Evolving Reactive Micromanagement Controller for Real-Time Strategy Games

[1]*Martin ČERTICKÝ*, [2]*Michal ČERTICKÝ*

[1]Dept. of Cybernetics and Artificial Intelligence, FEI TU of Košice, Slovak Republic
[2]Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic

[1]martin.certicky@student.tuke.sk, [2]michal.certicky@agents.fel.cvut.cz

***Abstract*—Real-Time Strategy (RTS) games are a genre of video games representing an interesting, well-defined adversarial domain for Artificial Intelligence (AI) research. One of many sub-problems that RTS players need to solve is the micromanagement of individual units (simple agents carrying out player's commands) during combat. Numerous multi-agent reactive control mechanisms have already been developed to maximize the combat efficiency of controlled units. Majority of these mechanisms make use of numeric parameters that need to be fine-tuned in order to achieve desired behavior. Due to a large number of these parameters, assigning them manually is inconvenient and training them by machine learning methods usually takes a long time (search space is too large). To reduce the search space and accelerate the training, we propose a simple reactive controller with only eight parameters. We implement it for a classic RTS game StarCraft: Brood War and train the parameters using genetic algorithms. Our experiments demonstrate an impressive combat performance after only a small number of generations.**

***Keywords*—Genetic Algorithms, Evolution, RTS, micromanagement, StarCraft.**

I. Introduction

Real-time Strategy (RTS) games, as a genre of video games in which players manage economic and strategic tasks by gathering resources and building bases, increase their military power by researching new technologies and training units, and lead them into battle against their opponent(s), serve as an interesting domain for Artificial Intelligence (AI) research. They represent a well-defined, complex adversarial systems [1] which pose a number of interesting AI challenges in the areas of planning, dealing with uncertainty, domain knowledge exploitation, task decomposition and, most relevant to the scope of this article, spatial reasoning and machine learning [2].

Research in the area of RTS game AI is usually classified according three levels of abstraction: high-level strategy, middle-level tactics and low-level reactive unit control (referred to as "micromanagement" by RTS players). Reactive control, as a challenge addressed by this paper, aims at maximizing the effectiveness of individual units of different types in combat by moving them on the battlefield and selecting the attack targets in response to terrain and the activity of opponent's units.

Decision-making for reactive unit control can in general take centralized or decentralized (distributed) approach. Centralized approaches try to control the whole group of units at once by searching a game tree. For example, Churchill et al. [3] presented a variant of alpha-beta search and Wang et al. [4] employed a Monte-Carlo planning approach to the problem of micromanagement. Unfortunately, centralized solutions are usually too slow due to large search space and can only be applied to situations with low unit counts.

Decentralized approaches are much more common, because they scale better to situations with higher numbers of controlled units. The search space is significantly reduced by making the control decisions for each unit independently of others. Decentralized solutions, which often take advantage of potential fields or influence maps, have one thing in common each unit is controlled by a relatively simple controller algorithm. The unit controller is a state machine that reacts solely to current game state without performing any kind of lookahead search.

The controller always comes with several parameters that need to be fine-tuned in advance to achieve a desired effective behavior. There has been a significant amount of work using machine learning techniques like Reinforcement Learning, Bayesian Modelling or Genetic Algorithms (GA) to train the parameters of underlying controller algorithm automatically – multiple examples of research in this area can be found in a survey paper by Ontanón et al. [2]. However, a general drawback of decentralized micromanagement techniques is the high number of controller's parameters, leaving us with too many possible value assignments [2] (problem of dimensionality). For example, Liu et al. [5] used GA to train the values of 14 different parameters (chromosome encoded as a 60-bit string), Ponsen [6] used 20 genes to train 20 parameters for combat-related actions and Sandberg el al. [7] trained 21 parameters of their micromanagement controller.

To address the problem of dimensionality, we propose a controller with only eight parameters, as described in Section II. We train these parameters using the GA with the population of 32 individuals, roulette wheel selection method and uniform cross-over method, according to detailed description in Section III. Finally, the performance of the trained controller is discussed in Section IV.

