Modeling Unit Classes as Agents in Real-Time Strategy Games

Ulit Jaidee and Héctor Muñoz-Avila

Department of Computer Science & Engineering; Lehigh University; Bethlehem, PA 18015
{ulj208, munoz}@lehigh.edu

Abstract

We present CLASS QL, a multi-agent model for playing real-time strategy games, where learning and control of our own team’s units is decentralized; each agent uses its own reinforcement learning process to learn and control units of the same class. Coordination between these agents occurs as a result of a common reward function shared by all agents and synergistic relations in a carefully crafted state and action model for each class. We present results of CLASS QL against the built-in AI in a variety of maps using the Wargus real-time strategy game.

Introduction

Real-time strategy (RTS) games are a kind of games where players asynchronously create, manage, maneuver and maintain armies of units to defeat an opponent. Creating automated AI players is a difficult task that requires managing different, inter-related tasks. To address this difficult task, a body of work points to the need to have components in charge of the following managerial tasks (Scott, 2002):

1. **Building.** In charge of structure building, including keeping track of the order in which buildings must be constructed (e.g., a keep requires barracks).
2. **Units.** Responsible of creating new units and prioritizing the order in which units are created.
3. **Research.** Responsible for creating new technology that enables the creation of more powerful units.
4. **Resource.** Responsible for gathering resources (i.e., wood and gold), which is needed to construct units, buildings and invest on research.
5. **Combat.** Responsible for controlling units during combat; determining which units to attack, or whether to defend.
6. **Civilization.** Responsible for coordinating the managers.

While substantial progress has been made on AI players over the years, particularly in the AIIDE RTS competition, where entries have increasing sophistication levels (Buro & Churchill, 2012), it is noticeable that no entries used machine learning techniques to control the managerial tasks.

One of the main difficulties for applying machine learning algorithms is the large search space; indeed, for strategy games (even without taking into account real-time considerations), the search space has been shown to be exponential on factors such as the size of the map (e.g., the number of tiles in the map) and the number of units controlled (Madeira et al., 2006; Aha et al., 2005).

To address this problem, we developed CLASS QL, a multi-agent model for playing real-time strategy games. Each agent models only part of the state and is responsible for a subset of the actions; thereby significantly reducing the memory requirement for learning. CLASS QL’s agents learn and act while performing the tasks performed by the managers discussed before. However, CLASS QL’s doesn’t implement these managers. Instead, each agent is responsible for learning and controlling one class of units (e.g., all footmen) or one class of buildings (e.g., barracks). There is no coordinator agent; coordination between these agents occurs as a result of a common reward function shared by all agents and synergistic relations in a carefully crafted state and action model for each class. Our experiments demonstrate that CLASS QL can achieve a good performance level.

Related Work

There is a substantial body of work for learning in RTS games. Table 1 categorizes research on learning systems for RTS games according to the managerial tasks.

Many of the works described are capable of playing the complete RTS games and hence perform the 6 managerial tasks. But learning in those works is limited to some of these tasks and not all of them. For example, Weber et al. (2012) reports on a system that plays full RTS using the managers indicating above but only the unit and building manager is using learning. Hence, we classify it on category B in Table 1. Other works in this category includes Aha et al. (2005) which uses case-based reasoning techniques to retrieve a plan that executes a building order. The same is true for the work of Hsieh and Sun (2008), whose systems analyzes game replays to determine suitable unit and building creation orders. Also included in this
category is the work by Dereszynski (2011) which learns a probabilistic model.

**Table 1: Categories of works versus managerial tasks**

<table>
<thead>
<tr>
<th>Build</th>
<th>Unit</th>
<th>Research</th>
<th>Resource</th>
<th>Combat</th>
<th>Civ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>B</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>D</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>F</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Category A belongs to works that perform learning in combat tasks. Included in this category are works by Sharma et al. (2007) which combines case-based reasoning and reinforcement learning, Wender and Watson (2012), which uses reinforcement learning, Weber and Mateas (2009) uses data mining techniques including k-NN and logitBoost to extract opponent models from game replays of annotated traces. Othman et al. (2012) use evolutionary computation to control combat tactics such as indicating which opponent’s unit to attack. It plays the AI against itself to speed-up learning.

