Case-Based Goal-Driven Coordination of Multiple Learning Agents

Ulit Jaidee[1], Héctor Muñoz-Avila[1], & David W. Aha[2]

[1]Department of Computer Science & Engineering; Lehigh University; Bethlehem, PA 18015
[2]Navy Center for Applied Research in AI; Naval Research Laboratory (Code 5514);
Washington, DC 20375
{ulj208,munoz}@lehigh.edu | david.aha@nrl.navy.mil

Abstract. Although several recent studies have been published on goal reasoning (i.e., the study of agents that can self-select their goals), none have focused on the task of learning and acting on large state and action spaces. We introduce GDA-C, a case-based goal reasoning algorithm that divides the state and action space among cooperating learning agents. Cooperation between agents emerges because (1) they share a common reward function and (2) GDA-C formulates the goal that each agent needs to achieve. We claim that its case-based approach for goal formulation is critical to the agents' performance. To test this claim we conducted an empirical study using the Wargus RTS environment, where we found that GDA-C outperforms its non-GDA ablation.

Keywords: Goal-driven autonomy, case-based reasoning, multi-agent systems

1 Introduction

Goal reasoning is the study of introspective agents that can reason about what goals they should dynamically pursue (Klenk *et al.*, in press). Goal-driven autonomy (GDA) (Muñoz-Avila *et al.*, 2010; Molineaux *et al.*, 2010) is a model of goal reasoning in which an agent revises its goals by reasoning about discrepancies it encounters during plan execution monitoring (i.e., when its expectations are not met) and their explanation.

GDA agents have not been designed to learn and act with large state and action spaces. This can be a problem when applying them to real-time strategy (RTS) games, which are characterized by large state and action spaces. In these games, agents control multiple kinds of units and structures, each with the ability to perform certain actions in certain states, while competing versus an opponent who is controlling his own units and structures. To date, GDA agents that learn to play RTS games can be applied to only limited scenarios (e.g., Jaidee *et al.*, 2011) or control only a small set of decision-making tasks within a larger hard-coded system that plays the full game (e.g., (Weber *et al.*, 2012)).

To address this limitation, we introduce GDA-C, a partial GDA agent (i.e., it implements only two of GDA's four steps) that divides the state and action space among multiple reinforcement learning (RL) agents, each of which acts and learns in the environment. Each RL agent performs decision making for all the units with a

common set of actions. For example, in an RTS game, it will assign one RL agent to control all footmen, which is a melee combat unit, and another RL agent to control the barracks, which is a building that produces units (e.g., footmen).

That is, each RL agent α_k is responsible for learning and reasoning on a space of size $|S_k|\ |\mathcal{A}_k|$, where S_k is agent α_k's set of states and $\mathcal{A}_k$ is its set of actions. Thus, GDA-C's overall memory requirement, assuming n RL agents, is $|S_1||\mathcal{A}_1| + \ldots + |S_n||\mathcal{A}_n|$. This is a substantial reduction in memory requirements compared to a system that must reason with a space of size $|S||\mathcal{A}|$, where $S = \bigcup_{1 \le i < n} S_i$ and $\mathcal{A} = \bigcup_{1 \le i < n} \mathcal{A}_i$ (i.e., all combinations of states and actions).

Cooperation among GDA-C's agents emerges as a result of combining two factors: (1) all its agents share a common reward function and (2) it uses case-based reasoning (CBR) techniques to acquire/retain and reuse/apply its goal formulation knowledge.

We claim that agents which share the same reward function, augmented with coordination provided by GDA-C, outperform agents that coordinate by sharing only the reward function. To test this claim we conducted an empirical evaluation using the Wargus RTS environment in which we compared the performance of GDA-C versus CLASS_{QL} (Jaidee *et al.,* 2012), an ablation of GDA-C where the RL agents coordinate by sharing only the same reward function. We first compared GDA-C and CLASS_{QL} indirectly by testing both against the built-in AI in Wargus, a proficient AI that comes with the game and is designed to be competitive versus a mid-range player. We also compared their performance in direct competitions. Our main findings are:

- Versus the Wargus built-in AI, GDA-C outperformed CLASS_{QL}
- GDA-C also outperformed CLASS_{QL} in most direct comparisons

Our paper continues as follows. In Section 2 we describe related work, and then present a formalization of the problem we are studying in Section 3. Section 4 discusses the RL agents and Section 5 presents the GDA-C algorithm. Section 6 discusses the states and actions defined in Wargus, while Section 7 presents the empirical evaluation. Finally, Section 8 concludes with future work suggestions.

2 Related Work

Weber et al. (2012) report on EISBot, a system that can play a complete RTS game. EISBot plays complete games by using six managers (e.g., for building an economy, combat), only one of which uses GDA (i.e., it selects which units to produce). The GDA system GRL (Jaidee *et al*., 2012) plays RTS game scenarios were each side starts with a fixed number of units. No buildings are allowed and hence no new units can be produced, which drastically reduces the GRL's state and action space. In contrast to these and other GDA systems that play RTS games (e.g., Weber *et al.*, 2010), GDA-C controls most aspects of an RTS game by assigning units and buildings of the same type to a specialized agent.