We implemented and tested our solution in a classic RTS game *StarCraft: Brood War* by Blizzard Entertainment, which was accessed via application programming interfaces BWMirror[1] and BWAPI[2].

II. Unit Controller

We propose a simple unit controller, implemented as a finite state machine with as few adjustable parameters as

[1]http://github.com/vjurenka/BWMirror/
[2]http://github.com/bwapi/bwapi/

possible to address the problem of dimensionality – low number of parameters to fine-tune reduces the assignment search space and makes the training faster. The controller is called individually for each unit on each logical frame of the game (approximately 23.81 times per real-world second on the *"Fastest"* game speed). On each frame, a unit is either left untouched or is assigned a new command. There are two types of commands that can be assigned by the controller: `move(Position pos)` or `attack(Unit enemy)`. See the flow chart describing the controller's implementation in Figure 1.

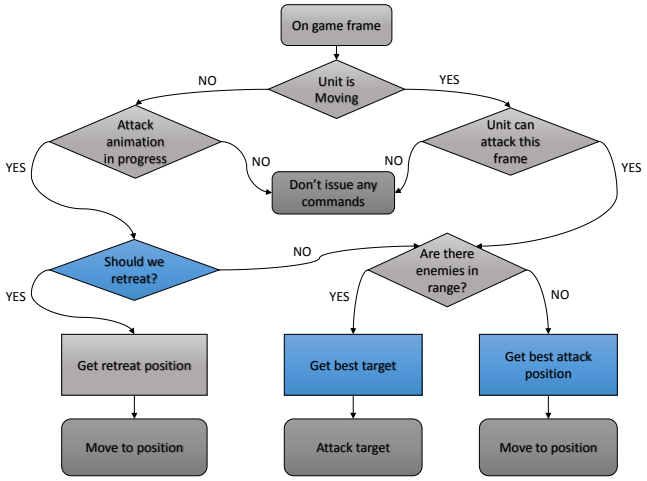

Fig. 1. Flow chart describing the reactive unit controller. Parts of the controller using parameters trained by GA are highlighted by the blue color.

First, the controller checks if a given unit is currently executing a `move` command. If it is, it checks if the unit is able to attack at this frame (units in StarCraft must wait a specified number of frames, called *"weapon cooldown"*, between individual attacks). If the unit is able to attack and there are some enemy units (targets) in its attack range, the controller selects the best target and issues an `attack` command (target selection is explained in subsection II-A). If there are no targets in range, the best position from its surrounding area is selected (subsection II-B) and the unit receives a `move` command sending it there. If the unit is already executing an *attack* command on the current frame but it's not in the middle of attack animation (attack animation should not be interrupted), it has two options: continue attacking or retreat to a safer position[3]. The decision to retreat is parametrized and described in subsection II-C, but the selection of retreat position is hard-coded – the controller simply computes the retreat vector directed away from the biggest enemy threats.

Following three subsections describe all the functions containing the parameters that were fine-tuned by the genetic algorithm. The total number of parameters is eight and each of them has a value from interval $[0, 1]$.

A. Selecting an Attack Target

The following function is used to select the most appropriate target from all the enemy units in our unit's attack range. The function scores enemy units and outputs the one with the highest score to be used as an argument for the `attack` command.

$$Score = (D \cdot p_1) - (HP \cdot p_2) + (L \cdot p_3)$$

where D is the damage of a given enemy unit (multiplied by 3 to increase its significance without normalising the value) and HP is the sum of its remaining Hit Points and Shields. The L variable equals 100 if the sum of unit's remaining Hit Points and Shields is lower than our unit's damage. Otherwise, the L variable equals 0. Each of the variables in this function is multiplied by an evolved parameter (p_1, p_2, p_3).

B. Selecting an Attack Position

Next function selects the most appropriate position for a unit to move to when it intends to attack enemy targets but has no targets in range. This function computes the score of a collection of walk tiles surrounding our unit (see Figure 2) and returns the position of the one with the highest score. This position is used as an argument for the `move` command.