Works in category C not only use learning techniques for unit and building creation tasks but also use learning for research tasks. Synnaeve and Bessière (2011) model this learning problem as a Bayesian model. Ponsen et al. (2006) uses a technique called dynamic scripting (Spronck, 2006) to control unit and building creation and research. A script is a sequence of gaming actions specifically targeted towards a game such as in this case Wargus. Scripts are learned by combining reinforcement learning and evolutionary computation techniques.

Category E belongs to works that uses learning for resource gathering tasks. Included in this category is work by Marthi et al. (2005), which uses concurrent ALISP in Wargus games. The basic premise of that work is the user specifying a high-level LISP program to accomplish Wargus tasks and reinforcement learning is used to tune the parameters of the program.

Young and Hawes (2012) use evolutionary learning to manage conflicts that arise between conflicting goals, which can be resource gathering as well as for unit and building creation (Category D). Their focus on goal management is in line with an increasing interest on the general topic of goal-driven autonomy as it pertains to RTS games (Scott, 2002; Rørmark, 2009; Perez, 2011; Synnaeve & Bessière, 2011). In micro-management the complex problem of playing an RTS game is divided into tasks. These tasks are accomplished by specialized components or agents. This is the principled follow by the 6 managers and similar architectures in implementations of RTS games (Scott, 2002; Hagelbäck and Johansson (2008) divides control through multiple (non-learning) agents. Each agent controls regions of interest for combat tasks.

**The CLASSQL Algorithm**

The multi-agent CLASSQL manages a collection of learning agents, one for every class of unit/buildings. Each CLASSQL’s agent performs a feedback loop with the environment (in this case the game Wargus), which is typical of reinforcement learning agents (Figure 1). In each iteration, the agents extract information from the state and using the reward signal from the environment, determine the actions that units under their control need to achieve. These actions are high-level (e.g., attack enemy workers) and translated into multiple game-level actions.

**CLASSQL** uses the following conventions:

- The set of classes $\mathcal{C}$ is the set of $n$ classes $\{C_1, C_2, ..., C_n\}$, where each $C_i$ is the $i^{th}$ class of units in the game, such as footmen, knights, and peasants.
- The set of class-actions $\mathcal{A}$ is the set of $\{A_1, A_2, ..., A_n\}$, where $A_i$ is the set of actions that the units in the $i^{th}$ class can perform.
- The set of class-states $\mathcal{S}$ is the set of $\{S_1, S_2, ..., S_n\}$, where each $S_i$ is the set of states that the units in the $i^{th}$ class can be at.
- The set of learning-matrices $\mathcal{M}$ is the set of $\{M_1, M_2, ..., M_n\}$, where $M_i$ is the learning-matrix of the $i^{th}$ class. Learning-matrix is a data structure that is
used to store data by a selected learning method. For example, as in our implementation of CLASSQL, if Q-learning is used as the learning method, then each $M_i$ is a Q-table $Q_i(s,a)$ indicating the estimated value of action-value function of taking action $a$ in state $s$. Our implementation uses the standard Q-learning update.

**Q-learning update.** The estimated value of action-value function of taking action $a$ in state $s$ at the time $t^{th}$ is denoted as $Q(s_t, a_t)$. Q-learning is defined by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Where $r_{t+1}$ is the reward observed, the parameter $\alpha$ controls the learning rate ($0 < \alpha \leq 1$), and $\gamma$ is a factor discounting the rewards obtained so far ($0 \leq \gamma < 1$).