Many GDA systems manage expectations that are predicted outcomes from the agent's actions. Most work on GDA assumes deterministic expectations (i.e., the same outcome occurs when actions are taken in the same state). These expectations are computed in a number of ways. Cox (2007) generates instances of expectations by

using a given model of abstract explanation patterns. Molineaux et al. (2011) use planning operators to define expectations. Borrowing ideas from Weber et al. (2012), GDA-C uses vectors of numerical features to represent the states and expects that actions will increase their values (e.g., sample features include total gold generated or number of units, both of which a player would like to increase). When this does not happen (i.e., when this constraint is violated), a *discrepancy* occurs.

When most GDA algorithms detect a discrepancy between an observed and an expected state, they formulate new goals in response. Some systems use rule-based reasoning to select a new goal (Cox, 2007), while others rank goals in a priority list and use truth–maintenance techniques to connect discrepancies with new goals to pursue (Molineaux *et al.*, 2010). Interactive techniques have also been used to elicit new goals from a user (Powell *et al.*, 2011). GDA-C instead learns to rank goals by using RL techniques based on the performance of the individual agents.

GDA-C has some characteristics in common with GRL (Jaidee *et al.*, 2012), which also uses RL for goal formulation. However, GRL is a single agent system and, unlike GDA-C, cannot scale to play complete RTS games.[1]

3 Multi-Agent Setting

The task we focus on is to control a set Γ of agents $\alpha_1, \ldots, \alpha_n$, where each belongs to one class c_k in $C = \{c_1, c_2, \ldots, c_n\}$. Each class c_k has its own set of class-specific states S_k. The collection of all states is denoted by S (i.e., $S = \bigcup_{1 \le k \le n} S_k$). Each agent α_k can execute actions in $\mathcal{A}_k$ for every class specific state.

A stochastic policy is a mapping $\pi_k: S_k \rightarrow \{(a, p) | a \in \mathcal{A}_k, p \in [0,1]\}$. That is, for every state $s \in S_k$, $\pi_k(s)$ defines a distribution $\{(a_1, p_1), \ldots, (a_n, p_n)\}$, where a_i is an action in $\mathcal{A}_k$ and p_i is the expected return from taking action a_i in state s and following policy π_k thereafter. The return is a function of the rewards obtained. For example, the return can be defined as the summation of the future rewards. Our goal is to find an optimal policy $\pi_k^*: S_k \rightarrow \{(a, p) | a \in \mathcal{A}_k, p \in [0,1]\}$ such that π_k^* maximizes the expected return.

It is easy to prove that, given a collection of n independent policies $\pi_1, \ldots, \pi_n$ where each π_k maximizes the returns for class k, then $\pi = (\pi_1, \ldots, \pi_n)$ is an optimal policy. As we will see in Section 4, GDA-C uses this fact by running n RL agents, one for each class c_k. If each converges to an optimal policy, their n-tuple policies will be an optimal policy for the overall problem. This results in a substantial reduction of the memory requirement compared to a conventional RL agent that is attempting to learn a combined optimal policy $\pi^* = (\pi_1, \ldots, \pi_n)$ where each π_i must reason on all states and actions. This conventional RL agent will require $|S| \times |\mathcal{A}|$ space, where $S = \bigcup_{1 \le i < n} S_i$ and $\mathcal{A} = \bigcup_{1 \le i < n} \mathcal{A}_i$ (i.e., counting all combinations of state n-tuples times all combinations of n-tuple actions).

[1] This means that the player starts with limited resources, units, and structures but can (1) harvest additional resources, (2) build any structure, (3) train any unit, (4) research any technology, and (5) control the units to defeat an opponent.

In contrast the n agents $\alpha_1, \ldots, \alpha_n$ each attempt to learn an optimal policy π^*_k, which requires $|S_1 \times \mathcal{A}_1| + \ldots + |S_{n-1} \times \mathcal{A}_n|$ space (i.e., adding the memory requirements of each individual agent α_k). The following inequality holds:

$$|S \times \mathcal{A}| \geq |S_1 \times \mathcal{A}_1| + \ldots + |S_{n-1} \times \mathcal{A}_n|,$$

assuming that $\forall_{i,j}\,(i \neq j)\ (\mathcal{A}_i \cap \mathcal{A}_j = \{\} \wedge S_i \cap S_j = \{\})$. This is common in RTS games where the actions that a unit of a certain type can take are disjoint from the actions of units of a different type. Under these assumptions, and for $n \geq 2$, the expression on the right is substantially lower than the expression on the left. For example, assuming $\forall_k$ $|S_k| = t$ and $|\mathcal{A}_k| = m$, then the LHS is equal to $(n \times t \times n \times m)$ whereas the RHS is equal to $(n \times t \times m)$. That is, the space saved is $(1 - \frac{1}{n}) \times 100\%$. The following table summarizes some of the savings for these assumptions:

Table 1: Space saved by GDA-C compared to a conventional RL agent

n	% of saved space	n	% of saved space	n	% of saved space
1	0	4	75	10	90
2	50	5	80	20	95

In our work, we use Q-learning (Sutton and Barto, 1998) to control each of the α_k agents. Thus, our baseline system consists of n Q-learning agents that are guaranteed, after a number of iterations, to converge to an optimal policy. We refer to this baseline system as CLASS_{QL} because each Q-learning (QL) agent controls a class of units in Wargus.

4 Case Bases and Information Flow in the GDA-C Agent

We now discuss how case-based reasoning techniques are used in GDA-C to manage goals on top of CLASS_{QL}. Figure 1 depicts a high-level view of the information flow in GDA-C, which embeds the standard RL model (Sutton and Barto, 1998). GDA-C has two threads that execute in parallel. First, the GDA thread selects a goal, which in turn determines the policy that each RL agent will use and refine. Second, the CLASS_{QL} thread performs Q-learning to control each of the α_k agents.

