Evolving Game State Evaluation Functions for a Hybrid Planning Approach

Xenija Neufeld
Faculty of Computer Science
Otto von Guericke University
Magdeburg, Germany,
Crytek GmbH, Frankfurt, Germany

Sanaz Mostaghim
Faculty of Computer Science
Otto von Guericke University
Magdeburg, Germany

Diego Perez-Liebana
School of Electronic Engineering
and Computer Science
Queen Mary University of London
London, United Kingdom

***Abstract*—Real-time games often require a combination of long-term and short-term planning as well as interleaved planning and execution. In our previous work, we introduced a hybrid planning and execution approach, in which high-level strategical planning is performed by a Hierarchical Task Network Planner and micro-management is done through Monte Carlo Tree Search. We use evaluation functions that represent weighted sums of selected game features as an interface between the two hierarchy levels.**

In this work, we present a way of automatically evolving the weights of these evaluation functions in order to improve the efficiency of the execution of high-level tasks. We compare the agent using the evolved evaluation functions with the one using manually created evaluation functions against state-of-the-art controllers in the Real Time Strategy game environment microRTS.

I. Introduction

Real Time Strategy (RTS) games represent challenging simulation environments for many other real-time planning problems. An agent acting in such an environment needs to create a plan within a short amount of time while working with a large search space. Additional challenges are non-determinism coming from the players' actions, as well as the environment itself, dealing with heterogeneous units, and – in most cases – partial observability.

However, despite these uncertainties, the agent working in such an environment is usually required to follow a long-term plan or a high-level strategy *and* react to changes adapting its behavior on the micro-management level. Thus, in order to stay reactive it needs to constantly monitor the environment and interleave planning and execution.

One of the approaches that has proven to work well on the micro-management level in large search spaces is Monte Carlo Tree Search (MCTS) and its variations. These approaches have been used in multiple game environments [1]–[3]. However, due to a high number of units that an agent controls in an RTS game and the number of actions that each unit can execute, the resulting high branching factor and the limited computation time usually allow MCTS to plan only a few steps ahead. For that reason, MCTS – at least when dealing with actions on a unit level – cannot be used for strategical *long-term* planning.

On the other hand, planning approaches can be used for long-term planning. Specifically a Hierarchical Task Network (HTN) planner can be used to create plans in a similar manner to human thinking: decomposing a high-level task into subtasks until reaching a certain level of action granularity and providing a plan as a sequence of actions. However, again, due to large branching factors when dealing with unit actions and the high costs derived from the manual design of the task network, a (HTN) planner is not suitable to be used for unit micro-management. Instead, it can be used on an abstract high-level. Another disadvantage of using a planner creating *detailed* long-term plans with micro-actions is the fact that due to the high dynamics of a game environment, the plan is very likely to fail during its execution requiring a new plan. In contrast, MCTS does not create a long plan but its flexibility and reactive capabilities allows it to make appropriate decisions at each time step, returning only one action that is considered best in the current state.

Combining the advantages of both an HTN planner and MCTS, we have introduced a hybrid approach in our previous work [4]. The hybrid approach uses an HTN planner for high-level planning and MCTS for micro-management. The tasks produced by the planner guide the action selection of MCTS through evaluation functions. Thus, instead of saying *how* a task should be executed, we describe *what* should be optimized by MCTS in order to finish the task. That way, the agent is able to stay reactive to small changes in the environment using MCTS to make decisions in every step considering the current game state and the current task. Additionally, a monitoring system detects failures on the higher level and triggers a re-planning whenever it is required.

Initial experiments of the hybrid approach in the research environment *microRTS*[1] [5], [6] have shown that when using different evaluation functions for distinct tasks the agent shows emergent behaviors. Additionally, the agent was able to follow long-term plans while staying reactive to the opponent's actions and changing its high-level strategy, if needed.

However, the first experiments also indicated that changing the evaluation functions that represented the high-level tasks

[1] microRTS: https://sites.google.com/site/microRTSaicompetition

might have a big impact on the agents behavior. Every task's evaluation function was a weighted sum of sub-functions that optimized different values. For example when *attacking* the opponent, the agent should 1) maximize the health points of friendly units while 2) minimizing the distance to the opponent's base as well as 3) minimizing the health points of the opponent's units. When changing the weights assigned to these 3 sub-functions, the agent would show different levels of aggressiveness.

Finding the right balance between the weights *manually* turned out to be a difficult task (and led to subjective results). For that reason, in this work, we propose using an Evolutionary Algorithm (EA) to learn the optimal weights for each task's evaluation function. We compare the agent using the learned weights with the one using the initial manually defined weights testing both against multiple benchmark opponents in microRTS.

The rest of the paper is structured as follows: Section II describes related work and Section III gives a more detailed insight into the hybrid approach. Section IV describes the application of an Evolutionary Algorithm to the weights of the Hybrid Approach. Section V gives insights into the experimental work and its results. Section VI concludes with our findings and proposes possible future work.

