Strategy Generation for Multiunit Real-Time Games via Voting

Cleyton Silva, Rubens O. Moraes, Levi H. S. Lelis, and Kobi Gal

***Abstract*—Real-time strategy (RTS) games are a challenging application for artificial intelligence (AI) methods. This is because they involve simultaneous play and adversarial reasoning that is conducted in real time in large state spaces. Many AI methods for playing RTS games rely on hard-coded strategies designed by human experts. The drawback of using such strategies is that they are often unable to adapt to new scenarios during gameplay. The contribution of this paper is a new approach, called strategy creation via voting (SCV), that uses a voting method to generate a large set of novel strategies from existing expert-based ones. Then, SCV uses an opponent modeling scheme during the game to choose which strategy from the generated pool of possibilities to use. By repeatedly choosing which strategy to use, SCV is able to adapt to different scenarios that might arise during the game. We implemented SCV as a bot for μRTS, a recognized RTS testbed. The results of a detailed empirical study show that SCV outperforms all approaches tested in matches played on large maps and is competitive in matches played on smaller maps.**

***Index Terms*—Artificial intelligence, computational intelligence, multiagent systems, machine learning algorithms.**

I. Introduction

REAL-TIME strategy (RTS) games require players to reason in real-time in adversarial scenarios with simultaneous moves and large state spaces [1]. In RTS games the players control units to gather resources, build structures, and battle the opponent. The RTS games we consider can be cast as two-player zero-sum simultaneous-move games, thus, they represent a general class of research scenarios. Also, we focus on games in which there is no fog of war, the only source of imperfect information comes from the simultaneous moves.

RTS typically involve a large action space, which grows exponentially with the number of units controlled by the player, making it challenging to generate strategies in the game without some form of human intervention. The state of the art typically relies on expert-designed hard-coded strategies [1]. Such strategies can include high-level rules such as "if controlling at least 5 combat units, then attack the opponent" and low-level rules such as "do not cause more damage than the necessary to eliminate an enemy unit from the game" to define the actions of all units at any game state. In some cases, the expert strategies are used as a basis for developing novel strategies. For example, Puppet Search (PS) [2] is a method that searches over the parameter space of rules (e.g., the number of units required to attack the opponent in our example) instead of the much larger original action space.

The problem of approaches that rely on hard-coded strategies is that they are predictable, and thus, exploitable by the opponent. Even methods that use search to alter the parameters of a hard-coded strategy such as PS might be exploitable as the rules comprising the strategies do not change, but only the parameters.

The main contribution of this paper is a system called strategy creation via voting (SCV). There are two main steps to SCV. In its first step, SCV generates a large number of strategies from a small pool of existing hard-coded ones. That is, for a given set of existing strategies Σ_I, our voting method is able to generate a strategy for each subset in the power set of Σ_I. Our voting scheme defines a strategy from a subset S of Σ_I by defining the action of each unit according to the majority of the strategies in S. As a result, each unit might behave according to a different strategy in the subset, thus, allowing novel and effective strategies to emerge.

The second step of SCV is a strategy selection step to choose, which strategy to use during game play. This is achieved by defining a set of opponent types and using supervised learning to infer the opponent type during the match. For a given predicted opponent o_t, SCV chooses a strategy that maximizes the player's utility against o_t. SCV adapts to the opponent's behavior by repeatedly choosing which strategy to use next from a large pool of them.

Combining strategy generation with strategy selection during game play allows SCV to adapt to its opponent strategies over time and to avoid being exploited. For example, SCV can use a strategy that requires the player to collect a substantial amount of resources in order to expand its base to a different location on the game's map. Suppose then that SCV infers during the game that the opponent is preparing an attack to the player's current base. SCV's strategy selection scheme allows the algorithm to switch to a strategy that counteracts the opponent's plans. In this case, instead of expanding its base, SCV can choose to build up its defenses, bracing for the incoming attack. This change in strategy is only possible because SCV is able to generate a large and diverse pool of strategies, even when the initial set of strategies is small.

Manuscript received February 17, 2018; revised May 11, 2018; accepted June 13, 2018. Date of publication June 19, 2018; date of current version December 23, 2019. This work was supported by FAPEMIG and CAPES. *(Corresponding author: Levi H. S. Lelis.)*

C. Silva, R. O. Moraes, and L. H. S. Lelis are with the Departamento de Informatica, Universidade Federal de Vicosa, Vicosa 36570-900, Brazil (e-mail: cleyton.rodrigo.silva@gmail.com; rubensolv@gmail.com; levilelis@gmail.com).

K. Gal is with the Department of Software and Information Systems Engineering, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel (e-mail: kobig@bgu.ac.il).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TG.2018.2848913

We evaluate SCV with an extensive empirical analysis on μRTS, an RTS game used in artificial intelligence (AI) competitions. The results show that SCV outperforms expert-based strategies and state-of-the-art methods in matches played in large maps and is competitive in matches played in smaller maps. We also show that SCV requires considerably less computation time than competing schemes. We then present a variant of SCV that uses the remaining time allowed for planning to refine its strategy with heuristic search.

Our approach provides insights for practitioners interested in other multiunit domains. In principle, our voting scheme can be applied to any problem domain in which one controls a group of units (e.g., robots) to jointly solve a task. SCV can be particularly useful in domains for which it is time consuming to obtain expert-designed strategies as our voting method can be an effective alternative to generate a large set of effective strategies from a small set of existing ones.

II. Related Work

This paper relates to two separate lines of research: strategy design for adversarial games and strategy selection methods.

Heuristic search is a common approach to strategy design in RTS games. Algorithm such as variations of Alpha Beta [3], [4] and Monte-Carlo tree search [5], [6] are able to adapt to the opponent's strategy during game play. The problem of search-based strategies is that they do not scale to large games as the branching factor of RTS games grows exponentially with the number of units controlled. For example, while Ontañón [6] uses maps with 8×8 and 12×12 grid cells to test a search-based approach. In this paper, we use maps as small as 8×8 and as large as 128×128 in our experiments.

Search-based approaches tend to be more effective if combined with hard-coded strategies. For example, PS [2] defines a search space over the parameter values of a hard-coded strategy. PS adapts to some extent to the opponent's strategy during the game. However, since PS changes only the parameter values, it is unable to encounter strategies that are fundamentally different from the hard-coded strategy provided as input. Similarly to PS, strategy tactics (STT) [7] also searches in the space of parameter values of a hard-coded strategy. However, STT balances the search over the space of parameters with a search in the actual state-space through the NaiveMCTS search method [6]. Our work differs from these algorithms in that our voting method is able to generate strategies that can be fundamentally different from the hard-coded strategies provided as input. That is, our voting scheme can change not only the parameters used in the rules, but also combine rules from different strategies to form novel ones.

Another line of research focuses on combat scenarios that arise in RTS games. These works have used either learning algorithms [8], [9] or heuristic search and genetic algorithms [5], [10]–[13]. These works differ from ours since they focus on a particular aspect of the game, which are the combats. By contrast, the methods we consider in this paper were designed to play the entire RTS game.

Our work is inspired by Marcolino *et al.* [14], which use a voting scheme to play the game of go. Their system accounts for a set of strategies that vote to decide the player's action. By contrast, we use a voting scheme to decide the action of each unit controlled by the player. Our voting scheme exploits the multiunit structure of the game to generate a large pool of novel strategies. Both our work and Marcolino *et al.*'s are inspired by the "wisdom of the crowds" idea [15], which suggests that combining and aggregating information from groups may increase the quality of the solution as compared to each of the individual solutions.

