Combining genetic algorithm and swarm intelligence for task allocation in a real time strategy game

Anderson R. Tavares, Gianlucca Lodron Zuin, Héctor Azpúrua, Luiz Chaimowicz
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil
{*anderson,gzuin,hector.azpurua,chaimo*}*@dcc.ufmg.br*

***Abstract*—Real time strategy games are complex scenarios where multiple agents must be coordinated in a dynamic, partially observable environment. In this work, we model coordination as a task allocation problem, in which specific tasks must be properly assigned to agents. We employ a task allocation algorithm based on swarm intelligence and adjust its parameters using a genetic algorithm. A fitness estimation method is employed to accelerate execution of the genetic algorithm. To evaluate this approach, we implement this coordination mechanism in the AI of a popular video game: *StarCraft: BroodWar*. Experiment results show that the genetic algorithm successfully adjusts task allocation parameters. Besides, we assess the trade-off between solution quality and execution time of the genetic algorithm with fitness estimation.**

***Keywords*-Task allocation; Evolutionary algorithms; Real-time strategy**

I. Introduction

Real Time Strategy (RTS) games have become one of the most successful genres in the game industry. Normally, RTS games are played at a extremely fast pace and players have to deal simultaneously with several different objectives such as resources collection, base construction, technology improvements and battles against enemy armies [1]. Due to their characteristics, RTS games became excellent testbeds for artificial intelligence (AI) research [2]. Particularly, the coordination of multiple units, or agents, in RTS games is an interesting research topic and has several commonalities with other complex scenarios such as rescue operations in disaster situations [3] or cooperative robotics [4].

Agent coordination in complex scenarios is one of the greatest challenges in multiagent intelligent systems and normally involves the optimized use of resources by different agents to accomplish a global goal. In general, complex scenarios comprise a set of agents that must perform multiple tasks in a dynamic and partially observable environment, where existing tasks can disappear and new tasks can arrive. Therefore, task allocation becomes an important part of the coordination problem. This work builds on [5], where a task allocation approach is proposed for the complex scenarios determined by the RTS game *StarCraft: BroodWar* (*StarCraft* for short). A genetic algorithm is employed to adjust the parameters of Swarm-GAP, a probabilistic, scalable algorithm for task allocation based on swarm intelligence [6]. This paper presents an extended discussion on fitness estimation, employed to accelerate the genetic algorithm, and compared performance in matches between the implemented approach and *StarCraft* tournament bots.

Experiments in matches against *StarCraft's* native AI show that the genetic algorithm is useful to adjust task allocation parameters for Swarm-GAP. The proposed approach performs similarly to a configuration of Swarm-GAP parameters adjusted by hand, outperforming it in a specific case. Moreover, manual parameter adjustment requires domain knowledge, e.g., which game units would be more suitable for which tasks, whereas the proposed approach is domain-independent in this sense, and thus can be easily extended to other coordination domains.

Genetic algorithm execution consumes considerable time because evaluation of individuals depends on the execution of *StarCraft* matches. Thus, we test a method to estimate fitness of some individuals in order to accelerate genetic algorithm execution [7]. When the individual is not evaluated, its fitness is estimated based on its parents'. We vary the probability of forced evaluations to compare execution time and solution quality of the genetic algorithm. Experiments in this subject show that the obtained speedup is inversely proportional to the probability of forced evaluation. Moreover, our results indicate points where time savings occur without loss of solution quality.

The remainder of this paper is organized as follows: Section II introduces some basic concepts regarding task allocation, specifically the Extended Generalized Assignment Problem (E-GAP) and the Swarm-GAP algorithm. Section III discusses some related work both in the fields of task allocation in complex scenarios and AI for RTS games. Section IV discusses the adoption of Swarm-GAP for task allocation in *StarCraft*, while Section V presents the genetic algorithm used to evolve Swarm-GAP parameters. Experiments with the proposed approach are presented in Section VI while Section VII brings the conclusion and directions for future work.

II. Task allocation

A. E-GAP

Task allocation is concerned with the assignment of tasks to agents in order to maximize a global metric of performance, usually related to the skills of the agents to perform each task. In dynamic environments, task allocation can be modeled with the E-GAP (Extended Generalized Assignment Problem) formalism [8]. E-GAP generalizes the GAP (Generalized Assignment Problem), which is NP-complete [9].

The E-GAP can be formalized as follows: let $\mathcal{I}$ be the set of agents and $\mathcal{J}$ the set of tasks. Each agent $i \in \mathcal{I}$ has r_i resources to perform tasks. Each task $j \in \mathcal{J}$ consumes c_{ij} resources of agent i, when it performs the task. Each agent i has a capability $k_{ij} \in [0, 1]$ to perform task j. Capability can be regarded as the skill of the agent to perform the task.

An allocation matrix $A_{|\mathcal{I}|\times|\mathcal{J}|}$ has its elements a_{ij} set to 1 if agent i performs task j, and 0 otherwise. In this model, only one agent can perform a given task instance.

In E-GAP, the total reward W is calculated as the sum of agents' rewards along t timesteps. In one timestep, reward is calculated by considering the capability of the agents to perform the tasks they were assigned. A delay cost d_j^t is applied as a penalty for not allocating task j in timestep t, as Eq. 1a illustrates. Reward calculation along the timesteps captures the dynamics of the environment, i.e., reward in a given timestep depends on the tasks and agents that exist in that timestep. Equation 1b determines that agents must allocate tasks within their resource limits and Eq. 1c determines that a task can be performed by only one agent. Thus, large tasks must be broken down into smaller tasks that can be performed by a single agent. E-GAP also considers task interdependence, but this aspect is not investigated in this work.

$$W = \sum_t \sum_{i^t \in \mathcal{I}^t} \sum_{j^t \in \mathcal{J}^t} k_{ij} \times a_{ij}^t - \sum_t \sum_{j^t \in \mathcal{J}^t} (1 - \sum_{i^t \in \mathcal{I}^t} a_{ij}^t) \times d_j^t \tag{1a}$$

$$\text{subject to:} \quad \forall t \forall i^t \in \mathcal{I}^t, \sum_{j^t \in \mathcal{J}^t} c_{ij}^t \times a_{ij}^t \leq r_i^t \tag{1b}$$

$$\text{and:} \quad \forall t \forall j^t \in \mathcal{J}^t, \sum_{i^t \in \mathcal{I}^t} a_{ij}^t \leq 1 \tag{1c}$$

B. Swarm-GAP

Swarm-GAP is an approximate algorithm for E-GAP, inspired by the division of labor in social insects. In swarms, or colonies of social insects, in general there are hundreds or thousands of members that work without explicit coordination. From the aggregation of individual actions of colony members, complex behaviors emerge. One characteristic of swarms is the ability to respond to changes in the environment by adjusting the numbers of members performing each task.

Observations about swarm behaviors are the base of the model presented in [10], where tasks have associated stimulus[1] and individuals have response thresholds for each task. Let $s_j \in [0, 1]$ be the stimulus associated with task $j \in \mathcal{J}$ and $\theta_{ij} \in [0, 1]$ be the response threshold of individual (agent) i to task j. The tendency, or probability, of individual i to engage in task j is given by $T_{ij} \in [0, 1]$, calculated using Eq. 2.

$$T_{ij} = \frac{s_j^2}{s_j^2 + \theta_{ij}^2} \tag{2}$$

In swarms, due to polymorphism, individuals may be more able to perform certain kinds of tasks. This characteristic is captured in Eq. 3, which determines the response threshold of individual i to task j according to its capability ($k_{ij} \in [0, 1]$) to perform task j.

$$\theta_{ij} = 1 - k_{ij} \tag{3}$$

The goal of Swarm-GAP is to allow agents to individually decide which task they will engage in a simple and efficient way, minimizing computational effort and communication between agents [6]. With Swarm-GAP, agents communicate via a token based protocol. When a given agent perceives new tasks, it creates a token with these tasks. The agent can receive tokens from other agents too. Either way, the token holder has the right to decide in which tasks of the token it will engage. The token with the remaining tasks is passed to a random agent that has not held the token before. This is formalized in Alg. 1, which is executed by each agent independently.

Although the execution of Swarm-GAP is simple, good performance requires a good adjustment of all parameters, i.e., we need to model the stimuli s_j for each task, the resources r_i that each agent has, as well as the capability k_{ij} and task cost c_{ij} for each agent and task. Finding a combination of these parameters that yields a good performance of Swarm-GAP can be very time-consuming specially in scenarios with several different types of agents and tasks.

[1]Stimulus intensity may be associated with pheromone concentration, the number of encounters with other individuals performing the task or another characteristic that individuals can measure.