II. Related Work

Over the last decade, multiple works have explored Monte Carlo techniques in several RTS game environments. The work described in [7] introduced *MCPlan* which was using an abstract state space and MCTS for the strategical level in *OpenRTS*. This work has shown promising results in RTS games.

[8] used the Upper Confidence Tree (UCT) approach in *StarCraft*. Using UCT for the tactical battle layer, this work implemented multiple additional systems for the strategical and economical decision making as well as for micro-management of units.

Similarly, [9] used UCT for the tactical layer in the game *Wargus* dealing with abstract game states. This approach was reasoning about groups of units while the micro-management of single units was taken care of by the game engine.

Further works that have explored UCT approaches in *StarCraft* are [10] and [11]. In [10], the vanilla version was extended to a UCT *Considering Duration* (UCTCD). It was used for tactical movement and was able to outperform the built-in AI of StarCraft. However, it was not able to perform well when dealing with a high number of units.

The work described in [11] outperformed the previous work even for big numbers of units by abstracting the search space. Instead of dealing with micro-actions of single units, it implemented more complex scripts which were assigned to groups of units.

In a similar way, [12] used MCTS for army maneuvering in *StarCraft*. It dealt with abstract game states and an abstracted high-level forward model for action simulation. Additionally, it used abstract evaluation functions for MCTS. Although, this approach was not able to search deep enough using MCTS and thus could not outperform the built-in AI of *StarCraft*, it has shown that abstracting the search space could provide meaningful actions while decreasing the branching factor.

The works described in [13], [14] have compared multiple Monte Carlo approaches in *microRTS* specifically. All approaches were used on the micro-action level. These works have shown that the sampling strategy *naiveMCTS* outperforms algorithms such as UCT or Alpha-Beta search, especially when dealing with large branching factors. For that reason, for both this and our previous work [4], we use naiveMCTS. Later, the vanilla version of naiveMCTS was improved by guiding the search through a pre-learned probability distribution of unit actions as described in [15]. This version outperformed the original naiveMCTS agent.

Outside of RTS games, MCTS was also used in the General Video Games Playing environment as described in [2]. Here, an agent was developed to play *any* of the games provided by the framework. In order to optimize the agent's performance, this work searched for optimal parameters of MCTS such as the search depth or the exploration factor. Similar to the work presented in the present paper, the performance of MCTS is meant to be optimised by tweaking evaluation functions instead of its parameters. Agents with adapted parameters performed similarly or better than the vanilla version of the algorithm in different games.

Planning techniques have also been used in RTS game environments. [16] introduced Adversarial planning with HTNs (AHTN) wich combined game tree search and HTN planning. As opposed to classical game tree search, this work allowed for simultaneous and durative actions dropping the assumption of a turn-based game. All AHTN agents with an HTN depth of more than 1 outperformed previous benchmark agents in microRTS.

The work described in [17] proposed a different hierarchical approach combining Hierarchical Adversarial search on high levels with either Alpha-Beta search or Portfolio search for micro-actions. This approach outperformed agents that used only Alpha-Beta search, Portfolio Search or UCT when dealing with more than 72 units in *SparCraft* (a StarCraft combat simulator)

Multiple approaches have been abstracting the action space by using scripts that assigned actions to units following pre-defined rules. [18] introduced the Puppet Search which forwarded the game state through scripts and limited the search space exposing only a few choice points to the search algorithm. This approach performed similar to the best benchmark agent in StarCraft. Later, it was extended by a convolutional neural network (CNN) [19]. The CNN was used for script selection while the low-level tactics was left to game tree search. This approach won the microRTS competition in 2017 as the *StrategyTactics* agent.

Further works that used scripts include the winner of the 2018 microRTS competition *Tiamat* [20], [21] and one of the top scoring agents *Capivara* [22]. [20] proposed assigning the same scripts to units of the same type. Building up on this idea, [23] explored the assignment of different scripts to subsets of units in combination with naiveMCT. Here, the scripts restricted the actions available to certain units when searching with naiveMCTS. In [21], the rule-based scripts were used to generate a larger set of strategies by changing the parameter values of the rules. Afterwards, an evolutionary algorithm was used to find an optimal subset of the generated strategies. In [24] new scripts were created from the existing ones using a voting system.

Evolutionary approaches have been applied to different parts of MCTS and tested in multiple game environments previously. [25] evolved a new *default policy* to be used in MCTS (see Section III) in the game Ms Pac-Man. The evolved agent outperformed both a random and a hand-crafted controller.

A particularly relevant previous work to the one presented in this paper is [26]. Their authors evolved the evaluation functions used to evaluate board states in the game Reversi. It implemented a full Genetic Algorithm that evolved all parts of the evaluation functions (including the arithmetical and logical operators). In contrast to this, we keep these parts of our evaluation functions static (they are manually created according to our expert knowledge and the requirements of a high-level task) and evolve the weights only (see Section III). This work has shown that evolving evaluation functions can greatly improve the performance of an MCTS agent.

Another approach that evolved the weights used in evaluation functions of MCTS is described in [27]. However, in contrast to other approaches that performed the evolution and the evaluation of individuals offline (as we do), [27] proposed integrating the evaluation of individuals' fitness with each iteration of the MCTS approach. This approach has shown that it is possible to improve the agent's performance during one single game.

