ISSN 1330-3651 (Print), ISSN 1848-6339 (Online)
https://doi.org/10.17559/TV-20190721135322
Preliminary communication

Applying LCS/XCS to the RTS Games Domain

Damijan NOVAK*, Domen VERBER

Abstract: Real-Time Strategy games (RTS) are representatives of the highest class of computational complexity in computer game genres. To cope with the high complexity of the state-action space of RTS game worlds, various Machine Learning algorithms are being used and researched extensively. In this article, we apply eXtended Classifier Systems (XCS) to the domain of RTS games. The XCS algorithm belongs to a Learning Classifier Systems (LCS) group known for their adaptability, generalisation, and scalability. We build the game agent named AIXCS. It uses a group of XCS algorithms, which generate a set of unit-actions used in the RTS game. The AIXCS operates without prior learning from the game runs and in tight timing constraints. The AIXCS was put to the test against other game agents in the micro RTS game environment, with positive results regarding successful game operation at runtime.

Keywords: AI; game agent; LCS; micro RTS; real-time strategy games; XCS

1 INTRODUCTION

Real-Time Strategy (RTS) games are a sub-genre of strategy computer games based on classical board games like chess or tic-tac-toe. The basic gameplay of RTS games usually takes place on some fictitious terrain, where players battle with each other. Usually, the players may also have to gather resources and manage the economy to build new units and structures. An important aspect of an RTS game is the construction of a base, which offers protection, unit upgrades, research capabilities, etc. Players need to master strategy, tactics and low-level unit behaviour, sometimes called micro management or reactive control, to win the game.

RTS games are high complexity games [1] because: the number of involved units is high (at one single time it can surpass several hundred units); information presented to the player can be imperfect (e.g. parts of a map can be hidden) or incomplete (e.g. information, about what kind of units the opponent chose to play with, is unknown to the player); randomness, uncertainty and non-deterministic behaviour can be included in the game engine operation; temporal continuity (the previous action constrains future action(s)); etc. This high computational complexity motivated an article from Buro [2], which was published in 2003, with the idea that RTS games can be used not just for gaming, but also as testbeds for Artificial Intelligence (AI) research.

The researchers tackled the RTS game domain with many types of Machine Learning (ML) techniques, for example, Monte-Carlo Tree Search, Deep Q Networks, rule mining, etc. The review of different computational intelligence methods (and successful applications) [3] and significant AI system (including evolutionary computation and deep RL) [4] shows rapid and extensive research regarding the ML domain. We decided to narrow it by limiting ourselves to its Reinforcement Learning (RL) category, which is based on reward received from the environment, and does not need prior large databases of game data (e.g. game replays of professional human players). In the article by Stanescu et al. [5] it was also nicely stated: "Without a large number of high-quality records, reinforcement learning techniques will likely need to be considered in future work".

Therefore, the main agenda of the article is to combine the RL and RTS game domains by creating a complete RTS game playing agent based on the Learning Classifier Systems (LCS). The agent will be tested against other respected game agents of the field. The motivation of choosing the LCS came from a survey written by Shafi and Abbass [6], which examined usage of LCS in games. The survey showed that LCS literature on the domain of RTS games is very scarce, and further research in the matter is very relevant.

LCS belongs to the category of the reinforcement learning (RL) and the Genetic Algorithms (GA). It uses evolutionary algorithm approach (along with other heuristics) to acquire rules regarding the environment. The learning process is used continually, with the goal of finding an optimal solution to a given problem.

There are many different types of LCS algorithms. We decided to use eXtended Classifier Systems (XCS) [7]. The decision was made on behalf of its efficient generalisations over complete state-action space. This is necessary for optimal gameplay. Otherwise the solution will be stuck in some local optima. However, efficient generalisation is also needed. The computational complexity of RTS game worlds is too prohibitive to allow exhaustive search of every state-action pair in the search space. Applying XCS to the game domain will test the suitability of such algorithms for operations and making decisions in a real-time game environment.

In previous research the usage of XCS in RTS games was studied in Rudolph et al. [10], where they created four separate micro management agents, each with its specific behaviour. They compared all the created agents playing between each other in basic scenarios. They proposed, but did not implement the inclusion of strategic command. They also did not include the resource gathering and production of units and structures. In contrast, we developed a complete RTS game agent that also incorporates resource management and production, as well as the strategic module.

Domains of RTS games and LCS algorithms were also combined before by researchers Tspanos et al. in [8], with a simplified version of the original LCS framework called Zeroth-level Classifier System (ZCS) [9]. The authors stated that the adaptation of an initial random set of rules through ZCS and repetitive game-playing, resulted in policies that dominated the static rule-based player.

For simulation and evaluation of the game agent, we decided to use micro RTS game engine [11]. It offers a simpler environment than fully-fledged RTS games like StarCraft™. Other researchers also use the same game engine, which allows for comparison among their solutions. For operations inside the micro RTS environment, a game agent named AIXCS was designed, which utilises the XCS algorithms. Structure of the game agent and its interaction with the game environment can be seen in Fig. 1. The structure of the presented game agent is universal enough and could also be used in other game genre environments wherever multiple units need to be operated simultaneously by the strategy level.

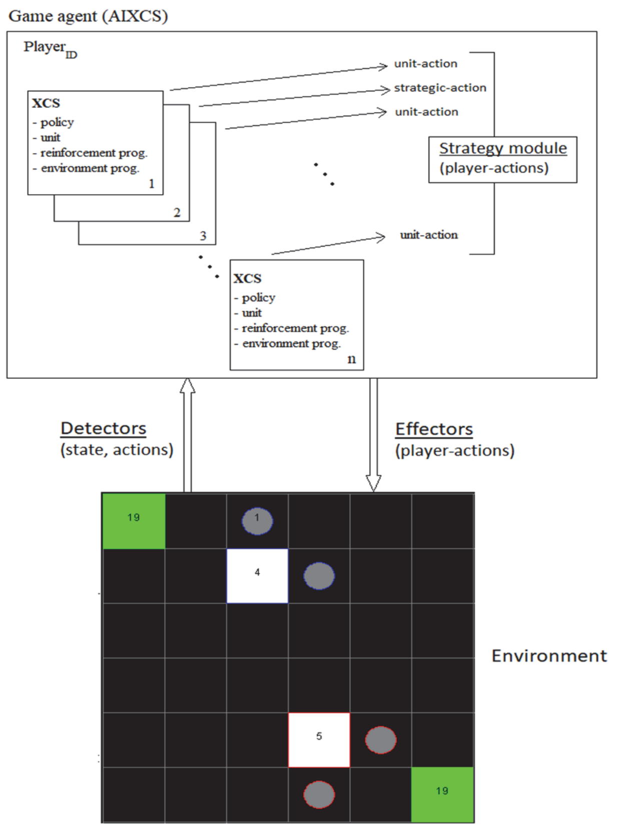

Figure 1 Game agent structure and its interaction with the game environment

Our game agent operates without any prior training [12]. This is because the international competition rules usually prohibit saving data between matches for additional processing, or because pre-processing is not possible. Otherwise, if interested, more about the prior learning of XCS agents in RTS games can be found in [12]. We must also note that, due to the inner complexity of our game agent, which includes multiple XCS algorithms and many game policies at play, we decided to include some limitations into the initial design. The most notable limitation is that we currently only process one game state in advance (i.e. no chaining of states). This limitation will help us understand better how XCS operates in the RTS game domain.