Algorithm 1 Swarm-GAP

```
When tasks are perceived: $token \leftarrow$ {perceived tasks}
When message is received: $token \leftarrow$ {received token}
for all task $j \in token$ do
    if $random() < T_{ij}$ and $r_i > c_{ij}$ then
        Engage in task $j$
        $token \leftarrow token \setminus \{j\}$
        $r_i \leftarrow r_i - c_{ij}$
    end if
end for
Send $token$ to random agent that didn't see the token before
```

III. Related work

A. Task allocation in complex scenarios

An approximate and decentralized algorithm for the task allocation problem in complex scenarios is LA-DCOP [8], where the modeling of the task allocation problem as E-GAP is introduced. In LA-DCOP, agents communicate and achieve coordination through a token-based protocol, which inspired the one used in Swarm-GAP (see Section II-B). Agents perceive tasks on the environment and create a token containing the tasks. The token holder, based in a global threshold, decide in which tasks it should engage. The token with the remaining tasks is communicated to other agents. In LA-DCOP, agents must allocate tasks in order to maximize the sum of their capabilities, respecting their resource limitations. This can be reduced to the binary knapsack problem (BKP), which is NP-complete. Therefore, the effectiveness of LA-DCOP depends on the method that solves the BKP. Each agent solves several instances of the BKP during the complete process of allocation.

LA-DCOP and Swarm-GAP are very similar regarding the token-based protocol for coordination. The difference in both algorithms lies in the way the agents allocate tasks. In LA-DCOP, the process corresponds to solving an instance of the BKP whereas in Swarm-GAP, agents allocate tasks in a probabilistic fashion inspired by the division of labor in social insects.

Swarm-GAP and LA-DCOP are compared in the RoboCup Rescue domain [6], a simulated disaster response scenario with heterogeneous agents. The execution of constrained tasks, where several agents are needed to perform a task, is also studied. This particular restriction is not approached in our proposal.

The task allocation problem can also be modeled through the formalism of coalition formation. In this formalism, a coalition structure[2] is determined and coalitions of agents are assigned to tasks. An approach that uses this formalism is presented in [11]. The authors model coalition formation as a Markov Decision Process (MDP). Initially, the generated MDP has an intractable state-action space. Authors present a method for parallel partition of the generated MDP so that efficient algorithms can be applied. In fact, authors apply this methodology to problems with hundreds of agents and tasks. However, their experiments are performed with homogeneous agents and they must deal with a single task type, i.e., simulated firefighting.

Branch-and-bound fast-max-sum (BnB FMS) is an anytime algorithm that advances the state-of-the-art regarding task allocation in large-scale scenarios [12]. BnB FMS searches the coalition structure that maximizes a global utility, which considers the contribution of each agent in its coalition when it performs a set of tasks. The algorithm prunes its search space by reducing the number of tasks and coalitions that need to be evaluated. The pruning techniques keep the correctness and robustness of the algorithm when the environment is dynamic. Performed experiments showed that BnB FMS reaches a global utility 23% higher than previous state-of-the-art algorithms, with 31% less computing time and 25% less messages than other algorithms.

In this paper we use Swarm-GAP to perform task allocation, augmenting it with a genetic algorithm to tune its parameters. This is a novel approach, specially in a RTS game scenario, as discussed in the next section.

B. Artificial intelligence in RTS games

Developing intelligent systems for RTS games is a complex problem because, in addition to intrinsic constraints of the game, such as partial observation, there are two types of decision making: micro-management, responsible for single unit behavior (i.e. unit positioning, target selection and retreat during battles) and macro-management, responsible for sequencing the construction of structures, resource management and strategy. In this sense, task allocation is related with macro-management.

Probably these aspects make real-time strategy games less researched than their turn-based counterparts [13], despite the great number of know open challenges such as: adversarial real-time planning, decision making under uncertainty of partially observable domains, learning from experience or observation, opponent modeling, spatial and temporal reasoning, navigation, resource management and collaboration [14], [15].

Common techniques used to tackle these challenges include:

[2]A coalition structure is a partition of the set of agents. Each subset of the partition is a coalition.

- Dynamic scripting [16], [17]. In dynamic scripting, each agent has a rule base, where each rule consists of actions activated in certain conditions. Rules are inserted in an active set or removed from it according to their performance. A drawback is that this technique requires domain knowledge to build the rule base;
- Game-tree search in abstract representations. To handle the huge state spaces of RTS games, game-tree search algorithms are extended and used in abstract game representations, e.g. [18], [19]. However, commercial RTS games, such as *StarCraft*, usually do not provide a forward model of the world, so that actions cannot be simulated in order to obtain successor states. This way, search-based techniques cannot be directly applied;
- Use of Bayesian models. In [20], authors implement Bayesian models for micro-management. Their approach outperform *StarCraft's* native AI and several other bots on unit micro-management;
- Potential fields, developed first as a navigation method to avoid obstacles in robotics [21], can be seen used in conjunction with a modified $A*$ algorithm for navigation in *StarCraft* game [22]. Their results have shown improvements over the navigation using only the $A*$ algorithm. Other works implement potential fields in conjunction with fuzzy measurements for micro-management in the game *Warcraft III* [23]. Their results have shown success on minimizing the score of the opponent on most cases;
- Evolutionary algorithms. This method is generally used to adjust the parameters of other techniques. Although evolutionary algorithms have been implemented in RTS games, the focus has been on unit micro-management [24], [25]. In the present work, the focus is on macro-management, which is related to task allocation (i.e., which tasks should be done), whereas micro-management is concerned with how the tasks should be done.

An example of evolutionary algorithm applied to evolve parameters of a RTS game player can be found in [13]. The authors define a rule-based player with adjustable parameters that are evolved with genetic algorithm. The approach is tested in a game called Planet Wars. In this game, a match takes place in a map containing several planets, each one with a number of space ships in it. The action of a player consists in sending space ships to other planets in order to conquer them. The player who conquers all enemy's planets is the winner. This scenario is simpler than the one studied here (see Section IV-A). In our scenario, the number of distinct actions (or tasks) that need to be performed is substantially higher.

IV. Swarm-GAP in *StarCraft*

A. StarCraft *game*

StarCraft is a RTS game, whose domain has several real-world characteristics, some of which are enumerated below.

- The environment is continuous in space and time (or at least it is discretized in a reasonably thin granularity);
- Partial observability: a player can only access information within the visual range of his units and buildings;
- Dynamicity: due to the actions of several agents, the environment is always changing, thus demanding quick decisions;
- The decision process is sequential, meaning that actions taken now affect which actions can be done in the future.

In RTS games, there is a need to perform hundreds of actions per minute. Actions are divided in several tasks involving resource gathering, creation of new units, construction of buildings and attacks to the enemy [26].

StarCraft has an application programming interface, called BWAPI [27], designed to foster the development of artificial intelligence techniques. BWAPI is capable of retrieving the same information and sending the same commands a human player is allowed in *StarCraft*.

In *StarCraft*, there are three races with different characteristics and unique strategies [28]:

- Protoss, characterized by powerful units that demand a higher amount of resources to produce;
- Zerg, characterized by attacking with large amounts of cheap units;
- Terran, which has units of intermediate power and cost.

In *StarCraft* there are two types of resources to produce units and buildings: mineral and gas. To win a game, a player must destroy all the buildings of his opponent. Figure 1 presents a game screenshot.

B. *Running Swarm-GAP in StarCraft*

In order to execute Swarm-GAP algorithm in *StarCraft*, we implemented a software-controlled player (bot), called GASW (abbreviation of genetic algorithm Swarm-GAP) through BWAPI. In this bot, task allocation among agents is performed according to Alg. 1.

GASW bot plays with the Terran race, using 7 out of 17 available Terran buildings and 3 out of 13 Terran units. Although via Swarm-GAP we obtain which tasks will be performed, we need to hand-code how these tasks will be performed in the game. This hand-coded behavior can be complex depending on the unit type. Thus, due to the

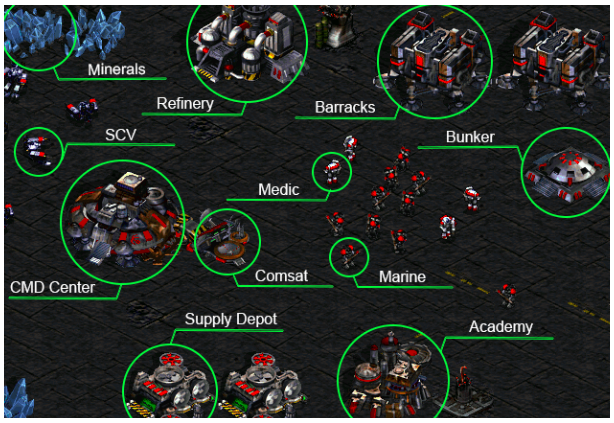


Figure 1: *StarCraft* screenshot of a Terran base. Each unit and structure has a specific function in the game.

complexity of implementing the task execution behavior of every unit inside the game, we model only a subset containing the basic buildings and units available to GASW bot.