The two case bases, *Policies* and *GFCB*, are learned from previous instances (e.g., previously played Wargus games). Given a policy $\boldsymbol{\pi}$, a *trajectory* is a sequence of states $< \boldsymbol{s_0}, \ldots, \boldsymbol{s_m} >$ visited when following $\boldsymbol{\pi}$ from the starting state $\boldsymbol{s_0}$. Any such state in this trajectory is a goal that can be achieved by executing $\boldsymbol{\pi}$. The policy is assigned the last state in a trajectory as its goal. The case base ***Policies*** is a collection of pairs $(g, \boldsymbol{\pi}_g)$, where $\boldsymbol{\pi}_g$ is a policy that should be used when pursuing goal g. GDA-C stores such pairs as it encounters them.

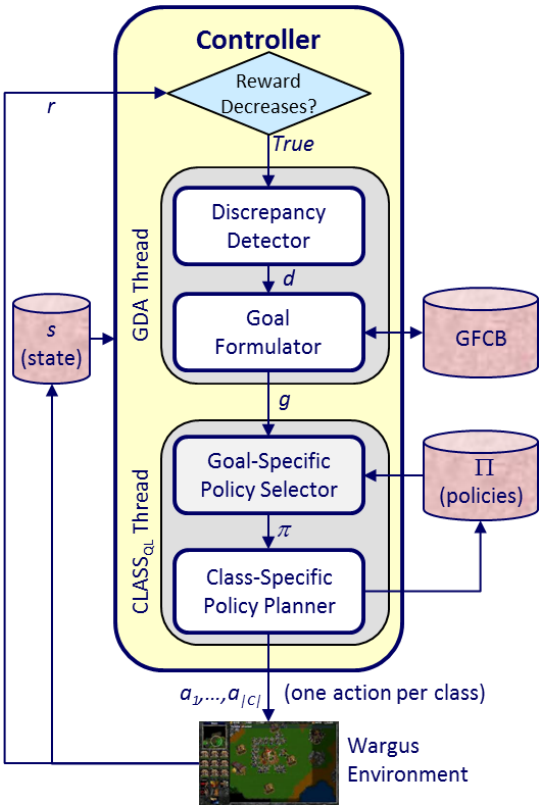

Figure 1: Information flow in GDA-C

The other case base assists with goal formulation. When a discrepancy d occurs between the expected state X and the actual state observed by the ***Discrepancy Detector***, this discrepancy is passed to the ***Goal Formulator***, which uses GFCB to formulate a new goal. ***GFCB*** maintains, for each (current) goal discrepancy pair, (g,d), a collection $\{(g_1,v_1),..,(g_m,v_m)\}$, where g_i is a goal to pursue next and v_i is the expected return of pursuing it. It outputs the next goal g to achieve.

The ***Goal-Specific Policy Selector*** selects a policy π based on the current goal g. The ***Class-Specific Policy Learner*** learns policies for new goals and refines the policies of existing goals. It uses Q-learning to update the Q-table entry $Q(s,a)$, given current state s and action taken a, as well as next state s' and next reward r (Sutton and Barto, 1998).

In many environments, there is no optimal policy for all situations. For example, in an adversarial game, a policy might be effective against one opponent's strategy but not versus others. By changing the goal when the system is underperforming, GDA-C changes the policy that is being executed, thereby making it more likely to adjust to different strategies.

We now provide formal definitions for the GDA process. Here we assume a state is represented as a vector $s = (v_1, \ldots, v_n)$ of numeric features, where v_i is a value of a feature f_i. Borrowing ideas from Weber et al. (2012), the agent uses ***optimistic expectations***. An expectation is ***optimistic*** iff $v'_i \not< v_i$, where *expectation* $e = (v'_1, \ldots, v'_n)$ and *previous state* $s = (v_1, \ldots, v_n)$. We use optimistic expectation *implicitly* in our algorithm. That is, if the previous state is $s = (v_1, \ldots, v_n)$ and, after executing an action, we reach a current state $s' = (v'_1, \ldots, v'_n)$ such that, for some k, $v'_k < v_k$ holds, then a ***discrepancy*** occurs. We represent a discrepancy as a vector of Boolean values $d = (b_1,\ldots,b_n)$, where b_k is *true* iff $v'_k < v_k$ holds. Basically, the agent expects that actions will not decrease the features' values. As we will see in Section 6, our state model consists of numeric features (e.g., the numbers of our own units) whose values the agent expects will remain the same or increase, but not decrease.

5 The GDA-C Algorithm

GDA-C coordinates the execution of a set of RL agents and how they learn. GDA-C uses an online learning process to update the Policies and GFCB case bases. Each GDA-C agent has its own individual Q-table. All q-values in Q-tables are initialized to zero. In each iteration of the algorithm, only some ***units*** (i.e., class instances such as peasants and archers) will be ready to execute a new action because others may be busy. Every unit records the state when it starts executing its current action. This is necessary for updating values in Q-tables. Below we present the pseudo-code of GDA-C, followed by its description.