This paper is structured as follows: Section 2 gives a description of the microRTS simulation environment. In section 3 the general LCS structure and the details of the XCS algorithm are given. The XCS interaction with reinforcement and the environment programme is provided in Section 4. Section 5 explains the use of XCS for the full RTS gameplay. Section 6 explains the syntactic and semantic validation of classifiers (internal rules of XCS) and how to manage them in the RTS environment. Section 7 describes the experiment set-up and the results. Discussion is given in section 8. Section 9 offers the conclusion, current limitations of the agent and outlines for future work.

2 microRTS

microRTS is a simple simulation environment, designed with similar rules to the fully-fledged RTS games (e.g. Command & Conquer™ and StarCraft). It provides an Open AI integration API to create and evaluate different solutions quickly. It is used in many research articles and competitions between different solutions at prestigious conferences [13].

The microRTS game contains four types of mobile units (Worker, Light, Heavy and Ranged), two kinds of structures (Base and Barracks), one type of resource and a wall. The base produces workers [11]. Workers gather resources and build structures.They have only a limited firepower. Barracks produce military units (Light, Heavy and Ranged). Military units vary in their capabilities (e.g. a Ranged unit has a longer attack range). The wall is used as a physical barrier in the environment.

The simulation environment of the microRTS allows different game scenarios. Among others, it is possible to define the size of the map, the starting positions and types of the structures and units, starting position of the resources, etc. It also allows executing the game with the GUI turned off, which makes simulation much faster. Many game agents are pre-included. Some of them are further described in the section 7.

An example of a microRTS game state with all the environment elements is shown in Fig. 2. For greater clarity, only the description of the environmental elements for the first player, positioned at the bottom right corner, was included. The opponent starts at the top left corner.

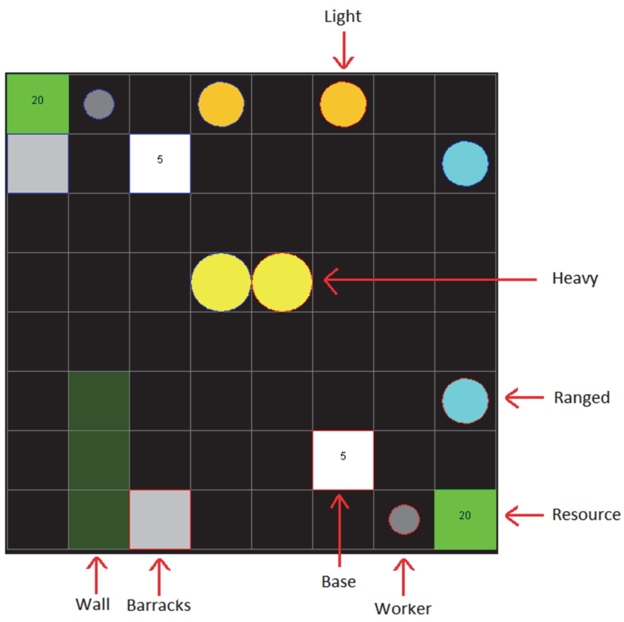

Figure 2 The example of microRTS game state

3 LCS/XCS
3.1 LCS

LCS are rule-based systems. At the core they comprise a population of classifiers: $C = \{C_1, C_2, \ldots, C_n\}$. Classifiers take the form of "IF condition THEN action" [14]. Condition is a vector that represents some game state. During the execution, each condition is matched with the actual input (state variables) provided by the environment. Each condition vector element is formed from the set of {0, 1, #}[15]. Values 0 and 1 represent the situation where classifier bits must exactly match the corresponding bits of the state variable. The symbol # represents a don't care value where the value in the state variable is ignored. For example, the condition 11#10 matches both 11010 and 11110 values of the state vector. This allows for generalisation of the condition. The action part of the classifier specifies the action that the classifier is propagating. Some examples of actions are: "produce worker left", "harvest", "wait", etc. Some groups of LCS's can introduce other attributes.

LCS learning, evolving of existing rules, as well as creation of new rules comes on an account of interaction with an environment from which it receives some numerical reward. Reward is the indication of the benefit of the actions. The goal is the maximisation of the amount of the reward gathered in the long run [16]. To achieve that goal, LCS learns an action-value function [17]. This function maps state-action pairs into a payoff (a real number). The payoff is used to guide the search for new rules, which should, ideally, be better at making more effective decisions.

During LCS operation, adaptability, generalisation, and scalability take place. LCSs are adaptive [14], and capable of runtime learning in rapidly changing situations by exploiting their GA. GA's purpose is to evolve the population of classifiers to improve the solution of the problem. New set of rules should be better in terms of being more accurate, or in being able to predict a higher payoff. Generalisation is also an important feature.The system that generalises properly can a) Represent in a compact form what it has learned, and b) Can apply what it has learned to previously unseen situations [16].

3.2 XCS

Because of our requirement of LCS to cope with the large and very complex state-action space of the RTS environment, the LCS's group XCS was chosen, for which literature [18] shows that it can learn a state-value function over the complete state-action space with efficient generalisations.

The implementation of our XCS algorithm is based on the work of Butz and Wilson [19]. Their work comprises a modular structured pseudo code with accompanying explanations. The XCS algorithm also interacts with reinforcement and the environment programme. The details of interaction and implementation are explained in Section 4.

XCS is a kind of the RL algorithms. It operates in an environment by taking some actions and for which it receives reward. The reinforcement component of XCS uses a form of Q-learning [20]. Q-learning allows agents to map states and actions to their utilities. The important part of Q-learning is the Markovian environment assumption, which states that any information needed to determine the optimal actions is reflected in the agent's state representation [21]. The process where the agent firstly observes the environment's output, consisting of a reward and the next state, and, secondly, upon that output executes a proper action, is called a Markov Decision Process (MDP) [22]. MDP specifies a finite set of states and actions. As long as all of those actions are sampled repeatedly in all states and action-values are represented discretely, Q-learning delivers convergence to the optimum action-values with probability one [23].