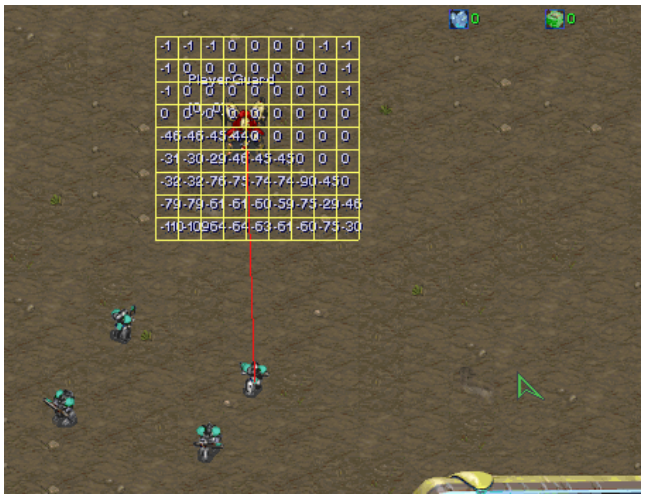

Fig. 2. Walk tiles around our unit with the assigned scores. Walk tiles in range of multiple enemy units have the lowest score.

We use the following equation to score a walk tile. Again, the one with the highest score is used as a function's output.

$$Score = D_f \cdot p_4 - D_t \cdot p_5 - L \cdot p_6 - C \cdot p_7$$

where D_f is the sum of the damage our unit is able to inflict from the examined tile and D_t is the sum of the damage that the enemy units are able to deal to our unit if it is moved to that tile. The L variable is similar to the one from the previous method – it equals 100 if the sum of the damage of enemy units having range on examined tile is higher than remaining Hit Points and Shields of our unit (equals 0 if not). Variable C equals 50 if the path to the examined position crosses the path of a moving friendly unit or if that path collides with any non-moving unit (equals 0 if not). Again, all the variables are multiplied by evolved parameters (p_4-p_7).

C. Deciding to Retreat

The last parametrized function returns `True` if our unit should retreat to a safer location. The retreat decision is made if the following holds:

[3]We intentionally do not let units switch between the targets in their range because such switch causes a long pauses between the attacks in StarCraft. This, in general, negatively impacts the performance.

$$D \geq HP \cdot p_0$$

where D is sum of damage of all units currently attacking our unit and HP is the sum of remaining Hit Points and Shields of our unit. The HP variable is multiplied by an evolved parameter p_0.

III. Evolving the Parameters

It is known that evolutionary optimization by simulating fights can be easily adapted to any parameter-dependent micromanagement control model [2]. To train the parameters of our controller, we used three different scenarios – each scenario had our AI player face a different type of commonly used enemy units (more details can be found in the following section). Enemy units were controlled by standard built-in StarCraft AI script. The controller was trained using the population of 32 individuals during 15 generations.

A. Encoding

Every individual is represented by a vector of 8 real numbers from the interval $[0, 1]$, each representing a single parameter used in our controller.

B. Evaluation

The game ends if one of the players loses all the units or if the game hits the predefined time limit. After that, we evaluate the current individual using the sum of remaining Hit Points and Shields of all its surviving units and substract the Hit Points and Shields of surviving enemy units. The final fitness of an individual is averaged over 5 games to lower the randomness caused by the StarCraft mechanics and by varying behavior of the enemy AI script.

C. Selection Mechanism

We use a roulette-wheel selection method in our GA. Since negative fitness values are not acceptable for this method, we apply windowing to ensure the positive values during the selection. Exactly half of the population is selected for the parent role (while each individual may be selected multiple times). Slight form of elitism is also applied on the populations – the best member of each population is guaranteed to become a parent at least once.

D. Genetic Operators

1) Cross-over: The GA uses a uniform cross-over with the mixing ratio of 0.5. The offspring has half of the genes of each parent on average, as the cross-over points are chosen randomly throughout the gene.

