

2013

Integrated Learning for Goal-Driven Autonomy

ULIT JAIDEE
Lehigh University

Follow this and additional works at: <http://preserve.lehigh.edu/etd>



Part of the [Computer Engineering Commons](#)

Recommended Citation

JAIDEE, ULIT, "Integrated Learning for Goal-Driven Autonomy" (2013). *Theses and Dissertations*. Paper 1516.

This Dissertation is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

Integrated Learning for Goal-Driven Autonomy

by

Ulit Jaidee

A Dissertation

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Computer Engineering

Lehigh University

January 2014

© 2013 Copyright by Ulit Jaidee

All Rights Reserved

Approved and recommended for acceptance as a dissertation in partial fulfillment
of the requirements for the degree of Doctor of Philosophy

Defense Date

Dissertation Advisor/Committee Chair

Héctor Muñoz-Avila

Approved Date

Committee Members:

Richard Decker

Jeff Heflin

Michael Spear

To my parents

Mr. Kit Jaidee

Mrs. Jongdee Boonyupakorn

ACKNOWLEDGMENTS

I wish to thank the following persons and organizations whose support helped make this work possible:

- My advisor Professor Héctor Muñoz-Avila for his research direction and guidance not only for academic perspective but also living philosophy.
- My frequent co-author Dr. David Aha for many valuable ideas and comments.
- My parents Kit Jaidee and Jongdee Boonyupakorn who supported and encouraged me.
- My best friend Jerry Makos who always lovingly stood by me in every matter.
- My dear friend and tutor Karen Bedics for helping to improve my English and helping to review this manuscript.
- My committee members, Professor Jeff Heflin, Professor Michael Spear, and Professor Richard Decker for their helpful comments.

- National Science Foundation for partially supporting my work through grant NSF 1217888.
- Naval Research Laboratory for a partial grant.
- Thai Government for all the financial support and the scholarship to pursue my PhD degree.

Table of Contents

Acknowledgments	v
List of Tables	xii
List of Figures	xiv
List of Algorithms	xvii
Abstract	1
CHAPTER 1 Introduction	3
1.1 Prolog	3
1.2 Goal-Driven Autonomy.....	6
1.3 Research Question.....	8
1.4 Contributions.....	9
1.5 Brief Overview of the Research.....	10
CHAPTER 2 Background	16
2.1 Reinforcement learning.....	17
2.1.1 Policy.....	17
2.1.2 Q-learning.....	19
2.2 Case-Based Reasoning.....	20

2.2.1	Case Similarity	24
CHAPTER 3	Goal-Driven Autonomy	26
CHAPTER 4	Goal-Driven Autonomy with Hierarchical-Task Network	
	Planning	31
4.1	Hierarchical-Task Network	31
4.2	Goal-Driven Autonomy with Hierarchical-Task Network Planner	32
4.3	Example in the DOM Game.....	36
CHAPTER 5	Goal-Driven Autonomy with Case-Based Reasoning	38
5.1	Case-Based Goal Driven Autonomy	38
CHAPTER 6	Case-Based Learning of Expectations and Goal-Formulation	
	Knowledge	42
6.1	Definitions	42
6.2	LGDA Algorithm	44
6.3	Implementation and Example.....	45
CHAPTER 7	Integrated Learning of Goal-Driven Autonomy Elements	48
7.1	The GRL Algorithm	49
7.2	Example.....	53
CHAPTER 8	Goal-Driven Autonomy Coordination of Multiple Agents	56
8.1	CLASS _{QL} : Modeling Unit Classes as Agents.....	57

8.1.1	The CLASS _{QL} Algorithm	57
8.1.2	Application of CLASS _{QL} for Wargus.....	61
8.1.3	Modeling Wargus in CLASS _{QL}	66
8.1.4	State Representation	68
8.1.5	Actions.....	70
8.1.6	Analysis of CLASS _{QL}	74
8.2	GDA-C: Case-Based Goal-Driven Coordination of Multiple Learning Agents	77
8.2.1	Multi-Agent Setting.....	79
8.2.2	Case Bases and Information Flow in the GDA-C Agent.....	80
8.2.3	The GDA-C Algorithm.....	82
CHAPTER 9 Experimental Evaluation		87
9.1	The description of problem domains used for experiments	87
9.1.1	DOM: Domination game	88
9.1.2	Wargus.....	89
9.2	Empirical Evaluation of GDA-HTNbots.....	91
9.3	Empirical Evaluation of CB-GDA	95
9.4	Empirical Evaluation of the LGDA	100
9.4.1	Experimental Setup.....	100

9.4.2	Results	102
9.5	Empirical Evaluation of the GRL.....	105
9.5.1	Scenarios.....	106
9.5.2	Protocol and Results	107
9.6	Empirical Evaluation of the CLASS _{QL}	114
9.6.1	Experiment #1.....	114
9.6.1.1	Experimental Setup.....	114
9.6.1.2	Results.....	117
9.6.2	Experiment #2.....	118
9.6.2.1	Experimental Setup.....	118
9.6.2.2	Results.....	121
9.7	Empirical Evaluation of the GDA-C.....	123
9.7.1	Experimental Setup.....	124
9.7.2	Results	129
CHAPTER 10 Related Work		131
10.1	Planning Methods and Their Disadvantages Compared to Goal-Driven Autonomy	131
10.2	Integrations of Case-Based Learning and Reinforcement Learning	137
10.3	Goal-Driven Autonomy Agents and Their Integration of Learning	138

10.4	Learning Agents in Real-Time Strategy Games	141
10.5	Goal-Driven Autonomy Agents That Can Play RTS Games.....	144
CHAPTER 11 Conclusions		146
11.1	Final Remarks	146
11.2	Future Work	150
Bibliography		155
Appendix A Wargus Units and Structures		166
A.1.	Wargus Units.....	166
A.2.	Wargus Structures	171
Appendix B Scoring in Wargus		177
Curriculum Vitae		179

LIST OF TABLES

Table 1-1:	List of AI systems and their outstanding points of difference	14
Table 4-1:	Example explanations of discrepancies (with some expectations and observations shown), and the corresponding recommended goals.	37
Table 8-1:	All possible high-level actions for each Wargus class.....	72
Table 8-2:	Space saved by CLASS _{QL} compared to a conventional RL agent.	77
Table 9-1:	The adversaries in DOM game and their descriptions.....	91
Table 9-2:	Average Percent Normalized Difference in Game AI System vs. Opponent Scores (with average scores in parentheses).	94
Table 9-3:	Average Percent Normalized Difference in the Game AI System vs. Opponent Scores (with average Scores in parentheses)	97
Table 9-4:	Average percent normalized difference in the game AI system vs. opponent scores (with average scores in parentheses) with statistical significance.	97
Table 9-5:	Average Percent Normalized Difference for the Dynamic Game AI Systems vs. CB-GDA Scores (with average scores in parentheses)....	99
Table 9-6:	Average Utility Results from Experiment 1.	103

Table 9-7:	The results of using the q-table that was trained with one scenario (the first landscape) and tested with other unseen scenarios (the second and the third landscape) on the small, medium and large maps.	123
Table 9-8:	The average time of running a game for both experiments	127
Table 10-1:	Categories of works versus managerial tasks	142
Table A-1:	List of Wargus Units and their properties in both Human and Orc ...	166
Table A-2:	List of Wargus Structures and their properties in both Human and Orc.....	171
Table B-1:	Points earned from killing specific units or structures.....	177

LIST OF FIGURES

Figure 1.1:	No one needs to tell the boy why he should not touch a hot stove again. (The figure is retrieved from.....)	4
Figure 1.2:	A simplified flow of Goal-Driven Autonomy	7
Figure 2.1:	The agent-environment interaction in reinforcement learning.	18
Figure 2.2:	The Case-Based Reasoning Cycle.	21
Figure 2.3:	Driving from Allentown to New York City. (a) Without revising, driving from Allentown to Jersey City and then driving from Jersey City to New York City. (b) With revising, driving from Allentown to most of the way to Jersey City and then using bypass highway to New York City.	23
Figure 2.4:	Using Case-Based Reasoning to solve the problem of driving from Allentown to New York City.	24
Figure 3.1:	A Conceptual Model for Goal Driven Autonomy	27
Figure 3.2:	Data flow of Goal Driven Autonomy	30
Figure 4.1:	An example DOM game map with five domination locations (yellow flags), where small rectangles identify the respawning locations for the	

	agents and the remaining two types of icons denote each player's agents.	35
Figure 8.1:	Illustration how units in Wargus perform a decision making.....	59
Figure 8.2:	The interaction between CLASS _{QL} and its environment.....	62
Figure 8.3:	An example of a timeline at the beginning of the Wargus game.....	76
Figure 8.4:	Information flow in GDA-C.	81
Figure 9.1:	An example DOM game map with five domination locations (yellow flags), where small rectangles identify the respawning locations for the agents and the remaining two types of icons denote each player's agents.	93
Figure 9.2:	Results from Experiment 2: Average learning curves for comparing LGDA DOM performance vs. non-learning and ablated agents. The trend lines were generated using a polynomial fit to the raw curves.	105
Figure 9.3:	The results of the Wargus experiments: GRL vs. Retaliate (a) and vs. LGDA (b) on the medium map, and GRL vs. Retaliate (c) and vs. LGDA (d) on the large map. The x-axis plots the number of training episodes, while the y-axis plots the average utility (i.e., score difference of GRL versus another agent).....	112
Figure 9.4:	Results from the DOM experiments: (a) GRL vs. Retaliate and (b) GRL vs. LGDA. The x-axis plots the number of training episodes,	

	while the y-axis plots the average utility (i.e., score difference of GRL versus another agent).	113
Figure 9.5:	The screen capture of the small map from Wargus.	115
Figure 9.6:	The results of the experiments #1 from Wargus: Class _{Q-L} vs. (a) Land-Attack, (b) SR, (c) KR, (d) SC1 and (e) SC2 respectively.	116
Figure 9.7:	The detailed landscape of the (a) 1 st , (b) 2 nd , (c) 3 rd large maps. The highlighted squares are the locations of both teams.	121
Figure 9.8:	The results of the experiments #2 from Wargus game: CLASS _{QL} vs the built-in AI on the (a) small, (b) medium and (c) large maps.	123
Figure 9.9:	Landscapes and details of the small (a), medium (b), and large maps (c) that used for the experiment #1 and #2.	126
Figure 9.10:	The results of Experiment: GDA-C versus CLASS _{QL} on the small (a), medium (b), and large maps (c). Figures (a-2), (b-2), and (c-2) show the results as accumulative score.	129

LIST OF ALGORITHMS

Algorithm 1	Q-learning: An off-policy temporal-difference control algorithm.....	20
Algorithm 2	CB-GDA algorithm.....	39
Algorithm 3	LGDA algorithm.....	44
Algorithm 4	GRL algorithm.....	50
Algorithm 5	CLASS _{QL} algorithm.....	60
Algorithm 6	CLASS _{QL} algorithm for Wargus.....	64
Algorithm 7	GDA-C algorithm.....	83

ABSTRACT

Goal-Driven Autonomy (GDA) is an online planning framework that focuses on the integration of planning, execution and goal reasoning. Given a goal, a GDA agent generates a plan to pursue the goal. In addition, by using its expectations, the agent reasons about what the next observed state should be when the plan's actions are executed. If the expectation does not match the observed state, the GDA agent is able to suggest a new goal to be pursued.

In most GDA research, knowledge is handcrafted and later fed into the GDA agent by humans who are experts in a particular problem domain. Therefore, in this dissertation, we would like to investigate about how we can create GDA agents that have abilities to acquire knowledge by themselves and reuse that knowledge. The problem domains we focus are real-time strategy (RTS) games. We used two RTS games called DOM game and Wargus. We used Reinforcement Learning because it is an unsupervised learning method and we want our GDA agents to be autonomous.

Our research went through multiple steps. First, we built a GDA agent without integration of any learning methods. Later, we incrementally integrated learning methods to each component in the GDA architecture until we build a GDA agent that

could learn knowledge for all components. The experimental results show that we can create GDA agents that have the ability to acquire GDA knowledge by themselves.

CHAPTER 1

INTRODUCTION

It is not the strongest of the species that survives, nor the most intelligent, but the one most responsive to change.

— Charles Darwin

1.1 Prolog

In everyday life, living creatures face problems and try to overcome those problems. There are several methods to solve problems. In nature, living creatures mostly use the method called trial-and-error by repeating actions until success is achieved, or else they stop trying (Radnitzky, et al., 1993). In addition, creatures that can remember which experiences ensure their survival have another method to solve problems. They have ability to learn. Learning is one of the most natural methods that living creatures use for survival.

Research in artificial intelligence (AI) and specifically machine learning can enable computers to solve problems by themselves. Living creatures such as humans can learn directly from the environment around them. A little child may touch a hot

stove and then learn by himself without anyone telling him not to touch it again (Figure 1.1). Reinforcement learning (RL) agents can also adapt to a new environment. RL is a learning approach that learns how to map situations to actions, so as to maximize a numerical reward signal (Sutton, et al., 1998). RL agents learn naturally by interacting directly with their environment and observing the outcome. However, RL agents need a large memory space; they need a memory the size of the state-action space for the particular problem domain (Si, et al., 2012). In complex environments such as a real-time battle strategy team-based game, the state-action space is exponentially large according to the number of states and actions (Jaidee, et al., 2013; Weber, et al., 2010). Conventional RL methods alone are not practical enough to handle complex problems. Therefore, we need some new model of a learning method.



Figure 1.1: No one needs to tell the boy why he should not touch a hot stove again. (The figure is retrieved from (One Crafty Mother, 2010)).