III. Hybrid Planning and Execution

A. Hybrid Planning

In contrast to classical planning techniques where a goal is reached by searching through the space of states, Hierarchical Task Network Planners provide a way of solving a goal task by decomposing it into subtasks [28], [29]. That way, the search space can be decreased early in the search process removing branches in the task network that cannot be used to decompose the goal task given the current game state. However, manual creation of task networks is very time-consuming and requires good expert knowledge. Furthermore, using an HTN planner for strategical *and* tactical planning for multiple units in RTS games still leads to high branching factors towards the low levels of a decomposition tree. Additionally, due to the high dynamics of RTS games, a detailed long-term plan created by an HTN planner is likely to fail early in the execution phase. This leads to high re-planning frequencies and diminishes the advantage of long-term panning.

On the other hand, Monte Carlo Tree Search techniques have shown to perform well in large search spaces such as RTS games, even when planning for a high number of units (see Section II). However, usually, they need to run every time that a decision needs to be made returning only *one* currently optimal action. Thus, they cannot incorporate long-term strategical planning.

The hybrid planning approach is based on the idea of combining the advantages of both – HTN planners and MCTS [4]. This is done by using an HTN planner for strategical planning with abstract high-level tasks, defining each task in the final HTN plan as an evaluation function and using this function for state evaluation by MCTS. This way, each evaluation function implicitly describes *what* should be optimized when executing the corresponding HTN task, instead of explicitly defining *how* to execute this task. Thus, there is no need for developers to manually create rules or commands for each unit's behavior.

For example, when executing the HTN task *CollectResources*, the agent should maximize the number of resources. However, usually an agent should optimize multiple potentially conflicting objectives. Additionally to the collection of resources, it should try to stay alive by maximizing the health points of friendly units making them avoid enemy units on their way. Therefore, a task's evaluation function needs to take into account all objective functions required for this task balancing their importance through weights. Formally, an evaluation function f_τ of an HTN task τ is defined as a weighted sum of N evaluation functions f_j that optimize N different objectives x_j. Each objective's function is multiplied by the corresponding weight w_j as shown in Equation 1.

$$f_\tau = \sum_{j=1}^{N} w_j f_j(x_j) \tag{1}$$

This evaluation function is then passed to MCTS to find optimal unit actions. In general, combining tree search and Monte Carlo simulations, MCTS searches for an optimal solution of a Multi-Armed-Bandid (MAB) problem [30]. Following a *tree policy*, it starts with a root node expanding an action a. A common tree policy is Upper Confidence Bound (UCB1) [31] which balances between exploration and exploitation of actions and is shown in Equation 2. Each node stores the number of times that a node s was visited is stored as $N(s)$ on each node, as well as the number of times that a certain action a was applied in this node $N(s,a)$, and the average reward $Q(s,a)$ gained from its visit.

$$a^* = arg\ max_{a \in A(s)} \left\{ Q(s,a) + K \sqrt{\frac{ln\ N(s)}{N(s,a)}} \right\} \tag{2}$$

After applying the tree policy, the expansion step adds another node to the tree when a node is visited from which not every action has been explored. After the expansion step, a Monte Carlo simulation (or rollout) runs from a leaf node of the tree following a *default policy* until reaching another terminal criterion. This policy (in its default version) selects actions uniformly at random. The state reached at the end of the rollout is then evaluated using an evaluation function, in our case f_τ. The result of f_τ is back-propagated to visited nodes in this iteration, updating their statistics. Once the budget time is consumed, a recommendation policy selects the best action a^* to play it in the game.

In case of an RTS game, the recommendation policy needs to select an optimal *combination* of unit actions. Thus MCTS is dealing with a Combinatorial MAB (CMAB) problem. An overview of CMAB sampling techniques is provided in [14] and one of the most effective techniques has shown to be naiveMCTS [13]. This approach (naively) assumes that the reward distribution of multiple units' actions can be approximated as the sum of each unit's reward. This allows breaking the CMAB problem into one global MAB problem and a local MAB problem for each unit.

The approach uses an ϵ-greedy policy π_0 (where ϵ is the probability for exploration and $1 - \epsilon$ for exploitation) deciding whether to exploit the local MABs or to explore the global MAB. In case of exploration, it uses an ϵ-greedy policy π_l independently selecting an action for each unit and adding it to the global MAB. In case of exploitation, it uses a pure greedy policy π_g selecting an action combination. naiveMCTS has shown to outperform sampling techniques such as Upper Confidence Tree. For that reason, we are using its implementation in this work as described in [13].

B. Interleaved Planning and Execution Architecture

In order to stay reactive while following the long-term plan provided by the HTN planner, the agent needs to monitor the game environment and interleave high-level planning and execution. Figure 1 shows the architecture of the hybrid agent. The *Agent Controller* represents the interface between the *HTN planner*, *MCTS* and the game environment *microRTS*.