Inside the XCS algorithm [19] (Fig. 3) the knowledge about a problem is kept in classifiers. The algorithm uses four sets: Population set [P], Match set [M], Action set [A], and the Previous Action set $[A]_{-1}$. [P] contains all classifiers that exist in XCS at any time t. [M] includes all classifiers that match the current situation. [A] is formed out of the current [M], and consists of those classifiers from [M] that will be executed in the current cycle. $[A]_{-1}$ is a record of [A] from the previous XCS iteration. $[A]_{-1}$ is only used if we are dealing with a multi-step problem (i.e. the problem might require multiple steps in order to solve it successfully). Our current solution is implemented as a single-step problem and the $[A]_{-1}$ is not used.

```
initialize environment programme env
initialize reinforcement programme rp
initialize XCS // [P] creation; connection to the game policy
run_XCS()
----------------------------------------------------------------------
run_XCS():
1. do{
2.         σ = get situation from env
3.         generate [M] from [P] using σ // covering can occur
4.         generate PA from [M]
5.         act = select action according to [PA]
6.         execute act in env // simulation phase
7.         ρ = get reward from rp // evaluation of game state
8.         update set [A] using ρ with possibility of deleting
           in [P]
9.         run GA in [A] considering σ inserting in [P]
10. } while (termination criteria signal from rp is not met)
```

Figure 3 Pseudocode of the XCS algorithm

The pseudocode starts with the initialisation of the main components: The environment programme *env*, reinforcement programme *rp* and XCS. At the XCS initialisation phase the [P] is created, and a bond is formed from the XCS algorithm to the game policy. Next, the XCS cycle is started, which lasts until the termination criteria signal is sent from *rp*. At the beginning of the cycle, the input state σ (also called sensory information) is received from the environment. σ is then matched with [P] to create [M]. Matching is done by comparing specified bits of condition and of σ. During the execution of the algorithm some bits (or group of bits) of the condition can be set to a don't care value # [7]. This can occur at the covering and/or mutation phase.

If, after the creation of [M], the number of unique actions is lower than the number of all the available actions of the current search space, the covering operation occurs which covers all the available actions. After that, the Prediction Array (PA) is formed from the [M] [15], which consists of as many real values as there are unique actions. Each value is the estimate of the pay off that the system expects when an action is executed (Eq. (1)).

$$\mathrm{PA}(\alpha)=\frac{\sum_{cl.a=a\&cl\in[\mathrm{M}]} cl.p\cdot cl.f}{\sum_{cl.a=a\&cl\in[\mathrm{M}]} cl.f} \tag{1}$$

Legend: cl - classifier, a - action, p - prediction, f - fitness.

Pay off is a fitness-weighted average of the predictions of all classifiers in [M] that advocate that action [24]. It is calculated by considering each classifier in [M] and taking the sum of its prediction (*cl.p*) multiplied by its fitness to the prediction value total for that action (*cl.f*). This sum is then divided by the sum of fitnesses for that action.

After the PA has been created, the algorithm selects the action with the highest prediction, and executes that action in the *env*. This is also known as the simulation phase. After the simulation is complete, the *rp* returns reward ρ. Using the ρ, updating of [A] occurs (with the possibility of deleting in [P]). Lastly, GA is called upon an [A] (considering σ inserting in [P]).

XCS differs from all other LCS's variants in its rule fitness for GA. Rule fitness is not based on the amount of reward received, but purely upon the accuracy of predictions of reward [25]. Another important difference is that GA takes place in the [A] (implicit niching) and not in the [P]. In XCS, each classifier also keeps certain additional parameters (alongside condition, action and prediction estimates) needed for functioning. Those parameters are: Prediction error ε (a value which reflects the deviation of prediction from the actual reward), fitness f (the accuracy of the prediction of the classifier), experience *exp* (it represents the number of times the classifier has belonged to an [A]), time stamp *ts* (it denotes the last occurrence of a GA in an [A] to which this classifier belonged), action set size *as* (it estimates the average size of the action sets this classifier has belonged to), and numerosity *num* (the number of micro-classifiers [ordinary classifiers] this classifier-which is technically called a macro-classifier-represents) [10, 19].

4 REINFORCEMENT AND ENVIRONMENT PROGRAMMES

The Reinforcement and Environment programmes are important interfaces that help XCS with its operations (Fig. 4) [19]. They provide the XCS algorithm with information on an as-needed basis. With these two interfaces the raw information from the RTS domain is processed to the input format requirements of an XCS algorithm.

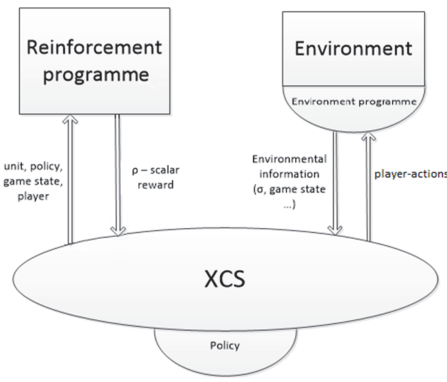

Figure 4 Interaction between XCS, environment, game policy and both programmes

4.1 Reinforcement Programme

The main objective of the reinforcement programme is to return the reward to the XCS algorithm for the specific unit that has executed an action. XCS uses this reward to update its [A]. Reward is calculated over the game state that was provided using the evaluation function that the game policy holds. Evaluation function is usually implemented as a subtraction of max and min player scores of the current game state.

To help with evaluation of the game state, we first tried evaluation policies that were already part of the micro RTS package, but initial testing did not return the results we anticipated. This was because those policies were made to fit better with other kinds of algorithms, e.g. Puppet Search [26]. So, we decided to create a specific interface, which uses domain knowledge and extracts relevant features from the environment to be used for evaluation of state. The group of features that we selected as relevant includes: Number of friendly and enemy units by categories (worker, light, heavy and ranged), flags which signal if a friendly or an enemy base is under threat or if it was already damaged, number of friendly and enemy bases / hit points / units / stockpiled resources. These relevant features were selected by hand from the group of features during initial pre-testing (for selection of some of the features we found inspiration in the works of Tspanos et al. [8] and Lujan [12]). The final selection was done through simple matching of which feature was showing better reward results regarding a given game policy. We also took advice from [27], which said: "… reward can be configured using a range of flags, but we strongly discourage disingenuous engineering of the reward function…". This advice was reflected in pre-testing by not applying the simple tactics (e.g. 2 vs. 2 light units), but on playing the bigger scenarios (e.g. full game starting with one base and one worker) while making observations as to which reward feature made a difference in overall performance.