In some situations we do not learn from experience. Actually, some things that are dangerous, harmful, or life-threatening, we should never learn by trying and observing

the result by ourselves. For example, smoking a cigarette is not regarded as healthy. We know because someone told us, or we read research connecting to the fact. We do not need to really smoke a cigarette to know that is harmful (however, about sixty years ago, if your great-grandparents went to see a doctor, it is possible that the doctor may have recommended smoking more cigarettes. At that time, people did not know the real effects of smoking.) Similarly, computer agents can smartly perform tasks by following the rules already established by experts. However, this kind of method is neither autonomous nor dynamic.

In autonomous learning, an agent has to act and observe directly from its environment. If agents can share what they learn, their learning speed will increase. For example, a delivery team needs to reschedule their work shifts. Each one in the team who will drive on the new schedule should learn the best direction to drive from the previous drivers on the team who previously drove the route on the same day and time. Learning from someone who works exactly in the same position and situation can help in saving a lot of time. This is more efficient than trying to find the best driving direction from scratch every time the driver schedule changes.

When the environment or situation around us changes, we must adapt in what we do in everyday life (and possibly we must learn more). Imagine a situation where a person always takes a bus of the same line daily to go to the office. The person hears on the morning news that there was an accident on the road used by the bus line. Because the person is aware of the heavy traffic that may occur as a result of the accident, he will need to plan some new ways to go to his office. How AI agents

would be able to do this? What will the agent do? In other words, can we build an agent that can independently make these decisions?

It may take a while for Reinforcement Learning agents before they can adapt to changing environments. This long adaptation period is due, in a large part, to the RL exploration phase, in which an agent must almost blindly use trial-and-error to test new actions and develop a policy to maximize its expected future rewards.

1.2 Goal-Driven Autonomy

As mentioned in the previous section, when the environment or situation changes dramatically, some type of learning agents such as reinforcement learning agents can take a long period of time before they can learn to adapt. To overcome this issue, we study Goal-Driven Autonomy with the integration of learning methods. Instead of having only one plan for all kinds of situation, Goal-Driven Autonomy uses a collection of plans to pursue different goals that are designed to handle different situations.

Goal-Driven Autonomy is an online planning framework that focuses on the integration of planning, execution and goal reasoning. Given a goal, a GDA agent generates a plan (i.e., a sequence of actions) to pursue the goal. The agent also reasons with expectations about what the state should be when the plan is executed. If expectation doesn't match the state, the GDA agent will suggest a new goal to be pursued instead.

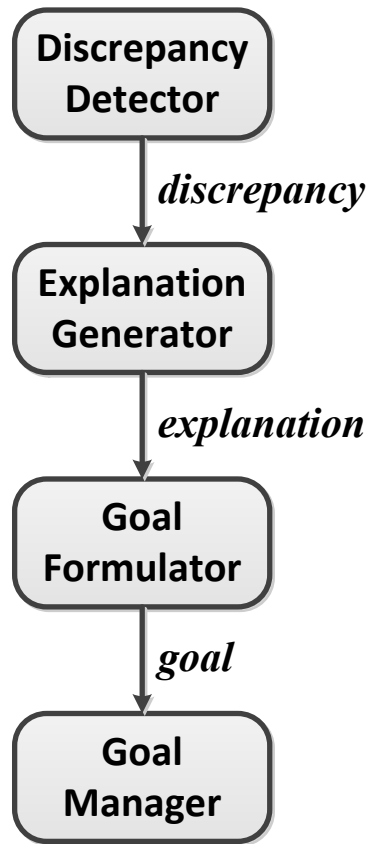


Figure 1.2: A simplified flow of Goal-Driven Autonomy

Figure 1.2 shows the flow of Goal-Driven Autonomy. If some unexpected situation occurs, a discrepancy between the expected situation and actual situation will be computed by the Discrepancy Detector. Next, an explanation is generated by the Explanation Generator. After that, a new goal will be chosen from the Goal Formulator. The Goal Manager will be the one that executes the new goal.

1.3 Research Question

In most GDA research, knowledge is handcrafted by human experts in a problem domain. This knowledge is fed into the GDA agent. For example, in HTNbots a collection of rules is given that maps for given discrepancies that may dictate what goal should be pursued next (Munoz-Avila, et al., 2010). This knowledge may require a team of programmers and experts working together. Hence, a lot of time and money could be invested to handcraft such knowledge.

Unfortunately, human expertise is expensive and handcrafting such knowledge is time consuming. Therefore, we will study the following research question in this thesis:

Can we create GDA agents that have the ability to acquire knowledge by themselves and reuse this knowledge?

We are interested in studying this question in the context of complex environments. A complex environment has the following characteristics (Russell, et al., 2003):

- **Partially observable:** the information that agents observe from their environment is not complete. There is some information that is hidden.
- **Stochastic:** the actions that agents take can have multiple possible outcomes.

- **Multiagent:** there are multiple agents interacting under the environment.

1.4 Contributions

The following is a summary of the scientific contributions of this dissertation to the state-of-the-art in integrated learning for Goal-Driven Autonomy (GDA) research:

- **First integration of Case Based Reasoning (CBR) and Goal-Driven Autonomy.** CB-GDA is the first GDA system that integrates case based reasoning. CB-GDA uses case-based planning techniques to formulate goals by deriving inferences from the game state and the agent's expectations.
- **First GDA system to automatically learn state expectations.** LGDA is a GDA agent that automatically acquires knowledge by using a case based learning to map (state, action) pairs to a distribution over expected states, and (goal, discrepancy) pairs to a value distribution over discrepancy-resolution goals. It also uses a reinforcement learning method to learn the goals' expected values.
- **First GDA system capable of learning and reusing goal-specific policies.** Our GDA agent, named the Goal Reasoning Learner (GRL), integrates case based learning and reinforcement learning processes to learn and reuse goal-specific action policies, state expectations, and goal selection knowledge.

- **First learning agent that can learn multiple real-time strategy games managerial tasks.** CLASS_{QL} and GDA-C are the first learning agent (and first GDA agent) that are capable of learning on 5 out of 6 managerial tasks¹ (Table 10-1) that are needed for creating automated players of real-time strategy games (Scott, 2002).

1.5 Brief Overview of the Research

Early in my research, we compared the performance of Goal-Driven Autonomy alone without integrating any learning methods to hand-coded AI engines crafted by human and a reinforcement learning agent. Without integration of any learning methods, the four components shown in the Figure 1.2 are hand-coded by a human. Later in my research, we incrementally integrated learning methods to each component in the GDA flow. Ours is the first GDA agent learns the main components.

In our studies, the first system called GDA-HTNBots (Munoz-Avila, et al., 2010) (see Section 4.2), reasons about the events occurring in its environment, changes its own goals in response to them, and replans to satisfy these changed goals. To do this, GDA-HTNbots constantly monitors its environment for unexpected changes and dynamically formulates a new goal when appropriate. GDA-HTNbots is

¹ The resource task is the only one task that our agents do not learn because we built a simple algorithm to maintain equilibrium among resources. Thus, the agents are not necessary to take time learn it.

a partial instantiation of GDA model, and present a limited empirical study of its performance. The results support our primary claim: agent performance in a team shooter domain with exogenous events can be improved through appropriate self-selection of goals for some conditions (see Section 9.2).

The system that was developed after the GDA-HTNBots is called CB-GDA (Muñoz-Avila, et al., 2010), the first GDA system to employ case-based reasoning (CBR) methods (for more details, see Section 2.2). CB-GDA uses two case bases to dynamically generate goals. The first case base relates goals with expectations, while the latter's cases relate mismatches with (new) goals. The empirical study of CB-GDA on the task of winning games was defined using a complex gaming environment called DOM which will mention more details and descriptions in Section 9.1.1.

The study revealed that CB-GDA outperforms a rule-based variant of GDA when executed against a variety of opponents. CB-GDA also outperforms a non-GDA replanning agent against the most difficult of these opponents and performs similarly against the easier ones. In direct matches, CB-GDA defeats both the rule based GDA system and the non-GDA replanner (for more details of the results, see Section 9.3).

As for the third system, learning GDA (LGDA) was introduced (see more details in Section 6.2). LGDA (Jaidee, et al., 2011) is a GDA algorithm that learns two types of cases from observed discrepancy resolution episodes: (1) expectation cases, which map state and action pairs to a distribution over expected states, and (2) goal formulation cases, which map goal-discrepancy pairs to a distribution of expected

values over discrepancy resolution goals. LGDA learns these through an integration of case-based reasoning (CBR) and reinforcement learning (RL) methods. It models goal formulation as an RL problem in which a goal’s value is estimated based on the expected future reward for achieving it. The claim of LGDA is that this integration can learn to perform as well as a non-learning GDA agent that employs expert knowledge, and can outperform GDA agents that use only CBR or RL. We report LGDA’s comparative evaluation on a task involving the control of a team in a domination video game (DOM). The results show that LGDA outperforms most built-in hand-coded opponents (i.e., adversaries) and significantly outperforms its ablated versions. Finally, LGDA learns to perform almost as well as a non-learning GDA variant, whose case knowledge was hand-crafted by a domain expert such that it also significantly outperforms these adversaries (see Section 0).

Next, we introduce GRL (Goal Reasoning Learner) (Jaidee, et al., 2012), a case-based GDA agent (see more details in Section 7.1). Unlike previous work integrating RL and CBR and work on GDA, GRL embeds GDA in an RL cycle by learning and reusing the three kinds of cases mentioned above. GRL is the first GDA agent capable of learning and reusing goal-specific policies. Our hypothesis is that, as a result of this capability, GRL can fine-tune strategies by exploiting the episodic knowledge captured in its cases. We report performance gains versus a state-of-the-art GDA agent and an RL agent for challenging tasks in two real-time gaming domains (see Section 9.5).

A well-known limitation of RL is that given the large size of the action and state space, it is very difficult to use RL algorithms to control the full scope of real-time strategy games (Jaidee, et al., 2013; Weber, et al., 2010). The state contain a lot of data about the number of resources, detail information about each units of each classes of each teams, information about the map, etc. In the same way, the action-space of RTS game is also very huge. The set of all possible action is composed of building various structures, upgrading structures, upgrading abilities of each classes, training units, attack some unit, harvest resources, etc. In RL, its space is not just state-space or action-space, but state-action-space. Therefore, without some add-on technique, reinforcement learning method alone is not practical for any experiments on real-time strategy games. Therefore, before I built a GDA integrated with learning, I thought it was a better idea to investigate a method to handle the mentioned problems first.

CLASS_{QL} (Jaidee, et al., 2013) is a multi-agent model for playing real-time strategy games (see Section 8.1). It was introduced to reduce the problem of space in RL for 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.

The last system that will be discuss in this thesis is GDA-C (Jaidee, et al., 2013), a partial GDA 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. We will discuss more details about GDA-C in Section 8.2.

All the systems that are mentioned previously are listed in Table 1-1 to show their capability and differences from other systems.

Table 1-1: List of AI systems and their outstanding points of difference

Systems	Description	Points of difference
GDA-HTNbots (Munoz-Avila, et al., 2010)	<ul style="list-style-type: none"> <input type="checkbox"/> First system on a simple implementation of GDA (with an HTN planner and using SHOP for interpretation) and its demonstration on the simple DOM game environment. <input type="checkbox"/> Assume a lot of domain knowledge to be given by the user in the form of HTN syntax. 	<ul style="list-style-type: none"> <input type="checkbox"/> No learning <input type="checkbox"/> Using HTN & SHOP <input type="checkbox"/> Lot of handcrafted domain: HTNs and GDA elements
CB-GDA (Munoz-Avila, et al., 2010)	<ul style="list-style-type: none"> <input type="checkbox"/> The first investigation on case-based planning and GDA. <input type="checkbox"/> Calculate expectation dynamically depending on action. <input type="checkbox"/> The (pending) goals that used by Goal Manager and Goal Formulator is well planned by user. 	<ul style="list-style-type: none"> <input type="checkbox"/> Using CBR, but no learning. <input type="checkbox"/> Cases are given as input.

Systems	Description	Points of difference
LGDA (Jaidee, et al., 2011)	<ul style="list-style-type: none"> ❑ An extension of CB-GDA that focuses on automated case acquisition. ❑ Learning two types of cases: expectation cases and goal formulation cases. ❑ Experimented on two domains: DOM and Wargus. 	<ul style="list-style-type: none"> ❑ Using RL & CBR to learn expectations & goal formulation cases. ❑ Goals and policies are given as input
GRL (Jaidee, et al., 2012)	<ul style="list-style-type: none"> ❑ An extension of LGDA that can learn all case bases: (Policy Case Base, Expectation Case Base, Goal Formulation Case Base). ❑ All case bases can be empty at the first run. 	<ul style="list-style-type: none"> ❑ GRL is the first system that can learn all case bases. (ICCBR 2012)
CLASS _{QL} (Jaidee, et al., 2013)	<ul style="list-style-type: none"> ❑ New RL algorithm for multiple agents. ❑ Dividing the state and action space among cooperating learning agents ❑ Each agent has its own q-table ❑ Each agent's unit has its own previous state, previous action, and previous reward for updating the q-table of its class 	<ul style="list-style-type: none"> ❑ CLASS_{QL} is an RL that can play complete Wargus game
GDA-C (Jaidee, et al., 2013)	<ul style="list-style-type: none"> ❑ A case-based goal reasoning algorithm built on top of CLASS_{QL}. ❑ Two interacting threads: CLASS_{QL} and GDA. 	<ul style="list-style-type: none"> ❑ GDA-C is the first case-based goal reasoning system that can play complete Wargus games.

CHAPTER 2 BACKGROUND

“The best thing for being sad,” replied Merlin, beginning to puff and blow, “is to learn something. That’s the only thing that never fails. You may grow old and trembling in your anatomies, you may lie awake at night listening to the disorder of your veins, you may miss your only love, and you may see the world about you devastated by evil lunatics, or know your honor trampled in the sewers of baser minds. There is only one thing for it then — to learn. Learn why the world wags and what wags it. That is the only thing which the mind can never exhaust, never alienate, never be tortured by, never fear or distrust, and never dream of regretting. Learning is the only thing for you. Look what a lot of things there are to learn.”