In the beginning of a match, the Agent Controller forwards the current game state to the HTN planner. Using the predefined planning domain (task network) the planner creates an abstract high-level plan such as *CollectResources, BuildAndDefend, AttackOpponent* and returns this plan as an ordered list of tasks to the Agent Controller. Taking the first task of the plan – *CollectResources* – which contains a corresponding evaluation function, the controller forwards the function to MCTS. Using this function, MCTS takes care of the micromanagement of the task. It computes the optimal *player action* (combination of actions of all free units) and returns it to the controller which then forwards it to the game environment. The actual execution of the player action is then performed by microRTS itself.

In the following frames (time steps), the agent checks whether any of the units can execute any action. If this is possible, the controller first checks whether the currently executed task (*CollectResources*) is still valid given the current game state. This check allows the agent to react to changes in the game world. For example if the opponent attacks first, our agent will stop collecting resources and will try to prevent the attack. In this case, it will report a plan failure to the HTN planner and trigger re-planning. However, if the task is still valid, the agent will then check whether it has been reached – thus whether enough resources have been collected. The end

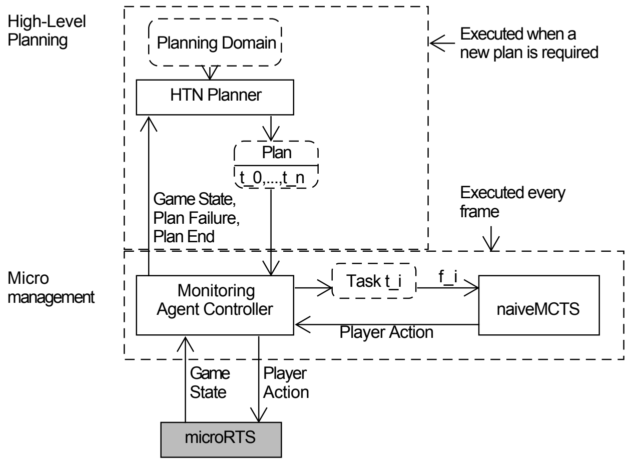

Fig. 1: Architecture of the hybrid planning and execution approach for microRTS.

of each task is defined through post-conditions. Depending on the result of this check, the agent controller will either continue using the current evaluation function or it will switch to the next task in the high-level plan (also checking for its validity) providing a new evaluation function.

First experiments of the hybrid approach have shown that the combination of the two planning approaches allows the agent to stay reactive while showing emergent unit behaviors when executing different high-level tasks [4]. Furthermore, tweaks of the subfunctions weights had a big impact on the behaviors. However, balancing these weights turned out to be a difficult task. For that reason, we propose evolving these weights for every task separately in order to learn evaluation function that best represent HTN tasks and allow MCTS to complete the task in shortest time.

IV. Evolution of Weight Vectors

As already mentioned in Section III, the hybrid planning approach uses evaluation functions as an interface between the HTN planner and MCTS. Each evaluation function is a weighted sum of subfunctions that optimize different objectives. It is possible that two different tasks have the same set of subfunctions with different weights prioritizing some objectives over the others. However, it is difficult to balance these weights manually in such a way that the resulting evaluation functions allow MCTS to achieve the corresponding HTN tasks as fast and precisely as possible.

In order to learn the weights, we propose using an evolutionary algorithm. We represent an individual i by the weight vector of the corresponding evaluation function f_τ which is described in Equation 1. Each gene $g_j \in i$ represents the weight $w_j \in [0.0, 1.0]$. The initial population is created with each weight set randomly to a value between 0.0 and 1.0. For genetic operators, we use uniform crossover between two neighboring individuals. Here, the values of each gene pair can be switched with a 50% probability. Additionally, we use mutation with each gene's value having a 50% chance of being re-set to a new value between 0.0 and 1.0.

We aim to find weights that allow achieving the corresponding task τ *in the shortest time*. For that reason, we propose using the difference between the maximum allowed time of a match t_{max} and the time that an individual i uses to finish the task t_i as a measurement of an individual's fitness, which is to be *minimized* as shown in Equation 3. The fitness $F(i)$ is measured for each match that an individual plays. Note, that t_i starts with the execution of the task and ends when the task is either successfully reached or fails.

$$F(i) = \begin{cases} -(t_{max} - t_i) & \text{if task } \tau \text{ reached} \\ (t_{max} - t_i) & \text{if task } \tau \text{ failed} \end{cases} \tag{3}$$

Since we are minimizing the overall time that an individual needs to finish a task, we take the negative value of the difference for successful matches and the positive value for unsuccessful matches. This way, the individuals that *reach* a task fastest get higher rewards and those that *fail* fastest get higher penalties.