Terran race was chosen because it is the only race whose basic combat unit can target ground and air enemies, which would increase the chances of victory.

All units and buildings used by Swarm-GAP are shown in Fig. 1. Their function is described in the list that follows.

- Buildings:
  - Command center: receives gathered resources and produces workers (SCV).
  - Comsat: allows periodic scans in the map, useful for scouting.
  - Supply depot: are required to increase the number of units that can be created.
  - Barracks: produces marines and medics.
  - Academy: is required for the production of medics and allows the research of combat upgrades for marines.
  - Refinery: is required to collect gas, which is needed to train medics and to perform upgrades at the Academy.
  - Bunker: defensive building that provides shelter for up to 4 ground units. Allows marines to attack enemies with increased range.
- Units:
  - SCV: worker unit that gathers resources, constructs and repairs buildings.
  - Marine: ranged combat unit. Can target air and ground enemies.
  - Medic: auxiliary combat unit that heals other units.

In GASW bot, three types of agents allocate tasks according to Alg. 1: SCV, marine and a commander agent. SCV and marine have counterparts in the game. Commander is an abstract agent, responsible for deciding when SCV, marine and medic units should be produced. The behavior of the medic unit is hand-coded as it is an auxiliary unit created to keep other units alive for longer periods.

Table I presents the tasks that agents must allocate via Swarm-GAP. Cells are marked where an agent can perform the given task.

Table I: Agent-task compatibility for tasks allocated via Swarm-GAP

Task	**SCV**	**Marine**	**Commander**
Gather minerals	√		
Build barracks	√		
Build supply depot	√		
Build academy	√		
Build refinery	√		
Build command center	√		
Repair building	√		
Explore map	√	√	
Attack	√	√	
Train SCV			√
Train medic			√
Train marine			√

For the execution of Swarm-GAP algorithm, stimulus must be modeled for all tasks and agent capabilities must be modeled for every compatible agent-task combination. Some game-related tasks are not allocated via Swarm-GAP thus they do not appear in Table I. These tasks include: build Comsat, Bunker and gather gas. Allocation and performance of these tasks are hand-coded in the following way: the Comsat is built after 15 minutes of game, one Bunker is built near each Command Center and three SCVs are allocated for gas collection in the Refinery. This configuration was set after preliminary experiments to allow basic scouting (with Comsat), basic defenses (with Bunker) and a suitable gas collection rate (with three SCVs at refinery).

In *StarCraft*, agents can perform only one task at a time. This eliminates the need to model agent resources and task costs, represented by r and c in Swarm-GAP (Alg. 1). These are needed to address the situation where agents can perform multiple tasks simultaneously.

Also, in *StarCraft*, units are not limited with local information. Swarm-GAP is a decentralized algorithm that works with agents limited by local information by using the token-based protocol for communication. We exploit the common knowledge of game units by making a global “pool” of tasks perceived by all units instead of using communication among them.

In the E-GAP model, in which Swarm-GAP is based, a task can be performed by only one agent. However, in *StarCraft*, several tasks, such as attacks, must be executed by multiple agents. In E-GAP, such tasks are decomposed into several smaller, inter-related tasks. However, this requires prior knowledge on how to decompose a task. We address this issue as follows: to allow an agent to engage in a previously allocated task, we instantiate a new instance of that task. This approach maintains conformity with the E-GAP model and does not require prior knowledge on task decomposition.

The behavior of GASW bot is controlled by 27 parameters: one stimulus parameter for each task in Tab. I, in a total of 12, one capability parameter for each compatible agent-task combination, in a total of 14 (9 for the SCV, 2 for the marine and 3 for the commander), and one parameter that controls the size of an attacking group of marines. The last parameter is not related to execution of Swarm-GAP algorithm, but controls an important aspect of GASW bot.

In *StarCraft*, we can associate stimulus with the importance and/or urgency of a task. Moreover, we can associate capability (k) with the suitability of an agent to perform a task. For example, a worker unit is expected to prefer resource gathering than combats, although it could engage in combats if they're very important (e.g. to defend a base under attack). In this case, this combat task should have a high stimulus so as to goad the worker unit to perform it. A sensible adjustment of stimulus and capability parameters has great impact on how task allocation reflects the actual importance of tasks and skills of units. This motivated us to adopt the genetic algorithm for parameter adjustment, as further discussed in Section V.

V. Optimizing Swarm-GAP parameters

A. Genetic algorithm

Finding a good combination of Swarm-GAP parameters may be impractical for large scenarios if done manually. To address this issue, we employ a genetic algorithm to automatically find a combination of parameters that maximizes a global metric of agent performance.

Briefly, a genetic algorithm (GA) is a metaheuristic that mimics the process of natural selection. An initial population is generated, the fitness of its individuals is evaluated, individuals are selected to produce the population of the next generation, genetic operators are then applied, and the process is repeated until a stop criteria is satisfied. The stop criteria is usually related to the solution convergence or number of generations. Genetic algorithms are usually applied to complex optimization problems [29].

In our approach, an individual is given by a chromosome represented by an array of 27 parameters that control the behavior of GASW bot (see Section IV-B). The domain of the 26 parameters related to Swarm-GAP algorithm (stimulus and capabilities for agent-task combinations) is the set of real values from 0 to 1 spaced by 0.05: $\{0, 0.05, 0.10, ..., 0.90, 0.95, 1.0\}$. The set $[0, 1]$ was discretized in this way in order to reduce the search space of the genetic algorithm without significant loss in precision for the control of Swarm-GAP algorithm. The domain of the last parameter of GASW bot (size of attacking marine group) is the set $\{6, 8, 10, ..., 20, 22, 24\}$, i.e., the set of even integers between 6 and 24, inclusive. Odd integers are not considered for search-space reduction as well.

B. Fitness function

Evaluation of an individual in our approach is based on the performance of GASW bot in matches against *StarCraft*'s native AI, or SC Bot for short. Our bot implements Swarm-GAP, loading the parameters contained in the chromosome of the individual to be evaluated in order to perform allocation of the game-related tasks presented in Table I.

We "train" GASW bot by executing the genetic algorithm against SC Bot controlling one of the three races and test the best individual found against SC Bot controlling the same race. We evaluate four fitness functions for the genetic algorithm:

1) *Victory Rate*: Performance improvement of a bot can be related to the number of wins it achieves against an opponent (or a set of opponents). Thus, a fitness function related to the rate of victories in a number of matches is a reasonable choice. In fact, *Fernandez-Ares et. al.* [30] use this function to evaluate its bot in Planet Wars game. The main drawback of this function is the time required to run the genetic algorithm: each individual has to play a number of matches to have its victory rate calculated.
2) *Score ratio:* At the end of a *StarCraft* match, each player receives a score related to its overall game performance. The overall score aggregates components regarding resource collection, structure construction and unit management (army deployment and attacks to enemy forces). Score ratio is then calculated by GASW's score divided by its opponent's score, as in [5].
   The rationale behind this fitness function is that the winner achieves greater overall score than the loser of a match, because it outperforms the loser in combat and/or in economic aspects.
3) *Unit-based fitness*: This fitness function is given by dividing our bot's unit management score by its opponent's. It is similar to the overall score ratio fitness but it uses only the score component regarding unit management. This score component is based on the number and power of units a player has created for itself and destructed from the enemy. A player who

creates a powerful unit adds greatly to his score. The same happens when it destroys a powerful unit owned by the opponent.
The rationale behind this fitness function is that the goal in an RTS game is to eliminate enemy forces and this combat-oriented fitness function reflects directly a player's combat performance. The other score components (resources and buildings) also affect game performance but one cannot win a match without good combat performance.

4) *Time-based fitness*: The time-based fitness is given by Eq. 4, where d is the match duration, d_{max} is the maximum duration of a match and $F(d) : \mathbb{R} \to [0, 1]$ is the fitness function.

$$F(d) = \begin{cases} 1 - \frac{d}{2 * d_{max}}, & \text{in case of victory} \\ \frac{d}{2 * d_{max}}, & \text{in case of defeat} \end{cases} \tag{4}$$

This function is used in [31]. If the bot wins the match, its fitness will range from 0.5 to 1, and it receives higher fitness for faster victories. On the other hand, if the bot loses the match, its fitness will range from 0 to 0.5, and it receives higher fitness for longer matches. That is, the bot is rewarded for rapid victories and for lasting longer in defeats.