Our approach relates to works in AI using machine learning to choose among several candidate algorithms for solving optimization problems. We mention some notable examples, referring the reader to a survey by Kotthoff *et al.* [16] for further discussion. Xu *et al.* [17] builds models for predicting the running time of heuristic solvers for SAT problems that are based on linear regression models. Guerri and Milano [18] used a decision tree to choose among several possible algorithms for determining the winner in combinatorial auctions. Other works used reinforcement learning and classification techniques to choose the branching rule to use when solving SAT problems [19] or to switch during the search between different heuristics for solving quantified Boolean formulas [20]. The focus of all of these works is optimization problems with no domain uncertainty. Our setting differs due to the inherently different nature of a multiagent setting in which players outcomes depends on the strategies of others. Ilany and Gal [21] used an algorithm selection approach for determining negotiation strategies, but they did not consider simultaneous games, and assumed a sufficiently large set of existing strategies. Finally, Aha *et al.* [22] used cases to map states to strategies in an RTS game. Our approach is similar to Aha *et al.*'s as it selects a strategy to counteract the strategy it predicts the opponent to be following.

III. Preliminaries

An RTS match can be described as a finite zero-sum two-player simultaneous move game, and be denoted as $(\mathcal{N}, \mathcal{S}, s_{\text{init}}, \mathcal{A}, \mathcal{R}, \mathcal{T})$, where we have the following.

1) $\mathcal{N} = \{i, -i\}$ is the set of *players*.
2) $\mathcal{S} = \mathcal{D} \cup \mathcal{F}$ is the set of *states*, where $\mathcal{D}$ denotes the set of *nonterminal states* and $\mathcal{F}$ the set of *terminal states*. Every state $s \in \mathcal{S}$ includes the joint set of *units* $\mathcal{U}^s = \mathcal{U}^s_i \cup \mathcal{U}^s_{-i}$, for players i and $-i$, respectively. We write $\mathcal{U}$, $\mathcal{U}_i$, and $\mathcal{U}_{-i}$ instead of $\mathcal{U}^s$, $\mathcal{U}^s_i$, and $\mathcal{U}^s_{-i}$ whenever the state is clear from the context.
3) $s_{\text{init}} \in \mathcal{D}$ is the start state of a match.
4) $\mathcal{A} = \mathcal{A}_i \times \mathcal{A}_{-i}$ is the set of joint actions. $\mathcal{A}_i(s)$ is the set of legal *player actions* i can perform at state s. Each action $a \in \mathcal{A}_i(s)$ is denoted by a vector of n *unit actions* $(m_1, \ldots, m_n)$, where $m_k \in a$ is the action of the kth *ready unit* of player i.[1] A unit u is not ready at s if u is performing an action. We denote the set of ready units of players i and $-i$ as $\mathcal{U}^r_i$ and $\mathcal{U}^r_{-i}$. For unit u, we write $a[u]$ to denote the action of u in a.
5) $\mathcal{R}_i : \mathcal{F} \to \mathbb{R}$ is a *utility function* with $\mathcal{R}_i(s) = -\mathcal{R}_{-i}(s)$, for any $s \in \mathcal{F}$, as matches are zero-sum games.
6) $\mathcal{T} : \mathcal{S} \times \mathcal{A}_i \times \mathcal{A}_{-i} \to \mathcal{S}$ is the *transition function*, which determines the sucessor of a state s for a set of joint actions taken at s.

[1] We write "action" instead of "player action" or "unit action" whenever it is clear from the context that we are referring to an action of a player or unit.

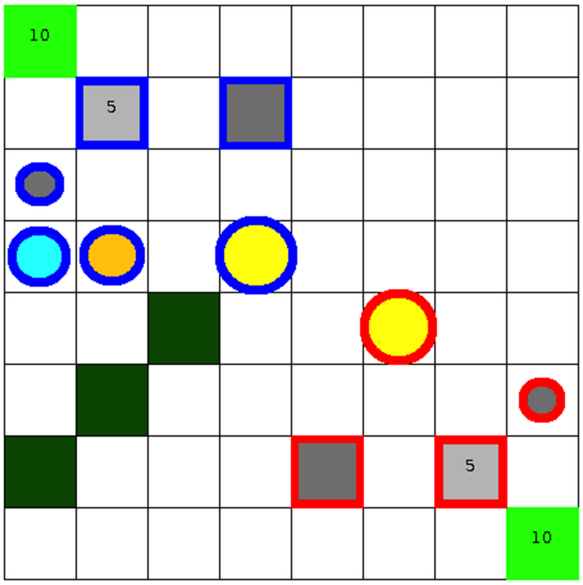

Fig. 1. A μRTS state of a game being played on a 8×8 map.

A *decision point* of player i is a state s in which i has at least one ready unit. A *pure strategy* is a function $\sigma : \mathcal{S} \rightarrow \mathcal{A}_i$ for player i mapping a state s to an action a. Although in general one might have to play a *mixed strategy* to optimize the player's payoffs in simultaneous move games [23], we follow current state-of-the-art RTS methods and consider only pure strategies in this paper [2], [3], [5], [11], [12].

IV. The μRTS Domain

In this section, we detail μRTS, an RTS game developed for research purposes that is actively used as a competition testbed for evaluating and comparing state-of-the-art approaches [6], [24].[2] Fig. 1 shows a screenshot of a μRTS match played on a 8×8 grid map. Each player is given a color (blue or red); units are presented as circles and squares with a contour color of the associated player. The contour of units in Fig. 1 were emphasized to facilitate visualization.

a) Unit types: μRTS has the following types of units: workers, light units, ranged units, heavy units, base, and barracks. The units that can move and attack are represented by a circle, the other units are represented by a square. Small dark-gray circles represent workers, large yellow circles heavy units, small light-blue circles ranged units, and small orange circles light units. Light-gray squares represent bases (the number in the base shows the amount of resources available to the player). The dark-gray squares represent barracks.

b) Map layout: The dark-green squares on the map are walls that cannot be traversed by units. Light-green squares are resources that can be collected by workers (the amount of resources that can be collected is displayed in the square). The layout of walls on the map might influence the strategies chosen by the players. For example, players can choose to build their structures where they are surrounded by walls as a way of protecting their base against attacks of the opponent.

c) Hit points: Every unit has an amount of hit points that indicates the amount of damage the unit can suffer before being eliminated from the game. Workers and ranged units have fewer hit points than light and heavy units, and barracks. The base has more hit points than any other unit. Some of the units can attack an enemy unit. If unit u attacks enemy unit u', then u reduces the hit points of u' according to its inflicted damage, which is determined by the unit's type. Workers and ranged units cause the least amount of damage, light and heavy units cause more damage per attack. Workers, light, and heavy units u can only attack enemy units that are in cells orthogonally adjacent to the grid cell u occupies. Ranged units u can attack any enemy unit that is at an Euclidean distance of three grid cells or less from u.

d) Action scheme: We refer to game cycles as the smallest unit of time in the game. RTS games typically have from ten to 50 game cycles per second [6]. Most of the unit actions require ten game cycles to be executed, but some of the unit actions (e.g., build a base) take much more than ten game cycles to complete (i.e., actions have different durations). Any unit can take a no-op action, which means that the unit waits until the next game cycle. All units other than base and barracks can move one grid cell at a time (up, down, left, and right). Only one unit can occupy a given grid cell at a time. If two units move simultaneously to the same grid cell, then the game server overwrites their actions with the no-op action.

e) Collecting and spending resources: Bases and barracks cannot move nor attack, but the former can train workers and the latter can train light, heavy, and ranged units—all at a cost of resources. Workers can build bases and barracks at the cost of resources. Workers can also collect resources (one unit of resource at a time). Once collected, the resource must be delivered to the base by the worker. Once the collected resource is delivered at the player's base, it can be spent to train other units or build bases and barracks.

f) Utility function: The game uses a utility function introduced by Ontañón [6], which we refer as Ψ in this paper. Ψ computes a score for each player—score(i) and score$(-i)$—by summing up the cost in resources required to train each unit controlled by the player weighted by the square root of the unit's hit points. The Ψ value of a state is given by player i's score minus player $-i$'s score. The Ψ value is then normalized to a value in $[-1, 1]$ through the following formula $\frac{2*\text{score}(i)}{\text{score}(-i)+\text{score}(i)} - 1$.