— T.H. White, *the Once and Future King*

2.1 Reinforcement learning

Reinforcement Learning (RL) is a learning system which can learn to map situations to actions in order to maximize a numerical reward (Sutton, et al., 1998). In most forms of machine learning, statistical pattern recognition and artificial neural networks are form of supervised learning where learning agents are told which actions they should undertake for each situation using examples provided by a knowledgeable external supervisor. This is an important kind of learning, but alone it is not adequate for learning from interaction. We are human and, since birth, we learn by interacting with our environment. Many things that we learn we do not have a teacher to tell us what to do, but constantly we interact directly with our environment.

Likewise, Reinforcement Learning agents learn to select rational actions by trial-and-error to find out which actions feedback the most reward in the long run. Therefore, RL is a powerful learning model to help deal with interactive problems. Anyhow, an RL agent must be able to receive a state signal and a numerical reward signal from the environment in some form of representation and then respond with some actions that affect the state back to the environment (Sutton, et al., 1998).

2.1.1 Policy

In Reinforcement Learning Cycle, learning of a RL agent comes from interaction between taking actions selected from current situation to the environment and observing the outcome which respond back in form of signals of new situations and

numerical values called *rewards*. In general, one of the main objectives of the agent is to try to maximize the rewards in the fullness of time.

Figure 2.1 show how an agent and its environment interact to each other. At a time step t , where $t = 0,1,2,3, \dots$, the agent chooses an action a_t from actions available in state s_t . One time step ($t + 1$) later the agent observes the environment's state s_{t+1} and a numerical reward, r_{t+1} .

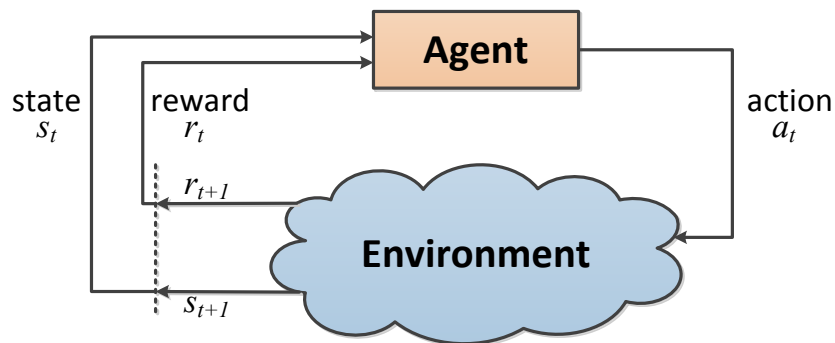


Figure 2.1: The agent-environment interaction in reinforcement learning. (Sutton, et al., 1998)

At each time step, the agent uses the numerical reward r_{t+1} that is just observed to update the probabilities of selecting action a_t from state s_t and uses the state s_{t+1} to select the next action a_{t+1} . In some RL method, state s_{t+1} is also taken into account for updating the probabilities of selecting action a_t from state s_t . Roughly speaking, a *policy*, denoted π , is a mapping from states to actions. In addition, a policy is a mapping from states to probabilities of selecting each possible action. A policy $\pi_t(s, a)$ is the probability that $a_t = a$ if $s_t = s$. The goal of the agent is to search for a best mapping for its policy to maximize the total amount of reward in the long run. (Sutton, et al., 1998).

2.1.2 Q-learning

Q-learning is a reinforcement learning methods that is designed to learn action-value function. The *action-value function* for policy π of taking action a in state s , denoted $Q^\pi(s, a)$, as the expected return starting from s , taking the action a , and thereafter following policy π (Watkins, 1989):

$$Q^\pi(s, a) = E_\pi\{R_t | s_t = s, a_t = a\}$$

The estimated value of action-value function of taking action a in state s at the time t th is denoted as $Q(s_t, a_t)$. The simplest form of Q-learning, one-step 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 from s_t , the learning rate α is such that $0 < \alpha \leq 1$, The discount factor γ is such that $0 \leq \gamma < 1$. The learning rate, denoted as α , determines to what extent the newly acquired information will override the old information. A factor of 0 will make the agent not learn anything, while a factor of 1 would make the agent consider only the most recent information. The discount factor, denoted as γ , determines the importance of future rewards. A factor of 0 will make the agent "opportunistic" by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward. The Q-Learning algorithm is shown in pseudo code below (Sutton, et al., 1998).

Algorithm 1 Q-learning: An off-policy temporal-difference² control algorithm (Sutton, et al., 1998).

```
Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
  Initialize  $s$ 
  Repeat (for each step of episode):
    Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
    Take action  $a$ , observe  $r, s'$ 
     $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
     $s \leftarrow s'$ 
  until  $s$  is terminal
```

2.2 Case-Based Reasoning

Case-Based Reasoning (CBR) is a branch of artificial intelligence, based on human problem solving, in which new problems are solved by recalling and adapting the solutions of similar past problems that are experiences stored in human memory. Roughly defined, Case-Based Reasoning is the process of solving new problems based on the solutions of similar past problems. A doctor who treats cancer patients by recalling other cases of patients who have similar symptoms is using Case-Based Reasoning. A jazz musician, who has to improvise a solo in a song that he has never played before, can do it by using his past experience and adapting it to the new song, is also using Case Based Reasoning. Not only is Case Based Reasoning a powerful method for computer reasoning, but also a dominant behavior in everyday human

² Temporal-difference (TD) methods can learn directly from raw experience without a model of the environment's dynamics and without waiting for a final outcome.

problem solving. Even more, some believe all reasoning is based on past cases that are personally experienced.

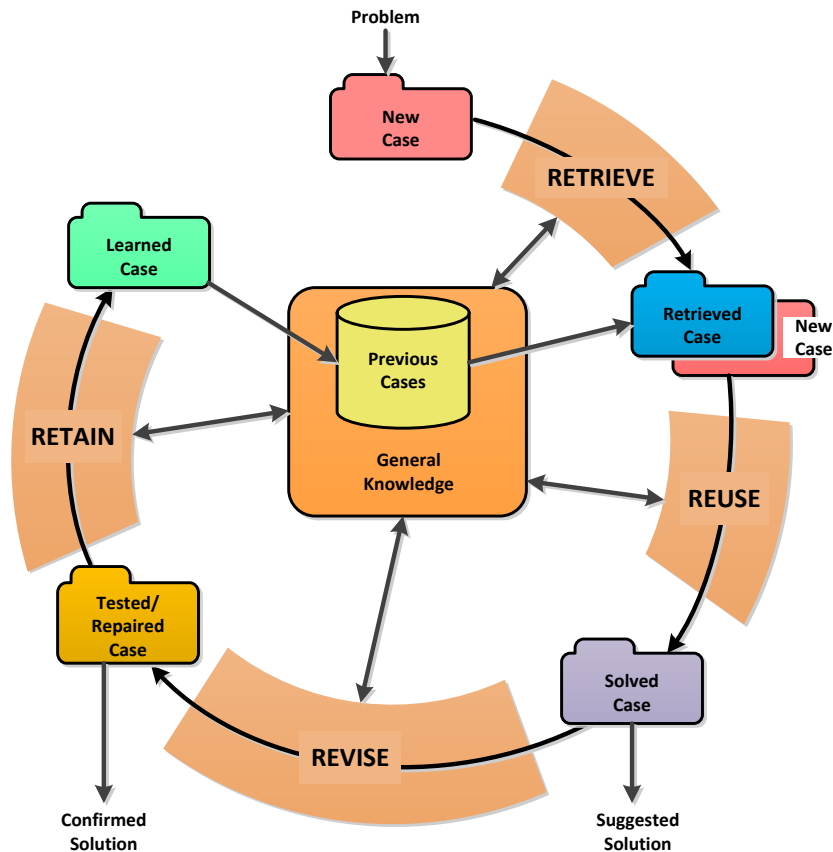


Figure 2.2: The Case-Based Reasoning Cycle (Aamodt, et al., 1994).

At the highest level of generality, a Case-Based Reasoning cycle may be described by the following four processes (Lenz, et al., 1998):

- 1) **RETRIEVE** - the most similar case or cases.

Given a target problem, retrieves from memory any cases relevant to solving it. A case consists of a problem, its solution, and in general, annotations about how the solution was derived. For example, consider an

analogous situation; suppose Jerry who lives in Allentown wants to go to New York City by driving his own car, but he has never traveled there before. The most relevant experience he can recall is one in which he used to drive to Jersey City. The directions he follows for driving to Jersey City, together with justifications for decisions made along the way, constitutes Jerry's retrieved case.

- 2) **REUSE** - the information and knowledge of the retrieved case to solve the problem.

Map the solution from the previous case to the target problem. This may involve adapting the solution as needed to fit the new situation. In the “driving to New York City” example, Jerry must adapt his retrieved solution to include the addition of driving from Jersey City to New York City.

- 3) **REVISE** - the proposed solution.

Having mapped the previous solution to the target situation, test the new solution in the real world (or a simulation thereof) and, if necessary, revise it. Suppose Jerry, who drove his car from Allentown to Jersey City and from Jersey City to New York City, found that there will be an unnecessary delay caused if he drives through Jersey City (Figure 2.3-a). If he drove most of the way to Jersey City and then used a bypass highway (in this case, Pulaski Skyway) to avoid the traffic in Jersey City, he will get to New

York City faster. This suggests the following revision: do not take the exit to Jersey City but continue on the highway (Figure 2.3-b).



(a)



(b)

Figure 2.3: Driving from Allentown to New York City.
(a) Without revising, driving from Allentown to Jersey City and then driving from Jersey City to New York City.
(b) With revising, driving from Allentown to most of the way to Jersey City and then using bypass highway to New York City.

4) **RETAIN** - the parts of this experience likely to be useful for future problem solving.

After the solution has been successfully adapted to the target problem, the agent stores the resulting experience as a new case in its memory. Jerry

records his new-found direction for driving to New York City, thereby increasing his set of stored experiences and better preparing him for the future.

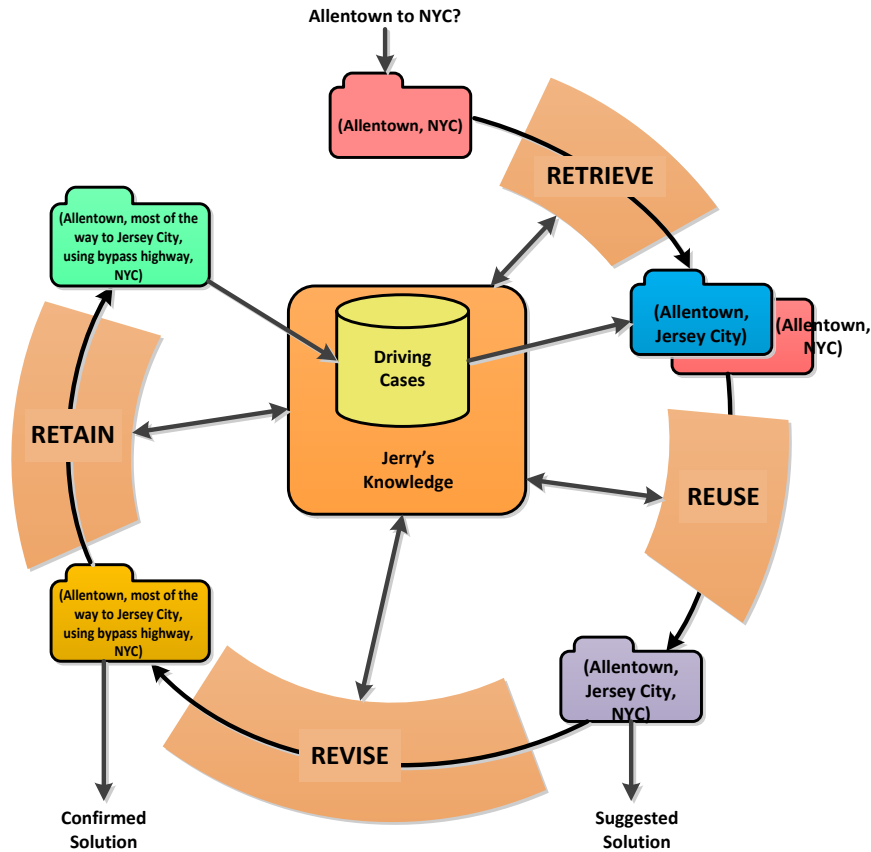


Figure 2.4: Using Case-Based Reasoning to solve the problem of driving from Allentown to New York City.

2.2.1 Case Similarity

To compare one case with other cases in case base, it is not proper way to match them using only just exact matching. Indeed, it is almost a bad idea when we think about various kinds of data in cases such as real numbers, strings or complex

structures. Case similarity is a method which can return a value of similarity between a case in a problem and cases in a case base. The value 1 is to highest similarity, while the value 0 is to lower one. There are numerous methods to calculate similarity and for the one we use in the paper is weighted hamming distance with α parameter. Let $X = (X_1, \dots, X_n)$ and $Y = (Y_1, \dots, Y_n)$ are two cases that we want to calculate similarity. First, the simple matching coefficient is the weighted simple matching coefficient that introduces a weight $\omega_i > 0$ for each attribute such that $\omega_1 + \dots + \omega_n = 1$. The weight allows expressing the importance of the attribute for the similarity (Richter, 2007).

$$\text{sim}_{H,\omega}(X, Y) = \sum_{i=1..n, X_i=Y_i} \omega_i$$

Second, the simple matching coefficient results from weighting the number of equal attribute values different than the number of unequal attribute values. This results in a non-linear strictly monotonic increasing function of the number of equal attributes. A parameter α ($0 < \alpha < 1$) determines the concrete shape of this function (Richter, 2007).

$$\text{sim}_{H,\omega,\alpha}(X, Y) = \frac{\alpha \cdot \text{sim}_{H,\omega}(X, Y)}{\left(\alpha \cdot \text{sim}_{H,\omega}(X, Y)\right) + \left((1 - \alpha) \cdot (1 - \text{sim}_{H,\omega}(X, Y))\right)}$$

CHAPTER 3

GOAL-DRIVEN AUTONOMY

We learn from failure, not from success!