GDA-C (Δ, Π, GFCB, $\mathcal{C}$, $\mathcal{A}$, ε, g_0) =

1: $s' \leftarrow$ GETSTATE(); $d' \leftarrow$ CALCULATEDISCREPANCY(s', s'); $\pi \leftarrow \Pi(g_0)$; $g \leftarrow g_0$
2: //-------- GDA thread --------
3: **while** episode continues
4: $s \leftarrow$ GETSTATE()
5: WAIT(Δ)
6: $r \leftarrow U(s) - U(s')$ // s' is the prior state
7: **if** $r < 0$ **then**
8: $d \leftarrow$ CALCULATEDISCREPANCY(s', s)
9: GFCB $\leftarrow$ Q-LEARNINGUPDATE(GFCB, d', g, d, r)
10: $g \leftarrow$ GET(GFCB, d, ε) // ε-greedy selection
11: $\pi \leftarrow \Pi(g)$
12: $s' \leftarrow s$; $d' \leftarrow d$
13: //-------- CLASS$_{QL}$ thread --------
14: **while** episode continues
15: $s \leftarrow$ GETSTATE()
16: **parallel for** each class $c \in \mathcal{C}$ // this loop controls agent α_c
17: $s_c \leftarrow$ GETCLASSSTATE(c, s)
18: $\mathcal{A}_c \leftarrow$ *getClassActions*($\mathcal{A}$,c); $A \leftarrow$ GETVALIDACTIONS($\mathcal{A}_c$, s_c)
19: $\pi_c \leftarrow \pi(c)$
20: **for** each instance $u \in c$ // this loop controls each unit or instance of class c
21: if u is a new instance then
22: $a'_u \leftarrow$ do-nothing; $s'_u \leftarrow s_c$
23: **if** instance u finished its action **then**
24: $r_u \leftarrow$ U(s_c) – U(s'_u) // U(s) is the utility of state s
25: $\pi_c \leftarrow$ Q-LEARNINGUPDATE(π_c, s'_u, a'_u, s_c, r_u)
26: $a \leftarrow$ GETACTION(π_c, ε, s_c, A)
27: EXECUTEACTION(a)
28: $s'_u \leftarrow s_c$; $a'_u \leftarrow a$
29: **return** Π, GFCB

GDA-C has two threads that execute in parallel and begin simultaneously when a game episode starts. The GDA thread (lines 3-12) selects a goal, which in turn determines the policy $\pi = (\pi_1, \ldots, \pi_n)$ that each RL agent will use and refine. The

CLASS_{QL} thread (Lines 14-28) performs Q-learning control on each of the α_k agents. When the GDA thread is deactivated (which is how our baseline system CLASS_{QL} works), the CLASS_{QL} thread refines the *same* policy from the beginning of the episode to the end. When the GDA thread is activated, the policy that CLASS_{QL} refines is the *most recent* one selected by the GDA thread.

GDA-C receives as input a constant number Δ (a delay before selecting the next goal), a policy case base Π, a goal formulation case base (GFCB), a set of classes $\mathcal{C}$, a set of actions $\mathcal{A}$, a constant value ε (for ***ε–greedy*** selection in Q-learning, whereby the action with the highest value is chosen with a probability $1-\varepsilon$ and a random action is chosen with a probability ε), and the initial goal g_0.

The GDA thread: The variable s' is initialized by observing the current state, d' is initialized with a null discrepancy (e.g., CalculateDiscrepancy(s', s')), and a policy π is retrieved from Π for the initial goal g_0 (all in Line 1). While the episode continues (Line 3), the current state s is observed (Line 4). After waiting for Δ time (Line 5), the reward r is obtained by comparing the utilities of current state s and previous state s' (Line 6). Our utility function calculates, for a given state, the total "hit-points" of the controlled team's units and subtracts those of the opponent team. When a unit is "hit" by other units, its hit-points will be decreased. A unit "dies" when its hit-points decrease to zero. If the reward is negative (Line 7), a new goal (and hence a new policy) will be selected as follows. First, the discrepancy d between s' and s is computed (Line 8). GFCB is then updated via Q-learning, taking into account previous discrepancy d', current goal g, discrepancy d, and reward r (Line 9). Then ε-greedy selection is used to select a new goal g from GFCB with discrepancy d (Line 10). Next, a new policy π is retrieved from Π for goal g (Line 11). Policy π will be updated in the CLASS_{QL} thread. Finally, previous state s' and discrepancy d' are updated (Line 12).

The CLASS_{QL} thread: While the episode continues (Line 14), the current state s is updated (Line 15). For each class c in the set of classes $\mathcal{C}$ (Line 16), the class-specific state s_c is acquired from s (Line 17). Agents from different classes have different sets of actions that they can perform. Therefore, a set of valid actions A must be obtained for each class s_c (Line 18). π_c is initialized with the policy for class c, which depends on the overall policy π updated in the GDA thread (Line 19). For each instance (or *unit*) u of class c (Line 20), if u is a new instance, initialize its state and action (Line 21-22). If u finished its action then calculate the reward r_u and update the policy π_c via Q-learning (Line 23-25). A new action is selected based on policy π_c using ε-greedy action selection (Line 26). Finally, the action is executed and the previous state s'_u and previous action a'_u are updated (Lines 27-28).

When the episode ends, GDA-C will return the policy case base Π and the goal formulation case base GFCB (Line 29).

Although at any point each agent α_k is following and updating a policy π_k, this does *not* mean that all units controlled by α_k will execute the same action. This is due to a combination of three factors. First, even when two units u and u' start executing the same action at the same time, there is no guarantee that they will finish at the same time. For example, if the action is to move u and u' to a specific location L, one of them might be hindered (e.g., engaged in combat with an enemy unit). Hence, u and u' might reach L at different times and therefore the subsequent actions they execute might differ because the state may have changed between the times that they arrive at L. Second, actions are stochastic (chosen with the ε-greedy method). Third, the policies are changing over time as a result of Q-learning or even altogether as a result

of the GDA thread. Therefore, at different times, even if in the same state, units might perform different actions.