2) Mutation: We apply a 10% chance of uniform mutation for each individual carrying out to the next generation. The mutation operator is applied after cross-over of the population is done to prevent mutating the individuals prematurely and possibly losing well-scored offsprings.

IV. Performance

In the training scenarios, our AI player controlled a squad of *"Dragoon"* type units (widely used StarCraft unit with an average attack range). Three scenarios with different types of enemy units were chosen for the training due to difference in desired behavior depending on enemy unit's attack range. When facing melee (close-combat) or short-ranged units, the Dragoons have a major advantage if they learn to apply so-called *"kiting"* behavior (a hit-and-retreat technique commonly used in RTS games). On the other hand, too frequent retreating while facing enemy units with the same or longer attack range might lead to decrease in the performance.

The performance was compared to standard built-in StarCraft AI script (the variant used in "custom game" mode). We computed the fitness of the built-in AI using the same fitness function and averaged the value over 5 games.

A. Dragoons vs. Zealots scenario

Zealots belong among the most commonly used melee units in StarCraft. After the initial generation, the majority of individuals adopted similar values of first parameter (handling the retreat decisions) and showed an elusive kiting behavior patterns during their games. After only a few generations, the results of the controller were fairly close to absolute optimum (see Figure 3). The controller was outperforming the native AI massively.

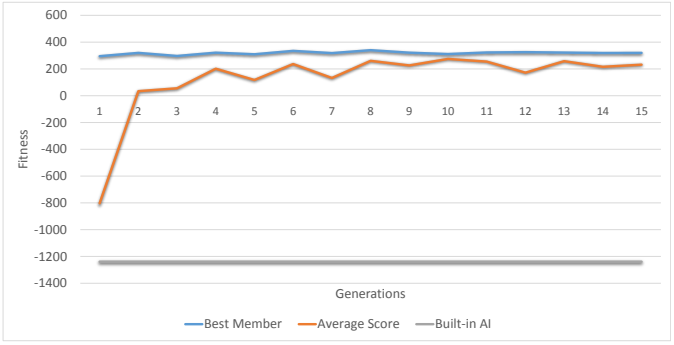

Fig. 3. Dragoons vs. Zealots scenario: Fitness of the best individual (blue line) and the average fitness of the population (red line) over time (15 generations) compared to the fitness of built-in AI (grey line).

B. Dragoons vs. Marines scenario

Marines were chosen for the second scenario as the cheap and common short-ranged unit. The convergence of the GA was not as fast as in first scenario (see Figure 4), yet the behavior of the AI player ended up being quite similar. The units became elusive, taking advantage of their superior range and kiting the enemy units. It should be noted that with the improving performance of the AI player, the games became longer. This was due to dragoons kiting the enemy units, attacking only if they were in safe distance.

C. Dragoons vs. Dragoons (mirror match scenario)

The last experiment was performed on a mirror match scenario, where the AI player faced a set of units identical to its own. While the first two scenarios resulted in an observable behavioral difference compared to native AI, this scenario was not able to produce the results as impressive as the previous

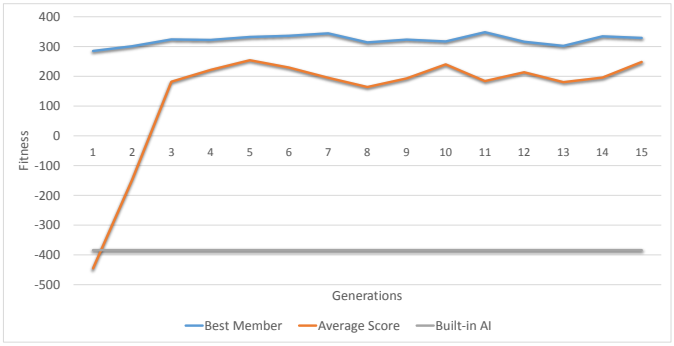

Fig. 4. Dragoons vs. Marines scenario: Fitness of the best individual (blue line) and the average fitness of the population (red line) over time (15 generations) compared to the fitness of built-in AI (grey line).

two. Kiting the enemy army with same number of units and the same attack range is ineffective – even moving the units may result in worse scores. Our individuals gradually learned to reduce their movement and trained only the target selection. After a few generations the AI player developed a behavior similar to that of built-in AI scripts, which is also reflected in the comparable fitness values (see Figure 5).