— Bram Stoker

The conception of Goal-driven autonomy (GDA) is inspired by Cox’s work (Cox, 2007). The first GDA systems were reported in (Munoz-Avila, et al., 2010; Molineaux, et al., 2010). GDA is a process that integrates planning, execution, and goal reasoning for online planning in autonomous agents (Klenk, et al., 2010). Figure 3.1 illustrates how GDA extends Nau’s model of online planning (Nau, 2007). The GDA model primarily expands and details the scope of the Controller, which interacts with a *Planner* and a *State Transition System* Σ (an execution environment). System Σ is a tuple (S, A, V, γ) with a set of states S , a set of actions A , a set of exogenous events V , and state transition function $\gamma: S \times (A \cup V) \rightarrow 2^S$, which describes how the execution of an action or the occurrence of an event transforms the environment from one state to another. For example, given an action a in state s_i , γ returns the updated state s_{i+1} .

The Planner receives as input a planning problem (M_Σ, s_c, g_c) , where M_Σ is a model of Σ (the environment), s_c is the current state, and $g_c \in G$ is a goal that can be satisfied by some set of states $S_g \in S$. The Planner outputs a plan $p_c = \{A_c, X_c\}$, which is a sequence of actions $A_c = [a_c, \dots, a_{c+n}]$ paired with a sequence of expectations $X_c = [x_c, \dots, x_{c+n}]$. Each $x_i \in X_c$ is a set of state constraints corresponding to the sequence of states $[s_c, \dots, s_{c+n}]$ expected to occur when executing A_c in s_c using M_Σ .

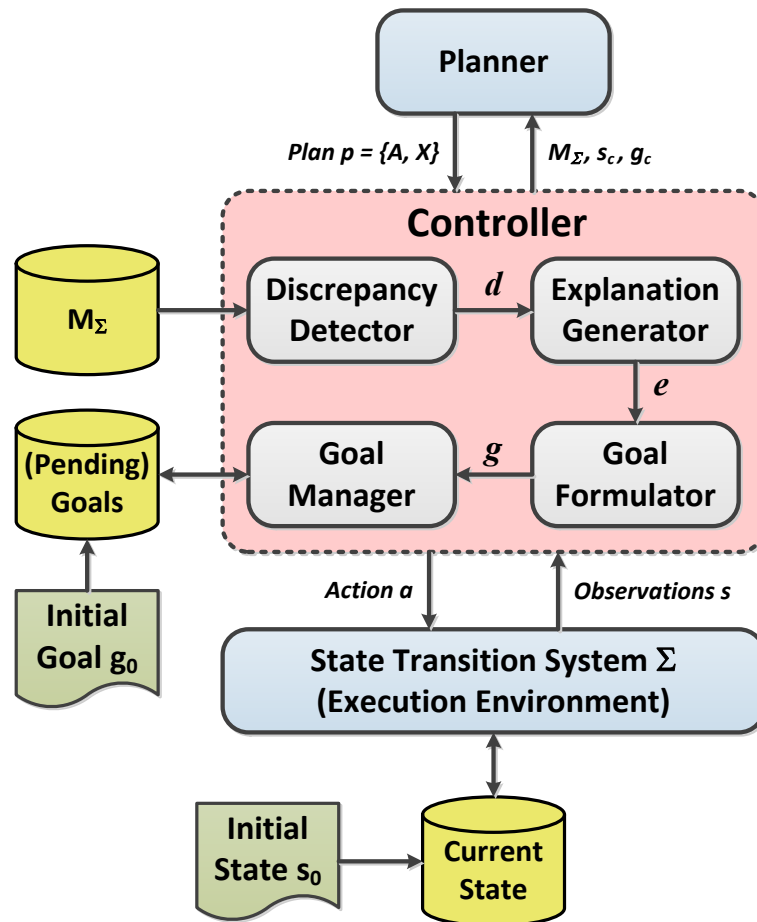


Figure 3.1: A Conceptual Model for Goal Driven Autonomy (Munoz-Avila, et al., 2010; Molineaux, et al., 2010)

The Controller sends the plan's actions to Σ and processes the resulting observations. The GDA model takes as input initial state s_0 , initial goal g_0 , and M_Σ , and sends them to the Planner to generate a plan p_0 consisting of action sequence A_0 and expectations X_0 . When executing A_0 , the Controller performs the following four knowledge intensive tasks, which distinguish the GDA model:

1. **Discrepancy detection:** This compares the observations s_c obtained from executing action a_{c-1} in state s_{c-1} with the expectation $x_c \in X$ (i.e., it tests whether any constraints are violated, corresponding to unexpected observations). If one or more discrepancies $D_c \subseteq D$ are found, then they are given to the following function.
2. **Explanation generation:** Given a state s_c and a set of discrepancies $D_c \subseteq D$, this hypothesizes one or more explanations $e_c \in E$ of D_c 's cause(s), where e is a belief about (possibly unknown) aspects of s_c or M_Σ .
3. **Goal formulation:** This creates a goal $g_c \in G$ in response to a set of discrepancies D_c , given their explanation $e_c \in E$ and the current state $s_c \in S$.
4. **Goal management:** Given a set of existing/pending goals $G' \subseteq G$ (one of which may be the focus of the current plan execution) and a new goal $g_c \in G$, this may update G' to create G'' (e.g., by adding g_c and/or deleting/modifying other pending goals) and will select the next goal $g' \in G''$ to be given to the Planner. (It is possible that $g = g'$.)

GDA makes no commitments to specific types of algorithms for the highlighted tasks, and treats the Planner as a black box. This description of GDA's conceptual model is necessarily incomplete due to space constraints. For example, it does not describe the reasoning models used by Tasks 1-4 (each of which may perform substantial inference) nor how they are obtained, it assumes multiple plans are not simultaneously executed, and it does not address goal management issues such as goal prioritization or goal transformation (Cox, et al., 1998).

Figure 3.2 illustrates the data flow of Goal Driven Autonomy. The GDA agent interacts to the environment by giving an action a_t at time t to the environment and then receiving a reward r and state s_{t+1} at the next discrete time $t + 1$ back from the environment.

Inside the GDA agent, assume at the time t , the agent receives a pair of a state s_t and a reward r_t from the environment. The discrepancy detection detects a discrepancy d between the state s_t and the expectation x_t which was predicted in the previous discrete time $t - 1$ by the expectation formulator. It then sends the detected discrepancy d to the goal formulator. The goal formulator is an independent module in the GDA agent to learn how to pick the next goal g_{t+1} from the goal base. It needs the reward r_t from the environment, the discrepancy d from discrepancy detection and the current goal g_t as its inputs to decide the next goal g_{t+1} . After that, the goal manager will take an action a_t to the environment base on the state s_t and the next goal g_{t+1} received. However, right before the agent will send the action a_t , the

expectation formulator will learn and predict the next expectation x_{t+1} by using the information from the state s_t and the action a_t .

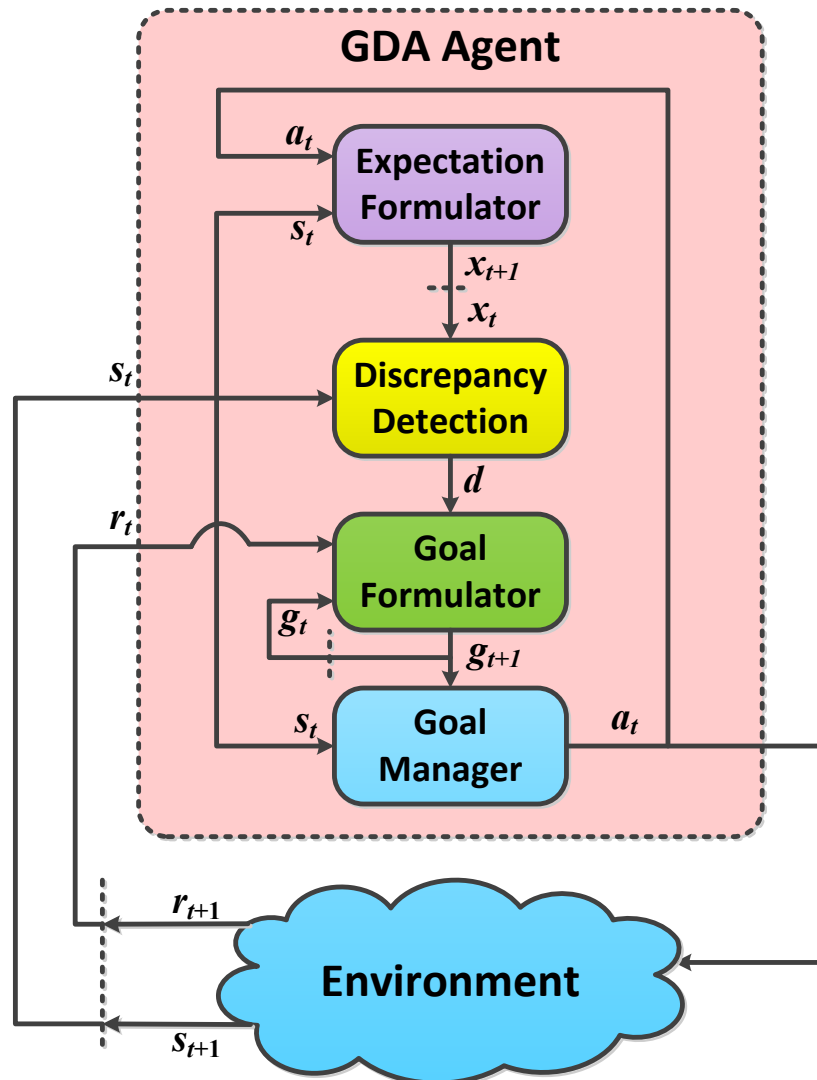


Figure 3.2: Data flow of Goal Driven Autonomy

CHAPTER 4

GOAL-DRIVEN AUTONOMY WITH HIERARCHICAL-TASK NETWORK PLANNING

4.1 Hierarchical-Task Network

The hierarchical task network (HTN) is one of many methodologies used in automated planning. The idea behind HTN is that, in the real world, many tasks can be given in the form of networks for example mathematical problems, military missions, going somewhere by using only public transportations. There are distinctive benefits of using HTN such as preventing exponential explosion during plan generation³ and faster speed of computation comparing to other planning methods.

Planning problems are identified in the hierarchical task network by providing a set of tasks, which can be (Erol, et al., 1994):

1. primitive tasks (actions that can be executed)
2. compound tasks (sequences of actions)

³ Plan generation is the problem of generating a sequence of actions that transform an initial state into some desired state (Ghallab, M.; Nau, D.S.; Traverso, P., 2004).

3. goal tasks (tasks of satisfying a condition)

The difference between primitive tasks and the others is that the primitive tasks can be executed directly. Compound and goal tasks both require a sequence of primitive tasks to be performed. However, goal tasks are specified in terms of conditions that have to be made true, while compound tasks can only be defined in terms of other tasks via the task network.

A task network is a set of tasks and constraints among them. And, it is possible that a task network can be used as the precondition for another compound or goal task. This way, one can express that a given task is possible only if a set of other actions are done in such a way that the constraints among them are satisfied. In addition, a task network can determine that a condition is necessary for a primitive action to be executed. When this network is used as the precondition for a compound or goal task, the compound or goal task requires the primitive action to be executed and that the condition must be true for its execution to successfully achieve the compound or goal task (Erol, et al., 1994).

4.2 Goal-Driven Autonomy with Hierarchical-Task Network Planner

GDA-HTNbots is an extension of HTNbots in which the controller performs the four tasks of the GDA model. HTNbots uses SHOP to generate game-playing

strategies for DOM based on an external hierarchical task network (HTN). These strategies are designed to control a majority of the domination locations in the game world. Whenever the situation changes (i.e., when the owner of a domination location changes), HTNbots generates a new plan. Therefore, HTNbots is a dynamic replanning system. It calls SHOP to find the first method that is applicable to a given task, and uses it to generate subtasks that are recursively decomposed by other methods into a sequence of actions to be executed in the environment.

Unlike HTNbots, GDA-HTNbots reasons about its goals, and can dynamically formulate which goal it should plan to satisfy. It controls plan generation in two ways: first, it determines when the planner must start working on a new goal. Second, it determines what goal the planner should attempt to satisfy. GDA-HTNbots extends HTNbots to instantiate the GDA conceptual model as follows:

- ***State Transition System*** (Σ) (task environment): We apply GDA-HTNbots to the task of controlling an agent playing DOM. We described this task and game environment in the preceding section, and describe an example of this application in the next section.
- ***Model of the State Transition System*** (M_{Σ}): We describe the state transition function for DOM using SHOP axioms and operators. Exogenous events are not directly modeled in SHOP. HTNbots play DOM by monitoring the game state and replan as needed.

- **Planner:** GDA-HTNbots uses SHOP, although other planners can be used. Given the current state s_c (initially s_0), current goal g_c (initially g_0), and M_Σ , it will generate an HTN plan p_c designed to achieve g_c when executed in Σ starting in s_c . This plan includes the sequence of expectations X_c determined by the HTN's methods that are anticipated from its execution.
- **Discrepancy Detector:** This continuously monitors p_c 's execution in Σ such that, at any time t , it compares the observations of state s_t provided by Σ with the expected state x_t . If it detects any discrepancy d_t (i.e., a *mismatch*) between them, then outputs d_t to the Explanation Generator.
- **Explanation Generator:** Given a discrepancy d_t for state s_t , this generates an explanation e_t of d_t . GDA-HTNbots tracks the history of the game by counting the number of times agents from the opposing team have visited each location. Using this information and the discrepancy d_t , GDA-HTNbots identifies an explanation e_t , which is the strategy that the opponent is pursuing.
- **Goal Formulator:** Given an explanation e_t representing the opponent's current GDA-HTNbots formulates a goal g_t using a set of rules of the form:

if e then g

The new goal g_t directs GDA-HTNbots to counter the opponent's strategy.