CLASSQL$(C, A, M) =$

1. **while** episode continues
2. **parallel for** each class $C \in C$
3. $s \leftarrow \text{OBSERVESTATE}()$
4. $r \leftarrow \text{OBSERVEREWARD}()$
5. $s \leftarrow \text{GETSTATEFORCLASS}(s, C)$
6. $A \leftarrow \text{GETACTIONSFORCLASS}(A, C)$
7. $M \leftarrow \text{GETLEARNINGMATRIXFORCLASS}(M, C)$
8. $L \leftarrow \{\}$
9. **for** each unit $u \in C$
10. **if** $u$ is a new unit then
11.    $s_u \leftarrow s; a_u \leftarrow \text{idle-action}$
12. **else** if unit $u$ is idle or finished its action **then**
13.    $M \leftarrow \text{UPDATE}(M, s_u, a_u, s, r)$
14.    $a \leftarrow \text{GETACTION}(A, M)$
15.    $L \leftarrow \text{Append}(L, \{u, a\})$
16.    $s_u \leftarrow s; a_u \leftarrow a$
17. **end if**
18. **end for**
19. **return** $M$

CLASSQL receives as inputs: a set of classes $C$, a set of class-actions $A$, and a set of Q-tables $M$. During an episode(Line 1), for each class $C$ working in parallel(Line 2) will execute Lines 3 to 17; so Lines 3 to 17 are executed independently for each agent of class $C$. Our implementation runs one thread for each CLASSQL agent.

**Loop of each CLASSQL agent.** The state and reward are observed directly from the environment (Lines 3-4). Then, the observed state $s$ is filtered specifically for class $C$ (Line 5). This filtering is necessary because different classes need different kinds of information. Moreover, the benefit of doing this is to help reducing the size of the observed state (we analyze the size reduction in the next section). Next, the set of actions $A$ of class $C$ is retrieved from the set $A$ (Line 6). Afterwards, the Q-table $M$ of class $C$ is retrieved (Line 7). The set of units’ action $L$ is initialized as an empty list (Line 8). For each unit $u$ of class $C$ (Line 9), if the unit $u$ is just created, then its previous state $s_u$ is initialized to the current state $s$ and its previous action $a_u$ is initialized to the idle-action (Lines 10-11). If the unit $u$ is idle or finished its action (Line 12), the Q-table is updated using the Q-learning update (Line 13). Then, an action $a$ is chosen from the set $A$ (Line 14). A pair of unit and its action is appended to the list $L$ (Line 15). Afterward, the previous state $s_u$ and previous action $a_u$ of unit $u$ are saved for the future use (Line 16). In the last step we execute the actions from the list $L$ (Line 17).

After the episode is over, the set of Q-tables $M$ is returned (Line 18).

**Analysis of CLASSQL**

As explained before each CLASSQL agent $i$ maintains its own Q-table $M_i$ for all units of class $C_i$. Where $M_i : S \times A_i \rightarrow [0,1]$. Agent $i$ controls all units of class $C_i$. Assume a greedy policy $\pi_i$ extracted from each $M_i$ and for each $s \in S_i$:

$$\pi_i(s) = \text{ArgMax}_{a \in A_i} [M_i(s, a)]$$

That is, $\pi_i(s)$ picks the action $a$ that has the maximum value $M_i(s, a)$ ($\pi_i$ is often referred to as the greedy policy). In Line 14, each agent will typically pick the same action as the greedy policy most of the time (i.e., with some high probability $1 - \varepsilon$, where $\varepsilon$ is probability of random action in $\varepsilon$-greedy policy). However to guarantee that optimal policies are learned, it will pick a random action with a probability $\varepsilon$ (this is commonly known as $\varepsilon$–greedy action selection).