4.2 Environment Programme

The main operations that are covered by the Environment programme are: Executing the combined player-actions for a given unit on the game state; receiving all the available unit actions for all the units on the playfield; filtering the unit actions based on the specific game policy; getting sensory information (used by XCS during [M]`s creation); getting purified (higher abstraction) data from the game state (XCS uses it for initial creation or covering of classifiers and while executing GA over [A]). The Environment programme is also responsible for two very important methods of syntactic and semantic validations of the classifier and of the action in play.

5 USE OF XCS FOR THE FULL-BLOWN RTS GAMEPLAY

In this Section, we connect the domain of Machine Learning (XCS algorithm) with the RTS game domain by presenting the components further required for creation of the full-blown RTS game agent (Fig. 1). In the first part of the Section, we are presenting five policies. One game policy is associated with exactly one XCS algorithm. These policies provide the units with a specific behaviour (e.g. production management), and cover all aspects of the RTS game play (including strategic command, because one game policy is specially dedicated to such a purpose). Second, we describe how each unit's actions, present on the game map, are combined to form the Player Actions with the strategy module's help. The XCS algorithm selects a unit's actions by following game policy agenda. In the third part, we present the map coding representation, which allows the XCS algorithm to compare the condition of its classifiers to the environment sensory information (the sensory input is provided by the environment programme).

5.1 Game Policy

Each XCS algorithm`s behaviour is defined by its game policy. Connection to the game policy is established at the start of the XCS creation and initialisation. We used two groups of policies: Tactical and strategic. Tactical policies operate with direct calls to the game engine with the commands for the specific units. Strategic policies use commands which are of higher abstraction and are not directly related to the game engine calls (e.g. more workers must go harvesting).

Tactical policies are set to an XCS that is associated with a single unit, while strategic policies oversee general gameplay, so they influence multiple units. We created five policies that cover all the actions that can be executed in the micro RTS game environment. The first one is the production policy, responsible for creating new units, and for constructing base and barrack buildings. The second is the tactical military policy, which defines if the unit movement command is needed, if it must retreat, or the unit has to attack a specific location. The third game policy is similar to the second one, with the difference that the move command has specific direction (up, down, left and right). The third game policy replaces the second one, when there are nearby enemy units. The range threshold of how far away the enemy is positioned can be set in advance. The fourth is a tactical harvest policy, specifying if a worker will wait, or if it will go harvesting. The fifth is a strategic military policy, updating two weights that determine the global behaviour of the game agent either towards the attack or the harvest. The weights can go up, down, or they can stay the same. Weights values are transferred (kept) between the game states.

Complete game agent behaviour is formed when each unit has at least one XCS algorithm set and connected to the game policy. The strategic policy is an on/off choice. It is turned on by default. A game agent can operate using only the tactical policies.

Each game policy contains a common set of parameters used by XCS during its running time:

- size of set [P],
- number of actions,
- learning rate values,
- bit size of condition,
- GA probabilities, etc.

The game policy also includes the following important variables:

- exact number and list of actions that it is associated with (e.g. production of a worker),
- a flag to signal if the game policy is tactical or strategic, and
- a flag that signals if for condition presentation the XCS uses the detailed game map representation with all the elements included (used by default), or if it uses aggregated environment information (e.g. the number of enemy workers).

Actual values of constants used by an agent and its XCS algorithms are presented in Section 7.

5.2 Player-Actions and Their Use by Strategy Module

When a player makes a decision and issues an action to a single unit, we call it unit-action. The game field can contain many units, and each is capable of executing several actions; a unit can always execute a wait action. A set of all unit-actions issued by a player in each decision cycle is called player-actions (for that purpose, micro RTS implements a class *PlayerAction* [28]).

The player-actions may contain commands for multiple units at the same time. Many game related algorithms already have the capability to use player-actions operations directly (e.g. game trees [29]). On the other hand, XCS is not capable to use PlayerActions directly, and can only return a single (the best) unit-action. Our solution to deal with the problem was to cover every single unit that is under our command with one (or more) XCS, and then keeping the best action from each unit in an object called *combined*.

Some units can be under the control of more than just one XCS algorithm, because they have different types of modus operandi, and, thus, can be covered by more than one game policy (e.g. a worker can be used for harvesting or for attacking). The decision, which XCS should be used at specific time, is made according to the weights from the strategic module. E.g. workers can switch between harvesting and attacking modes, based on which weight is of higher value.

5.3 Representation of Classifier and Sensory Input for Usage in microRTS

To match condition of classifier and environment input σ, they must have the same length and the same representation. They are both built upon the representation of the environment (map). To create the representation of the map, we decided to take a different approach than those seen in the work of Tspanos et al. [8]. We opted to use full map representation to create a binary vector instead of purified information retrieved and abstracted from the map (e.g. the number of friendly workers). This way we do not need crafting of abstraction by hand. Additionally, we also preserve more spatial information presented on the map.

The two-dimensional structure of the map cells was transformed into a one-dimensional vector. The content of each cell is encoded with four bits.The encoding is shown in Tab. 1. For example, if the map contains 8 × 8 cells, a one-dimensional condition of 256 bits - [map] × [4 bits per cell] = 8 × 8 × 4, is created.

Table 1 Map coding table

Name of the environmental element	Representation in bits (4 bits needed)
Base (Friendly)	0 0 0 0
Barracks (Friendly)	0 0 0 1
Worker (Friendly)	0 0 1 0
Light (Friendly)	0 0 1 1
Ranged (Friendly)	0 1 0 0
Heavy (Friendly)	0 1 0 1
Base (Enemy)	0 1 1 0
Barracks (Enemy)	0 1 1 1
Worker (Enemy)	1 0 0 0
Light (Enemy)	1 0 0 1
Ranged (Enemy)	1 0 1 0
Heavy (Enemy)	1 0 1 1
Resource	1 1 0 0
Wall	1 1 0 1
Empty	1 1 1 0
Hidden (for future research, dealing with partial observability)	1 1 1 1

6 CLASSIFIER CORRECTNESS

6.1 Syntactic and Semantic Validations of Classifiers

To make sure that results from classifiers during all phases of XCS operation are valid, and that the XCS cycle would not stop unintentionally, syntactic and semantic validation of classifiers during XCS processing is implemented. Syntactic validation checks if the action can be executed for a specific condition. If the action and condition are not compatible, the classifier is disregarded for future use. Semantic validation is made directly on the active game state. The basic XCS algorithm does not include the points of reference where exactly the syntactic and semantic checking should be performed. For this, we considered the work of Vasilyev [30]. He suggests that the syntactic validity is checked at the stage of classifier formation, and the semantic one is checked directly during the game playtime.