- **Goal Manager:** GDA-HTNbots employs a trivial goal management strategy. Given a new goal g_t , it immediately selects this as the current goal, which the Controller submits to the Planner for plan generation.

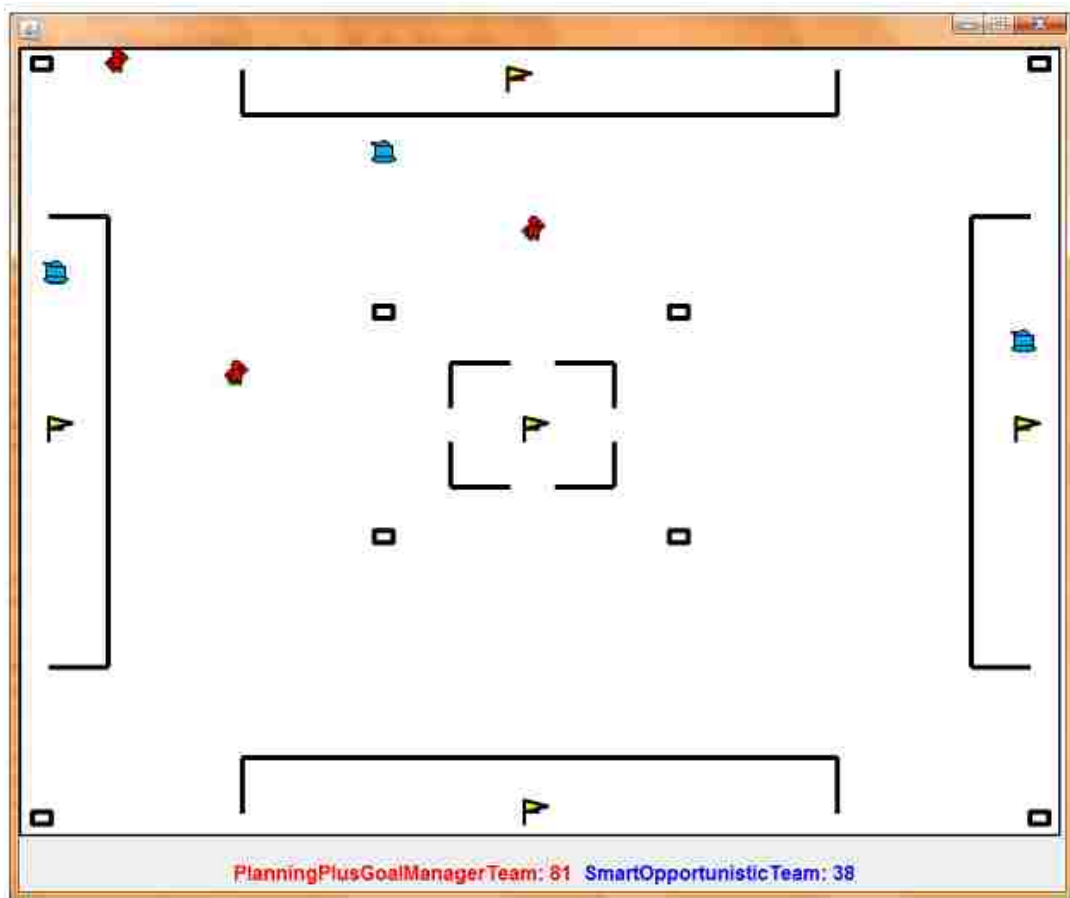


Figure 4.1: An example DOM game map with five domination locations (yellow flags), where small rectangles identify the respawning locations for the agents and the remaining two types of icons denote each player's agents.

4.3 Example in the DOM Game

We report on a case study in which the system's task is to control a team of agents in DOM. Figure 4.1 shows an example of a map in a domination game with five locations. Our scenario began with the following initial state and goal:

Initial State (s_0): This includes the locations of all the agents in the game and which team (if any) controls each domination location.

Initial Goal (g_0): The initial goal is to win the game (i.e., be the first to cumulate 20,000 points). GDA-HTNbots sends g_0 to SHOP, which generates a plan to dispatch GDA-HTNbots' agents to each domination location and control them. Given the uncertainties about the opponent's actions and the stochastic outcome of engagements, this plan may not yield the expected results. For example, Table 4-1 presents some sample explanations for the DOM game (we do not display the full state due to space constraints). The first row highlights a situation where the bot3 agent was expected to be at location 2, but this did not happen. By examining the history of enemy agents at that location, GDA-HTNbots assumes the opponent is executing a strategy to heavily defend location 2. Using the explanation goal rule set, GDA-HTNbots counters this strategy by setting a goal to have bot3 at an alternative location, namely location 1.

Table 4-1: Example explanations of discrepancies (with some expectations and observations shown), and the corresponding recommended goals.

Discrepancy	Explanation	Next Goal
$x_t: \text{Loc}(\text{bot3}, \text{loc2})$ $s_t: \neg\text{Loc}(\text{bot3}, \text{loc2})$	Defended(loc2)	Loc(bot3, loc1)
$x_t: \text{OwnPts}(t) > \text{EnemyPts}(t)$ $s_t: \text{OwnPts}(t) > \text{EnemyPts}(t)$	EnemyCtrl(loc1) EnemyCtrl(loc2)	OwnCtrl(loc2)

The second row shows a discrepancy where GDA-HTNbots expected to, over the last time period t , earn more points than the enemy. However, this did not happen because the enemy controlled two of three (total) locations. The rule set determines that the next goal should be to control one of the locations controlled by the opponent (e.g., loc2). Given this, our system generates a plan to send two agents to location 2.

This example illustrates how GDA-HTNbots explains discrepancies by reasoning about the opponent's strategies. This enables GDA-HTNbots to formulate goals that counter the opponent's actions.

CHAPTER 5

GOAL-DRIVEN AUTONOMY WITH CASE-BASED REASONING

*Experience is what you get when you didn't get what you wanted. And
experience is often the most valuable thing you have to offer.*

— Randy Pausch, The Last Lecture

In this chapter, we focus on CB-GDA, the first GDA system to employ Case-Based Reasoning (CBR) methods (Lopez de Mántaras, et al., 2005). CB-GDA uses two case bases to dynamically generate goals. The first case base relates goals with expectations, while the latter's cases relate mismatches with (new) goals.

5.1 Case-Based Goal Driven Autonomy

Our algorithm for case-based GDA uses two case-bases as inputs: the *planning* case base and the *mismatch-goal* case base. The planning case base (PCB) is a collection of triples of the form (s_c, g_c, e_c, pl) , where s_c is the observed state of the world (formally, this is defined as a list of atoms that are true in the state), g_c is the goal being pursued (formally, a goal is a predicate with a task name and a list of

arguments), e_c is the state that the agent expects to reach after accomplishing g_c starting from state s_c , and pl is a plan that achieves g_c . The mismatch-goal case base (MCB) is a collection of pairs of the form (m_c, g_c) , where m_c is the mismatch (the difference between the expected state e_c and the actual state s_c) and g_c is the goal to try to accomplish next. In our current implementation both PCB and MCB are

Algorithm 2 CB-GDA algorithm

```

CB-GDA ( $D, A, g_{init}, PCB, SIM_g(), th_g, MCB, SIM_s(), th_s, SIM_m(), th_m$ ) =
// Inputs:
//  $D$ : Domain simulator
//  $A$ : The CBR intelligent agent
//  $g_{init}$ : Initial goal
//  $PCB$ : Planning case base
//  $MCB$ : Mismatch-goal case base
//  $SIM_g()$ : return true iff goals are similar
//  $SIM_s()$ : return true iff states are similar
//  $SIM_m()$ : return true iff mismatches are similar
//  $th_g$ : the threshold value used by the  $SIM_g()$  function.
//  $th_s$ : the threshold value used by  $SIM_s()$  function.
//  $th_m$ : the threshold value used by  $SIM_m()$  function.
// Output: the final score of simulation  $D$ 

1: run( $D, A, g_{init}$ )
2: while status( $D$ ) = running do
3:    $s_i \leftarrow$  currentState( $D$ )
4:    $g_i \leftarrow$  currentGoal( $A, D$ )
5:   while  $SIM_g$ (currentTask( $A$ ),  $g_i$ ) do
6:     wait( $t$ )
7:      $e_c \leftarrow$  retrieve( $PCB, g_i, s_i, SIM_s(), th_g$ )
8:      $s_D \leftarrow$  currentState( $D$ )
9:     If  $e_c \neq s_D$  then
10:        $g_{c'} \leftarrow$  retrieve( $MCB, e_c, mismatch(e_c, s_D), SIM_m(), th_m$ )
11:       run( $D, A, g_{c'}$ )
12: return game-score( $D$ )

```

defined manually. In Section 5.2, we will discuss some approaches we are considering to learn both automatically.

The algorithm above displays our CBR algorithm for GDA, called CB-GDA. It runs the game D for the GDA-controlled agent A , which is ordered to pursue a goal g_{init} . Our current implementation of A is a case-based planner that searches in the case base PCB for a plan that achieves g_{init} . The call to run (D, A, g_{init}) represents running this plan in the game (Line 1). While the game D is running (Line 2), the following steps are performed. Variables s_i are initialized with the current game state (Line 3). And then, variables g_i is setup for agent's goal (Line 4). The inner loop continues running while A is attempting to achieve g_i (Line 5). The algorithm waits a time t to let the actions be executed (Line 6). Given the current goal g_i and the current state s_i , agent A searches for a case (s_c, g_c, e_c) in PCB such that the binary relations $SIM_s(s_i, s_c)$ and $SIM_g(g_i, g_c)$ hold and returns the expected state e_c (Line 7). $SIM(a, b)$ is currently an equivalence relation which is a Boolean relation that holds true whenever the parameters a and b are similar to one another according to a similarity metric $sim(\cdot)$ and a threshold th (i.e., $sim(a, b) \geq th$). Since the similarity function is an equivalence relation, the threshold is 1. The current state s_D in D is then observed (Line 8). If the expectation e_c and s_D do not match (Line 9), then a case (m_c, g_c) in MCB is retrieved such that mismatch m_c and mismatch(e_c, s_D), are similar according to $SIM_m(\cdot)$; this returns a new goal $g_{c'}$ (Line 10). Finally, D

is run for agent A with this new goal $g_{c'}$ (Line 11). The game score is returned as a result (Line 12).

From a complexity standpoint, each iteration of the inner loop is dominated by the steps for retrieving a case from PCB (Line 7) and from MCB (Line 10). Retrieving a case from PCB is of the order of $O(|PCB|)$, assuming that computing $SIM_s(\cdot)$ and $SIM_g(\cdot)$ are constant. Retrieving a case from MCB is of the order of $O(|MCB|)$, assuming that computing $SIM_m(\cdot)$ is constant. The number of iterations of the outer loop is $O(N/t)$, assuming a game length of time N . Thus, the complexity of the algorithm is $O((N/t) \times \max\{|PCB|, |MCB|\})$.

We claim that, given sufficient cases in PCB and MCB , CB-GDA will successfully guide agent A in accomplishing its objective while playing the DOM game. To assess this, we will use two other systems for benchmarking purposes. The first is HTNbots, which we discussed in Section 4.1. As explained before, it uses Hierarchical Task Network (HTN) planning techniques to rapidly generate a plan, which is executed until the game conditions change, at which point HTNbots is called again to generate a new plan. This permits HTNbots to react to changing conditions within the game. Hence, it is a good benchmark for CB-GDA. The second benchmarking system is GDA-HTNbots, which implements a GDA variant of HTNbots using a rule-based approach (i.e., rules are used for goal generation), in contrast to the CBR approach we propose in this paper.

CHAPTER 6

CASE-BASED LEARNING OF EXPECTATIONS AND GOAL-FORMULATION KNOWLEDGE

In this chapter, we introduce learning GDA (LGDA), a GDA algorithm that learns two types of cases from observed discrepancy resolution episodes: (1) expectation cases and (2) goal formulation cases. LGDA learns these through an integration of Case-Based Reasoning (CBR) and reinforcement learning (RL) methods.

6.1 Definitions

Let S be the set of states that an agent can visit, G the goals that an agent can pursue, and A the actions that can be executed. We define an *expectation case base* (ECB) as a mapping $S \times A \rightarrow 2^{S \times [0,1]}$ from the current state and selected action to a probability distribution over expected next states (i.e., actions can be non-deterministic). LGDA clusters ECB cases that involve the same action and have similar states, and learns a state probability distribution for each cluster.

In LGDA (detailed in Section 6.2), $\text{GET}(\text{ECB}, s, a)$ returns, as the expected state, the one with maximal probability among the ECB cluster $\chi_{s,a}$ whose state is most similar to s and has the same action a . If no such cluster exists, then $\text{UPDATE}(\text{ECB}, s, a, s')$ will create one. Otherwise, it will update the probability distribution for $\chi_{s,a}$.

A *goal formulation case base* (GFCB) instead maps the current goal g_t and discrepancy d_t into a distribution over the expected values, $G \times D \rightarrow 2^{G[0,1]}$, for formulated goals. That is, multiple goals g' may be formulated to resolve d when pursuing g . Cases with the same goal and similar discrepancies are clustered together in GFCB. LGDA uses Q-learning to track the expected value for each g' . In LGDA, $\text{GET}(\text{GFCB}, g, d, g')$ returns the expected value q from cluster $\chi_{g,d}$ when g' is formulated. If no such cluster exists, it returns 0 and initializes $\chi_{g,d}$. Function call $\text{UPDATE}(\text{GFCB}, g, d, g', q')$ updates the value of q for g' in $\chi_{g,d}$.