V. Hard-Coded Strategies

Hard-coded strategies for RTS games are represented as rules that are created by domain experts. The state-of-the-art in RTS games have incorporated expert strategies, whether to play effectively against an opponent, or to use as starting points in a search for good strategies [1], [25]. As an example of a hard-coded strategy, Algorithm 1 describes the strategy known as the worker rush (WR),[3] which receives as input a state s and returns a vector a containing an action for each ready unit. The strategy consists of building a base (if the player does not start with one) and allocating one worker to collect resources. WR constantly trains workers (line 5) and sends them to attack the

[2]https://github.com/santiontanon/microrts/wiki

[3]WR was implemented by S. Ontañón and is available in μRTS codebase.

Algorithm 1: Worker Rush.

Require: a state s with set of ready units $\mathcal{U}_i^r$ and $\mathcal{U}_{-i}^r$.
Ensure: an action vector a for player i.
1: initialize all positions of a with no-op
2: **for** each $u \in \mathcal{U}_i^r$ **do**
3: **if** u is a Base **then**
4: **if** player i has resources to train a Worker **then**
5: $a[u] \leftarrow$ train a Worker
6: **else if** u is a Worker **then**
7: **if** i has no Base and has resources to build a Base **then**
8: $a[u] \leftarrow$ build a Base
9: **else if** i has no Worker harvesting resources **then**
10: $a[u] \leftarrow$ harvest resources
11: **else**
12: $a[u] \leftarrow$ attack enemy closest to u
13: **else if** u is able to move and attack **then**
14: $a[u] \leftarrow$ attack enemy closest to u
15: return a

enemy (line 12). That is, every worker u will attack its closest enemy e. If there is no enemy within u's attack range, the attack action for u means that u will move toward the closest enemy unit hoping that it will be able to attack in the next decision point. WR assigns an attack-the-closest-enemy action to every unit that is able to move and attack and is not a worker (see line 1). It is useful to be able to control such units as s_{init} might contain units which are not trained by the strategy. WR returns the special no-op action for barracks as it initializes all actions in vector a with no-op (see line 1) and it does not alter the action for barracks.

A major drawback of hard-coded strategies is that they are often unable to adapt to the opponent's strategy. For example, WR sends workers to attack the enemy even if they do not stand a chance against the opponent's defense.

VI. SCV Algorithm

We mitigate the lack of adaptability of hard-coded strategies with an algorithm we call SCV. SCV is composed of two steps. In the first step, SCV generates a large pool of strategies from an initial set of hard-coded ones via voting (Section VI-A). In the second step, SCV selects during the match, depending on the opponent's behavior, which strategy to use next (Section VI-B). In Section VI-C, we show how to combine the two steps into the overall SCV.

A. Strategy Creation via Voting

In this section, we describe SCV's voting method, which receives as input a (possibly small) pool Σ_I of strategies and outputs a large set of strategies Σ. Let 2^{Σ_I} be the power set of the strategies in Σ_I. For each subset S in 2^{Σ_I} the voting approach generates a new strategy σ. We generate σ from a subset of strategies S by using a majority voting method. That is, each ready unit u in the current state of the game is assigned the action a that receives the majority of the votes for u in subset S. The rationale behind this scheme is the possibility of creating novel strategies without additional expert knowledge by combining existing ones. Consider the following illustrative example of how our voting scheme generates novel strategies.

Example 6.1: Let $S = \{\sigma_1, \sigma_2, \sigma_3\}$ be a set of strategies and σ be the strategy generated via voting with S. Here, σ_1 trains workers and sends them to attack. σ_2 allocates a worker to build a barracks and then to collect resources. The barracks is then used to train light units. Similarly to what σ_1 does with its workers, σ_2 sends light units to attack the enemy as soon as they are trained. σ_3 also builds a barracks and allocates a single worker to collect resources. However, σ_3 trains ranged units instead of light units. Moreover, the ranged units are positioned around the base, forming a defense line. Depending on its tie-breaking rule, our voting scheme generates through S a novel strategy σ, which trains ranged units which are sent to attack the enemy as soon as they are created. σ emerges because the actions of the base is determined by the majority of the strategies: σ_2 and σ_3, which is to train a single worker; the action of the worker is also determined by σ_2 and σ_3, which is to first build a barracks and then collect resources. There is a draw in the voting of the action of the barracks: σ_1 votes for the no-op action, σ_2 votes for training a light unit, and σ_3 votes for training a ranged unit. If the draw is broken in favor of σ_3, then the barracks would produce a ranged unit whenever possible. The action of the ranged units is determined by σ_1 and σ_2, which is to attack the enemy.

We always break ties by favoring the strategy that comes first in a total ordering of the strategies in S. The total ordering we consider is arbitrary. Although one could use several orderings to obtain an even larger pool of novel strategies, in this paper, we consider a single ordering and show that this approach already generates a large set of useful strategies.

SCV assumes that the strategies in Σ_I follow the same procedure for selecting workers to build structures. For example, if $\Sigma_I = \{\sigma_1, \sigma_2, \sigma_3\}$ σ_1, and σ_2 issue the action "build structure A at position x and y" to two different workers, then the action can be outvoted, despite the majority of the strategies in Σ_I issuing the same build action. SCV also assumes that the strategies in Σ_I choose the same location in which structures are built. For example, if σ_1 issue the action "build A at position x_1 and y_1" and σ_2 issue the action "build A at position x_2 and y_2" to the same worker, then the action can be outvoted if $(x_1, y_1) \neq (x_2, y_2)$, despite the majority of the strategies in Σ_I issuing an action to build A.

SCV requires a minimal size of 3 strategies in S to generate a strategy that is different than the strategies in S. If S contains a single strategy, then the strategy generated through voting will trivially be equal to the strategy in S. If $S = \{\sigma_1, \sigma_2\}$ with the total ordering defining that σ_1 is preferred over σ_2, then for every unit u in a given state s, σ_1, and σ_2 will either agree or disagree with respect to u's action at s. If they agree, then the resulting strategy will be equal to σ_1 and σ_2. If they disagree, the tie will be broken in favor of σ_1 and the strategy output by voting will be equal to σ_1. Thus, SCV considers only the strategies generated from subsets with size in $[3, |\Sigma_I|]$.