After training, we assess performance as the rate of victories in a number of matches, since the goal in RTS games is to win as many matches as possible. We remark that the fitness function that reflects this goal more closely is victory rate. It is used as the baseline for comparison with the other functions (see Section VI-A).

C. Accelerating the genetic algorithm

As fitness evaluation is a time-consuming task, we accelerate the genetic algorithm by estimating fitness of some individuals instead of actually evaluating them [7]. Fitness estimation of an individual considers similarity with its parents and their reliability. Similarity between two individuals i and j is denoted by $\rho_{ij} \in [0, 1]$ and reliability of individual i is denoted by $w_i \in [0, 1]$.

Let individual a be the parent of c. Similarity between them, is calculated via Eq. 5, where A is the chromosome of a, C is the chromosome of c and max_i and min_i are the maximum and minimum value in the domain of the variable in locus i of a chromosome, respectively.

$$\rho_{ac} = 1 - \frac{1}{|A|} \sum_{i=1}^{|A|} \frac{abs(A[i] - C[i])}{max_i - min_i} \tag{5}$$

In Eq. 5, each summand represents the normalized difference (i.e. between 0 and 1) of values in locus i of the chromosomes. The average of these normalized differences gives us a degree of divergence between the two chromosomes. Similarity is the complement of the degree of divergence.

Reliability of an individual is calculated according to the reliability of its parents and the similarity between parents and child. Equation 6 formalizes this, where a and b are the parents of c. This equation ensures that the reliability of child c is closer to that of the most similar and reliable parent.

$$w_c = \frac{(w_a \rho_{ac})^2 + (w_b \rho_{bc})^2}{w_a \rho_{ac} + w_b \rho_{bc}} \tag{6}$$

Finally, fitness of an individual c (f_c) is estimated as the average of its parents' fitness weighted by their reliability times similarity with the child. This calculation is shown in Eq. 7, where a and b are the parents of c.

$$f_c = \frac{f_a w_a \rho_{ac} + f_b w_b \rho_{bc}}{w_a \rho_{ac} + w_b \rho_{bc}} \tag{7}$$

If the estimated fitness of individual c falls below a given threshold ($\tau \in [0, 1]$), the actual fitness is evaluated and assigned to f_c. In this case, its reliability w_c is set to 1. Note that fitness of a child estimated via Eq. 7 lies between its parents' fitness. Moreover, reliability of a child calculated via Eq. 6 is lower than the highest reliability of its parents. This is desirable, because with successive estimations, reliability should drop below the threshold and an actual evaluation must take place.

A value of reliability threshold (τ) close to 1 results in many actual fitness evaluations, slowing down the genetic algorithm. On the other hand, a value of τ close to 0 results in many successive fitness estimations which may yield fitness values that differ too much from the actual fitness of the individuals. Thus, τ is an important parameter of this fitness estimation method and should be carefully adjusted.

In order to ensure that some individuals are evaluated instead of estimated in every generation, a probability of evaluation ($p_e \in [0, 1]$) is employed. Individuals with reliability above the threshold are evaluated with probability p_e. This parameter can be adjusted to force evaluation of a desired portion of individuals. The closer p_e is to 1, the slower is the execution of the genetic algorithm. On the other hand, we have more actual fitness evaluations, increasing overall reliability of the population.

For a complete description of the fitness estimation method adopted in this paper, the reader may refer to Salami and Hendtlass [7].

Algorithm 2 formalizes our approach. In this algorithm, $P^{(n)}$ is the population in generation n, the maximum number of generations is η and the population size is κ. Method $select\_parents$ selects two individuals from the population. Method $crossover\_and\_mutation$ receives two individuals, performs crossover according to a crossover probability, mutates the individuals according to the mutation probability and returns the two individuals. Note that selection, crossover and mutation methods are not specified and can be chosen according to the situation. For example, in this paper we use tournament selection and one-point crossover (see Section VI).

Algorithm 2 Genetic algorithm for Swarm-GAP

```
 1: $P^{(0)} \leftarrow$ {initial random population}
 2: for all $p \in P^{(0)}$ do
 3:     $f_p \leftarrow Evaluate(p)$
 4:     $w_p \leftarrow 1$
 5: end for
 6: for $n \in [0..\eta]$ do
 7:     $P^{(n+1)} \leftarrow \emptyset$
 8:     while $|P^{(n+1)}| < \kappa$ do
 9:         $(a, b) \leftarrow select\_parents(P^{(n)})$
10:         $(c, d) = crossover\_and\_mutation(a, b)$
11:         $f_c \leftarrow$ fitness estimated via Eq. 7
12:         $w_c \leftarrow$ reliability calculated via Eq. 6
13:         if $w_c < \tau$ or ($w_c \geq \tau$ and $random() < p_e$) then
14:             $f_c \leftarrow Evaluate(c)$
15:             $w_c \leftarrow 1$
16:         end if
17:         /* repeat lines 11-16 for individual $d$ */
18:         $P^{(n+1)} \leftarrow P^{(n+1)} \cup \{c, d\}$
19:     end while
20: end for
```

D. Evolving Swarm-GAP in StarCraft

This section describes our architecture used to implement the genetic algorithm that adjusts the parameters of task allocation used by GASW bot via Swarm-GAP algorithm.

GASW bot communicates with *StarCraft* via BWAPI calls. However, during the execution of the genetic algorithm, GASW bot is responsible only for the evaluation of an individual, i.e., subroutine *Evaluate* in Alg. 2. The remainder of Alg. 2 is implemented in the genetic algorithm controller module (GA controller). This module places files with data of the individuals to be evaluated in a specific directory that GASW bot reads. After evaluation of an individual, which corresponds to a *StarCraft* match, GASW bot writes the match outcomes in the same directory, where the external module extracts score information and calculates fitness of the evaluated individual.

Figure 2 illustrates the implemented architecture. At experiment beginning, the GA controller loads a configuration file with genetic algorithm parameters (crossover and mutation probabilities, reliability threshold, etc.).

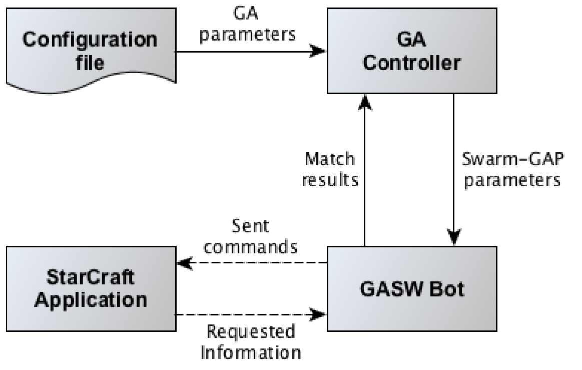


Figure 2: Architecture of the proposed approach. Solid lines represent data exchanged via files. Dashed lines represent data exchanged via BWAPI calls.

VI. Experiments and discussion

In this work we perform three sets of experiments. First, in Section VI-A, we test the four fitness functions defined in Section V-B. Second, in Section VI-B, we test the fitness estimation method of Section V-C in terms of solution quality and execution time. Finally, in Section VI-C, we assess genetic algorithm convergence and performance, comparing our approach with random task allocation, Swarm-GAP with parameters adjusted by hand and *StarCraft* AI tournament bots.

Matches of all experiments, except the reenacted AI tournament of Section VI-C, were played against *StarCraft*'s native AI, or SC Bot for short. SC Bot is able to play with the three races, using different strategies and army compositions. SC Bot is competitive against beginner human players.

Unless otherwise stated, all matches are played in the same map, which is a two-player map with no islands[3]. Since GASW bot has no flying units, it would be unable to defeat an opponent who uses isolated regions to build structures. Also, a match is interrupted when it reaches one hour of in-game time. In this case, GASW bot is considered a loser for the victory rate fitness. The other fitness functions are calculated normally.

Our genetic algorithm was executed with selection by tournament with 2 participants and one-point crossover, where a crossover point on both parents' chromosomes is randomly

[3]The used map is Astral Balance, edited to make all regions reachable by land.

selected. All data beyond that point is swapped between parents to produce the children. Also, we employ elitism, adding the best individual from a generation to the next one. Mutation occurs by randomly setting the value in a locus. Prior experiments were performed in order to determine the genetic algorithm parameters, which are:

- Crossover probability: 0.9;
- Mutation probability: 0.01 per locus;
- Number of generations: 50;
- Population size: 30 individuals.

Unless otherwise stated, all results are averaged over 30 repetitions.

A. *Fitness functions*

In this section we test the four fitness functions defined in Section V-B. After genetic algorithm execution, the best individual plays 150 matches against SC Bot. The goal of this test is to assess the actual effectiveness of the different fitness functions regarding the ultimate goal in RTS games, i.e., to win matches.