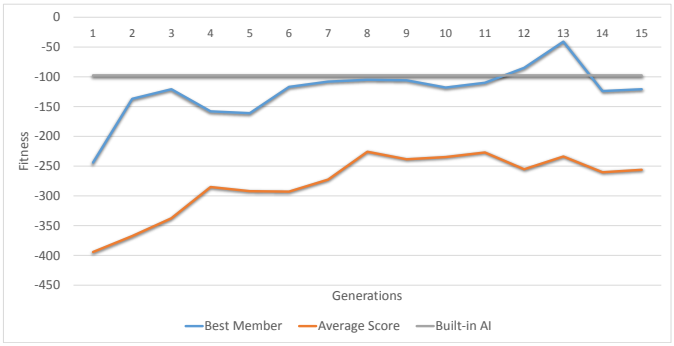

Fig. 5. Dragoons vs. Dragoons scenario: Fitness of the best individual (blue line) and the average fitness of the population (red line) over time (15 generations) compared to the fitness of built-in AI (grey line).

In two out of three scenarios, our AI player surpassed the built-in AI's performance significantly. The controller was performing best while facing melee or short-ranged units. When facing long-ranged units (the mirror scenario), the controller was performing slightly worse than the native AI.

V. Conclusion

In this paper, we addressed the dimensionality problem, typical for current parametrized RTS micromanagement solutions. The problem is caused by the large number of numeric parameters of the control mechanisms, that need to be fine-tuned in order to achieve a desired effective behavior.

To reduce the search space and accelerate the training of these parameters, we proposed a simple reactive controller with only eight parameters. We implemented it for a classic RTS game StarCraft: Brood War and trained the parameters using genetic algorithm with roulette-wheel selection and uniform cross-over methods.

Thanks to the reduction of search space, less than 10 generations with the population size of 32 proved sufficient for GA to converge in every experimental scenario.

Compared to a built-in StarCraft AI, the performance proved to be exceptionally good in two out of three experimental scenarios (in the third one, it was comparable to that of the built-in AI). The results depended on the type of enemy units presented in each scenario. The hit-and-retreat behavior of the AI player that emerged in two out of three scenarios was extremely effective against melee or short-range units. It proved to be ineffective against long-range enemy units and was suppressed after a few generations to be replaced with more static behavior.

The proposed controller has proven to be effective in typical StarCraft combat situations only after a short training by genetic algorithm.

References

[1] M. Buro, "Call for AI research in RTS games," in *Proceedings of the 4th Workshop on Challenges in Game AI*, 2004, pp. 139–142.

[2] S. Ontanón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A survey of real-time strategy game AI research and competition in StarCraft," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 5, pp. 293–311, 2013.

[3] D. Churchill, A. Saffidine, and M. Buro, "Fast heuristic search for rts game combat scenarios." in *AIIDE*, 2012.

[4] Z. Wang, K. Q. Nguyen, R. Thawonmas, and F. Rinaldo, "Monte-carlo planning for unit control in starcraft," in *2012 IEEE 1st global conference on consumer electronics (GCCE). IEEE, New York*, 2012, pp. 263–264.

[5] S. Liu, S. J. Louis, and C. Ballinger, "Evolving effective micro behaviors in rts game," in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*. IEEE, 2014, pp. 1–8.

[6] M. Ponsen, "Improving adaptive game ai with evolutionary learning," Ph.D. dissertation, Citeseer, 2004.

[7] T. W. Sandberg and J. Togelius, "Evolutionary multi-agent potential field based ai approach for ssc scenarios in rts games," Ph.D. dissertation, Masters thesis, University of Copenhagen, 2011.