In our implementation, syntactic validations must be made at population set creation, at match set creation (whenever covering operations occur) and at the crossover operations of the GA algorithm. In contrast, the semantic validation is made before the *combined* object is executed on the active game state. The purpose of syntactic validation is to make a basic check if a single unit-action is being legal against the environment. For this we must first create the synthetic game state from the classifier conditions. This state only holds the information about units and structures on their rightful positions, but without the details of what actions they are all presently executing. Second, we check the legality of execution of each classifier's unit-actions by utilising the micro RTS method *isUnitActionAllowed.* For example, the syntax validation will fail if the map contains a unit that wants to move left, but there is a wall present.

Semantic validation checks if the set of unit-actions (player-actions present in the *combined* object) passes execution (through utilisation of microRTS game state API methods) on the deep copy of the game playing game state. Creation of the deep copy is needed so that the original game play state does not become corrupted. Unit-actions that are not allowed are filtered out from the player-actions set that holds them. If there are not enough resources to complete the specific action request (violation of resource usage), the action is also not allowed.

6.2 Managing Classifiers in the RTS Environment

Managing the condition and action of a classifier in the RTS environment is not a trivial procedure. When XCS initialises, it starts with fixed number of classifiers in its set [P]. The condition and action for those classifiers are chosen randomly. The choosing of an action is straight forward. The action is chosen randomly from the list of currently available actions considering the game. On the other hand, the condition of a classifier cannot be created completely at random. The opposite would frequently produce conditions that are not feasible, e.g. a condition where the whole map is made just of the resources. Consequently, classifiers with non-feasible conditions would be constantly removed from the initial [P].

To cope with this problem, we designed our own solution in three steps. First, we made a copy of the original game state. Second, on that copy, we executed a random player-actions set for both players. We must consider actions from both players; otherwise the reward received just by one player would not be correct (i.e. enemy units holding still while our friendly units are performing actions is not realistic). Third, we create the condition of the classifier from the copy of the game state by using the encoding presented in Tab. 1. The result is a set [P] of randomly created classifiers, which are syntactically correct.

In this scenario, there is a possibility that several classifiers with the same condition and action are created in the process. This could lead to stopping of learning due to premature convergence at local optima [31]. This topic will be investigated further in our future work.

7 EXPERIMENT AND RESULTS

The experiment was designed to test our game agent's performance against the other game agents. The experiment was executed using two cases. First, we used a simple battle scenario: A map with the size of 4 × 4 cells with only military units (2 vs. 2 light units). Battles constitute a significant part of the RTS games, and a good performance is desired in such localised matches. Second, we used a full-blown RTS game: A map of the size 8 × 8 cells, comprising the full game configuration. This will help us evaluate our agent better against dynamic opponents during operations in runtime-only game mode (i.e. without prior pre-learning of classifiers).

7.1 Experimental Set-Up

The experiment was carried out on an i7-3770k CPU computer @ 3.50 (turbo: 3.9) GHz, 4 cores (only one core was used for experimentation), 16 GB RAM, OS Windows 10 Pro and Java version 1.8.0. The experiment was set in the latest version of microRTS environment at the time of preparing this article [32]. We set up our experiment following the guidelines from the article [11], where each player starts with one base and one worker. The experiment was executed in a perfect information environment (fully observable and deterministic). Maps used for both scenarios are included inside the microRTS package under the names "melee 4 × 4 light2" and "basesWorkers 8 × 8".

The experiment was executed by using the class *CompareAllAIsObservable*, which is part of the micro RTS *tests* package. We made three changes to that class regarding our experiment: Tests were limited to the two before mentioned maps; game agents PuppetSearchMCTS (utt) and our AIXCS agent were added to the list of the agents to test; the experiment was run only to test our agent against the others. Each experiment was designed to run with AIXCS first for 10 matches against every other agent. Then, we switched the sides for another 10 matches. When players switch their sides, their starting position on the map also changes. Everything else inside the experiment class was left unchanged. There was no transfer learning from one match to another.

The parameters of the experiment were pre-set as follows: Each match was limited to 3000 cycles/frames (with max. inactive cycles set to 300), continuing = true, time = 100, max_actions = 100, max_playouts = −1, playout_time = 100, max_depth = 10 and randomised_ab_repeats = 10.

XCS constants (internal working parameters) were set as follows: N = 83 ([P] size number), α = 0.05 (learning rate for updating ρ), β = 0.01 (learning rate for p, ε and f), δ = 0.1 (specifies the fraction of the mean fitness in [P], below which the fitness of a classifier may be considered in its probability of deletion), θGA = 25 (GA threshold), ε_0 = 10 (error below which classifiers are considered to have equal accuracy), θdel = 20 (deletion threshold), v = 5 (power parameter), χ = 0.5 (crossover probabilities), μ = 0.01 (mutation probability), θsub = 20 (subsumption threshold), P# = 0.33 (probability of using a # in one attribute in condition when covering), pI = εI = fI = 0 (used as initial values in new classifiers), pexp = 0.045 (exploration probability), [A] subsumption and GA subsumption are set to false, and for condition presentation the whole game map was used with all the information included. γ (discount factor) is currently not set and not used, because our XCS algorithm, at this point, does not execute multi-step problems.

7.2 Game Agents in Testing

In the experiment we included the following game agents which were all part of the microRTS package [5, 11, 33]:

- RandomAI: The game agent makes a completely random selection of actions.
- RandomBiasedAI: Based on RandomAI, but the probability of choosing to attack or harvest action is five times higher than that of selecting any other actions.
- LightRush: The agent uses only one worker to mine resources. Gathered resources are then first spent to build one barracks structure. All the rest of the resources are forwarded towards Light combat units' production, which are sent immediately to attack the nearest target.
- RangedRush: Similar to LightRush, except it builds a Ranged combat unit instead of Light combat units.
- WorkerRush: Similar to LightRush and RangeRush, except that the barracks are not needed (the workers are built directly from the base structure).
- PortfolioAI: Uses a collection of four predefined AI's (RandomBiasedAI, LightRush, RangedRush, and WorkerRush) to create combinations of two game agents for a single match. It keeps scores (by utilisation of the evaluation function) of the matches and counts how many times matches occurred with the same set up of two game agents. It returns the action from the AI that obtained the best results (Minimax).
- IDRTMinimax: The game agent uses the Minimax algorithm, but not its standard version. The algorithm that the game agent is using, is not defined by the game agent`s moves, but rather the time. Here, acronym RT stands for Real-Time. The ID acronym represents Iterative-Deepening, because the algorithm uses up all the available time to search in a tree as deeply as possible.
- IDRTMinimaxRandomized: Similar to the previous game agent, but with the special mechanics of the randomised alpha-beta algorithm (the algorithm has better assessment for situations where players execute moves simultaneously).
- IDABCD: Alpha-Beta Considering Duration is similar to RTMinimax, but with modifications that allow for operations like durative moves (more info can be found in the work of Churchill et al. [34]).
- MonteCarlo: A standard Monte Carlo search algorithm.
- MonteCarlo (with max_actions): A standard Monte Carlo search algorithm, but with the limit of only considering a subset of all the possible actions that can be executed in each game state which is of the size max_actions.
- NaiveMCTS (four versions [11]): Standard Monte Carlo search but using Naïve sampling. Four variations (which differ in their initial parameter settings) were used in the experiment (all versions` parameters were set by the game agents` author and were not changed in any way): NaiveMCTS#1 (max_depth = 1, ε_l = 0.33, ε_0 = 0.75), NaiveMCTS#2 (max_depth = 1, ε_l = 1.00, ε_0 = 0.25), NaiveMCTS#3 (max_depth = 10, ε_l = 0.33, ε_0 = 0.75) and NaiveMCTS#4 (max_depth = 10, ε_l = 1.00, ε_0 = 0.25). Note: max_depth set to 1 transforms NaiveMCTS into NaiveMC.
- UCT: A standard real-time UCT algorithm with Upper Confidence Bound 1 (UCB1) sampling policy.
- DownSamplingUCT: Each node in the search tree can only go up to a maximum predefined number of actions, and is therefore not allowed to cover all the available actions at one time (note: Random choosing of actions stays the same