Victory rate fitness is calculated as the rate of victories in 5 matches. In this case, we reduce population to 15 individuals to prevent the genetic algorithm from taking too long to execute due to the multiple matches required to evaluate each individual.

Results are shown in Fig. 3, averaged over 15 repetitions. Each repetition consists of genetic algorithm execution and test of the best individual in 150 matches against SC bot. Error bars show the 95% confidence intervals. The best individuals are tested against the same race they “trained” during genetic algorithm execution. The different fitness functions are grouped by adversary race because in this way we can assess the distinct functions under the same conditions. The baseline for comparison is victory rate, because it directly mirrors the metric used in the test (rate of victories in a number of matches).

There is no statistical difference among the distinct fitness functions from a same group as Fig. 3 visually suggests and one-way ANOVA and Tukey’s HSD confirm. That is, under the same conditions, all fitness functions are statistically similar. The only statistical difference was found between score ratio and victory rate against Protoss. This result suggests that it is advantageous, in terms of execution time, to employ fitness functions other than victory rate, as it performs much more evaluations than other functions without gains in solution quality.

However, a major factor limit a stronger claim about the performance similarity of the distinct fitness functions. GASW bot is limited in the units and buildings it uses, thus, its

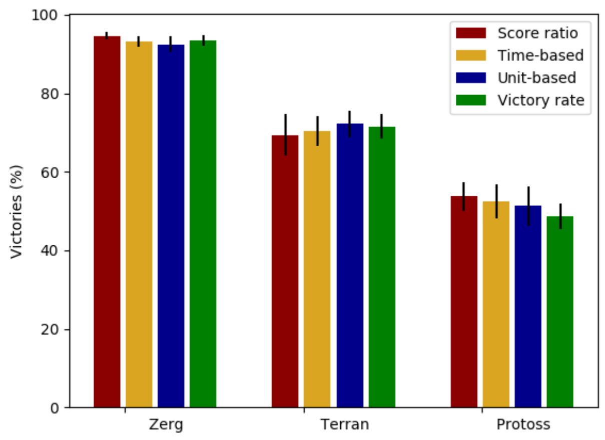


Figure 3: Rate of victories (%) in 150 matches against Protoss, Terran and Zerg adversaries controlled by SC Bot. Error bars are the 95% confidence intervals in 15 repetitions.

“near-optimal” parameter configuration might not be difficult to find regardless of the fitness function that directs the evolution. Here, near-optimal means the best that GASW bot can perform, given its limitations and not the best possible performance of a software-controlled player.

For the remaining experiments in this paper, we adopt the score ratio fitness, because it is statistically similar to the baseline (victory rate) besides running faster. Moreover, compared to the remaining functions, score ratio measures player performance compared to its opponent, combining the major aspects of gameplay (resources, units and building management), instead of considering them separately.

B. *Fitness estimation*

In this section we analyse the fitness estimation method described in Section V-C. Tests are made to assess the quality of the estimation method in the presence of noise. In this work, noise in the fitness function comes in two ways. First, it depends on the probabilistic way that agents allocate tasks in Swarm-GAP. Second, *StarCraft* is an adversarial game. Thus, match results and the calculated fitness also depend on the adversary actions, which may be randomized in order to become more difficult to predict.

Noise can be harmful for the performance of genetic algorithm with fitness estimation. For instance, in [5], fitness achieved during evolution was superior with fitness estimation but performance in terms of victory rate was inferior. This happened because an individual who represents a bad configuration of parameters can eventually win a match and receive high fitness (which is measured as the score ratio defined in Section V-B). With fitness estimation, this individual may not be evaluated again, thus propagating across generations. This may direct the search of the genetic algorithm to the neighborhood of that individual, which would be less promising than other regions of the parameter space. In [5], when an individual has reliability above the threshold τ, it is forcedly evaluated with probability $p_e = 0.1$. In this section we analyze the performance obtained with different values of p_e. A greater p_e results in more individuals evaluated instead of estimated. This may increase the genetic algorithm solution quality at the expense of time to evaluate more individuals.

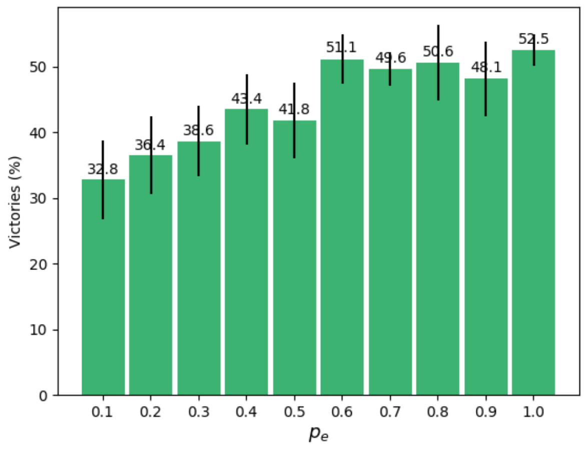
Experiments were performed with score ratio fitness (see Section V-B) and reliability threshold $\tau = 0.5$ (in Alg. 2). Performance is the rate of victories of the best individual found by the genetic algorithm in 150 matches against SC Bot controlling the same race used during evolution. Execution time is measured in hours as the sum of the time spent to evaluate individuals across all generations. Results are averaged over 30 repetitions. Error bars in plots are the 95% confidence intervals. The baselines are the values obtained with $p_e = 1.0$ (where all individuals are always evaluated). To evaluate statistical differences we use one-way ANOVA and Tukey's HSD.

Figure 4 shows performance and execution time of the genetic algorithm against the Protoss adversary controlled by SC Bot. In this case, performance with $p_e \in \{0.1, 0.2, 0.3\}$ is significantly inferior to the baseline, with p-values of $\{0.0000004, 0.0001239, 0.0032373\}$ respectively. For $p_e \geq 0.4$, difference in performance compared to baseline is not statistically significant (p-values are all above 0.15). Execution time, as expected, grows in proportion to p_e. All times were significantly different from the baseline (p-value near 0.00003).

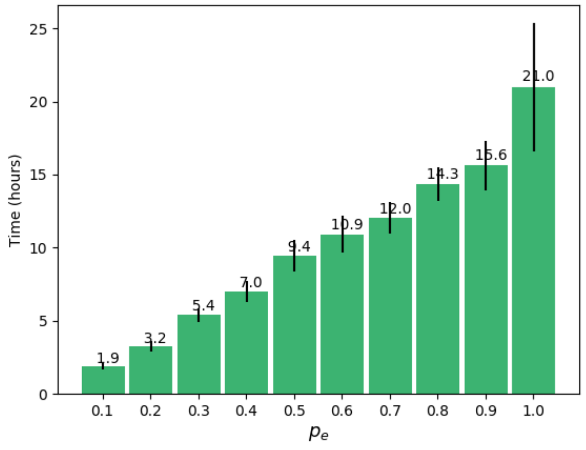
Figure 5 shows performance and execution time against the Terran adversary controlled by SC Bot. Performance with $p_e \in \{0.1, 0.2\}$ is significantly inferior to the baseline, with p-values of $\{0.0000821, 0.0003312\}$ respectively. For $p_e \geq 0.3$, the difference in performance compared to the baseline is not statistically significant (all p-values are above 0.10). As with Protoss, execution time grows in proportion to p_e.

Against a Zerg adversary (Fig. 6), performance of individuals found with all values of p_e, except 0.2, is statistically similar to the baseline, as p-values of Tukey's HSD are above 0.05. The victory rate is close to 100% in all experiments. This may happen because the emerging strategy of GASW bot is effective against the Zerg adversary controlled by SC Bot (see Section VI-D). Against this adversary, even poor parameter configurations might be able to win matches. Execution time also grows in proportion to p_e.

These results are aligned with those in [5], where fitness estimation was executed with $p_e = 0.1$. Here we show that, for Protoss and Terran adversaries, performance obtained with this value is significantly inferior to that of the genetic algorithm without fitness estimation.



(a) Victory rate.



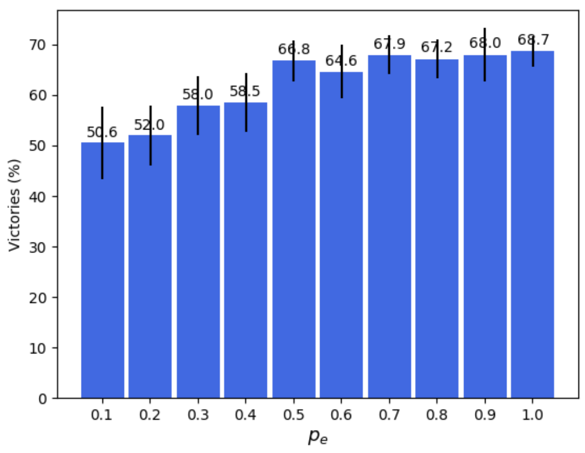
(b) Time spent.