6 States and Actions in Wargus

In this paper, we use Wargus in our experiments. Wargus is a widely used testbed for adversarial environments (e.g., (Aha *et al*., 2005; Judah *et al.*, 2010; Mehta *et al*., 2009; Ontañón and Ram, 2011)). In Wargus decision making must be conducted in real time. Wargus follows a rock-paper-scissors model for unit-versus-unit combat. For example, archers are strong versus footmen but weak versus knights. For these reasons, Mehta et al. (2009) argue that Wargus is a good research testbed for studying agent-based control methods. Each type of unit defines a unique class c so that every unit in that class can execute a set of actions $\mathcal{A}_c$. For example, an Archer can shoot an enemy from a distance while Gryphon Rider can fly across any barriers. Analogously, we also model each type of building (e.g., a Blacksmith, which can improve a unit's defense and damage, and a Barracks, which produces units such as Archers and Footmen for a specified amount of resources) as a class. In total, we modeled the following 12 classes:

1. Town Hall
2. Blacksmith
3. Lumber Mill
4. Church
5. Barrack
6. Knight
7. Footman
8. Archer
9. Ballista
10. Gryphon Rider
11. Gryphon Aviary
12. Peasant Builder

Each unit type has a different state representation. To reduce the number of states, we discretized features (*italicized* below) with many values (e.g., we used 18 bins for gold, where bin 1 means 0 gold and bin 18 corresponds to more than 4000). We also measure the distances from an enemy's units to the controlled player's camp using Manhattan distance. The features of the state representations per class are:

- Town Hall: *food*, *peasants*
- Blacksmith, Lumber Mill and Church: *gold*, *wood*
- Barrack: *gold*, *food*, *footmen*, *archer*, *ballista*, *knight*
- Knight, Footman, Archer, Ballista and Gryphon Rider: *our footmen*, *enemy footmen*, *number of enemy town halls*, *enemy peasants*, *enemy attackable units that are stronger than our footmen*, *enemy attackable units that are weaker than our footmen*
- Gryphon Aviary: *gold*, *food*, *gryphon rider*

- Peasant Builder: *gold*, *wood*, *food*, *number of barracks*, *lumber mill built?*,[2] *blacksmith built?*, *church built?*, *gryphon built?*, *path to a gold mine?*, *town hall built?*

$CLASS_{QL}$ (and, hence, GDA-C) reasons with composite actions such as "knight attack enemy camp", which are composed of several primitive actions such as selecting a building in the enemy camp, navigating to that building, and attacking it. Below is the list of all possible actions per class (by default every class can perform the action *do-nothing*):

- Town Hall: train peasant, upgrade to keep/castle
- Blacksmith: upgrade sword level 1, same but 2, upgrade human shield level 1, same but 2, upgrade ballista level 1, same but 2
- Lumber Mill: upgrade arrow level 1, same but 2, elven ranger training, ranger scouting, research longbow, ranger marksmanship
- Church: upgrade knights, research healing, research exorcism
- Barrack: train a footman, train an elven archer/ranger, train a knight/paladin, train a ballista
- Knight, Footman, Archer, Ballista, Gryphon Rider: wait for attack, attack the enemy's town hall/great hall, attack all enemy's peasants, attack all enemy's units that are near to our camp, attack all enemy's units that have their range of attacking equal to one, same but more than one, attack all enemy's land units, attack all enemy's air units, attack all enemy's units that are weaker (the enemy's units that have hit-points less than those of us), and attack all enemy's units (no matter what kind)
- Gryphon Aviary: train a gryphon rider
- Peasant Builder: build farm, build barracks, build town hall, build lumber mill, build black smith, build a stable, build a church, and build a gryphon aviary.

Our reward function is:

$$\text{total-hit-points(controlled team)} - \text{total-hit-points(enemy team)}$$

Each unit and building is assigned a number of hit points based on their type (e.g., Paladins have more than Peasants). Games are typically played until either the controlled team or the enemy is reduced to 0 points, at which time it loses the game.

7 Empirical Study

We measured the performance of GDA-C versus its ablation $CLASS_{QL}$ in experiments on small, medium, and large Wargus maps whose sizes are 32×32, 64×64, and 128×128 cells, respectively. In each map, we have two opponent teams (human and orc). Each starts with only one Peasant/Peon (i.e., a unit used to harvest resources and construct new buildings), one Town Hall/Great Hall, and a nearby gold mine. Each competitor also starts on one side of a forest that divides the map into two

[2] The question mark signals that this is a *binary* feature.

parts. We added this forest and walls to provide opponents with sufficient time to build their armies. Otherwise, our algorithms will learn an efficient early attack (called a "rush"), which will end the game when the opponents have produced only a few units or buildings.

7.1 Experimental Setup

We conducted two experiments. In the first, we compared the performance of each algorithm (i.e., GDA-C or $CLASS_{QL}$) against Wargus's built-in AI. The built-in AI in Wargus is quite good; it provides a challenging game to an average human player. In the second, we instead compared their performance in a direct competition. We use five adversaries (defined below) and the Wargus' built-in AI to train and test each algorithm. These adversaries can construct any type of unit unless otherwise stated:

- *Land Attack*: This tries to balance offensive/defensive actions with research. It builds only land units.
- *Soldier's Rush*: This attempts to overwhelm the opponent with cheap military units early in the game.
- *Knight's Rush*: This attempts to quickly research advanced technologies, and launch large attacks with the strongest units in the game (knights for humans and ogres for orcs).
- *Student Scripts*: We included the top two competitors that were created by students for a classroom tournament.