as in the standard UCT, but it only operates until the maximum allowed number of actions is met).

- UCTUnitActions: This search tree speciality is in the fact that it is based on unit-actions rather than on player-actions.
- PuppetSearchMCTS: An adversarial search framework based on scripts that can expose choice points to a look-ahead procedure (for this experiment we used a basic configurable script with a unit type table (utt)). Here, Monte Carlo adversarial search tree was used to search over sequences of puppet moves (which gave the algorithm its name).

7.3 Results

Experiment results carried for the map "melee4 × 4light2" are shown in Tab. 2.

Table 2 Wins / ties / losses of AIXCS against other game agents in the "*melee4 × 4light2*" map

Column of game agents against which AIXCS plays	AIXCS (Wins/ties/losses)
RandomAI	20 / 0 / 0
RandomBiasedAI	19 / 0 / 1
LightRush	10 / 10 / 0
RangedRush	7 / 11 / 2
WorkerRush	6 / 14 / 0
PortfolioAI	7 / 13 / 0
IDRTMinimax	0 / 20 / 0
IDRTMinimaxRandomized	0 / 18 / 2
IDABCD	12 / 5 / 3
MonteCarlo	20 / 0 / 0
MonteCarlo (with max_actions)	20 / 0 / 0
NaiveMCTS#1	0 / 13 / 7
NaiveMCTS#2	1 / 12 / 7
NaiveMCTS#3	3 / 6 / 11
NaiveMCTS#4	5 / 6 / 9
UCT	0 / 11 / 9
DownSamplingUCT	2 / 5 / 13
UCTUnitActions	3 / 4 / 13
PuppetSearchMCTS (utt)	9 / 11 / 0
AIXCS	0 / 20 / 0

Simple scenario results show that, against game agents, RandomAI, RandomBiasedAI, MonteCarlo and MonteCarlo (with max actions), AIXCS show superior performance. Establishing the performance dominance against purely random oriented agents is always a good first step when testing (evolutionary) game agents (i.e. the performance of the tested game agent must be above the category of random methods. Otherwise, there would not be much confidence about the efficiency.

Against game agents: LightRush, RangedRush, WorkerRush, PorfolioAI, IDABCD and basic PuppetSearchMCTS (utt), AIXCS delivers a strong performance, which results in either a win or a tie. This is a clear indication that AIXCS's internal logic of operation gets enough information from one game state for XCS algorithms to compute relevant classifiers, and to choose player-actions that are able to overturn a non-basic game agent.

When battling against agents of IDRTMinimax and IDRTMinimaxRandomized, games end mostly in a tie. Against all variants of NaiveMCTS and UCT, there is a strong tilt towards a tie or a loss. With DownSamplingUCT and UCTUnitActions there is a very high chance of a loss. Ties and losses are most likely due to the XCS algorithms' lack of foresight (i.e. currently they only observe one game state in advance) when connected to the tactic policies.

Experiment results carried for the map »*basesWorkers8* × *8*« are shown in Tab. 3.

Table 3 Wins / ties / losses of AIXCS against other game agents in the "*basesWorkers8* × *8*" map

Column of game agents against which AIXCS plays	AIXCS (Wins/ties/losses)
RandomAI	19 / 1 / 0
RandomBiasedAI	16 / 3 / 1
LightRush	0 / 0 / 20
RangedRush	0 / 0 / 20
WorkerRush	0 / 0 / 20
PortfolioAI	1 / 1 / 18
IDRTMinimax	4 / 16 / 0
IDRTMinimaxRandomized	5 / 15 / 0
IDABCD	7 / 8 / 5
MonteCarlo	13 / 7 / 0
MonteCarlo (with max_actions)	17 / 3 / 0
NaiveMCTS#1	1 / 2 / 17
NaiveMCTS#2	0 / 1 / 19
NaiveMCTS#3	0 / 0 / 20
NaiveMCTS#4	0 / 1 / 19
UCT	0 / 6 / 14
DownSamplingUCT	2 / 10 / 8
UCTUnitActions	0 / 1 / 19
PuppetSearchMCTS (utt)	1 / 0 / 19
AIXCS	3 / 14 / 3

In the second experiment, game agents played a full microRTS game, starting with one base and one worker. Matches against LightRush, RangedRush, WorkerRush and Portfolio (which is mostly a mix of rush AI's), therefore, end up with defeat. The defeat is also against NaiveMCTS variants, PuppetSearchMCTS, UCT and UCTUnitsActions. All of those are MCTS based AI's. Against a basic UCT game agent there are no wins, but it manages to get to a tie in six out of the twenty matches.

Based on the data shown in Tab. 3, AIXCS clearly outperforms both variants of random, MonteCarlo and Minimax game agents. Wins and ties are also favoured towards AIXCS when dealing with IDABCD. We can establish that the good performance against randomly based agents has been preserved when switching from a simple scenario to the full-blown RTS game. The experiment data also show that, although Minimax and IDABCD are tree-based agents, which can traverse over multiple game states, they do not utilise their full potential (i.e the complexity of RTS games is probably too high for them).

The second experiment established that, at this point, AIXCS does not do well when dealing with beginning rush tactics. This was somehow expected, because an 8x8 map is relatively small, and there is little time to counter hard-coded and well-tuned rush tactics. So, without offline processing or deeper traversing through the game states, good enough countermeasures could not evolve that could deal with constant attacks on the nearby units. Consequently, the base is eventually destroyed by rushing units, and our game agent cannot produce new units anymore. We can establish that beginning rushes are at this stage too hard for our XCS algorithms.