We further reduce the number of subsets used by considering subsets of size $[3, N]$ with $N < |\Sigma_I|$. This is because the time required for training a system that uses the strategies generated with our voting scheme increases with the number of strategies generated (as we explain in Section VI-B), and the parameter

N allows one to control the number of strategies generated. Moreover, we prefer to generate strategies from smaller rather than larger subsets because the running time of a voting-based strategy depends on the size of the subset used to generate the strategy. Even simple strategies such as WR runs shortest path algorithms such as A* [26] (e.g., to find the closest enemy unit), which can use a substantial portion of the time allowed for planning, depending on the state. We show in our experiments with μRTS that $N = 4$ allows one to generate effective strategies that use only a small fraction of the time allowed for planning for most of the decision points.

We denote as S_N the set of all subsets of Σ_I with size $[3, N]$ and as Σ_N the strategies generated via voting from the subsets in S_N. Finally, we denote $\Sigma = \Sigma_N \cup \Sigma_I$.

B. Strategy Selection

In order to adapt to different scenarios that occur during the game, SCV uses a strategy selection scheme to choose during the match, which strategy from Σ one should use next. Our strategy selection algorithm accounts for the units the opponent currently controls as well as the layout of the map in which the match takes place. We assume a group of n opponent types $O = \{o_1, o_2, \ldots, o_n\}$ and r maps $M = \{m_1, m_2, \ldots, m_r\}$. We then play each strategy in Σ against each opponent in O in each map in M. We store in a lookup table T the Ψ value of the end-game state of each match played. We store one table T for each map tested.

We used an information gain method [27] to choose which features to use to describe the opponent. Namely, the method chose the following features: number of workers, light, ranged and heavy units, bases, and barracks controlled by the opponent. We also considered other features such as the total amount of resources in the opponent's base, which were eliminated by the information gain method. We record the chosen features for every ten decision points of the game, as well as the opponent type being played while playing all $|\Sigma| \times n \times r$ matches. The collected data is used to train a logistic regression model that, given a set of opponent features, outputs a value p_j between 0 and 1 to each opponent type o_j such that $\sum_{j=1}^{n} p_j = 1$. The p_j values can be interpreted as the probability of the enemy being of type j.

As mentioned in Section IV, the layout of the map plays an important role in the strategy used by the players during the game. That is why in order to select a good strategy from Σ we also account for the map layout. Although one could use richer features, we simply account for the in-game distance, as computed by μRTS's pathfinding system, between the player's base to the opponent base. The map classifier we use is simple: our player assumes it is playing on the training map for which the distance of the bases is the most similar to the distance of the bases on the map of the match in question.

Given the predicted probability distribution over opponent types o_j and the predicted map m_k, SCV selects a strategy from Σ that maximizes the player's expected end-game Ψ value, which is given by

$$\arg\max_{\sigma \in \Sigma} \sum_{j=1}^{n} p_j \cdot T_{m_k}(\sigma, o_j)\,. \tag{1}$$

Algorithm 2: SCV Training.

Require: set of opponent types O, training maps M, basic set of expert strategies Σ_I, and maximum subset size N.
Ensure: set of strategies Σ, opponent type classifier C_o, map classifier C_m, a lookup table T_m for each map m in M.

1: $S_N \leftarrow$ all subsets with size in $[3, N]$ of Σ_I
2: $\Sigma_N \leftarrow$ generate a strategy σ for each subset in S_N via voting.
3: $\Sigma \leftarrow \Sigma_N \cup \Sigma_I$
4: **for** each map m in M **do**
5: **for** each opponent type o in O **do**
6: **for** each strategy σ in Σ **do**
7: $T_m(\sigma, o) \leftarrow \Psi$ value of σ versus o in m.
8: Store opponent features F_o for every 10 decision points of the match of σ versus o; store the opponent type o as the label of F_o; store the distance of the players' bases.
9: train a opponent classifier C_o and a map classifier C_m on the features collected.
10: return Σ, classifiers C_o and C_m, and lookup tables T.

Here, $T_{m_k}(\sigma, o_j)$ is the Ψ value of the strategy σ against o_j for the predicted map m_k. Next, we describe the overall SCV algorithm.

C. Overall SCV Algorithm

In this section, we show how to combine Algorithms 2 and 3 to form the SCV approach. We begin with SCV's training step (Algorithm 2), which requires as input a set of opponent types O (each opponent type is a hard-coded strategy), training maps M, basic strategies Σ_I, and maximum size N of the subsets composing S_N. The algorithm returns a set of strategies Σ, a classifier for opponent type C_o and a classifier for maps C_m, as well as a set of lookup tables T, one for each training map. SCV generates one strategy for each subset of strategies in S_N via voting. We denote as Σ_N the set of strategies thus generated (line 2). We then define Σ as the union of Σ_N and Σ_I (line 3). In its training step, SCV plays each strategy in Σ against each opponent type in every training map. The result of the matches are stored in lookup tables T (see line 2). SCV stores in memory the opponent features F_o as well as the map feature. After all matches are played and their Ψ values stored, SCV trains a classifier C_o for identifying opponent types as well as a map classifier C_m. SCV returns the set Σ of vote-generated strategies, the classifiers, and the lookup tables.

Algorithm 3 uses the components produced in Algorithm 2 to play μRTS matches in real time. Algorithm 3 receives a state s, classifiers C_o, and C_m, window size w, time step t indicating the decision point that s represents in the game, a default strategy σ_d, the strategy σ_{t-1} returned by SCV in the previous time step $(t-1)$, the set of strategies Σ, and the lookup tables T. If s represents a decision point that happens early in the game (i.e., $t < w$), then SCV returns the default strategy σ_d (line 1). The reason SCV uses a default strategy σ_d in the beginning of the game is that both players normally start the game by training at least one worker, thus, making different opponent strategies almost indistinguishable in the beginning of the match. After the wth decision point, SCV starts to use its classifiers to choose

Algorithm 3: SCV.

Require: state s, classifiers C_o and C_m, window size w, time step t, default strategy σ_d and strategy returned in previous time step σ_{t-1}, pool of strategies Σ, lookup tables T.
Ensure: strategy from Σ.
1: **if** $t < w$ **then**
2: return σ_d
3: **if** $t \mod w = 0$ **then**
4: $T_m \leftarrow C_m(s)$
5: $\vec{p} \leftarrow C_o(s)$
6: return $\arg\max_{\sigma \in \Sigma} \sum_{j=1}^{n} p_j \cdot T_m(\sigma, o_j)$
7: return σ_{t-1}

a strategy that maximizes the expected end-game value (line 6). SCV classifies the opponent and map only in time steps that are multiples of w (see line 3).

In order to choose a strategy that maximizes the expected end-game Ψ value, SCV uses the map classifier C_m to select the most similar map m to the map used in the match in question (line 4). Map m indicates the lookup table T_m to be used. SCV also uses the opponent type classifier to obtain a probability distribution $\vec{p}$ over the possible opponent types (line 5). Once T_m and $\vec{p}$ are defined, SCV returns the strategy in Σ that maximizes the expected end-game Ψ value according to (1) (line 6). If $t \geq w$ and $t \mod w \neq 0$, then SCV returns σ_{t-1}, the strategy used in $t-1$.

As we show in Section VII, SCV often uses only a small fraction of the time allowed for planning to choose a strategy and then an action for the player to perform. We show that the remaining time allowed for planning can be used to refine the strategies generated by SCV, by introducing a variant of SCV we call SCV+. Similarly to how STT refines its own strategy [7], SCV+ refines a strategy selected by Algorithm 3 with a variant of Alpha Beta called ABCD [3]. The ABCD search is performed as follows. For every state s representing a decision point, we verify if s includes units engaged in combat (i.e., units with enemy units within attack range). If that is the case, then we create a partial state of the game state that contains only the player's and the opponent's units engaged in combat. We run ABCD for this partial state for the remaining time allowed for planning and use the actions, thus, encountered to control the units in the partial state; all other units act according to the strategy select by Algorithm 3 as usual. Note that this strategy refinement step could be performed with other search algorithms such as NaiveMCTS [6].