To ensure there is no bias because of the landscape, we swapped the sides of each team in each round. Also, to prevent race inequities, in each round each team plays once with each race (i.e., human or orc).

In Experiment 1, we trained GDA-C and $CLASS_{QL}$ by playing one game versus each of the five adversaries. We then tested GDA-C and $CLASS_{QL}$ by playing one game against the Wargus's built-in AI. The performance metric is:

$$(\text{wins}(\text{GDA-C}) - \text{wins}(\text{built-in AI})) \; - \; (\text{wins}(\text{CLASS}_{QL}) - \text{wins}(\text{built-in AI})),$$

where ***wins(A)*** is the number of wins for team ***A***. For Experiment 2, we trained GDA-C and $CLASS_{QL}$ with all five adversaries and then tested them in combat against each other. We report results for the average of ten runs, where the performance metric is:

$$\text{wins}(\text{GDA-C}) - \text{wins}(\text{CLASS}_{QL})$$

Table 2: The average time of running a game for both experiments

Map size	One game	Experiment 1	Experiment 2
small	31 sec	25 hours	38 hours
medium	3 min 27 sec	115 hours	172 hours
large	11 min 28 sec	191 hours	286 hours

In Experiment 1, the matches pitting the two algorithms versus the built-in AI took place after training GDA-C and $CLASS_{QL}$ against each of the other five adversaries

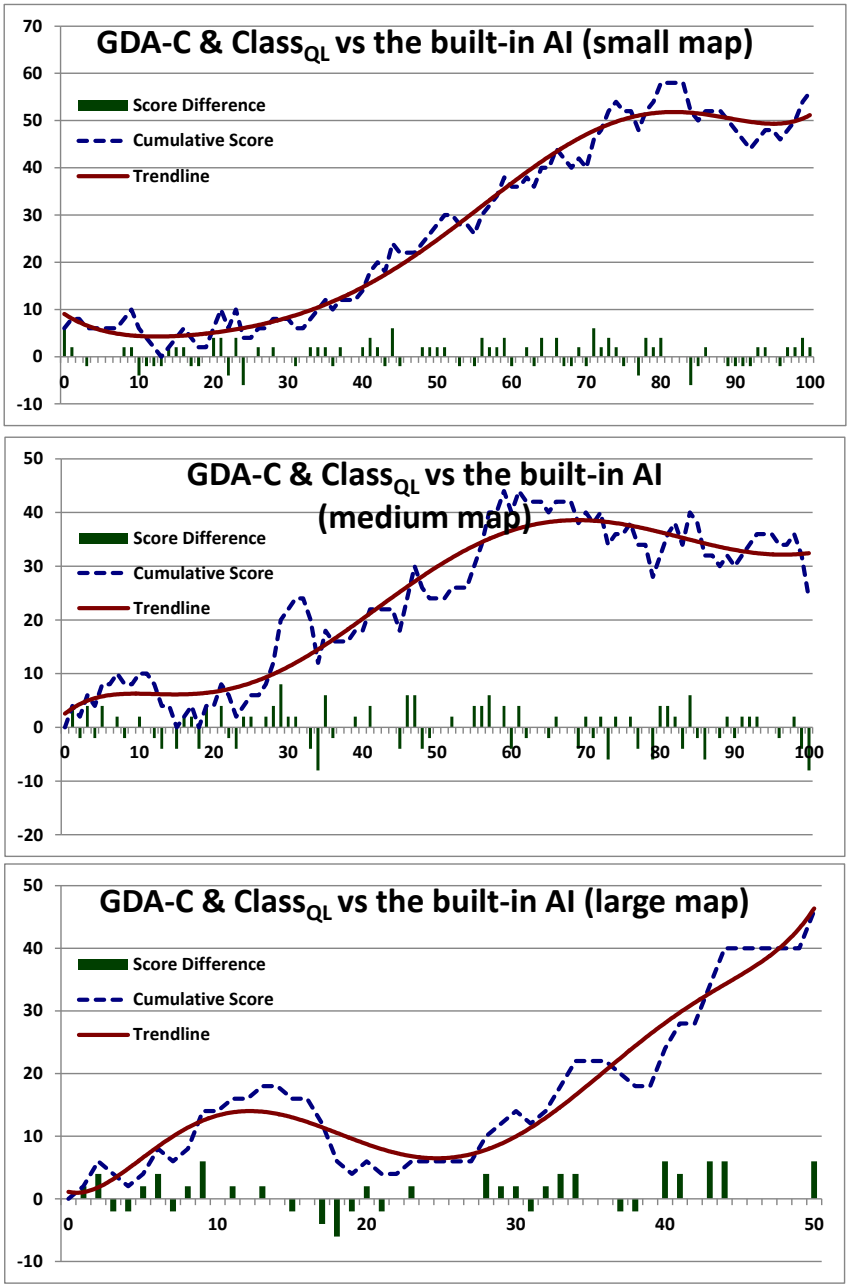

Figure 2: The results of Experiment 1: The relative performance of GDA-C versus CLASS$_{QL}$ playing against the built-in Wargus AI on the three maps