LGDA receives as input the policies $\Pi: G \times S \rightarrow A$ that are implemented by hard-coded *adversaries*. The call $\Pi(g)$ returns the policy π for G , while $\pi(s)$ returns the action that is pursued in state s (if multiple such actions exist, one is randomly selected). These input policies are static, not learned. LGDA instead learns the case bases (1) ECB and (2) GFCB. This learned knowledge is dynamic; their application varies based on the environment's state in which the actions are executed.

6.2 LGDA Algorithm

Expectations can be learned by recording the occurrence frequency of (s, a, x) triples. The interpretation of the most frequent triple (s, a, x) among those in (s, a) in the ECB is that, when s is the current state, it is most likely that x will be the next state when a is executed. We use s , s' , and x for the previous, current, and expected states, respectively

Let g , g' , and g'' be the previous, current, and next goals, respectively. LGDA uses Q-learning to learn the values of goals in each GFCB cluster. The following pseudo code provides details and is documented below.

Algorithm 3 LGDA algorithm

```

LGDA( $s_0, g_0, d_0, \Delta, \text{ECB}, \text{GFCB}, \alpha, \gamma, \varepsilon, G, \Pi$ )
1:  $s \leftarrow s_0; x \leftarrow s_0; a \leftarrow \phi; g \leftarrow g_0; g' \leftarrow g_0; d \leftarrow d_0$ 
2: while the game-playing episode continues
3:   wait( $\Delta$ );  $s' \leftarrow \text{GETSTATE}()$ 
4:    $\text{ECB} \leftarrow \text{UPDATE}(\text{ECB}, s, a, s')$ 
5:    $d \leftarrow \text{CALCULATEDISCREPANCY}(s', x)$ 
6:    $q \leftarrow \text{GET}(\text{GFCB}, g, d, g')$ 
7:    $r \leftarrow U(s') - U(s)$ 
8:    $q' \leftarrow q + \alpha(r + \gamma \text{ARGMAX}_{g_i \in G}(\text{GET}(\text{GFCB}, g, d, g_i)) - q)$ 
9:    $\text{GFCB} \leftarrow \text{UPDATE}(\text{GFCB}, g, d, g', q')$ 
10:  if  $r < 0$ 
11:    if  $\text{RANDOM}(1) \geq \varepsilon$ 
12:       $g'' \leftarrow \text{ARGMAX}_{g_i \in G}(\text{GET}(\text{GFCB}, g, d, g_i))$ 
13:    else  $g'' \leftarrow \text{RANDOM}(|G|)$ 
14:     $g \leftarrow g'; g' \leftarrow g''$ 
15:     $\pi \leftarrow \Pi(g'); a' \leftarrow \pi(s')$ 
16:     $x \leftarrow \text{GET}(\text{ECB}, s', a')$ 
17:     $\text{EXECUTEACTION}(a')$ 
18:     $s \leftarrow s'; a \leftarrow a'$ 
19: return  $\text{ECB}, \text{GFCB}$ 

```

LGDA initializes previous state s to the initial state s_0 , action a to the null action, previous goal g and current goal g' to the dummy goal g_0 , and discrepancy d to dummy value d_0 (Line 1). Entries with these dummy values are not added to the case base, and will be assigned to non-dummy values after the algorithm's first iteration. During a game-playing episode (Line 2), LGDA periodically waits and observes the current state s' (Line 3), which it uses to update the distribution of expected states when taking action a in s (Line 4). It then calculates the discrepancy d between the current and expected states (Line 5) and uses it to retrieve GFCB's estimated q value for formulating g' (Line 6). It then computes the reward (Line 7), updates the q value using the Q-learning formula (Line 8), and records it in the GFCB (Line 9). If the agent is performing poorly (Line 10), LGDA retrieves a new goal g'' from GFCB using ε -greedy exploration (Lines 11-13), and updates its previous and current goal (Line 14). LGDA then retrieves the next action a' using policy π , where π is the policy in Π for goal g' . (Line 15), retrieves an expected state x from the ECB (Line 16), executes a' (Line 17), and updates previous state s and action a (Line 18). Finally, when the game-playing episode ends, it returns the revised case bases.

6.3 Implementation and Example

An LGDA agent must determine how to cluster cases using a similarity metric and re-cluster when necessary. Our implementation was inspired by the design of Retaliate (Smith, et al., 2007), an RL agent that we will use for benchmarking. For

LGDA, we ensure that the states S and actions A represented in ECB and GFCB, as well as the state utility U , are the same as those in Retaliate. Theoretically, this permits a fair comparison between LGDA and Retaliate.

Retaliate was applied to control one team's actions in a domination game called DOM (see Section 0). Retaliate selects the actions for the friendly team's bots. Its state representation includes only information on domination location ownership. The *state* is a vector (l_1, l_2, \dots, l_d) , where d is the number of domination locations and l_k indicates the team which owns location k . For a 2-team game and b bots per team, this reduces the number of states to d^3 and the space of actions to $(2b)^d$. The utility U of state s is defined by the function $U(s) = F(s) - E(s)$, where $F(s)$ is the friendly team's score and $E(s)$ is the enemy's score. The discrepancy between states s and s' is a d -dimensional vector (v_1, v_2, \dots, v_d) , where v_i is true if s and s' have the same value in coordinate i and false otherwise.

Given this representation, LGDA's cases implement a simple similarity metric: two states are deemed similar if they have the same feature values for domination location ownership. Analogously, two discrepancies are similar if they mismatch on the same features. Thus, after a case is assigned to a cluster, it will never be reassigned.

Example: Suppose the domination locations in the current game are $(l_1, l_2,$ and $l_3)$ and there are three bots per team (b_1, b_2, b_3) . Each location l_i can be in one of the three states: un-owned (U), owned by the friendly team (F), or owned by the enemy

(E). Therefore, state (E, F, F) denotes that the first domination location is owned by the enemy and the others are owned by the friendly team. Suppose the previous state s is (U, E, U) , the current state s' is (F, E, U) , the expected state x is (F, F, U) , and the friendly team's previous actions were $(b_1 \rightarrow l_1, b_2 \rightarrow l_2, b_3 \rightarrow l_1)$. After updating the relevant ECB distribution (Line 4), LGDA will calculate the discrepancy d between the current and expected states (Line 5). Here, d is (true, false, true), where *true* means they match. After calculating the value and updating the (Lines 6-9), suppose the reward is negative, and that LGDA will retrieve/formulate a new goal. Suppose the current goal g' is to control the first half plus one of the domination locations, and the next goal g'' that was retrieved from the GFCB is to control all domination locations. Then the action a' that will be retrieved from policy π will be $(b_1 \rightarrow l_1, b_2 \rightarrow l_2, b_3 \rightarrow l_3)$. The expectation x that it retrieves from GFCB for executing action a is (F, F, F) (Line 16). LGDA will then execute a' and record the new values for the previous state s and action a (Line 18).

CHAPTER 7

INTEGRATED LEARNING OF GOAL-DRIVEN AUTONOMY ELEMENTS

In certain adversarial environments, reinforcement learning (RL) techniques require a prohibitively large number of episodes to learn a high-performing strategy for action selection. For example, Q-learning is particularly slow to learn a policy to win complex strategy games. We propose GRL, the first GDA system capable of learning and reusing goal-specific policies. GRL is a case-based goal-driven autonomy (GDA) agent embedded in the RL cycle. GRL acquires and reuses cases that capture episodic knowledge about an agent’s (1) expectations, (2) goals to pursue when these expectations are not met, and (3) actions for achieving these goals in given states. Our hypothesis is that, unlike RL, GRL can rapidly fine-tune strategies by exploiting the episodic knowledge captured in its cases. We report performance gains versus a state-of-the-art GDA agent and an RL agent for challenging tasks in two real-time video game domains.

7.1 The GRL Algorithm

We now present GRL, which incrementally learns expectations, goal formulation knowledge, and goal-specific policies. GRL uses Q-learning as its RL algorithm. Q-learning is frequently used as the prototypical RL algorithm due to its bootstrapping capabilities, which enables it to estimate state-action values based on other state-action values estimates. As a result, it tends to converge to optimal policies faster than other RL methods (Sutton, et al., 1998).

GRL receives as input the start state s_0 , a waiting time Δ , the Policy Case Base Π , the Expectation Case Base (ECB), the Goal Formulation Case Base (GFCB), the actions A , and some parameters. The parameters α and γ are the step-size and discount-rate parameters for Q-learning. Parameters ε_1 and ε_2 are for the ε -greedy selection of action and goals, respectively. Parameters c_a and c_b are used to learn new goals as will be explained later, and t is a threshold used to determine when two goals are similar to one another. GRL runs one episode of a game and returns updated values for Π , ECB, and GFCB.

GRL executes an iterative decision making cycle with the following steps: (1) identify discrepancies when they arise, (2) decide which goals to achieve to resolve any such discrepancies, and (3) perform actions to accomplish these goals. Simultaneously, GRL learns knowledge about state expectations, discrepancies, goals to achieve, and the actions to achieve these goals (e.g., goal-specific policies).

GRL has three phases: In Phase 1, which occurs during an episode, GRL uses and updates ECB and GFCB. Phases 2 and 3 occur immediately after an episode ends. In Phase 2 new goals are identified and in Phase 3 goal-specific policies are updated.

Algorithm 4 GRL algorithm

```

GRL( $s_0, \Delta, \Pi, \text{ECB}, \text{GFCB}, A, \alpha, \gamma, \varepsilon_1, \varepsilon_2, c_a, c_b, t$ ) =
  // Phase 1: Online execution and updating
1:   $s \leftarrow x \leftarrow s_0$ ;
    $a \leftarrow l_a \leftarrow l_b \leftarrow \emptyset$ ;
    $g \leftarrow g' \leftarrow g_0$ ;
    $d \leftarrow d_0$ ;
    $G \leftarrow \text{GETGOALS}(\Pi)$ 
2:  while episode continues
3:    WAIT( $\Delta$ )
4:     $s' \leftarrow \text{GETSTATE}()$  // Periodically observe the state
5:     $r \leftarrow U(s') - U(s)$  // Compute the reward
6:     $l_a \leftarrow \text{CONCAT}(l_a, \langle s' \rangle)$ ;
    $l_b \leftarrow \text{CONCAT}(l_b, \langle s, a, s', r \rangle)$ 
7:    ECB  $\leftarrow$  UPDATE(ECB,  $s, a, s'$ ) // Update ECB's distribution
8:     $a' \leftarrow \text{RANDOM}(|A|)$  // Random current action
9:    if  $\Pi \neq \emptyset$ 
10:      $q \leftarrow \text{GET}(\text{GFCB}, g, d, g')$  // Fetch/update Q value
11:      $q' \leftarrow q + \alpha(r + \gamma \text{ARGMAX}_{g_i \in G}(\text{GET}(\text{GFCB}, g, d, g_i)) - q)$ 
12:     GFCB  $\leftarrow$  UPDATE(GFCB,  $g, d, g', q'$ )
13:     if  $r < 0$  // Performing poorly?
14:        $d \leftarrow \text{CALCULATEDISCREPANCY}(s', x)$ 
15:       if  $\text{RANDOM}(1) \geq \varepsilon$  // Formulate next goal
16:          $g'' \leftarrow \text{ARGMAX}_{g_i \in G}(\text{GET}(\text{GFCB}, g, d, g_i))$ 
17:       else  $g'' \leftarrow \text{RANDOM}(|G|)$ 
18:        $g \leftarrow g'; g' \leftarrow g''$ 
19:        $\pi \leftarrow \Pi(g')$  // Retrieve a new policy
20:       if  $\text{RANDOM}(1) \geq \varepsilon_2$ 
21:          $a' \leftarrow \text{ARGMAX}_{a_i \in A}(\text{GET}(\Pi, \pi, s', a_i))$ 
22:        $x \leftarrow \text{ARGMAX}_{x_i \in X}(\text{GET}(\text{ECB}, s', a', x_i))$ 
23:       EXECUTE( $a'$ ) // Execute current action
24:        $a \leftarrow a'; s \leftarrow s'$ 
  // Phase 2: Goal extraction
25:   $G' \leftarrow \text{TOPFREQUENCY}(l_a, c_a, c_b)$ ;
    $G \leftarrow \emptyset$ 
26:  for-each  $g' \in G'$  // Iterate over the most frequent goals

```

```

27:       $H \leftarrow \emptyset$ 
28:      for-each  $(g, \pi) \in \Pi$     // Attempt to group  $g'$  with an existing goal
29:      if  $\text{SIMILARITY}(g, g') \geq t$  then  $H \leftarrow H \cup \{g\}$ 
30:      if  $H = \emptyset$  then  $H \leftarrow \{g'\}$  //  $g'$  is a new goal
31:       $G \leftarrow G \cup H$ 


---


// Phase 3: Policies revision
32:      for-each  $g \in G$ 
33:      if  $\Pi \neq \emptyset$  then  $\pi \leftarrow \Pi(g)$  else  $\pi \leftarrow \text{nil}$ 
34:      if  $\pi = \text{nil}$  then  $\pi \leftarrow \text{NEW}(g)$ ;  $\Pi \leftarrow \Pi \cup \{(g, \pi)\}$ 
35:      for-each  $\langle s, a, s', r \rangle \in l_b$ 
36:       $q \leftarrow \text{GET}(\Pi, \pi, s, a)$ 
37:       $q' \leftarrow q + \alpha(r + \gamma \text{ARGMAX}_{a_i \in A}(\text{GET}(\Pi, \pi, s', a_i)) - q)$ 
38:       $\pi \leftarrow \text{UPDATE}(\pi, s, a, q')$ 
39:       $\Pi \leftarrow \text{UPDATE}(\Pi, g, \pi)$ 
return  $\Pi, \text{ECB}, \text{GFCB}$ 

```

In Phase 1 (Lines 1-24), GRL applies and updates ECB and GFCB. It first initializes s and x to the initial state s_0 , action a to the null action, lists l_a and l_b to empty, g and current goal g' to the dummy goal g_0 , discrepancy d to dummy value d_0 , and G to the set of goals that can be accomplished by Π (Line 1) (i.e., policies are annotated with the goals they accomplish). During an episode (Lines 2-24), GRL periodically waits (Line 3) and then observes the current state s' (Line 4), calculates the reward r (line 5), and concatenates $\langle s' \rangle$ to l_a and $\langle s, a, s', r \rangle$ to l_b (Line 6) for use after the game episodes concludes. It then updates the distribution of expected states when taking action a in s (Line 7) and generates the current action a' randomly (Line 8). This guarantees that, if Π is empty, then GRL still has an action to perform. Otherwise (Line 9), it retrieves GFCB's estimated q value for formulating goal g' given (g, d) (Line 10), updates the new q' value using Q-learning (Line 11), and records it in the GFCB (Line 12). If the agent is performing poorly (Line 13), it then

calculates the discrepancy between the current and expected states (Line 14) and retrieves a new goal g'' from GFCB using ϵ -greedy exploration (Lines 15-17), and updates its previous and current goal (Line 18). GRL then retrieves a new policy from Π using goal g' as the index (Lines 19-21). It retrieves from the ECB the expected state x from executing a' , executes a' , and updates the previous action a , current action a' , and previous state s (Lines 22-24).