Figure 4: Performance and time spent of GA with fitness estimation against the Protoss adversary.

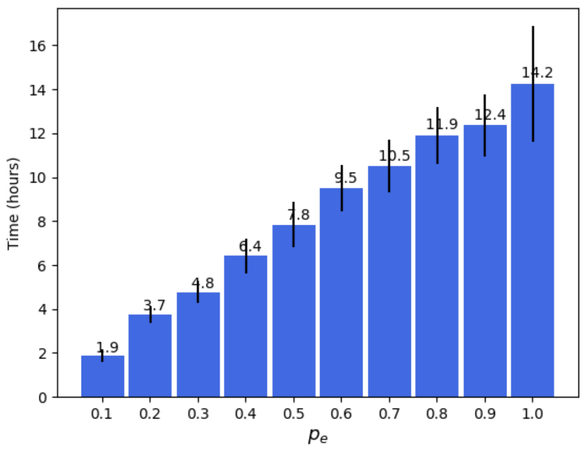
C. Genetic algorithm convergence and performance

1) Convergence: In this section, we analyze the convergence of the genetic algorithm by assessing the mean fitness of the population along generations. Experiments were performed with score ratio fitness (see Section V-B) and all individuals evaluated ($p_e = 1.0$ in Alg. 2).

Figure 7 shows fitness along the generations for GASW bot evolved by the genetic algorithm. The distance between lines shows that GASW bot performs better against Zerg race controlled by SC bot. Conversely, GASW bot performs

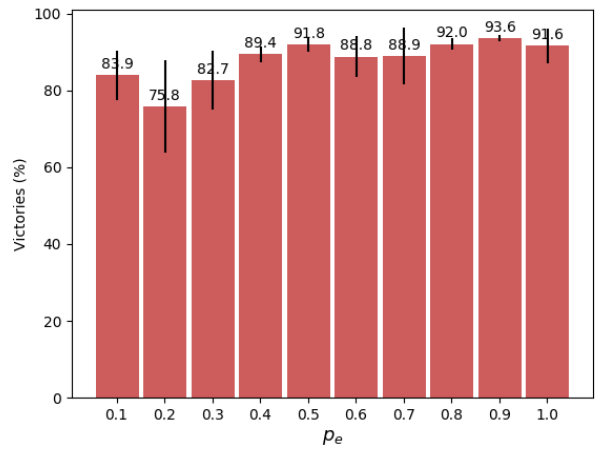


(a) Victory rate.

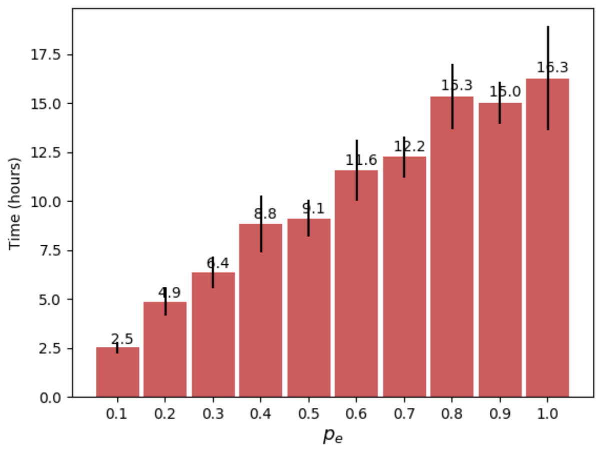


(b) Time spent.

Figure 5: Performance and time spent of GA with fitness estimation against the Terran adversary.



(a) Victory rate.



(b) Time spent.

Figure 6: Performance and time spent of GA with fitness estimation against the Zerg adversary.

worse against Protoss. A Terran adversary controlled by SC bot lies between them. In all cases, fitness stabilizes above 1.0 which indicates that individuals are winning the matches on average.

The results obtained in this experiment indicate that the genetic algorithm behaves as expected, that is, mean fitness increases along generations and stabilizes in a desirable performance. However, these results do not show how GASW bot performs compared to other approaches. This is done in Section VI-C2.

2) Compared performance: In this section we assess the performance of GASW bot compared to other approaches. We begin by testing GASW and other approaches against SC Bot and proceed by testing GASW in direct matches against *StarCraft* AI tournament bots.

In matches against SC Bot, the performance of the best individual found by GASW is compared to that of the bots described next.

- Random: in this bot, task allocation is made randomly. For each agent, given a list of tasks, one is chosen with uniform probability. This way, tasks are given equal importance during the allocation process.
- ManSW: this bot allocates tasks via Swarm-GAP, similarly to GASW. The difference is that the parameters of ManSW were configured by hand whereas GASW runs with parameters configured via genetic algorithm. We configured the parameters of ManSW bot according to empirical *StarCraft* knowledge and a few tests against

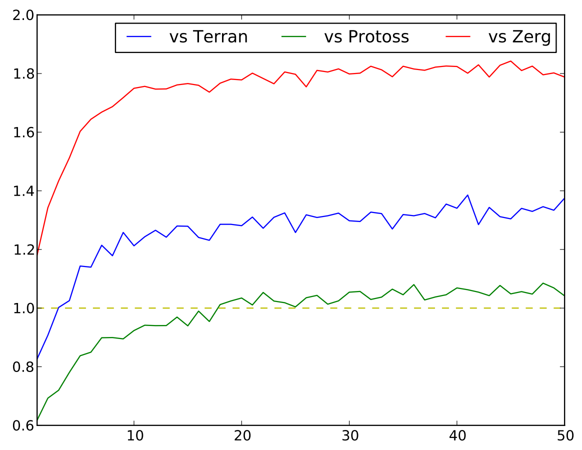


Figure 7: Genetic algorithm performance. When a match is won, fitness goes above 1.0 (baseline).

SC bot. The number of matches performed to adjust ManSW parameters is very small compared to that of the genetic algorithm for GASW.

- AIUR: this is a competitive bot that placed 3rd among 8 bots in AIIDE 2013 competition, 4th among 13 bots in IEEE CIG 2014 and 5th among 22 bots in AIIDE 2015[4]. Among all competitors, AIUR was the bot with best performance that we were able to obtain the source code, compile and successfully execute against SC Bot[5]. Briefly, in AIUR, several game-related tasks are divided among many different modules. At the beginning of a game, the bot initializes a "mood" that influences the adopted strategy (focus on resource collection, early attacks or defense). This bot does not perform reactive controls (micro-management) [15]. AIUR plays with Protoss race.

Random and ManSW bots have the same limitations of GASW: they play with Terran race using the same units and buildings. Moreover, task execution is the same in these bots. Difference in performance between Random and Swarm-GAP-based bots (ManSW and GASW) are due to the task allocation method. Difference in performance between ManSW and GASW are due to the quality of parameter adjustment.

Figure 8 shows the victory rate in 150 matches of all bots versus Protoss, Terran and Zerg adversaries controlled by SC Bot. Results are averaged over 30 repetitions. Error bars show the 95% confidence intervals. Statistical significance is evaluated with one-way ANOVA and Tukey's HSD. Results are grouped by race, so that we discuss approaches' performances under the same conditions.

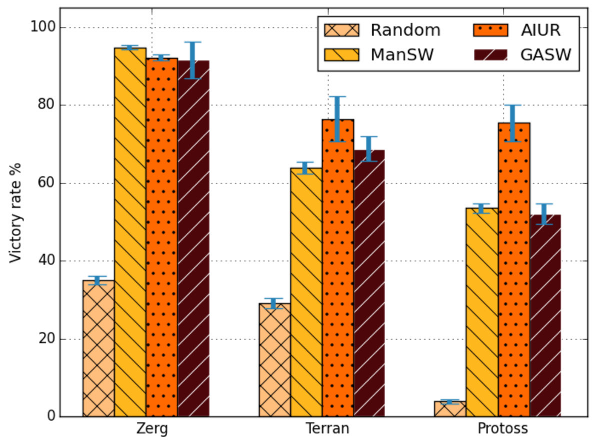


Figure 8: Victory rates (%) of approaches in 150 matches against Protoss, Terran and Zerg races controlled by SC bot.

Terran-based bots (Random, ManSW and GASW) had their best performance against a Zerg adversary controlled by SC Bot. Worst performance is observed against Protoss. Performance against a Terran adversary is between them.

Random had the worst performance among all bots. This is due to the poor task allocation that emerges from the uniformly random selection of tasks. The Swarm-GAP-based bots (ManSW and GASW) achieved significantly superior performance than Random. This shows that it pays off to employ Swarm-GAP to solve the task allocation problem posed by an RTS game.