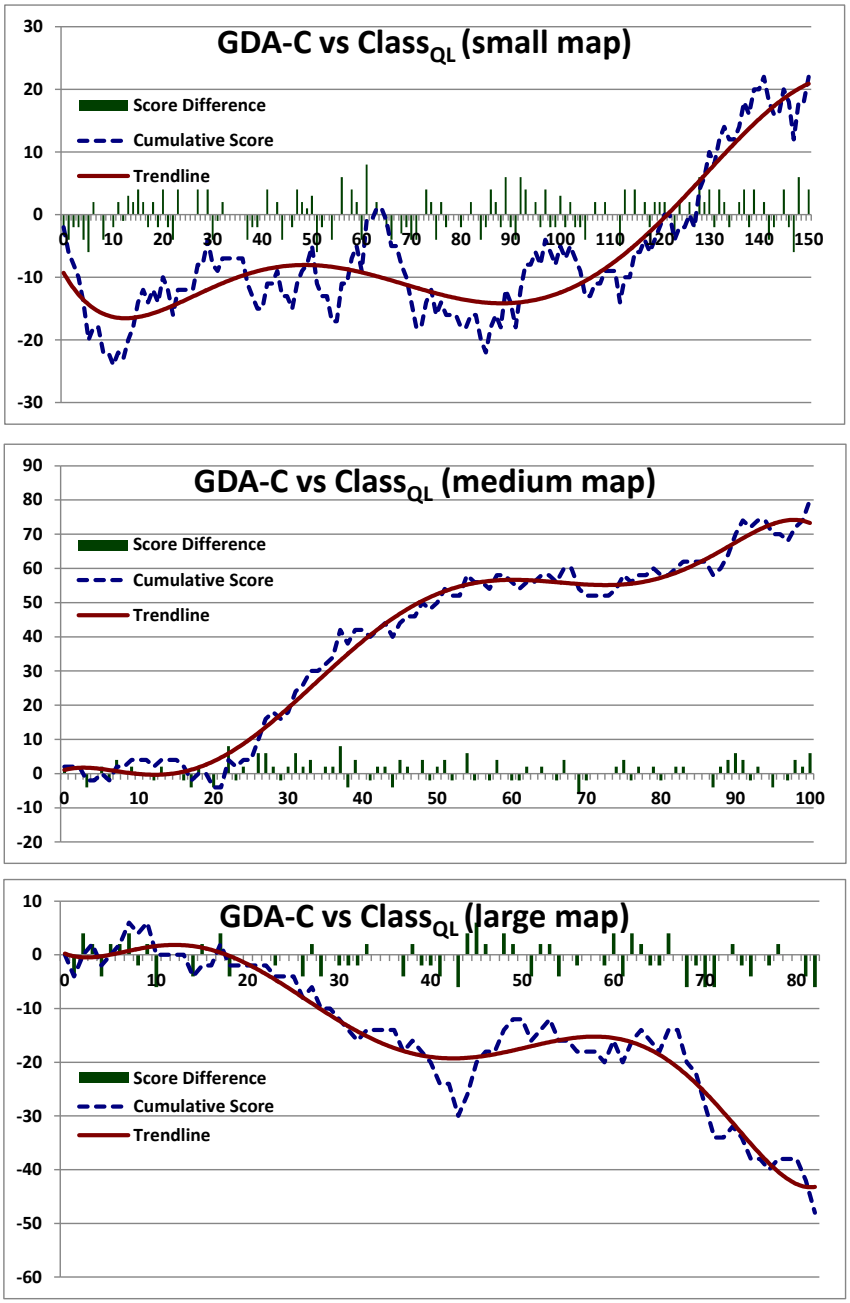

Figure 3: The results of Experiment 2: GDA-C versus CLASS$_{QL}$ on the small, medium, and large maps

for *n* games, where we varied $n = 0,1,2,\ldots,N$. Similarly, in Experiment 2 the matches pitting GDA-C versus CLASS_{QL} took place after training them against each of the adversaries for *n* games, where again $n = 0,1,2,\ldots,N$. The total number *N* of games varied as indicated in the results. Table 2 shows the running times for the experiments.

7.2 Results

Figures 2 and 3 display the results for Experiments 1 and 2, respectively. For both experiments each data point is the average of 10 tests, and the graphs display the results for the small, medium, and large maps. There are two curves: the score difference for each data point and the cumulative score difference up to that data point. The x-axis refers to the training iteration number.

Results for Experiment 1: For all three maps, both GDA-C and CLASS_{QL} outperform the built-in AI (not shown in the graphs) but GDA-C does so at a higher rate than CLASS_{QL}, as shown in Figure 2. These results illustrate the effectiveness of changing policies as GDA-C does when underperforming compared to sticking to the current policies and refining them by using reinforcement learning.

Results for Experiment 2: For the small map CLASS_{QL} initially outperforms GDA-C but its performance improves and it eventually outperforms CLASS_{QL}. From $x = 110$ (i.e., after 110 training iterations), it begins to outperform CLASS_{QL} and surpasses it by $x = 117$. For the medium map, the algorithms start evenly but then GDA-C quickly outperforms CLASS_{QL}. For the large map CLASS_{QL} outperforms GDA-C. We ran further iterations (not shown) and this trend continues. We believe that for the large map, CLASS_{QL} is learning a very good strategy, perhaps even optimal for the map, and GDA-C will continue to retrieve policies that cannot outperform the one executed by CLASS_{QL}. This suggests that, at some point, GDA-C should deactivate its GDA thread and continue only with the CLASS_{QL} thread. How we would identify such a point is a topic left for future research.