VII. Empirical Evaluation

We compare SCV and SCV+ to several competing schemes: the hard-coded strategies used to compose Σ_I, a baseline we call strategy basic selection (SBS) (Section VII-D), and state-of-the-art methods for the game of μRTS (Section VII-F).

A. Empirical Methodology

In our experiments all actions are deterministic and both players can see all units on the map, the only source of imperfect information comes from the simultaneous moves of the game. The player who eliminates the enemy's units is the winner of the match. Each player is allowed 100 ms for planning in every game cycle. In all experiments each algorithm plays 20 times against every other algorithm in each map tested. To ensure fairness, the players switch their starting location an even number of times on the maps tested. For example, for a map with starting locations A and B, Algorithm 1 starts in A with Algorithm 2 starting in B for ten matches; we then switch the starting positions for the remaining ten matches. All experiments are run on 2.6 GHz CPUs.

In order to test, if the strategies generated through our voting scheme are effective, we compare SCV with a baseline called SBS. The only difference between SCV and SBS is that the former uses voting to generate a large pool strategies Σ to select from during the match, while the latter selects strategies only from the initial strategies Σ_I. SCV and SCV+ use an arbitrary total ordering to break the ties in the voting method.

1) Maps: We use a large diversity of maps in our experiments. Namely, we use maps of size $x \times x$ with $x \in \{8, 16, 32, 64, 128\}$. Following the diversity of initial states used in μRTS's latest competition [24], we used maps in which each player starts with 1 base and 1 worker, 1 base, 1 worker and 1 barracks, 1 base, 2 barracks and 1 worker, and 4 bases and 4 workers. Every match is limited by a number of game cycles and the match is considered a draw once the limit is reached. We present the number of matches won by each algorithm, the matches finishing in draws are not accounted for in our tables and figures of results. The maximum number of game cycles is map dependent. We use the limits defined by Barriga *et al.* [2]: 3000, 4000, 5000, 6000, 8000, and 12000 game cycles for maps of size 8, 16, 32, 64, and 128, respectively. We train the SCV approaches and SBS on a set of maps and test them on a different set. Namely, for training, we use three maps of size 64, eight maps of size 128, and four maps of the remaining sizes. We use five maps of each size for testing.

2) Basic Strategies: The set of basic strategies Σ_I is composed of the following eight hard-coded strategies: worker defense (WD), WR, ranged defense (RD), ranged rush (RR), heavy rush (HR), light defense (LD), light rush (LR), and military rush (MR). These strategies are described in the appendix of this paper. Since we use all combinations of subsets of size 3 and 4 in addition to the basic strategies, our set Σ has a total of $\binom{8}{3} + \binom{8}{4} + 8 = 134$ strategies. We used the following hard-coded strategies as opponent types in our experiment: WR, LR, RR, HR, and MR. These strategies cover several styles of play that arise in the μRTS game. The default strategy for each map is determined by the strategy that obtained the largest average Ψ value during training.

As an example of the training running time, each match on a 128×128 map lasts for approximately 12 min, thus, it takes approximately $134 \times 5 \times 8 \times 12 = 64\,320$ min of CPU time to run SCV training step while considering the 134 strategies, five opponent types, and eight training maps. Although time consuming, this step is highly parallelizable as all the $134 \times 5 \times 8 = 5360$ matches can be run in parallel.

3) Competing Schemes: We test SCV against the following state-of-the-art methods: Adversarial hierarchical task network (AHT) [28], an algorithm that uses Monte Carlo tree search

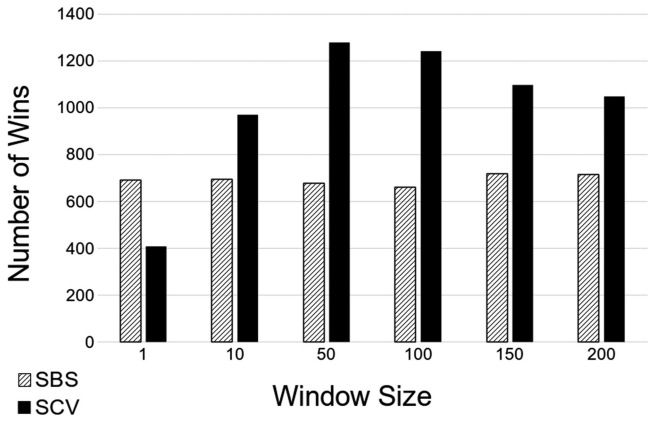

Fig. 2. Comparison of SBS and SCV with various w values. The maximum number of wins is 1 440.

and HTN planning; NaiveMCTS [6] (henceforth referred as Naïve), an algorithm based on combinatorial multiarmed bandit algorithms; an extension of Monte Carlo algorithms for dealing with partial observability called BS3 [29]; the MCTS version of PS and STT mentioned in Section II. All approaches except STT are used in their default settings, as provided in μRTS's codebase. STT is used as provided in STT's codebase.[4]

B. Evaluation of Logistic Regression

The logistic regression obtained a 90.83% accuracy in a 10-fold cross validation over our training data. The training data is composed of data collected from $134 \times 2 \times 5 \times 23 = 30\,820$ matches (134 strategies playing twice each of the five opponent types on the 23 training maps); we play each strategy twice against each opponent type on each map because we switch the starting location of the players to ensure fairness. Since the number of opponent types is five, a random classifier is expected to be 20% accurate on average, which suggests that our system is able to accurately detect the opponent type.

C. Evaluation of Window Size

We start our study by testing both SBS and SCV with different window sizes. In theory, search-based algorithms such as AHT, Naïve, BS3, PS, and STT can adapt themselves during gameplay to the strategy being used by their opponent. The window size w gives SBS and SCV similar capability. In every w decision points SBS and SCV can reassess their strategy and possibly switch to a different one.

We test SBS and SCV with various window sizes on the maps used for training. The y-axis in Fig. 2 denotes the total number of wins of each approach. The x-axis denotes the different approaches, where SBS(10) and SCV(10) denote SBS and SCV with $w = 10$. The method that performs best is SCV(50), which wins approximately 1300 matches against the different versions of SBS and SCV. The method that performed worst was SCV(1), which wins approximately 400 matches against the other versions SCV and SBS.

Different w values do not affect much the number of wins of SBS—all SBS variants win approximately 700 matches. By contrast, there is a large difference in the number of wins of SCV for different values of w. This is because SCV's voting scheme generates strategies that are more effective than others depending on the game scenario. In contrast with SBS, SCV does not have a dominant strategy in its pool of options and it often chooses to switch strategies.