As described in Section V-A, each individual is evaluated through multiple matches on multiple maps against multiple opponents. Trying to optimize (minimize) the individual's fitness for *all* maps and opponents, the total fitness of an individual $F_T(i)$ is computed as the difference between t_{MAX} (the maximum total time allowed for all matches on all maps against all opponents) and the sum of fitness values of all (M) matches $F(i)$ as shown in Equation 4. This way, starting with the maximum time that an individual could possibly require to achieve a task, we are optimizing towards 0.

$$F_T(i) = t_{MAX} - \sum_{match=1}^{M} F(i) \tag{4}$$

However, whether or not the execution of a task starts depends on the effectiveness of preceding tasks. For example an agent cannot build anything if it failed to collect enough resources. For that reason, it is important to evolve the evaluation functions in the order that the tasks would usually be scheduled (for example first the task *CollectResources*, then *BuildAndDefend*, and then *AttackOpponent*). In case a task does not even start its execution, we cannot evaluate the individual's fitness and repeat the match.

We follow a $(\mu + \lambda)$ strategy (with $\mu = \lambda$) with elitism keeping the best solutions of a population. Additionally, we save the μ best solutions found so far in an archive.

V. Experiments

A. General Experiment Setup

As already described in [4], we have created a simple HTN domain using the following 4 primitive tasks: *CollectResources*, which allows the agent to collect resources in the beginning of a game. *BuildAndDefend* is usually scheduled next making the agent build up his military force while staying close to and defending his own base. *PreventAttack* is used in case the agent controller detects an opponent's attack. In this case, the agent concentrates all his forces on destroying the attacking units. Finally, *AttackOpponent* is scheduled to perform an assault.

With the 4 tasks, the agent would use one of the 4 corresponding evaluation functions or individual types. The length of each individual type varied depending on the number of objective subfunctions of the corresponding tasks as follows: 7 for *CollectResources*, 15 for *BuildAndDefend*, 14 for *PreventAttack*, and 14 for *AttackOpponent*. In the initial experiments, we have noticed that the efficiency of our agent would change depending on the map size when using the same weight vectors for all map sizes. For example, a weight vector for the *AttackOpponent* evaluation function that would lead to a high winning rate on a small map would not make the agent *aggressive* enough on a bigger map. For that reason, we decided to evolve the 4 evaluation functions for small maps (8×8 cells), mid-size maps (16×16 cells), and big maps (24×24 cells and bigger) separately. That way, we have evolved 12 different evaluation functions in the current experiments.

Furthermore, we have selected different types (structures) of maps in order to optimize the agent's behavior in different environments. All maps provided by microRTS are symmetrical. Following the rules of the microRTS competition [5], the maximum time of a match was limited according to the map size as follows: 3000 frames for small maps, 4000 frames for mid-size maps, 5000 frames for big maps with a width of 24 cells, 6000 for 32 cells wide maps, and 8000 for maps of 64 cells width, and 12000 for bigger maps.

In case the agent did not start executing the task-to-learn in a match, the match was repeated. However, knowing that the HTN domain would never lead to some tasks to be scheduled on certain maps, these maps were let out of the evolution of these tasks. For example, the task *CollectResources* would not be scheduled on maps where the players started the game with enough resources already collected. Some small maps, on the other hand, required a quick assault, so that the agent would never execute the task *BuildAndDefend*, attacking with worker units only.

For the naiveMCTS part of our agent, we have used the following parameters that are set as default parameters by the naiveMCTS agent provided with microRTS: ϵ-greedy policies for π_0, π_l and π_g with $\epsilon_0 = 0.4$, $\epsilon_l = 0.3$ and $\epsilon_g = 0$ (for more details see [13]). The maximal tree depth for MCTS was 10, the maximal simulation time 100 frames, and the playout policy was *RandomBiasedAI*. The RandomBiasedAI agent is provided with the microRTS framework and its action selection is biased towards non-movement actions (collect, attack and return a resource) rather than movement actions.

Agents from the first experiment described below were trained against *Tiamat* [20]. The second experiment also used *SCVPlus* [24] and a pure *naiveMCTS* agent which was using exactly the same parameters as our agent and the *SqrtEF* evaluation function throughout the whole match as described in [13].

Afterwards, we have compared the evolved agents described below against the *Initial* agent that was using manually-crafted weight vectors. For this comparison, each agent played 50 games (25 on each player side) against each of the 3 opponents from the training set as well as each of the additional 3 test opponents: *Capivara* [22], *AHTN* [16], and *StrategyTactics* [19] on each of the maps used during the evolution. We have measured the *winning percentages* shown in Table I by summing the number of victories of the row-player against the column-player and half of the number of draws, dividing the sum by the number of matches and multiplying the result by 100.

B. First Experiment

In our first experiment, we have trained our agent against the winner of the 2018 microRTS competition – *Tiamat* [20]. During the evolution, each of the 12 evaluation function was evolved over 20 generations with a population size of $\mu = \lambda = 10$ individuals, a mutation probability of 50% and a crossover probability of 50%. During the first training, we let each *individual* play 6 matches (3 on each player side) against *Tiamat* only on each map of the corresponding size.

Having 10 final individuals of each task's evaluation function in the corresponding archives, we have selected the ones with the lowest score from each archive (the most optimal ones). The combination of these individuals trained against *Tiamat* is represented as I_{T20} in Table I.