Against the Terran adversary, the superiority of GASW's performance compared to ManSW is statistically significant (Tukey's p-value of 0.049). By looking at Protoss and Zerg adversaries, ManSW and GASW had statistically similar performance by Tukey's HSD test. P-values in these cases are 0.88 against the Protoss adversary and 0.10 against the Zerg adversary.

Compared to AIUR, the performance of GASW is statistically similar against the Zerg adversary but it is significantly inferior against Terran and Protoss adversaries (p-values < 0.00005).

In [5], performance of AIUR and GASW were similar. Here we use the 2014 version of AIUR, which was improved[6]. The limitation in terms of the units and buildings used by Swarm-GAP-based bots may explain both their inferiority

[4]See the *StarCraft* AI Competition archive: https://www.cs.mun.ca/~dchurchill/starcraftaicomp/archive.shtml

[5]Some adjustments in source code were needed to record match results in a suitable format for analysis.

[6]Improvements are listed on http://aiur-group.github.io/AIUR/#whats-new-since-aiidecig-2013-aiur-21

compared to AIUR and the similarity between themselves against Protoss and Zerg adversaries controlled by SC Bot.

The adoption of new units and buildings can improve performance by allowing the adoption of new strategies (see Section VI-D). This may help closing the gap between AIUR and the Swarm-GAP-based bots, especially against the Protoss adversary. Furthermore, regarding the similarity between Swarm-GAP-based bots, the search space would increase with the addition of parameters to control task allocation for other game units. This would increase the difficulty to find good parameter values by hand.

To evaluate GASW performance in direct matches against tournament bots, we reenact the AIIDE 2015 *StarCraft* AI tournament, which include the 2014 version of AIUR, in a one-vs-all mode. Since GASW bot was "trained" using only a single small map (an edited version of Astral Balance), this evaluation is made in a tournament map with similar characteristics: Benzene, a two-player map with no islands. Table II shows the reenacted tournament results. GASW dominates, winning more than 50% matches, about a fifth of AIIDE 2015 bots. Those bots were also the weakest overall in that tournament, as their rank shows. GASW is not competitive against stronger bots. Against AIUR, for example, it has won only 3.33% matches.

Table II: Performance (Win %) of GASW in 30 matches against AIIDE 2015 tournament bots. Their rank in that tournament is included.

Bot	Race	Win %	AIIDE 2015 Rank[a]
SusanooTricks	Protoss	96.67	22
Stone	Terran	93.33	19
Yarmouk	Terran	73.33	21
LetaBot	Terran	60.00	10
Oritaka	Terran	43.33	18
CruzBot	Protoss	23.33	16
TerranUAB	Terran	20.00	14
Cimex	Zerg	13.33	15
OpprimoBot	Terran	6.67	17
Bonjwa	Terran	3.33	20
Tyr	Terran	3.33	11
Aiur	Protoss	3.33	5
NUSBot	Protoss	0.00	13
GarmBot	Zerg	0.00	12
Xelnaga	Protoss	0.00	9
Skynet	Protoss	0.00	8
IceBot	Terran	0.00	7
Ximp	Protoss	0.00	6
UAlbertaBot	Random[b]	0.00	4
Overkill	Zerg	0.00	3
ZZZKBot	Zerg	0.00	2

[a] The tournament's 1st place, Tscmoo, is not shown because it could not be executed.
[b] In this case a race is randomly selected for the bot before each match.

The weak overall performance of GASW against stronger bots might be explained, in part, because GASW is trained against SC Bot, thus optimizing the task allocation parameters against this simpler adversary. Moreover, in general, strong tournament bots employ terrain analysis, opening libraries, scripted strategies and learning mechanisms [15]. Thus they have the flexibility to construct units and adopt behaviors that defeat the simple strategy of GASW, which is determined by the limited number of units and buildings it uses. In other words, the reenacted tournament also indicates that the performance of GASW could be better if it used more units and buildings, allowing the adoption of new strategies. This is further discussed in Section VI-D.

D. General discussion

Experiments in this paper suggest that employing Swarm-GAP for task allocation is a promising approach to tackle the challenge posed by a complex RTS game. Moreover, employing genetic algorithm for parameter adjustment pays off: it dismisses domain knowledge and the achieved performance can be superior to manually adjusted parameters.

Regarding genetic algorithm acceleration, fitness estimation is helpful, but, on the presence of noise, it may reduce solution quality. The parameter that controls the probability of forced evaluation must be adjusted to achieve time savings without loss of performance. Results found in this paper can be used as a guideline for future experiments.

Our results are influenced by the emergent strategy of the Swarm-GAP-based bots as they use only marines as combat units and medics as supporting units. This allows the emergence of a single strategy known in competitive play as "M&M"[7], which consists of attacking with a large amount of marines supported by medics. Both are cheap and less powerful Terran units, and the idea is to compensate their lack of power with their quantity.

M&M strategy is a good match against SC Bot playing as Zerg, because it also attacks with many units of reduced power. The supporting medics are able to properly heal marines in combat against such units. On the other hand, M&M is not a good strategy against SC Bot playing with Protoss, as it attacks with less units of greater power. These deal more damage against the marines, in a rate that the supporting medics are not able to keep up. Against SC Bot playing as Terran, M&M strategy performs well. Although Terran and Zerg have effective units against the Marine, SC Bot does not use them in most matches in the tested map. Against AIIDE 2015 *StarCraft* AI Tournament bots, M&M strategy proved useful only against weaker competitors, which are not able to properly come up with counter-strategies.

The adoption of more combat units would allow the emergence of strategies other than M&M, which can be more

[7]M&M stands for marines and medics: http://starcraft.wikia.com/wiki/Marines_and_medics

competitive, especially against the Protoss adversary controlled by SC Bot and the tournament bots. Moreover, fewer units and buildings to control result in fewer parameters to control task allocation. This implies a reduced parameter space so that a suitable combination of parameters may not be difficult to find. This explains the similar performance of the Swarm-GAP-based bots (with manual and genetic algorithm adjustment of parameters) against the Protoss and Zerg adversaries in Section VI-C2. The adoption of more units and buildings for the Swarm-GAP-based bots will result in more parameters to control task allocation. In this scenario of increased complexity, the advantages of employing the genetic algorithm may become more evident.

VII. Conclusion

A. Overview

In this work we tackle the problem posed by an RTS game with a task allocation approach. We employ a genetic algorithm to adjust the parameters of Swarm-GAP, a scalable task allocation algorithm. Our approach is tested in the popular RTS game *StarCraft: Brood War*. As the execution of the genetic algorithm takes considerable time, we employ the fitness estimation method of [7] to accelerate it. However, estimation may be harmful because the fitness function depends on match results, which are non-deterministic. Thus, we test several values for the parameter that forces evaluation of individuals instead of using estimated fitness. Our results can be used as guidelines to find the biggest time savings without loss of performance.

In experiments against *StarCraft*'s native AI (SC Bot), performance of our approach varies according to the race SC Bot uses. The emerging strategy of our approach is stronger against Zerg, has intermediate strength against Terran and has worst performance against Protoss. Configuring Swarm-GAP parameters via genetic algorithm achieves significantly superior performance against the Terran adversary than manually-configured Swarm-GAP. Both manual and genetic algorithm parameter configuration outperform Random task allocation, but both are inferior compared to the 2014 version of AIUR, one of the top-performing bots for *StarCraft* known to date.

Both the genetic algorithm and the *StarCraft* bots codebase is available[8].

B. Future work

There is room for improvement of the Swarm-GAP-based bots against the Protoss adversary controlled by SC Bot. In future studies of the proposed approach, more units and buildings could be incorporated into the Swarm-GAP-based bots. The adoption of more combat units will allow the emergence of new strategies and army compositions, which can be more competitive. Also, this will increase task allocation parameter space, which may reinforce the advantages of adopting the genetic algorithm for parameter adjustment. Future work could also adopt a stronger Zerg opponent than SC Bot.

Future studies can investigate other fitness estimation methods and techniques to deal with fitness noise, such as the ones surveyed in [32], for example.

To achieve tournament-winning performance, new aspects should be introduced into our architecture. This could include a library of game openings, hierarchical decision making, unit micro-management (reactive control) and terrain analysis [15]. Moreover, using different sets of parameters according to the game situation might be helpful. Usually, a match is divided in stages (e.g. early, mid and late game) and each stage might require a different parameter combination. Currently, we adopt a single set of parameters for task allocation during the entire match.

Other interesting topic for future work include the use of different task allocation algorithms. E-GAP-based algorithms, such as Swarm-GAP, assume that the value of a team never decreases if more agents become members [8]. In *StarCraft*, this does not hold in some situations. For example, a worker unit may disturb a team of attacking units by standing on their way. Algorithms based on coalition structure generation, such as BnB FMS [12], can handle this issue naturally, as they model the help or disturb caused by agents in each group they may enter.