After the experiment was completed, we also made some visual observations to see why so many matches against MCTS game agents end up in defeat. We noticed that AIXCS plays the games in a defensive-like style, while

MCTS based AIs are very offensive-oriented. MCTS game agents are also slightly better at combat micromanagement, because AIXCS, from time to time, fails to execute attack action when it should do so. E.g. sometimes a friendly worker, when positioned next to a cell of an enemy worker, fails to attack, even if that is the only logical solution. We must note that AIXCS does not lose the game straight away, but the combination of constant pressure and better micromanagement of MCTS game agents, eventually leads to destruction of the friendly base. Some of the MCTS (e.g. NaiveMCTS#4) gameplay behaviour of pressure and micromanagement looks very similar to the rush tactics.

The research challenge that we identified during the experiment is the need to make the XCS based game agent perform more offensively when facing a (strictly) offensive opponent. Additional experiments designed for performance evaluations against rushing opponents could reveal if a change of XCS parameters` values would change the overall evolutionary behaviour (e.g. switching towards higher exploration); changing strategic policies (e.g. for beginning, middle and end game); could provide enough countermeasures to get through the beginning of the game (if the rush tactic fails, the game agent that started it, is usually in an inferior position); and provide additional data (e.g. what kind of actions classifiers in population propagate when dealing with an attacking opponent) that could maybe offer clues on how to design an XCS algorithm that would cope with the RTS multi-game-state-steps.

8 DISCUSSION

Results are encouraging for simple scenarios, and to some degree for the full RTS game. We anticipated that the results of the second experiment would not be on a par win wise with the first experiment, because of the large number of XCS algorithms that must be executed in parallel in a limited time. With even larger maps there would not be enough processing time for multiple XCS algorithms to converge towards the best game possible. So appropriate scalability analysis is one of our top priorities to investigate in the future (e.g. through parallelisation of operational load across multiple cores). This would provide us with the overall picture of how evolutionary algorithmic components behave when used in the runtime game environment regarding the changing search-space sizes.

The groups of XCS's can operate in runtime mode, but the degree of establishing when and how exactly classifiers form good player-actions in current game state is not yet clear. More research is needed: Room for improvement lies in the optimisation of initial parameters; finding out which parameters have the biggest impact on the game play; analysis of what kind of classifiers formed in sets; establishing when generalisation towards good results starts forming; which new tactical or strategic policies are needed; are current game policies adding or subtracting from the expected behaviour; what kind of condition should classifiers use (full map representation or already extracted environment information); establishing a chain of game states that XCS traverses through; etc.

9 CONCLUSION

This paper has presented a game agent which uses a group of XCS algorithms that form a single player-actions` set as an output result and applied it to the RTS games domain. Five game policies and a strategy module were used to form an internal structure of operation. In its current form, three limitations are included in the design. Limitations imposed are that, currently, there is no chaining of game states while deciding the next best action through simulations (no computation done over multiple frames-also known as direct reward (environment)), the map coding Tab. 1 is currently only suitable for two players, and no algorithmic optimisation was used on many parameters. Because there is no chaining of states, an agent creates a new set of XCS algorithms with every new frame. We use the "One frame at a time" example seen in Monte-Carlo Tree Search (MCTS) algorithms, which are implemented alongside a microRTS engine source code. Every agent gets an allocation of a time slice for every game frame computation, during which it must return the player-actions that it wants to play with.

The reason to impose no chaining of states` limitation was to simplify the initial development of an agent and to better understand agents many dynamic parts:

- each game policy impacts game agents` behaviour,
- XCS uses many internal initialisation parameters and variables,
- XCS algorithms are computationally heavy,
- the rewarding scheme includes many score parameters which are set up through flags,
- XCS cooperation with the Reinforcement and Environment programmes, etc.

Experiments showed that, although a game agent at this stage does not defeat all of the game agents it plays against (e.g. Naïve based game agents), it delivers a good gameplay in full game scenario versus game agents based on random, Monte Carlo, Minimax techniques, and (with more than satisfactory results) also for Alpha-beta, considering duration. Therefore, learning classifier systems are worthy of further research and do show great promise for the game domain [6]. The stepping-stone for the further research was accomplished successfully.

Future work will focus on delayed reward (environment). This will include dealing with a more sophisticated reward system, which would be capable of acquiring the relevant reward and then delivering it over a chain of states. To support such a chain reward system, simulation parts, etc., will need to be reworked. This will include XCS's need to use the parameter for discount factor (γ) of rewards. Here, additional research and testing is needed, because literature shows that temporal-oriented rewards are not suitable for XCS algorithms [35], and that spatial-oriented reward will need to be considered. Other research attention should be given to the XCS being able to operate with a dynamic and changing number of actions in each game state. We also believe that the XCS based game agent shows (great) possibility of operating in a partially observable environment, but additional research is needed in that direction as well.

Acknowledgements

The authors acknowledge the financial support from the Slovenian Research Agency (Research Core Funding No. P2-0057).

10 REFERENCES

[1] Synnaeve, G. & Bessiere, P. (2016). Multi-scale Bayesian modeling for RTS games: An application to StarCraft AI. *IEEE Transactions on Computational intelligence and AI in Games*, *8*(4), 338-350. https://doi.org/10.1109/TCIAIG.2015.2487743

[2] Buro, M. (2003). Real-time strategy games: A new AI research challenge. *IJCAI'2003*, Mexico: Morgan Kaufmann, 1534-1535.

[3] Tang, Z., Shao, K., Zhu, Y., Li, D., Zhao, D., & Huang, T. (2018). A review of computational intelligence for StarCraft AI. *IEEE SSCI*, 1167-1173. https://doi.org/10.1109/SSCI.2018.8628682

[4] Arulkumaran, K., Cully, A., & Togelius, J. (2019). AlphaStar: An evolutionary computation perspective. *arXiv preprint*, arXiv:1902.01724. https://doi.org/10.1145/3319619.3321894

[5] Stanescu, M., Barriga, N. A., Hess, A., & Buro, M. (2016). Evaluating real-time strategy game states using convolutional neural networks. *IEEE CIG*, 1-7. https://doi.org/10.1109/CIG.2016.7860439

[6] Shafi, K. & Abbass, H. A. (2017). A survey of learning classifier systems in games [Review article]. *IEEE Computational Intelligence Magazine*, *12*(1), 42-55. https://doi.org/10.1109/MCI.2016.2627670