Assume that each agent $i$ has learned an optimal policy $\pi_i$. That is, when following the policy $\pi_i$, it maximizes the expected return for agent $j$, where the return is a function of the rewards obtained. For example, the return can be defined as the summation of the future rewards until the episode ends. It is easy to prove that, given a collection of $n$ independent policies $\pi_1, \ldots, \pi_n$ where each $\pi_i$ maximizes the returns for class $k$, then $\pi = (\pi_1, \ldots, \pi_n)$ is an optimal policy in $S \times (A_1 \times A_2 \times \ldots \times A_n)$ (where $S = \cup_{1 \leq k \leq n} S_k$ as defined in the previous section). This means that agents will coordinate despite the fact that each agent is learning independently. Admittedly, this assumption is not valid in many situations in RTS games since, for example, the agent barracks might produce an archer thereby consuming the resources needed for the peasant-builder to build a lumber mill. Nevertheless this represents an ideal condition that guarantees coordination.

For non-ideal (and usual) conditions, we observe coordination between agents. Figure 2 shows a typical timeline at the beginning of the game after CLASSQL has learned for several iterations. Games begin with a town hall and a peasant. The peasant agent orders the peasant to build a farm. The town hall agent orders the town hall to produce a second peasant. The peasant agent orders the second peasant to build a barracks and then orders the first peasant...
to mine gold (after it has finished the farm). Coordination emerges between the agents as a result of the reinforcement learning process; the decision by the town hall agent to create the second peasant is learned because it enables the peasant agent to order this peasant to produce the barracks. Bad orderings such as building the barracks prior to the second peasant will result in game lost (because the economy will be stagnant) and therefore over time discarded.

![Figure 2: timeline of units’ actions](image)

The CLASSQL agents require \(|S_1 \times A_1| + \cdots + |S_n \times A_n|\) space (i.e., adding the memory requirements of each individual agent \(a_k\)). In contrast, an agent reasoning with the combined states and actions would require \(|S \times A|\) space. Under the assumption that \(\forall (i,j)[i \neq j \rightarrow A_i \cap A_j = \{\} \land S_i \cap S_j = \{\}]\) hold, then the following inequality holds:

\[
|S \times A| \geq |S_1 \times A_1| + \cdots + |S_n \times A_n|
\]

for \(n \geq 2\), the expression on the right is substantially lower than the expression on the left. The action disjunction assumption is common in RTS games because the actions that a unit of a certain class can take are typically disjoint from the actions of units of other classes.

**Modeling Wargus in CLASSQL**

CLASSQL maintains 12 agents including unit-producing agents such as the Town Hall or the Barracks agents, combat agents such as knights and archers, research agents such as Church (e.g., allows to research the upgrade to convert knights into paladins, a stronger version of the knight) and the peasant agent which can gather resources (e.g., harvest gold form a mine) or create new buildings. In Table 2, we show the 12 agents we created in CLASSQL and the managerial tasks they performed. Missing in the table is the civilization managerial task because as explained in the previous section this is attained implicitly.

The reward function we used in our implementation is the difference between (A) the Wargus score of our team and (B) the Wargus score of the opponent. The Wargus score is the weighted summation of enemy units killed and enemy buildings destroyed. The weight reflects the cost of a unit (e.g., a knight has more weight than a peasant because the knight is more expensive to produce).

**Table 2: CLASSQL’s agents versus managerial tasks**

<table>
<thead>
<tr>
<th>Build</th>
<th>Unit</th>
<th>Research</th>
<th>Resource</th>
<th>Combat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Town Hall</td>
<td>✔️</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blacksmith</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lumber Mill</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Church</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barrack</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knight</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Foot</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Archer</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ballista</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gryphon Rider</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gryphon Aviary</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peasant</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Empirical Evaluation**

We conducted experiments for CLASSQL on small, medium, and large Wargus maps with the fog-of-war mode turned off.

5.1 Experimental Setup

At the first turn of each game, both teams start with only one peasant and one town hall. We have five adversaries: land-attack, SR, KR, SC1 and SC2. These adversaries are used for training our algorit hm. For testing it we use the built-in AI in Wargus. Learning only occurs while playing against the adversaries; when playing versus the built-in AI, learning is turned off. These adversaries come with the Warcraft distribution and have been used in machine learning experiments before (e.g., (Ponsen et al., 2006; Jaidee et al., 2011)). The build-in AI is capable of defeating average players and is a stronger player than the 5 adversaries used for training.