Afterwards, we repeated the first setup against *Tiamat*. However, we have changed the number of matches that each individual played to 2 (1 on each player side) and changed the mutation probability to 20%. With this setup, we evolved 30 generations of each evaluation function. Similarly to the first setup, we selected the best individuals from each archive for the test games against the 6 opponents. These individuals are represented as I_{T30} in Table I.

From the first experiments, we have noticed that *Tiamat* alone was too strong for training. This was especially obvious when training the *AttackOpponent* task on mid-size and bigger maps. In these cases the task started but was never reached throughout all 20 (30) generations. Thus, according to Equation 3, the evolutionary algorithm only optimized the time that an individual used to fail giving smaller penalties for longer execution trials. It could not reward successful trials.

C. Second Experiment

In order to have better chances of reaching tasks during the training phase and to train our agent to perform well against different types of opponents, we have used all 3 agents to train against in the next experiment (*Tiamat*, *SCVPlus*, and *naiveMCTS*). Similarly to the first setup, we have used a mutation probability of 50%, and a crossover probability of 50%. The population size was kept at 10 individuals and each individual played 3 matches against each opponent.

As already mentioned, we have evolved evaluation functions in the order that the corresponding tasks would usually be scheduled. Thus, for each experiment, we first ran a full evolutionary process for *CollectResourses*, then for *BuildAndDefend* and *PreventAttack* simultaneously and finally for *AttackOpponent*. This meant that, especially for mid-size and big maps, the evolution process of *all* tasks might take multiple hours. With the fact that the algorithm *repeated every match*, in which the agent *did not start* to execute the task-to-learn, the evolution process might even take several days. This became very obvious in the last experiment with 3 opponents to train against. For that reason, we limited the number of generations to 10.

After the evolution, similarly to the first experiments, we have first selected the best individuals from each task's archive. The combination of these individuals is represented as I_{F1} in Table I. Since this combination has shown better results than individuals from previous experiments, we have tested further combinations of archive individuals. Therefore, we have identified groups of *related* individuals (those with the same values in the same genes) in each archive. Using this information, we have taken the best individual of each group (different from the optimal one used for I_{F1}) and combined them to be used in the agents I_{F2} and I_{F3}.

As can be seen in Table I, I_{F1} – the agent using the combination of the evaluation functions with the best fitness – shows the highest performance gain for most maps compared to the *Initial* agent. Although, in some cases, combinations I_{F2} and I_{F3} also outperform agents that were trained against *Tiamat* only.

D. Results and Limitations

The experiments performed in this work have shown that, in general, an evolutionary algorithm can generate weight vectors for evaluation functions that lead to agent performance similar to and better than the performance of an agent using manually crafted weights. Although in most cases the agent was not able to outperform the two best controllers *Tiamat* and *Capivara*, the evolved agents had a performance gain compared to the initial agent which shows promising results for future work.

However, the biggest limitation of the proposed methodology lies in the fact that tasks that are usually scheduled later in a match (e.g. *AttackOpponent*) strongly rely on an optimal execution of preceding tasks (e.g. *BuildAndDefend*). If preceding tasks are not executed efficiently enough, the opponent might gain a great advantage not allowing our agent to reach or even start later tasks.

This leads to the problem that the algorithm either can not reward successful matches (it will only penalize matches, in which the task fails) or that a match has to be repeated (if the task-to-learn does not start at all). Considering that in the next match the same agent configuration will be tested against the same opponent, such a match might have to be repeated for a very high amount of times. Thus, in order to avoid such situations, we need a different way to deal with matches in which the task does not start.

FourBasesWorkers8x8	Tiamat	SCVplus	Naive-MCTS	Capivara	AHTN	Strategy-Tactics
Initial	85	96	90	80	100	98
I_{T20}	91	86	78	70	100	84
I_{T30}	80	91	63	60	100	86
I_{F1}	96	100	96	98	100	100
I_{F2}	97	100	96	98	100	100
I_{F3}	86	94	82	82	100	98

basesWorkers8x8A	Tiamat	SCVplus	Naive-MCTS	Capivara	AHTN	Strategy-Tactics
Initial	51	100	51	17	91	51
I_{T20}	38	94	57	16	85	72
I_{T30}	39	100	16	25	65	19
I_{F1}	74	100	74	29	85	77
I_{F2}	62	100	73	26	77	79
I_{F3}	46	99	57	10	96	53

NoWhereToRun9x8	Tiamat	SCVplus	Naive-MCTS	Capivara	AHTN	Strategy-Tactics
Initial	0	98	78	14	98	58
I_{T20}	0	95	80	27	94	86
I_{T30}	0	100	99	38	100	94
I_{F1}	10	99	100	46	100	93
I_{F2}	6	99	99	32	100	94
I_{F3}	0	84	32	27	92	41

basesWorkers16x16A	Tiamat	SCVplus	Naive-MCTS	Capivara	AHTN	Strategy-Tactics
Initial	0	97	64	0	85	37
I_{T20}	2	68	54	0	57	20
I_{T30}	10	98	82	0	64	58
I_{F1}	10	100	96	0	88	63
I_{F2}	2	100	93	0	98	58
I_{F3}	4	94	96	2	75	84