There is a lot of fluctuation in individual data points. For example, despite the cumulative trends in the medium map for Experiment 2, which show that GDA-C outperforms CLASS_{QL}, the reverse occasionally occurs (e.g., at $x = 70$). The reason for this fluctuation is that Wargus is a stochastic environment that introduces a lot of randomness in the outcomes of individual actions and, hence, in the overall outcome of individual games.

8 Conclusions and Future Work

We introduced GDA-C, an algorithm that divides the state and action spaces among multiple, cooperating RL agents, where each agent uses Q-learning to learn a different policy for controlling units of a single class. Because these agents share a common reward function, they can coordinate. GDA-C augments this coordination by using a partial goal-driven autonomy (GDA) agent to retrieve previously stored policies for the RL agents to apply and further revise. Our experiments demonstrate that GDA-C outperforms its ablation, CLASS_{QL}, in most situations.

For future work we want to explore two directions. First, we plan to make the state representation more general so it does not depend on the expectation that the feature's values must increase. To do this, we will borrow ideas from our previous GDA research (e.g., (Jaidee *et al.*, 2011; 2012)), in which we used more general state representations. Second, we will examine alternative GDA agents. GDA-C does not include two steps that are common to the GDA model, namely *discrepancy explanation* and *goal management*. We will assess the utility of generating explanations of discrepancies for GDA-C. That is, recent research on GDA (Molineaux *et al*., 2012) has demonstrated the value of using discrepancy explanations to determine which goals to select, and this may also be true for our studies. Alternative methods for goal management also exist. GDA-C simply replaces one goal with another, without considering, for example, whether the initial goal should simply be delayed. We will study more comprehensive strategies for goal management in our future research.

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

References

Aha, D.W., Molineaux, M., & Ponsen, M. (2005). Learning to win: Case-based plan selection in a real-time strategy game. *Proceedings of the Sixth International Conference on Case-Based Reasoning* (pp. 5-20). Chicago, IL: Springer.

Cox, M.T. (2007). Perpetual self-aware cognitive agents. *AI Magazine,* **28**(1), 23-45.

Jaidee, U., Muñoz-Avila, H., & Aha, D.W. (2011). Integrated learning for goal-driven autonomy. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*. Barcelona, Spain: AAAI Press.

Jaidee, U., Muñoz-Avila, H., & Aha, D.W. (2012). Learning and reusing goal-specific policies for goal-driven autonomy. In *Proceedings of the Twentieth International Conference on Case-Based Reasoning* (pp. 182-195). Lyon, France: Springer.

Judah, K., Roy, S., Fern, A., & Dietterich, T.G. (2010). Reinforcement learning via practice and critique advice. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*. Atlanta, GA: AAAI Press.

Klenk, M., Molineaux, M., & Aha, D.W. (in press). Goal-driven autonomy for responding to unexpected events in strategy simulations. To appear in *Computational Intelligence*.

Mehta, M., Ontañón, S., and Ram, A. (2009). Using meta-reasoning to improve the performance of case-based planning. *Proceedings of the Seventeenth International Conference on Case-Based Reasoning* (pp. 210–224). Seattle, WA: Springer.

Molineaux, M., Klenk, M., & Aha, D.W. (2010). Goal-driven autonomy in a Navy strategy simulation. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*. Atlanta, GA: AAAI Press.

Molineaux, M., Kuter, U., & Klenk, M. (2011). What just happened? Explaining the past in planning and execution. In T. Roth-Berghofer, N. Tintarev, & D.B. Leake (Eds.) *Explanation-Aware Computing: Papers from the IJCAI Workshop*. Barcelona, Spain.

Molineaux, M., Kuter, U., & Klenk, M. (2012). DiscoverHistory: Understanding the past in planning and execution. *Proceedings of the Eleventh International Conference on Autonomous Agents and Multiagent Systems* (pp. 989-996). Valencia, Spain: ACM Press.

Muñoz-Avila, H., Aha, D.W., Jaidee, U., Klenk, M., & Molineaux, M. (2010). Applying goal directed autonomy to a team shooter game.*Proceedings of the Twenty-Third Florida Artificial Intelligence Research Society Conference* (pp. 465-470). Daytona Beach, FL: AAAI Press.

Ontañón, S., & Ram, A. (2011). Case-based reasoning and user-generated AI for real-time strategy games. In P.A. González-Calero and M.A. Gómez-Martín (Eds.), *Artificial Intelligence for Computer Games*. Berlin: Springer-Verlag

Powell, J., Molineaux., M., & Aha, D.W. (2011). Active and interactive discovery of goal selection knowledge. In *Proceedings of the Twenty-Fourth Conference of the Florida AI Research Society*. West Palm Beach, FL: AAAI Press.

Sutton, R.S., & Barto, A.G. (1998). *Reinforcement learning: An introduction*. MIT Press, Cambridge, MA.

Weber, B., Mateas, M., & Jhala, A. (2010). Applying goal-driven autonomy to StarCraft. In *Proceedings of the Sixth Conference on Artificial Intelligence and Interactive Digital Entertainment*. Stanford, CA: AAAI Press.

Weber, B., Mateas, M., & Jhala, A. (2012). Learning from demonstration for goal-driven autonomy. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. Toronto (Ontario), Canada: AAAI Press.