[7] Wilson, S. W. (1995). Classifier fitness based on accuracy. *Evolutionary computation*. *3*(2), 149-175. https://doi.org/10.1162/evco.1995.3.2.149

[8] Tspanos, M. T., Chatzidimitriou, K. C. & Mitkas, P. A. (2011). A zeroth-level classifier system for real time strategy games. *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, *2*, 244-247. https://doi.org/10.1109/WI-IAT.2011.177

[9] Wilson, S. W. (1994). ZCS: A Zeroth Level Classifier System. *Evolutionary computation*, *2*(1), 1-18. https://doi.org/10.1162/evco.1994.2.1.1

[10] Rudolph, S., von Mammen, S., Jungbluth, J., & Hähner, J. (2016). Design and evaluation of an extended learning classifier-based starcraft micro ai. *European Conference on the Applications of Evolutionary Computation*, Springer, Cham, 669-681. https://doi.org/10.1007/978-3-319-31204-0_43

[11] Ontañón, S. (2013). The combinatorial multi-armed bandit problem and its application to real-time strategy games. *AIIDE*, 58-64.

[12] Lujan, A. (2009). Generation of rule-based adaptive strategies for games. *PhD Thesis*, Univ of Ottawa, Canada.

[13] Ontañón, S., Barriga, N. A., Silva, C. R., Moraes, R. O., & Lelis, L. H. (2018). The first microrts artificial intelligence competition. *AI Magazine*, *39*(1), 75-83. https://doi.org/10.1609/aimag.v39i1.2777

[14] L. Bull. (2015). A brief history of learning classifier systems: from CS-1 to XCS and its variants. *Evolutionary Intelligence*, *8*(2-3), 55-70. https://doi.org/10.1007/s12065-015-0125-y

[15] Urbanowicz, R. J. & Moore, J. H. (2009). Learning classifier systems: A complete introduction, review, and roadmap. *Journal of Artificial Evolution and Applications*. https://doi.org/10.1155/2009/736398

[16] Holmes, J. H., Lanzi, P. L., Stolzmann, W., & Wilson, S. W. (2002). Learning classifier systems: New models, successful applications. *Information Processing Letters*, *82*(1), 23-30. https://doi.org/10.1016/S0020-0190(01)00283-6

[17] Sutton, R. S. & Barto, A. G. (1998). Reinforcement Learning: An introduction. *MA: MIT Press*, Cambridge. https://doi.org/10.1109/TNN.1998.712192

[18] Butz, M. V., Goldberg, D. E., & Lanzi, P. I. (2003). Gradient descent methods in learning classifier systems: improving XCS performance in multistep problems. *Illigal Report 2003028*, Illinois Genetic Algorithms Laboratory.

[19] Butz, M. V. & Wilson, S. W. (2000). An algorithmic description of XCS. *In International Workshop on Learning Classifier Systems*, Springer, Berlin, Heidelberg, 253-272. https://doi.org/10.1007/3-540-44640-0_15

[20] Watkins, C. J. (1989). Learning from delayed rewards. *Ph.D. Thesis*, Cambridge university.

[21] Lin, L. J. & Mitchell, T. M. (1992). Memory Approaches to Reinforcement Learning in Non-Markovian Domains. Carnegie-Mellon University. Department of Computer Science.

[22] A. Schwartz. (1993). A Reinforcement Learning Method for Maximizing Undiscounted Rewards. *In Proceedings of the tenth international conference on machine learning*, 298, 298-305. https://doi.org/10.1016/B978-1-55860-307-3.50045-9

[23] Watkins, C. J. C. H. & Dayan P. (1992). Machine learning. *Technical Note: Q-Learning*, 8, 279-292. https://doi.org/10.1023/A:1022676722315

[24] Daneshfar, F. (2013). Intelligent load-frequency control in a deregulated environment: continuous-valued input, extended classifier system approach. *IET generation, transmission & distribution*, *7*(6), 551-559. https://doi.org/10.1049/iet-gtd.2012.0478

[25] Butz, M. V., Kovacs, T., Lanzi, P. L., & Wilson, S. W. (2004). Toward a theory of generalization and learning in XCS. *IEEE transactions on evolutionary computation*, *8*(1), 28-46. https://doi.org/10.1109/TEVC.2003.818194

[26] Barriga, N. A., Stanescu, M., & Buro, M. (2018). Game tree search based on non-deterministic action scripts in real-time strategy games. *IEEE Transactions on Games*, *10*(1), 69-77. https://doi.org/10.1109/TCIAIG.2017.2717902

[27] Samvelyan, M., Rashid, T., de Witt, C. S., et. al. (2019). The StarCraft Multi-Agent Challenge. *arXiv preprint*, arXiv:1902.04043v1 [cs.LG].

[28] Ontanón, S. (2016). Informed monte carlo tree search for real-time strategy games. *In 2016 IEEE Conference on CIG*, IEEE, 1-8. https://doi.org/10.1109/CIG.2016.7860394

[29] Saffidine, A., Finnsson, H., & Buro, M. (2012). Alpha-beta pruning for games with simultaneous moves. *Twenty-Sixth AAAI Conference on Artificial Intelligence*.

[30] Vasilyev, A. S. (1999). Classifier systems learning in dynamic environment. *Scientific proceeding of Riga Technical Unniversity 5. serija. Datorzinatne. Information technology and management science*, 5. sejums, 175-187.

[31] Squillero, G. & Tonda, A. (2016). Divergence of character and premature convergence: A survey of methodologies for promoting diversity in evolutionary optimization. *Information Sciences*, *329*, 782-799. https://doi.org/10.1016/j.ins.2015.09.056

[32] Ontañón, S. (2019). Retrieved from: https://github.com/santiontanon/microrts

[33] Ontañón, S. (2019). Retrieved from https://github.com/santiontanon/microrts/wiki/Artificial-Intelligence

[34] Churchill, D., Saffidine, A., & Buro, M. (2012). Fast Heuristic Search for RTS Game Combat Scenarios. *In Eighth Artificial Intelligence and Interactive Digital Entertainment Conference.*

[35] Tang, K. W. & Jarvis, R. A. (2005). Is XCS Suitable For Problems with Temporal Rewards?.*In CIMCA-IAWTIC'06*, *2*, 258-264.

Contact information:

Damijan NOVAK, Teaching Assistant & PhD. student
(Corresponding author)
University of Maribor, Faculty of Electrical Engineering and Computer Science,
Koroška cesta 46, 2000 Maribor, Slovenia
E-mail: damijan.novak@um.si

Domen VERBER, Assistant Professor, PhD
University of Maribor, Faculty of Electrical Engineering and Computer Science,
Koroška cesta 46, 2000 Maribor, Slovenia
E-mail: domen.verber@um.si