The five adversaries can construct any type of unit unless the strategy followed discards it (e.g., land-attack will only construct land units. So units such as gryphons are not built):

- **Land-Attack**: This strategy tries to balance between offensive/defensive actions and research. It builds only land units.
- **Soldier’s Rush (SR)**: It attempts to overwhelm the opponent with cheap military units in an early state of the game.
- **Knight’s Rush (KR)**: It attempts to quickly advance technologically, launching large offences as soon as
knights are available. Knight is the strongest unit in the game.

ClassQL is trained with the set of adversaries \{land-attack, SR, KR, SC1, SC2\} and then tested versus the built-in Wargus’ AI. We conducted experiments on the small map with size 32×32 tiles, the medium map with size 64×64 tiles and the large map with size 128×128 tiles. The details of the landscape of each map are shown in Figure 4. Each team starts in one side of the map. At the beginning, each team has one town hall and one peasant with a gold mine near the camp.

Our performance metric is:

\[
\text{Performance} = (\text{number of wins} - \text{number of loses}) \quad (1)
\]

In our experiments we first set CLASSQL against the built-in AI with no training \((m = 0)\). Then we play against each team from the set of adversaries and again test ClassQL versus the built-in AI. We repeat this until \(m = 50\). We repeat each match 10 times and compute the average metric.

Our performance metric provides a better metric than the difference in Wargus score (our score – opponent’s score) of the game because the lower score difference can mean a better performance than a larger score difference. This is due to how the Wargus score is computed. For example, our team can win the opponent very fast and the score we got is just 1735 and the game is over while the opponent got the score of 235 before the game end. In this case, the average score of (our team - opponent team) is just 1500. In another case, our team can win the opponent with the score of 3450, but the game takes very long time to run until the game is over; while the opponent team got the score of 1250. In this case, the average score of (our team – opponent team) is 2200, but it does not mean our team’s performance is better. In fact, its performance should be worse than the previous case because it takes longer time to win.

5.2 Results

Figure 5 shows the results of the experiments. The x-axis is the number \(m\) of training iterations and the y-axis is the performance metric (see Equation 1). In all three maps, the build-in AI starts winning, which is not surprising since CLASSQL has no training. After a few iterations CLASSQL begins to outperform the built-in AI and continues to do so for the remaining iterations.

We also conducted experiments, where we tested the AI learned by CLASSQL on one map after \(m = 50\) iterations and tested it against the built-in AI in other two unseen maps without any additional training. We repeated this experiment for the AI learned in the small, medium and large maps. Figure 4 shows the original large map used for learning and the two other large maps we used for testing. The results are shown in Table 3. In the original map used
for training (column labeled 1st map), CLASSQl is able to win almost all of the 10 games. The knowledge learned is effective in the other 2 maps (columns labeled 2nd map and 3rd map).

Table 3: The results of transfer learning.

<table>
<thead>
<tr>
<th>map size</th>
<th>1st map</th>
<th>2nd map</th>
<th>3rd map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Medium</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Large</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Conclusions

We presented CLASSQl, a multi-agent model for playing real-time strategy games. Each agent learns and controls units/buildings of the same class. Therefore each agent only uses part of the state information and reasons only with the actions that units of its class can perform. As a result, each agent needs to learn on a much reduced space compared to an agent that reasons with the combined states and actions. CLASSQl acts and learns on the high-level managerial tasks need to play effectively RTS games. Coordination between the CLASSQl agents occurs as a result of a common reward function shared by all agents and synergistic relations in a carefully crafted state and action model for each class.

For future work we will like to formulate more interesting conditions under which we can guarantee that the CLASSQl agents will coordinate to better understand the positive empirical results we obtained. We will also like to test CLASSQl with Starcraft, which is a more complex RTS game than Wargus having a larger state and actions space.

Acknowledgements. This work was supported in part by NSF grant 1217888.

References


