values that can be tuned by optimization algorithms (another Genetic Algorithm, for instance), increasing the performance of the evolved Commander.

As a final contribution, by the results obtained in this work it is reasonable to expect that a hyper-heuristic may also be successfully employed to generate codes for other kinds of games, widening the applications of the proposed ALChyper-GA.

REFERENCES

[1] I. Sakellariou, "Agent based modelling and simulation using state machines." in *SIMULTECH*. SciTePress, 2012, pp. 270–279.

[2] S. M. S. Nugroho, I. Widiastuti, M. Hariadi, and M. H. Purnomo, "Fuzzy coordinator based intelligent agents for team coordination behavior in close combat games," *Theoretical and Applied Information Technology*, vol. 51, May 2013.

[3] D. Isla, "Handling Complexity in the Halo 2 AI," in *Game Developers Conference '05*. Bungie Studios, Mar. 2005, pp. 1–11. [Online]. Available: $http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml$

[4] R. Pillosu, "Coordinating Agents with Behaviour Trees," in *Game/AI Conference '09*. Paris: Crytek, 2009, pp. 1–29. [Online]. Available: $http://staff.science.uva.nl/~aldersho/GameProgramming/Papers/Coordinating_Agents_with_Behaviour_Trees.pdf$

[5] M. Dyckhoff, "Decision Making and Knowledge Representation in Halo 3," Bungie Studios, Tech. Rep., Aug. 2008. [Online]. Available: $http://www.bungie.net/images/Inside/publications/presentations/publicationsdes/engineering/nips07.pdf$

[6] B. Knafla. (2011, Feb.) Introduction to Behavior Trees. $http://altdevblogaday.org/2011/02/24/introduction-to-behavior-trees/$. [Online]. Available: $http://altdevblogaday.org/2011/02/24/introduction-to-behavior-trees/$

[7] T. Bäck, "An overview of parameter control methods by self-adaptation in evolutionary algorithms," *Fundam. Inf.*, vol. 35, no. 1-4, pp. 51–66, Jan. 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=2379195.2379199

[8] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

[9] S. Bakkes, P. Spronck, and E. Postma, "Team: The team-oriented evolutionary adaptability mechanism," in *Entertainment Computing–ICEC 2004*. Springer, 2004, pp. 273–282.

[10] N. Cole, S. J. Louis, and C. Miles, "Using a genetic algorithm to tune first-person shooter bots," in *Evolutionary Computation, 2004. CEC2004. Congress on*, vol. 1. IEEE, 2004, pp. 139–145.

[11] A. M. Mora, R. Montoya, J. J. Merelo, P. G. Sánchez, P. Á. Castillo, J. L. J. Laredo, A. I. Martínez, and A. Espacia, "Evolving bot ai in unrealtm," in *Applications of Evolutionary Computation*. Springer, 2010, pp. 171–180.

[12] S. Priesterjahn, O. Kramer, A. Weimer, and A. Goebels, "Evolution of human-competitive agents in modern computer games," in *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*. IEEE, 2006, pp. 777–784.

[13] K. Sastry, D. Goldberg, and G. Kendall, "Genetic algorithms," in *Search Methodologies*. Springer, 2005, pp. 97–125.

[14] P. Ross, "Hyper-heuristics," in *Search methodologies*. Springer, 2005, pp. 529–556.

[15] G. L. Pappa, G. Ochoa, M. R. Hyde, A. A. Freitas, J. Woodward, and J. Swan, "Contrasting meta-learning and hyper-heuristic research: the role of evolutionary algorithms," *Genetic Programming and Evolvable Machines*, pp. 1–33, 2013.

[16] (2013, Jul.) aisandbox.com. [Online]. Available: http://aisandbox.com/

[17] (2013, Apr.) Kilroy commander. [Online]. Available: https://code.google.com/p/sqlitebot/wiki/Kilroy

[18] (2013, Apr.) Behavior-tree sample commander. [Online]. Available: https://code.google.com/p/knowthismusic/source/browse/ctf1/btSampleCommander.py

[19] L. Han, G. Kendall, and P. Cowling, "An adaptive length chromosome hyperheuristic genetic algorithm for a trainer scheduling problem," *SEAL2002*, pp. 267–271, 2002.

[20] P. Cowling, G. Kendall, and L. Han, "An investigation of a hyperheuristic genetic algorithm applied to a trainer scheduling problem," in *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, vol. 2. IEEE, 2002, pp. 1185–1190.

[21] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[22] K. Sastry, D. Goldberg, and G. Kendall, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer, 2005, ch. Genetic Algorithms, pp. 97–125.

[23] S. Zhao and L. Jiao, "Multi-objective evolutionary design and knowledge discovery of logic circuits based on an adaptive genetic algorithm," *Genetic Programming and Evolvable Machines*, vol. 7, no. 3, pp. 195–210, Oct. 2006.

[24] V. V. de Melo and G. L. C. Carosio, "Automatic generation of evolutionary operators: a study with mutation strategies for the differential evolution," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, 2013, pp. 188–193.

[25] A. Fukunaga, "Automated discovery of composite sat variable-selection heuristics," in *Eighteenth national conference on Artificial intelligence*. Menlo Park, CA, USA: American Association for Artificial Intelligence, 2002, pp. 641–648. [Online]. Available: http://dl.acm.org/citation.cfm?id=777092.777191

[26] E. K. Burke, M. R. Hyde, G. Kendall, and J. Woodward, "A genetic programming hyper-heuristic approach for evolving two dimensional strip packing heuristics," *IEEE Transactions on Evolutionary Computation*, vol. 14, no. 6, pp. 942–958, 2010. [Online]. Available: http://www.asap.cs.nott.ac.uk/publications/pdf/Hyde2010.pdf

[27] R. C. Barros, M. P. Basgalupp, A. C. P. L. F. de Carvalho, and A. A. Freitas, "A hyper-heuristic evolutionary algorithm for automatically designing decision-tree algorithms," in *GECCO*, 2012, pp. 1237–1244.

[28] L. Abednego and D. Hendratmo, "Genetic programming hyper-heuristic for solving dynamic production scheduling problem," in *International Conference on Electrical Engineering and Informatics (ICEEI 2011)*, Bandung, Indonesia, 17-19 Jul. 2011, pp. K3–2.

[29] N. Pillay and W. Banzhaf, "A study of heuristic combinations for hyper-heuristic systems for the uncapacitated examination timetabling problem," *European Journal of Operations Research*, vol. 197, no. 2, pp. 482–491, 1 Sep. 2009.