The result for SCV(1) contrasts with the intuition that reassessing, which strategy to use in every decision point of the game is more advantageous to SCV as a frequent verification scheme should allow the algorithm to adapt its strategies according to the opponent's behavior in a timely manner. A possible explanation for this result is the fact that we use an accurate but imperfect classifier, thus, posing a tradeoff between strategy selection accuracy and strategy adaptability. Calling the classifier more often (smaller w values) increases the chances of incorrectly classifying the opponent type at least once during a match, which can reduce a player's chances of winning the match. This is because classification mistakes of the opponent type could make SCV switch back and forth between competing objectives, thus, making the SCV's overall strategy less effective. For example, if a player wrongfully chooses to build an extra barracks instead of training heavy units, this decision can reduce the chances of the player winning the match as the resources that should have been spent training units was spent building an extra barracks. On the other hand, for larger w values, the classifier is called less often and SCV has a smaller chance of incorrectly identifying the opponent type at least once, but it also has fewer opportunities to adapt to the opponent's strategy or to recover from early misclassifications. Our results show $w = 50$ offers a good balance between selection accuracy and strategy adaptability.

Another possible reason for SCV(1) performing poorly is the training data including only Ψ values of matches played between a strategy and an opponent for the entire match. That is, the training data does not account for scenarios in which one switches strategies during the match. As an example of how this can be a problem, SCV might start following RD, which requires the player to train ranged units and place them around the base. The classifier might eventually detect that the opponent is following a strategy σ that is effective against RD and that HR is the best strategy against σ. What SCV knows is that HR is effective against σ if played from the beginning to the end of the match. SCV uses this knowledge as an approximation for how well HR will behave after the match is played for a number of time steps with RD. SCV with smaller w values will be more affected by the errors of such approximations as it considers switching strategies more often than SCV with larger w values. In order to deal with this problem we would have to account for strategy transitions during training, which would substantially increase the computational cost of SCV's training step.

We use SBS(150) and SCV(50) in all other experiments of this paper, as these are the w-values that the algorithms performed best in the training maps.

D. SCV versus Basic Strategies

Once we have defined that SBS performs best with $w = 150$ and SCV with $w = 50$, we compare them with the basic strategies in Σ_I. Fig. 3 shows the winning rate on the test maps.

[4]https://github.com/nbarriga/microRTSbot

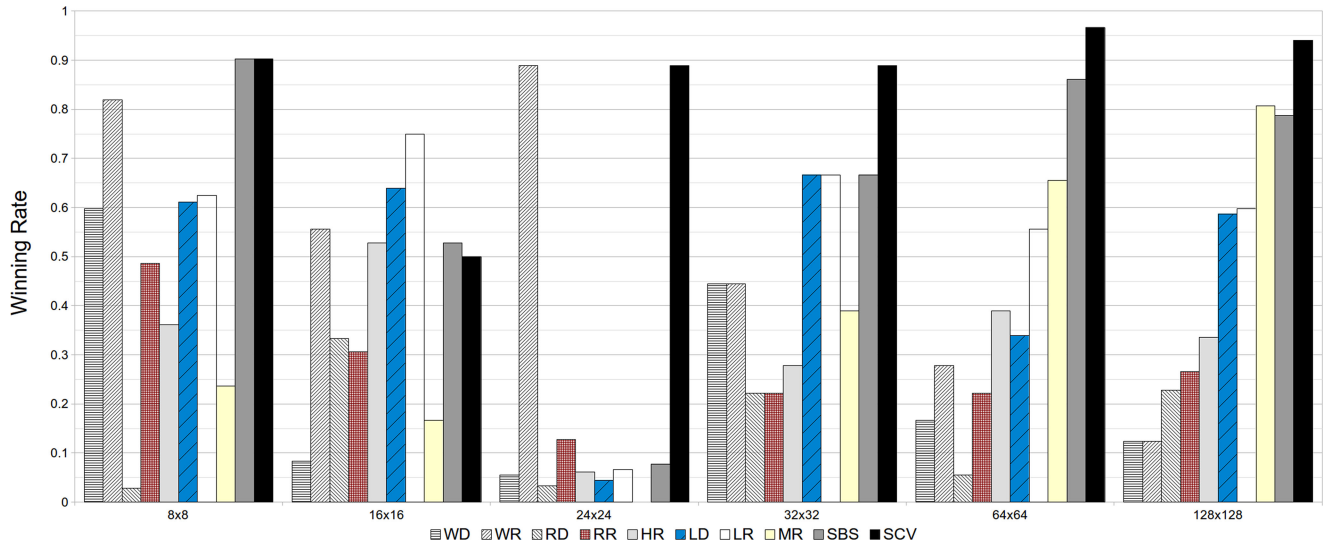

Fig. 3. SBS and SCV against the basic strategies.

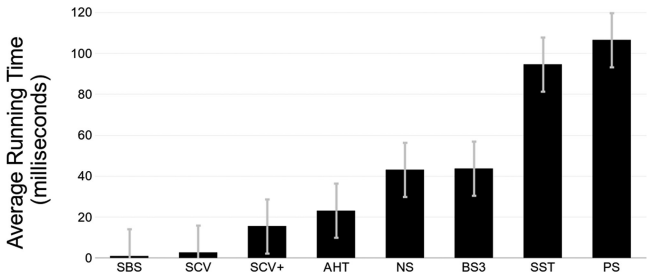

Fig. 4. Average running time in milliseconds of several approaches.

Overall SCV has a much larger winning rate than SBS and the other strategies. However, depending on the map size, some of the strategies from Σ_I present a large winning rate. For example, WR is competitive with SCV in maps of size 24 $\times$ 24 and LR is the best performing strategy in maps of size 16 $\times$ 16. We also observed that, for maps of size 128 $\times$ 128, MR is a dominant strategy in Σ_I and SBS almost always plays according to MR. This explains why the numbers of both MR and SBS are very similar for maps of that size. By contrast, SCV does not have a dominant strategy and is able to adapt to different scenarios that arise during the matches. These results suggest that SCV's voting scheme is able to successfully generate effective strategies by combining the basic strategies from Σ_I.

E. Evaluation of Running Time

We compare the running time of the algorithms used in our comparison of SCV and SCV+ with other approaches in all training maps. The average running time per game cycle in milliseconds and the standard deviations are shown in Fig. 4. In this comparison, we only account for SCV's and SCV+'s running time during the game. This is because, we assume that the training time is amortized over a large number of matches. Both SBS and SCV use a much smaller fraction of the time allowed for planning than the other approaches tested. SCV+ is 5.8 times slower than SCV, but it still does not use all the time available for planning, as in many decision points no units are engaged in combat and the SCV approaches are only invoked in game cycles with ready units.

F. SCV Versus State-of-the-Art Methods

In this section, we evaluate SCV and SCV+ against current state-of-the-art methods for μRTS. Table I presents the number of wins of the row player against the column player. Since there are five test maps and every method plays against each other 20 times in every map, the maximum number of wins is 100. We highlight the cells in which a method wins more matches than another. For example, in maps of size 8 $\times$ 8 SCV+ wins 65 and loses 34 matches against Naïve, thus, we highlight the cell containing the number 65. We also highlight the cell with the largest number of total victories for a given map size.

SCV+ is only outperformed in terms of total number of victories by STT in maps of size 16 $\times$ 16. Nonetheless, SCV+ has more victories against STT in matches played in maps of that size. Namely, SCV+ won 53 and lost 46 matches against STT. STT outperforms SCV+ in maps of size 24 $\times$ 24 and 32 $\times$ 32. As one increases the map size, the strategies encountered through SCV's voting system seem to be more effective than STT's strategies. This is because in maps of size 128 $\times$ 128 SCV+ defeated STT in 89 matches and lost only in one match.

SCV+ outperforms SCV and SBS in all settings tested, suggesting that both the set of strategies generated via voting and the heuristic search approach to refine the selected strategy contribute to improve the system's overall performance.