After an episode completes, GRL's Phase 2 extracts a set of goals to update their policies. It first identifies the set G' of most frequent states that appear in the most recent $c_a\%$ of visited states, where the frequency of these states must be at least a threshold value ($c_a \times c_b \times |l_a|$). For example, assume that $l_a = \{s_a, s_b, s_c, s_a, s_a, s_b, s_a, s_a, s_a, s_b\}$, $|l_a| = 10$, $c_a = 50\%$, and $c_b = 0.25$. Then the most recent 50% of l_a is $\{s_b, s_a, s_a, s_a, s_b\}$, and state s_a is the most frequent state among these (with frequency 3). The threshold value equals 1.75 (i.e., $0.5 \times 0.25 \times 10$), which means GRL will also include state s_b in G' because its frequency is 2. However, if $c_b = 0.1$, then $G' = \{s_a\}$ (Line 25). GRL then adds new goals from Π that are at least as similar to goals in G' as the threshold t (Line 29). Similarity between goals is computed using a linear combination of local similarity metrics, one for each of the state's features (Lopez de Mántaras, et al., 2005). More precisely, we assume cases to be vectors of n -dimensional features $X = \{x_1, \dots, x_n\}$. For computing similarity, we define a collection of local similarity metrics $\text{sim}_i()$, one per feature i , and a collection of weights α_i , which sum to 1. The aggregated similarity metric SIM_{agg} is defined as:

$$\text{SIM}_{\text{agg}}(X, Y) = \sum_{i=1}^n \alpha_i \cdot \text{sim}_i(x_i, y_i)$$

GRL groups goals by similarity to reduce the size of the Policies Case Base Π . However, if no similar goals exist, then GRL will interpret g' as a new goal (line 30).

In Phase 3, GRL refines or adds new policies. For each goal g in G (Line 32), if Π is not empty, then GRL will retrieve policy $\pi \in \Pi$ for this goal (Line 33). If either Π is empty or the policy associated with g is nil, a new policy π for g is created and π is added to Π (Line 34). It will then apply Q-learning to update π using the recent state transitions and rewards (Line 35-38) and update the (goal, revised policy) in Π (Line 39). Finally, GRL returns all the revised case bases (Line 40).

7.2 Example

Suppose in the real-time strategy (RTS) game Wargus a GRL-controlled agent is competing against one opponent (Wargus2012). Wargus is a combat game where each player controls a variety of units. In our experiments each player controlled mages, archers, knights, ballistae, and footmen. The objective of this game is to be the first to reach a predefined number of points, which are earned by killing the opponent's units. Some units award more points than others (e.g., killing a knight earns more points than a footman).

Assume each team begins with two footmen and two archers, and that the agent has already played many games. Thus, the case bases Π , ECB, and GFCB have recorded some results. For the Wargus state representation we use $s' = (u_1, u_2, \dots, u_n, e_1, e_2, e_m)$, where $u_i \in \mathbb{Z}$ denotes the number of remaining units of type i^{th} on our team and $e_i \in \mathbb{Z}$ denotes the same of type j^{th} for the opponent. Usually, n and m are equal (e.g., if the current state equals $(2,1,0,2)$, then our team has 2 footmen and one archer remaining while the opponent has only two archers). Actions in Wargus, denoted as $a' = (b_1, b_2, \dots, b_k)$, where each b_i is a unit type of the opponent such as $\{R = \text{archers}, F = \text{footmen}\}$, means that units of type i^{th} on GRL's team attacks opponent units of type b_i^{th} . For example, the action (R, F, F, \emptyset) means that a unit with ID 1 attacks an opponent archer, units with ID 2 and 3 attack opponent footmen, and units with ID 4 do nothing.

Suppose the current state s' is $s_{21} = (1,0,0,1)$ (i.e., GRL's team has only one footman left and the opponent has only one archer) and $r = -2$ (Lines 4-5). In Phase 1, GRL adds $\{s_{21}\}$ to l_a and $\{(s_{20}, a_{20}, s_{21}, -2)\}$ to l_b (Line 6). After updating the appropriate ECB distribution (Line 7), GRL will generate the random action a' and then calculate and update the q value of GFCB (Lines 10-12). Because the reward is negative, GRL will change to a new goal (Line 13). After finding the discrepancy $d_{21} = (T, T, T, F)$ between current state s_{21} and expectation $x_{21} = (1,0,0,0)$, it will choose a new goal g'' in an ε -greedy fashion (Lines 14-18). Using this new goal to retrieve a policy $\pi \in \Pi$, suppose it retrieves (by chance) greedy action $a'_{21} = (\emptyset, R, \emptyset, \emptyset)$ (i.e., send the remaining footman to attack an enemy archer) from policy

π (Lines 19-21). GRL then updates the previous state and action, computes expectation x , and executes action a' (Lines 22-24). Suppose that this action eliminates the opponent's units, which ends the game.

Phases 2-3 update Π . First, GRL uses TopFrequency to compute a set of new goals G' , and then searches for goals from Π that are similar to any members in G' to create a set G (Lines 25-31). Suppose $G' = \{(100,150,0,0)\}$ (e.g., we have 100 footmen and 150 archers while the enemy has 0 footmen and 0 archers), meaning that GRL won the episode because it destroyed all enemy units. Assume $\Pi = \{((96,96,0,0), \pi_1), ((0,0,10,20), \pi_2), ((150,100,0,0), \pi_3), ((105,104,0,0), \pi_4)\}$. Then $G = G'$ assuming there are no (sufficiently) similar cases in Π . Lines 39-48 will learn a policy π_5 , and $\{((100,150,0,0), \pi_5)\}$ will be added to Π . On the other hand, if $G' = \{(100,100,0,0)\}$ and assuming it would be (above-threshold) similar to the first and fourth goals in Π , then $G = \{(96,96,0,0), (105,104,0,0)\}$ and Lines 39-48 will update the policies π_1 and π_4 .

CHAPTER 8

GOAL-DRIVEN AUTONOMY COORDINATION OF MULTIPLE AGENTS

We, humans, can share knowledge that we have learned. Salespersons at the end of the week can have a meeting to share their selling experience, the problems that they had with their customers and other issues. By doing this, it helps those salespersons in the group to indirectly learn more and be ready for similar situations that might happen in the future. We already knew from Section 2.1 that a reinforcement learning (RL) agent learns directly by interacting with its environment without any supervisor. However, can reinforcement learning agent indirectly learn from other agents? Another issue, a well-known limitation of RL is that the space of possibilities is too large. This can be exemplified in many complex problems. For example, the state-space of a real-time combat strategy game called Wargus is very large. The state contains a lot of data about the number of resources, detailed information about each unit of each class of each team, information about the map, etc. In the same way, the action-space of Wargus is also very huge. The set of all possible action is composed of building various structures, upgrading structures, upgrading abilities of each class, training units, attacking some unit, harvesting resources, etc. In RL, its space is not just state-space or action-space, but state-action-

space. Therefore, without some add-on technique, reinforcement learning method alone is not practical for any experiments on real-time strategy games.

8.1 CLASS_{QL}: Modeling Unit Classes as Agents

We would like to introduce CLASS_{QL}, an application of the RL algorithm that learns directly and indirectly from the environment and reduces the problem of space that occurs in RL for real-time strategy games. In this dissertation, we show that we can reduce the size of the state-action space by having an individual Q-table of each class and filtering useful information that is customized for each class instead of using the observed state directly.

8.1.1 The CLASS_{QL} Algorithm

The multi-agent CLASS_{QL} manages a collection of learning agents, one for every class of unit/buildings. Each CLASS_{QL}'s agent performs a feedback loop with the environment, which is typical of reinforcement learning agents (Figure 8.2). 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 and translated into multiple problem domain actions.

CLASS_{QL} uses the following conventions:

- The *set of classes* \mathbb{C} is the set of n classes $\{C_1, C_2, \dots, C_n\}$, where each C_i is the i^{th} class of units in the problem domain. For example, as for Wargus game, we assign a class C_1 to control all footmen, another class C_2 to control all knights, and another class C_3 to control all peasants.
- The *set of class-actions* \mathbb{A} is the set of $\{A_1, A_2, \dots, A_n\}$, where A_i is the set of high-level actions that the units in the i^{th} class can perform.
- The set of class-states $S = \bigcup_{1 \leq k \leq n} S_k$ where each S_i is the set of states that the units in the i^{th} class can be at.
- The *set of learning-matrixes* \mathbb{M} is the set of $\{M_1, M_2, \dots, M_n\}$, 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 CLASS_{QL}, 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 α controls the learning rate ($0 < \alpha \leq 1$), and γ is a factor discounting the rewards obtained so far ($0 \leq \gamma < 1$).

Each class C controls all units of the same type. Each individual unit maintains the (state, action) pair of the state it last visited and the action it last performed. However, a unit is not able to make a decision on itself. The class's learning matrix (i.e., a Q-table) for the C class acts like a "brain" of each unit of class C . Figure 8.1 illustrates this idea.

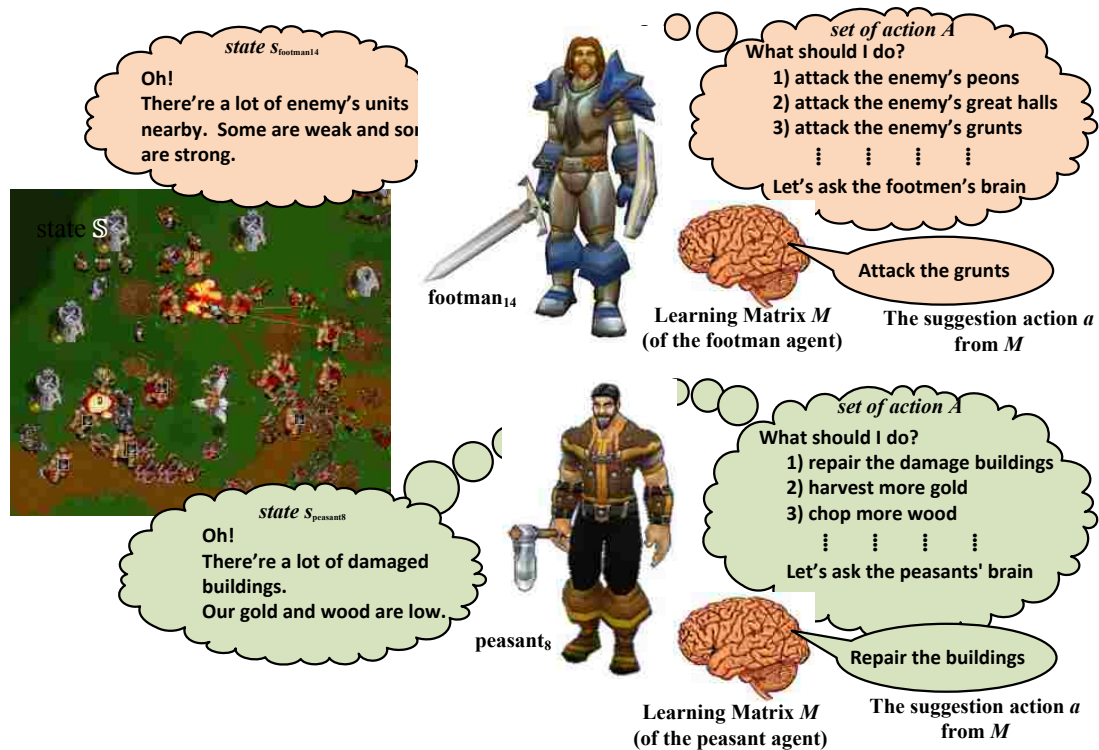


Figure 8.1: Illustration how units in Wargus perform a decision making

Algorithm 5 CLASS_{QL} algorithm

```

CLASSQL( $\mathbb{C}$ ,  $\mathbb{A}$ ,  $\mathbb{M}$ ) =
1: while episode continues
2:   parallel for each class  $C \in \mathbb{C}$ 
3:      $s \leftarrow \text{OBSERVESTATE}()$ 
4:      $r \leftarrow \text{OBSERVEREWARD}()$ 
5:      $s \leftarrow \text{GETSTATEFORCLASS}(s, C)$ 
6:      $A \leftarrow \text{GETACTIONSFORCLASS}(\mathbb{A}, C)$ 
7:      $M \leftarrow \text{GETLEARNINGMATRIXFORCLASS}(\mathbb{M}, C)$ 
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:      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:      EXECUTEACTION( $L$ )
return  $\mathbb{M}$ 

```

CLASS_{QL} receives as inputs: a set of classes \mathbb{C} , a set of class-actions \mathbb{A} , and a set of Q-tables \mathbb{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 CLASS_{QL} agent.