TwoBasesBarracks16x16	Tiamat	SCVplus	Naive-MCTS	Capivara	AHTN	Strategy-Tactics
Initial	0	83	57	0	96	2
I_{T20}	0	59	50	0	69	2
I_{T30}	2	94	77	0	100	13
I_{F1}	2	96	73	2	100	16
I_{F2}	0	93	85	0	96	8
I_{F3}	0	100	85	0	98	12

basesWorkers24x24A	Tiamat	SCVplus	Naive-MCTS	Capivara	AHTN	Strategy-Tactics
Initial	0	81	50	0	80	47
I_{T20}	0	52	50	0	100	48
I_{T30}	0	48	50	0	98	43
I_{F1}	0	72	50	0	100	32
I_{F2}	0	42	50	0	60	10
I_{F3}	0	54	50	0	94	44

DoubleGame24x24	Tiamat	SCVplus	Naive-MCTS	Capivara	AHTN	Strategy-Tactics
Initial	43	69	50	6	99	0
I_{T20}	50	72	50	8	68	8
I_{T30}	50	70	50	0	40	8
I_{F1}	50	69	50	6	99	0
I_{F2}	50	70	50	6	96	0
I_{F3}	48	68	50	4	80	2

BWDistantResources32x32	Tiamat	SCVplus	Naive-MCTS	Capivara	AHTN	Strategy-Tactics
Initial	0	0	50	0	10	3
I_{T20}	0	0	52	0	14	18
I_{T30}	0	0	58	5	28	6
I_{F1}	0	0	50	0	0	14
I_{F2}	0	0	52	0	16	10
I_{F3}	0	0	50	0	8	4

TABLE I: Winning percentage of the row player against the column player. Computed as the sum of victories of the row-player against the column-player and half of the number of draws, divided by the number of matches and multiplied by 100.

Moreover, with growing map size and difficulty the chances to achieve *any* task were decreasing. We assume that with a higher number of generations the proposed approach could achieve better results for bigger maps. However, the fact that tasks had to be evolved in a certain order meant that the total evolution time was increased a lot. At this points, co-evolution of tasks might be a better alternative to the sequential evolution proposed here.

Furthermore, the test games have shown that there are situations where it is not enough to improve the evaluation functions. Additionally, the high-level HTN domain needs to be improved. For example, on the map *NoWhereToRun9x8*, our agent struggled winning against *Tiamat* and *Capivara*. This map is the only one where players cannot reach each other from the beginning of the match because they are separated by a wall of resources. But they can create *ranged* units and shoot the opponent through the wall, which these two agents always did. However, since our agent was always maximizing the number of military units *of every type* during the *BuildAndDefend* task, it did not learn to *not create* melee units. In this case, a different HTN task could have been used enforcing the creation of ranged units and penalizing the creation of other units.

VI. Conclusions and Future Work

In this work, we have applied an evolutionary algorithm for the generation of weight vectors in weighted-sum evaluation functions. These functions are used by an MCTS approach to evaluate game states in the game environment microRTS. MCTS is guided through these evaluation functions to execute long-term tasks that are scheduled by a high-level HTN planner. This way, our agent is able to follow long-term plans while handling micro-management through MCTS and staying reactive to changes in the environment.

Although the agents evolved in the current experiments were not able to outperform the strongest state-of-the-art controllers, most of them showed a performance gain compared to the initial agent and outperformed other benchmark controllers. A big limitation of this approach are, however, very long evolution times. These result through a sequential evolution of evaluation functions according to a certain task order, the fact that the evaluation of each individual's fitness requires playing multiple matches, and the fact that a match has to be repeated if the task-to-learn does not start.

Future work involves further experiments with different evolution parameters. Additionally, due to the long evolution times when evolving the evaluation functions in a certain order, co-evolution of evaluation functions of different tasks seems to be a better option to deliver results faster. Further improvements of the evolution process might involve a full Genetic Programming approach evolving not only the weight vectors but the structure of the evaluation functions as described in [26].

In addition to the evolution of the evaluation functions, it is possible to evolve the HTN domain itself aiming for more efficient task transitions. For example, evolving the parameters used for pre- and post-conditions of HTN tasks in a similar way to the evolution of strategies described in [22]. Finally, the agent controller might be extended with adapting its behavior to an opponent's strategy deciding at run-time which of the evolved individuals to use.

References

[1] M. Ishihara, T. Miyazaki, C. Y. Chu, T. Harada, and R. Thawonmas, "Applying and improving monte-carlo tree search in a fighting game AI," in *Proceedings of the 13th International Conference on Advances in Computer Entertainment Technology*. ACM, 2016, pp. 27–32.

[2] C. F. Sironi, J. Liu, D. Perez-Liebana, R. D. Gaina, I. Bravi, S. M. Lucas, and M. H. Winands, "Self-adaptive MCTS for general video game playing," in *International Conference on the Applications of Evolutionary Computation*. Springer, 2018, pp. 358–375.

[3] D. Perez, P. Rohlfshagen, and S. M. Lucas, "Monte-carlo tree search for the physical travelling salesman problem," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2012, pp. 255–264.