Although we instantiated the approach in *StarCraft*, the general idea presented in this paper could be used in any task allocation domain. Modeling parameters for any other domain will require basic knowledge of the tasks associated with that domain. With the tasks at hand, the application of our methodology for task allocation and parameter adjustment is straightforward. At this level, our approach is domain-independent. However, the domain-specific part, regarding how tasks are performed, is challenging: as each domain has specific mechanics, there is no single approach that will be efficient for all domains. In this sense, task execution must be developed and tailored for every specific domain.

Examples of domains beyond RTS games which could benefit from our approach include robot soccer and autonomous coordination in hostile environments such as rescue operations in disaster situations. On those scenarios, task allocation is a vital part of multiagent coordination. The use of scalable task allocation algorithms such as Swarm-GAP and automatic parameter adjustment via genetic algorithms

[8]https://github.com/verlab/terranswarm

might be useful to align parameter values to actual agents skills and importance of tasks.

Acknowledgment

Authors would like to acknowledge the support of CAPES, CNPq and FAPEMIG in this research. We would like to thank Bernardo Biesseck and Daniel Balbino for helping with the experiment infrastructure. We also thank the anonymous reviewers for insightful remarks that helped improving the manuscript.

References

[1] R. L. de Freitas Cunha and L. Chaimowicz, "An Artificial Intelligence System to Help the Player of Real-Time Strategy Games," in *Proceedings of the 2010 Brazilian Symposium on Games and Digital Entertainment (SBGAMES),*, 2010, pp. 71 –81.

[2] M. Buro, "Real-Time Strategy Games: A New AI Research Challenge," in *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence, 2003, pp. 1534–1535.

[3] H. Kitano, "Robocup rescue: A grand challenge for multi-agent systems," in *Proc. of the 4th International Conference on MultiAgent Systems*. Boston, USA: Los Alamitos, IEEE Computer Society, 2000, pp. 5–12.

[4] R. Fierro, A. Das, J. Spletzer, J. Esposito, V. Kumar, J. P. Ostrowski, G. Pappas, C. J. Taylor, Y. Hur, R. Alur, I. Lee, G. Grudic, and B. Southall, "A framework and architecture for multi-robot coordination," *The International Journal of Robotics Research*, vol. 21, no. 10-11, pp. 977–995, 2002.

[5] A. R. Tavares, H. Azpurua, and L. Chaimowicz, "Evolving Swarm Intelligence for Task Allocation in a Real Time Strategy Game," in *Computer Games and Digital Entertainment (SBGAMES), 2014 Brazilian Symposium on*, November 2014, pp. 99–108.

[6] P. R. Ferreira, F. Dos Santos, A. L. C. Bazzan, D. Epstein, and S. J. Waskow, "RoboCup Rescue as multiagent task allocation among teams: experiments with task interdependencies," *Autonomous Agents and Multi-Agent Systems*, vol. 20, no. 3, pp. 421–443, 2010.

[7] M. Salami and T. Hendtlass, "A fast evaluation strategy for evolutionary algorithms," *Applied Soft Computing*, vol. 2, no. 3, pp. 156–173, 2003.

[8] P. Scerri, A. Farinelli, S. Okamoto, and M. Tambe, "Allocating tasks in extreme teams," in *Proc. of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Wooldridge, Eds. New York, USA: ACM Press, 2005, pp. 727–734.

[9] D. B. Shmoys and V. Tardos, "An approximation algorithm for the generalized assignment problem," *Mathematical Programming*, vol. 62, no. 3, pp. 461–474, 1993.

[10] G. Theraulaz, E. Bonabeau, and J. Deneubourg, "Response Threshold Reinforcement and Division of Labour in Insect Societies," in *Royal Society of London Series B - Biological Sciences*, vol. 265, 2 1998, pp. 327–332. [Online]. Available: http://citeseer.nj.nec.com/theraulaz98response.html

[11] M. A. Khan, D. Turgut, and L. Bölöni, "Optimizing coalition formation for tasks with dynamically evolving rewards and nondeterministic action effects," *Autonomous Agents and Multi-Agent Systems*, vol. 22, no. 3, pp. 415–438, May 2010.

[12] K. S. Macarthur, R. Stranders, S. D. Ramchurn, and N. R. Jennings, "A Distributed Anytime Algorithm for Dynamic Task Allocation in Multi-Agent Systems," in *Proc. of the 25th AAAI Conference on Artificial Intelligence*, 2011, pp. 701–706.

[13] A. Fernandez-Ares, A. M. Mora, J. Merelo, P. García-Sánchez, and C. Fernandes, "Optimizing player behavior in a real-time strategy game using evolutionary algorithms," in *Evolutionary Computation (CEC), 2011 IEEE Congress on*. IEEE, 2011, pp. 2017–2024.

[14] R. Lara-Cabrera, C. Cotta, and A. J. Fernández-Leiva, "A review of computational intelligence in RTS games," in *Foundations of Computational Intelligence (FOCI), 2013 IEEE Symposium on*. IEEE, 2013, pp. 114–121.

[15] S. Ontanon, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 5, no. 4, pp. 293–311, Dec 2013.

[16] P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma, "On-line adaptation of game opponent AI with dynamic scripting," *International Journal of Intelligent Games & Simulation*, vol. 3, no. 1, 2004.

[17] J. Ludwig and A. Farley, "Examining Extended Dynamic Scripting in a Tactical Game Framework," in *Artificial Intelligence and Interactive Digital Entertainment*, 2009.

[18] M. Stanescu, N. A. Barriga, and M. Buro, "Hierarchical Adversarial Search Applied to Real-Time Strategy Games." in *10th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 2014, pp. 66–72.

[19] A. Uriarte and S. Ontañón, "Improving Monte Carlo Tree Search Policies in StarCraft via Probabilistic Models Learned from Replay Data," in *12th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 2016, pp. 100–106.

[20] G. Synnaeve and P. Bessière, “A Bayesian model for RTS units control applied to StarCraft,” in *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*. IEEE, 2011, pp. 190–196.

[21] O. Khatib, “Real-time obstacle avoidance for manipulators and mobile robots,” *The international journal of robotics research*, vol. 5, no. 1, pp. 90–98, 1986.

[22] T. W. Sandberg and J. Togelius, “Evolutionary Multi-Agent potential field based AI approach for SSC scenarios in RTS games,” Ph.D. dissertation, Master’s thesis, IT University Copenhagen, 2011.

[23] P. H. Ng, Y. Li, and S. C. Shiu, “Unit formation planning in RTS game by using potential field and fuzzy integral,” in *Fuzzy Systems (FUZZ), 2011 IEEE International Conference on*. IEEE, 2011, pp. 178–184.

[24] C. Lin and C. Ting, “Emergent tactical formation using genetic algorithm in real-time strategy games,” in *Technologies and Applications of Artificial Intelligence (TAAI), 2011 International Conference on*, Nov 2011, pp. 325–330.

[25] E. A. Rathe and J. B. Svendsen, “Micromanagement in Starcraft using potential fields tuned with a multi-objective genetic algorithm,” Ph.D. dissertation, Norwegian University of Science and Technology, 2012.

[26] B. G. Weber, M. Mateas, and A. Jhala, “Building human-level AI for real-time strategy games,” in *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems*, 2011, pp. 329–336.

[27] BWAPI, “An API for interacting with StarCraft: Broodwar,” 2011. [Online]. Available: http://bwapi.github.io/

[28] Liquipedia, “Portal:Beginners - Liquipedia StarCraft Brood War Wiki,” 04 2012. [Online]. Available: http://wiki.teamliquid.net/starcraft/Portal:Beginners

[29] R. L. Haupt and S. E. Haupt, *Practical genetic algorithms*. John Wiley & Sons, 2004.

[30] A. Fernández-Ares, P. Garcıa-Sánchez, A. Mora, P. Castillo, and J. Merelo, “Designing Competitive Bots for a Real Time Strategy Game using Genetic Programming,” in *El Congreso de la Sociedad Española para las Ciencias del Videojuego, CoSECiVi*, 2014.

[31] R. de Freitas Pereira, C. F. M. Toledo, M. K. Crocomo, and E. do Valle Simões, “An Evolutionary Algorithm Approach for a Real Time Strategy Game,” in *Computer Games and Digital Entertainment (SBGAMES), 2008 Brazilian Symposium on*, 2008.

[32] Y. Jin and J. Branke, “Evolutionary optimization in uncertain environments - A survey,” *Evolutionary Computation, IEEE Transactions on*, vol. 9, no. 3, pp. 303–317, 2005.