The methods that do not use hard-coded strategies perform poorly in larger maps. That is, Naïve, BS3, and AHT lost all their matches against the methods that rely on hard-coded strategies (STT, PS, SBS, SCV, and SCV+) in maps of size 128 $\times$ 128. Naïve and BS3 perform so poorly in maps of that size that they draw all matches they play against themselves as they are unable to encounter each other on the map and all matches finish by reaching the 1200th game cycle. Expert-based strategies

TABLE I
SCV AND SCV+ AGAINST STATE-OF-THE-ART METHODS

Maps 8 × 8

	NS	BS3	AHT	STT	PS	SBS	SCV	SCV+	Total
NS	-	45	90	54	93	30	31	34	377
BS3	48	-	88	55	93	35	29	28	376
AHT	9	11	-	6	73	28	26	25	178
STT	40	38	93	-	85	27	29	28	340
PS	6	7	26	12	-	20	20	20	111
SBS	66	64	72	71	80	-	40	40	433
SCV	38	69	74	69	80	40	-	0	370
SCV+	65	70	75	72	80	40	100	-	502

Maps 16 × 16

	NS	BS3	AHT	STT	PS	SBS	SCV	SCV+	Total
NS	-	26	93	7	14	15	17	27	199
BS3	31	-	89	8	16	20	23	29	216
AHT	7	11	-	13	95	14	11	20	171
STT	88	85	87	-	93	41	37	46	477
PS	81	82	3	6	-	10	10	10	202
SBS	48	45	43	34	60	-	20	20	270
SCV	75	71	89	41	90	30	-	30	426
SCV+	71	69	83	53	90	40	40	-	446

Maps 24 × 24

	NS	BS3	AHT	STT	PS	SBS	SCV	SCV+	Total
NS	-	9	92	0	0	0	0	0	101
BS3	10	-	89	2	0	0	0	0	101
AHT	8	11	-	1	75	68	50	45	258
STT	99	95	99	-	54	10	49	53	459
PS	98	100	16	33	-	23	23	26	319
SBS	97	99	12	76	59	-	20	20	383
SCV	98	97	48	45	71	70	-	16	445
SCV+	96	97	55	37	73	70	63	-	491

Maps 32 × 32

	NS	BS3	AHT	STT	PS	SBS	SCV	SCV+	Total
NS	-	1	55	0	2	0	3	1	62
BS3	1	-	70	0	1	0	0	4	76
AHT	30	22	-	0	14	11	10	10	97
STT	100	100	100	-	75	21	58	58	512
PS	98	98	86	25	-	23	40	19	389
SBS	100	100	89	79	77	-	50	27	522
SCV	87	92	88	41	57	50	-	4	419
SCV+	84	90	90	40	81	72	91	-	548

Maps 64 × 64

	NS	BS3	AHT	STT	PS	SBS	SCV	SCV+	Total
NS	-	0	0	0	0	0	0	0	0
BS3	0	-	0	0	0	0	0	0	0
AHT	93	96	-	0	11	10	7	7	224
STT	100	100	100	-	35	45	30	41	451
PS	100	100	89	64	-	82	60	16	511
SBS	98	100	90	51	18	-	19	0	376
SCV	92	91	93	63	36	55	-	4	434
SCV+	92	93	92	53	79	89	90	-	588

Maps 128 × 128

	NS	BS3	AHT	STT	PS	SBS	SCV	SCV+	Total
NS	-	0	0	0	0	0	0	0	0
BS3	0	-	0	0	0	0	0	0	0
AHT	0	0	-	0	0	0	0	0	0
STT	60	53	97	-	3	30	0	1	224
PS	99	100	100	97	-	64	38	27	525
SBS	100	100	100	69	35	-	10	0	414
SCV	97	94	95	94	54	90	-	35	559
SCV+	98	95	94	89	69	100	62	-	607

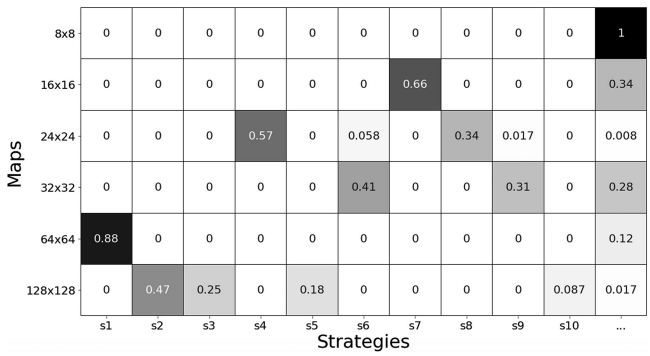

Fig. 5. Distribution of strategies chosen by SCV during matches played on maps of different sizes.

provide domain-specific information that allows one to guide their search towards more effective strategies.

G. Strategies Used in Matches

Fig. 5 shows how often different strategies are chosen in matches played in maps of different sizes. Strategies $s1$ through $s10$ are the ten strategies chosen more often by SCV in any given map. The number inside each cell shows the fraction of times a given strategy is chosen for a fixed map size; the numbers are rounded up to either two or three decimal places. For example, $s1$ is the strategy returned by SCV in 88% of the times the algorithm is invoked in maps of size 64 × 64. The column "..." shows the number corresponding to the sum of the fractions of the remaining strategies. Strategies $s1$ through $s10$ are all generated by SCV's voting scheme, which suggests that the strategies SCV generates through voting can be more effective than the strategies created by domain experts. This is because SCV's strategies are preferred by its selection method over the strategies created by domain experts. Fig. 5 also shows that several strategies can be effective for a given map size, which suggests that SCV's selection scheme allows the system to adapt to different scenarios that arise during a match.

VIII. CONCLUSION AND FUTURE WORK

In this paper we introduced SCV, a system that generates through voting a large pool of novel strategies from a small set of existing ones. In principle, SCV's voting scheme can be applied to any problem domain in which one controls a group of units (e.g., robots) to jointly solve a task. SCV can be particularly useful in domains for which it is time consuming to obtain expert-designed strategies as it can generate a large set of strategies from a small set of expert-based ones. In addition to a voting scheme, SCV uses a strategy selection system to adapt to the opponent's strategy during the game. An empirical study with μRTS highlighted the importance of expert-designed strategies for algorithms playing large-scale RTS matches. The methods that do not use hard-coded strategies were substantially outperformed in larger maps by methods that rely partially or fully on hard-coded strategies.

We also showed empirically that SCV uses on average only a small fraction of the time allowed for planning, which allows one to enhance SCV with a heuristic search algorithm to refine its selected strategy; we called the resulting algorithm SCV+. Our empirical study also showed that SCV is able to generate effective strategies with its voting scheme as SCV+ outper-

formed the current state-of-the-art methods in larger maps and was competitive in smaller maps.

Although we used a strategy selection method to showcase the potential of the set of strategies generated by voting, SCV's voting-based strategies could be used to enhance other algorithms. For example, PS could search in the parameter space of a strategy generated by SCV.