[4] X. Neufeld, S. Mostaghim, and D. Perez-Liebana, "A hybrid planning and execution approach through HTN and MCTS," in *Accepted for the International Conference on Automated Planning and Scheduling. Workshop on Integrating Planning, Acting, and Execution.*, 2019.

[5] S. Ontañón, "MicroRTS AI Competition, https://sites.google.com/site/micrortsaicompetition/," 2017.

[6] S. Ontañón, N. A. Barriga, C. R. Silva, R. O. Moraes, and L. H. Lelis, "The first microRTS artificial intelligence competition." *AI Magazine*, vol. 39, no. 1, 2018.

[7] M. Chung, M. Buro, and J. Schaeffer, "Monte carlo planning in RTS games." in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2005, pp. 117–124.

[8] D. Soemers, "Tactical planning using MCTS in the game of starcraft." Bachelors thesis, Department of Knowledge Engineering, Maastricht University, 2014.

[9] R.-K. Balla and A. Fern, "UCT for tactical assault planning in real-time strategy games." in *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 2009, pp. 40–45.

[10] D. Churchill and M. Buro, "Portfolio greedy search and simulation for large-scale combat in starcraft," in *Proceedings of the IEEE Conference on Computational Intelligence in Games*. IEEE, 2013.

[11] N. Justesen, B. Tillman, J. Togelius, and S. Risi, "Script-and cluster-based UCT for starcraft." in *Proceedings of the IEEE Conference on Computational Intelligence in Games*. IEEE, 2014.

[12] A. Uriarte and S. Ontañón, "Game-tree search over high-level game states in RTS games," in *Proceedings of the 10th Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014, pp. 73–79.

[13] S. Ontañón, "The combinatorial multi-armed bandit problem and its application to real-time strategy games," in *Proceedings of the 9th Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013, pp. 58–64.

[14] ——, "Combinatorial multi-armed bandits for real-time strategy games," *Journal of Artificial Intelligence Research*, vol. 58, pp. 665–702, 2017.

[15] ——, "Informed monte carlo tree search for real-time strategy games," in *Proceedings of the IEEE Conference on Computational Intelligence and Games*. IEEE, 2016, pp. 1–8.

[16] S. Ontañón and M. Buro, "Adversarial hierarchical-task network planning for complex real-time games," in *Proceedings of the 24th International Conference on Artificial Intelligence*. AAAI Press, 2015, pp. 1652–1658.

[17] M. Stanescu, N. A. Barriga, and M. Buro, "Hierarchical adversarial search applied to real-time strategy games." in *Proceedings of the 10th Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014, pp. 66–72.

[18] N. A. Barriga, M. Stanescu, and M. Buro, "Puppet search: Enhancing scripted behavior by look-ahead search with applications to real-time strategy games," in *Proceedings of the 11th Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015, pp. 9–15.

[19] ——, "Combining strategic learning and tactical search in real-time strategy games," *arXiv preprint arXiv:1709.03480*, 2017.

[20] L. H. S. Lelis, "Stratified strategy selection for unit control in real-time strategy games," in *International Joint Conference on Artificial Intelligence*, 2017, pp. 3735–3741.

[21] J. R. H. Mariño, R. O. Moraes, C. F. M. Toledo, and L. H. S. Lelis, "Evolving action abstractions for real-time planning in extensive-form games," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.

[22] R. O. Moraes, J. R. H. Mariño, L. H. S. Lelis, and M. A. Nascimento, "Action abstractions for combinatorial multi-armed bandit tree search," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. AAAI, 2018, pp. 74–80.

[23] R. O. Moraes, J. R. Mariño, L. H. Lelis, and M. A. Nascimento, "Action abstractions for combinatorial multi-armed bandit tree search," in *Proceedings of the 14th Artificial Intelligence and Interactive Digital Entertainment Conference*, 2018, pp. 74–80.

[24] C. R. Silva, R. O. Moraes, L. H. Lelis, and K. Gal, "Strategy generation for multi-unit real-time games via voting," *IEEE Transactions on Games*, 2018.

[25] A. M. Alhejali and S. M. Lucas, "Using genetic programming to evolve heuristics for a monte carlo tree search ms pac-man agent," in *Proceeding of the IEEE Conference on Computational Inteligence in Games*. IEEE, 2013, pp. 65–72.

[26] A. Benbassat and M. Sipper, "EvoMCTS: Enhancing MCTS-based players through genetic programming," in *Proceeding of the IEEE Conference on Computational Inteligence in Games*. IEEE, 2013, pp. 57–64.

[27] S. M. Lucas, S. Samothrakis, and D. Perez, "Fast evolutionary adaptation for monte carlo tree search," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2014, pp. 349–360.

[28] D. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila, "SHOP: Simple hierarchical ordered planner," in *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*. Morgan Kaufmann Publishers Inc., 1999, pp. 968–973.

[29] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, "SHOP2: An HTN planning system," *Journal of artificial intelligence research*, vol. 20, pp. 379–404, 2003.

[30] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[31] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.