APPENDIX A
HARD-CODED STRATEGIES

RR, HR, and LR were introduced by Ontañón [30], WD, RD, LD, and MR are introduced in this paper.

a) RR, HR, and LR: These strategies train a worker to collect resources and to build a barracks as soon as possible. Then, each of the rush strategies train specific units: RR trains ranged units, HR trains heavy units, and LR trains light units. Once trained, these units are sent to attack the nearest enemy. If the number of workers is reduced to zero, the base trains another worker, which is allocated to collect resources.

b) Worker Defense: This strategy assigns the first worker to collect resources while the base trains more workers. The workers stand at a distance from their base equal to the height of the map divided by two, forming a defense line. If an enemy unit e gets within a distance of also the height of the map divided by two from the workers, all units are sent to attack e.

c) RD and LD: These strategies behave similarly to WD, with the difference being that the defense line is formed by ranged and lights units for RD and LD, respectively. Since RD and LD train ranged and lights units, they build a barracks with their worker as soon as there is enough resources; the worker returns to collecting resources once the barracks is built.

d) MR: This strategy trains six workers. The first five workers collect resources, while the last worker is sent to build a base near the second closest set of resources from the player's base. Once the base is constructed, it trains five other workers, which also collect resources. Next, one of the workers builds a barracks nearby the first base. The barracks then repeatedly trains: a light, a ranged, and a heavy unit. These units are immediately sent to attack the nearest enemy unit.

ACKNOWLEDGMENT

The authors would like to thank S. Ontañón for organizing the 2017 μRTS competition and for providing support to μRTS's codebase. They would also like thank the anonymous reviewers for great suggestions.

REFERENCES

[1] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A survey of real-time strategy game AI research and competition in StarCraft," *IEEE Trans. Comput. Intell. AI Games*, vol. 5, no. 4, pp. 293–311, Dec. 2013.

[2] N. A. Barriga, M. Stanescu, and M. Buro, "Game tree search based on non-deterministic action scripts in real-time strategy games," *IEEE Trans. Comput. Intell. AI Games*, vol. 10, no. 1, pp. 69–77, Mar. 2018.

[3] D. Churchill, A. Saffidine, and M. Buro, "Fast heuristic search for RTS game combat scenarios." in *Proc. AAAI Conf. Artif. Intell. Interactive Digit. Entertain.*, 2012, pp. 112–117.

[4] A. Saffidine, H. Finnsson, and M. Buro, "Alpha-beta pruning for games with simultaneous moves," in *Proc. Conf. Artif. Intell.*, 2012, pp. 556–562.

[5] D. Churchill and M. Buro, "Portfolio greedy search and simulation for large-scale combat in StarCraft." in *Proc. Conf. Comput. Intell. Games*, 2013, pp. 1–8.

[6] S. Ontañón, "Combinatorial multi-armed bandits for real-time strategy games," *J. Artif. Intell. Res.*, vol. 58, pp. 665–702, 2017.

[7] N. A. Barriga, M. Stanescu, and M. Buro, "Combining strategic learning and tactical search in real-time strategy games," in *Proc. 13th Annu. AAAI Conf. Artif. Intell. Interactive Digit. Entertain.*, 2017, pp. 9–15.

[8] N. Usunier, G. Synnaeve, Z. Lin, and S. Chintala, "Episodic exploration for deep deterministic policies: An application to StarCraft micromanagement tasks," arXiv:1609.02993, 2016.

[9] S. Liu, S. J. Louis, and C. A. Ballinger, "Evolving effective microbehaviors in real-time strategy games," *IEEE Trans. Comput. Intell. AI Games*, vol. 8, no. 4, pp. 351–362, Dec. 2016.

[10] N. Justesen, B. Tillman, J. Togelius, and S. Risi, "Script- and cluster-based UCT for StarCraft," in *Proc. IEEE Conf. Comput. Intell. Games*, 2014, pp. 1–8.

[11] C. Wang, P. Chen, Y. Li, C. Holmgård, and J. Togelius, "Portfolio online evolution in StarCraft," in *Proc. Conf. Artif. Intell. Interactive Digit. Entertain.*, 2016, pp. 114–120.

[12] L. H. S. Lelis, "Stratified strategy selection for unit control in real-time strategy games," in *Proc. Int. Joint Conf. Artif. Intell.*, 2017, pp. 3735–3741.

[13] R. O. Moraes and L. H. S. Lelis, "Asymmetric action abstractions for multi-unit control in adversarial real-time scenarios," in *Proc. 32nd AAAI Conf. Artif. Intell.*, 2018, pp. 876–883.

[14] L. S. Marcolino, H. Xu, A. X. Jiang, M. Tambe, and E. Bowring, "Give a hard problem to a diverse team: Exploring large action spaces," in *Proc. AAAI Conf. Artif. Intell.*, 2014, pp. 1485–1491.

[15] J. Surowiecki, *The Wisdom of Crowds*. New York, NY, USA: Anchor, 2005.

[16] L. Kotthoff, I. P. Gent, and I. Miguel, "An evaluation of machine learning in algorithm selection for search problems," *AI Commun.*, vol. 25, no. 3, pp. 257–270, 2012.

[17] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown, "Satzilla: portfolio-based algorithm selection for sat," *J. Artif. Intell. Res.*, vol. 32, no. 1, pp. 565–606, 2008.

[18] A. Guerri and M. Milano, "Learning techniques for automatic algorithm portfolio selection," in *Proc. 16th Eur. Conf. Artif. Intell.*, vol. 16, 2004, pp. 475–479.

[19] M. Lagoudakis and M. Littman, "Algorithm selection using reinforcement learning," in *Proc. 7th Int. Conf. Mach. Learn.*, 2000, vol. 29, pp. 511–518.

[20] H. Samulowitz and R. Memisevic, "Learning to solve QBF," in *Proc. Nat. Conf. Artif. Intell.*, 2007, vol. 22, no. 1, pp. 255–260.

[21] L. Ilany and Y. Gal, "Algorithm selection in bilateral negotiation," *Autonom. Agents Multi-Agent Syst.*, vol. 30, no. 4, pp. 697–723, 2016.

[22] D. W. Aha, M. Molineaux, and M. J. V. Ponsen, "Learning to win: Case-based plan selection in a real-time strategy game," in *Proc. 6th Int. Conf., Case-Based Reasoning*, 2005, pp. 5–20.

[23] H. Gintis, *Game Theory Evolving: A Problem-centered Introduction to Modeling Strategic Behavior* (ser. Economics/Princeton University Press) Princeton, NJ, USA: Princeton Univ. Press, 2000.

[24] S. Ontañón, N. A. Barriga, C. R. Silva, R. O. Moraes, and L. H. S. Lelis, "The first microrts artificial intelligence competition," *AI Mag.*, vol. 39, no. 1, pp. 75–83, 2018.

[25] D. Churchill and M. Buro, "Hierarchical portfolio search: Prismata's robust AI architecture for games with large search spaces," in *Proc. AAAI Conf. Artif. Intell. Interactive Digit. Entertain.*, 2015, pp. 16–22.

[26] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, Jul. 1968.

[27] M. A. Hall and G. Holmes, "Benchmarking attribute selection techniques for discrete class data mining," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 6, pp. 1437–1447, Nov./Dec. 2003.

[28] S. Ontañón and M. Buro, "Adversarial hierarchical-task network planning for complex real-time games," in *Proc. Int. Joint Conf. Artif. Intell.*, 2015, pp. 1652–1658.

[29] A. Uriarte and S. Ontanón, "Single believe state generation for partially observable real-time strategy games," in *Proc. IEEE Conf. Comput. Intell. Games*, 2017, pp. 296–303.

[30] S. Ontañón, "The combinatorial multi-armed bandit problem and its application to real-time strategy games," in *Proc. AAAI Conf. Artif. Intell. Interactive Digit. Entertain.*, 2013, pp. 58–64.