Loop of each CLASS_{QL} agent. The state and reward are observed directly from the environment (Line 3-4). Then, the observed state s is filtered specifically for class C (Line 5). The reason why we have to do this is 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 \mathbb{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 \mathbb{M} is returned (Line 18).

8.1.2 Application of CLASS_{QL} for Wargus

The CLASS_{QL} algorithm that we describe in the previous section is applicable in any problem domain. However, if we would like to use the CLASS_{QL} algorithm for some particular problem domain, we might need to adapt the CLASS_{QL} algorithm according to the characteristics of the target problem domain we want to use. In this section, we will show how to apply CLASS_{QL} algorithm to Wargus.

As for Wargus, at the beginning of the game, units on the team take a lot of strategic actions, such as building structures, harvesting resources, and training units to prepare for battle. However, the scores of the game for all teams are still zero. This is because scores in Wargus will be increasing only from attacking the enemy team's

units (See Appendix B, for more detail about points gained from killing specific units). Without an immediate reward after taking an action, we cannot use any online reinforcement learning. This is why we have to save the list of state-action pairs to do an off-line update after the game ends (and we know that is the final score). The Q-learning update formula that is mentioned previously need four parameters $(s_t, a_t, r_{t+1}, s_{t+1})$ for updating the q-values. The reward's value r_{t+1} is a function of the final score as we will explain later.

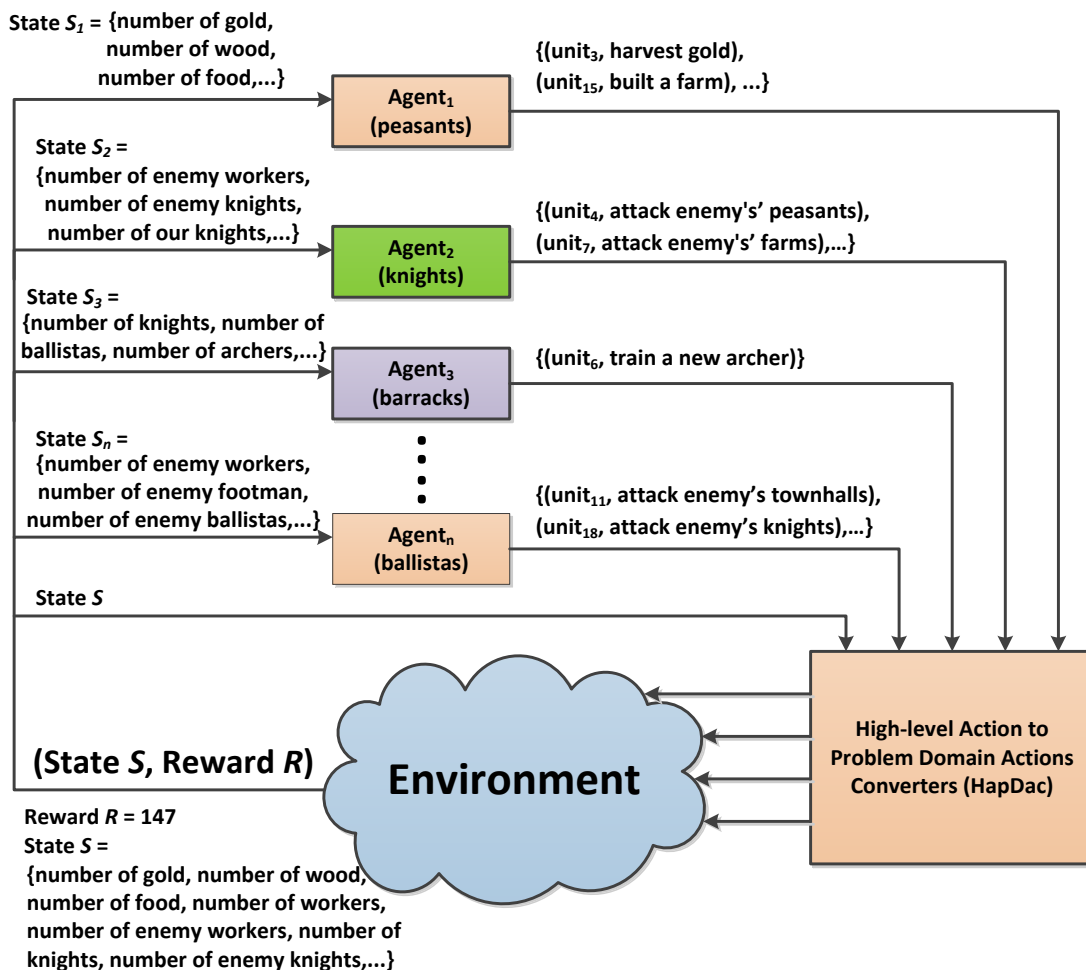


Figure 8.2: The interaction between CLASS_{QL} and its environment

Another reason why we do not store rewards at each time in the tracking list is because of the cooperation among different agents. The CLASS_{QL} agents cooperate because they all share the same reward.

Reward function. We decided to use reward $+1$, if the CLASS_{QL} agent's score is greater than the opponent's score; -1 , if the CLASS_{QL} agent's score is smaller than the opponent's score, and 0 , otherwise. We do not use the difference in score as the reward because, in Wargus, the score comes from killing enemy' units; therefore, a stronger team which can win and finish the game very fast can earn less score than the score of a weaker team who can win but take a longer time.

Using offline update has another benefit; because we are recording the list of (state, action) pairs, we can go back to early decisions in the game and assign the final reward.

CLASS_{QL} works in two phases. In the first phase, we use the Q-values learned in previous episodes to control the AI while playing the game. In the second phase, we update the Q-values from the sequence of triples (s, a, s') that occurred in the episode that just completed.

CLASS_{QL} initializes s to the initial state s_0 (Line 1). During an episode (Line 2), CLASS_{QL} periodically waits (Line 3) and then observes the current state s' (Line 4). Each class C in the set of classes \mathcal{C} (Line 5), creates the current state s' for the class C by customizing the observed state s' (Line 6). The reason why we have to do this is

that different classes need different kinds of information. The size of the observed state s' is quite expansive and contains various kinds of information. Each class requires some of the information uniquely.

Algorithm 6 CLASS_{QL} algorithm for Wargus

```

CLASSQL( $s_0, \Delta, Q, \mathcal{C}, \mathcal{A}, \alpha, \gamma, \varepsilon$ ) =
1:  $s \leftarrow s_0$ 
2: while episode continues
3:   wait( $\Delta$ )
4:    $s' \leftarrow \text{GETSTATE}()$ 
5:   parallel for each class  $C \in \mathcal{C}$ 
6:      $s' \leftarrow \text{GETABSTRACTSTATE}(s', C)$ 
7:      $A \leftarrow \text{GETVALIDACTIONS}(\mathcal{A}_C, s')$ 
8:      $Q \leftarrow Q(C)$ 
9:     for each unit  $c \in C$ 
10:      if unit  $c$  is idle
11:        if  $\text{RANDOM}(1) \geq \varepsilon$ 
12:           $a \leftarrow \text{ARGMAX}_{a' \in A}(Q(s', a'))$ 
13:        else
14:           $a \leftarrow \text{RANDOM}(A)$ 
15:          EXECUTEACTION( $a$ )
16:           $L_c \leftarrow \text{CONCAT}(L_c, \langle s_c, a_c, s' \rangle)$ 
17:           $s_c \leftarrow s'$ 
18:           $a_c \leftarrow a$ 
19:   end-while

  //----- After the game is over, update the q-tables -----
20:  $r \leftarrow \text{GETREWARD}$ 
21: for each class  $C \in \mathcal{C}$ 
22:    $Q \leftarrow Q(C)$ 
23:   for each unit  $c \in C$ 
24:     for each  $\langle s, a, s' \rangle \in L_c$ 
25:        $A \leftarrow \text{GETVALIDACTIONS}(\mathcal{A}_C, s')$ 
26:        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \text{ArgMax}_{a' \in A} Q(s', a') - Q(s, a)]$ 
27: return  $Q$ 

```

The next step is to create the set of valid actions A of class C under current situation (the current state s') (Line 7). We should not use \mathcal{A}_C (the set of possible

actions of class C) directly because some of the actions might not be applicable in the current state. For example, peasants can build farms. However, without enough resources, Wargus will ignore this command. Therefore, Line 7 prunes invalid actions. Any action randomly chosen from this set of actions is guaranteed to be a valid action. Next, the Q-table of class C is retrieved from the collection of Q-tables \mathcal{Q} (Line 8).

For each unit c of class C , if the unit c is idle, CLASS_{QL} retrieves an action a from the Q-table using ϵ -greedy exploration (Line 9-14). Notice that the algorithm chooses an action from the set of valid actions A , not from the set of possible actions \mathcal{A}_C . Then, the action a is executed (Line 15).

Because this is an offline learning method, Line 16 saves the set of s_c (the previous state s of unit c), a_c (the previous action a of unit c) and the current state s' for the Q-learning updates in the second phase. We wait until the end of the game to update the Q-values because we have found it experimentally to be more effective to use the outcome at the end of the game. This is why we have to save the list of state-action to perform the off-line update later. Afterward, we update the previous state s_c and the previous action a_c (Line 17-18).

As far as the reward is concerned, we would actually like to update the Q-table on the fly. However, the test base that we experiment on has some interesting characteristics. Since we use the score of the game to decide who the winner is, the

score for the game itself deserves to be rewarded or a part of a utility function to calculate the reward.

In the second phase, after calculating the reward r (Line 20), we use one-step Q-learning update and all the members in the list L_c of each individual unit c to update Q-values of the Q-table of each class C (Line 21-26). Finally, the set of Q-values is returned (Line 27).

8.1.3 Modeling Wargus in CLASS_{QL}

People with different profession have different roles and duties. For example, a president, a merchant, a scientist and a preacher, they have different duty in society. Thus, when they look into the world around them, they observe the world in different perspective. People are trend to carefully focus on the details of information that they need for their working and living. In addition, agents in CLASS_{QL} also behave the same way. When different agents observe information from their environment, they filter only some information that is important for their jobs. Peasants are just civilian. And, their jobs are about farming, harvesting resources, building and repairing structures. Thus, when they observe a state from environment, the information that they need are number of gold, wood and food that their team have, and number of barracks that are already built, etc. In the other hand, knights are warrior. Their duties are attack and defense enemies. So, the information that they need to focus on are the number of enemy workers, the number of enemy knights, and the number of knights in their team, etc.

We categorized units of a team into different class base on their duties. As for Wargus, we modeled units into 12 classes as list below:

1. Town Hall / Keep / Castle
2. Black Smith
3. Lumber Mill
4. Church
5. Barrack
6. Knight / Paladin
7. Footman
8. Elven Archer/ Elven Ranger
9. Ballista
10. Gryphon Rider
11. Gryphon Aviary
12. Peasant-Builder

Because the behaviors of peasants when they act as builders or harvesters are so different, we separate them in two different subsets. Harvesters can become builders to build some buildings/structures until the work is done. Then, they will return to working as harvesters again. There are two main kinds of resources to harvest: gold and wood. To harvest gold, peasants must find a path to a gold mine. To harvest wood, peasants must find a path to a forest. All resources which are already harvested, peasants will carry back with them to their nearest town-hall or great-hall

depending on their races. In some situations, the peasants also can act as repairers when a building is attacked.

There is no stable class in the list above because stable has no action, so the Q-table is not needed by the stable class. The peasant-harvester class also does not have its own Q-table. We create simple algorithms for the harvesters' job assignments to maintain the ratio of gold to wood at about 2:1. There are a few other missing classes such as Mages because usually they don't seem to work well when controlled using Wargus commands.

In Wargus, the sets of actions of each class are exclusive. So, we can make the state space of Q-table smaller by having individual Q-table of each class of unit. All q-values in each Q-table are zero initialized. However, each unit has its own previous state, previous action, and reward. Because each unit creates and finishes its action in a different game cycle, in each game cycle the previous state of each unit is different.

8.1.4 State Representation

Each unit type has different state representation. To reduce the number of states, we generalize levels for features that have too many values. For example, the amount of gold can be any value greater than zero. In our representation we have 18 levels for gold. Level 1 means 0 gold whereas level 18 means more than 4000 gold. We used the term "level number" for such generalizations. Here are the features of the state representations for each class:

- Peasant-Builder: (level of number of gold, level of number of wood, level of number of food, number of barracks, having lumber mill?, having blacksmith?, having church?, having Gryphon aviary?, having a path to a gold mine?, having a town hall?)
- Footman, knight, paladin, archer, ballista and Gryphon rider: (level of number of our footmen, level of number of enemy footmen, number of enemy town hall, level of number of enemy peasant, level of number of enemy attackable units that are stronger than our footmen, level of number of enemy attackable units that are weaker than our footmen)
- Town hall: (level of number of food, level of number of peasants)
- Barrack: (level of number of gold, level of number of food, level of number of footman, level of number of footmen, level of number of archers, level of number of ballista, level of number of knights/paladins)
- Gryphon Aviary: (level of number of gold, level of number of food, level of number of Gryphon Rider)
- Black Smith, Lumber Mill and Church: (level of number of gold, level of number of wood)

For peasants who are harvesters, we do not use any kinds of learning methodology to choose actions to take. We use some simple algorithms to balance between the amount of gold and the amount of wood about two units of gold per one unit of wood. Another work of harvesters is to repair damage structures if there are some that need to be repaired.

8.1.5 Actions

These are examples of actions that agents use to communicate with Wargus.

- Build(unit₈, location(25,46), FARM)
- Attack(unit₂,unit₅)
- Attack(unit₄, location(122, 59))
- Wait(unit₁₁)
- Harvest(unit₇, location(91,83))

The actions above are needed to specify a unit's ID or a point of location on the map. Thus, the size of the action space for these actions varies by the number of units and size of maps. However, this issue is not bad enough. For any kind of learning agent, agent who learns these actions may be not able to use its learned actions to other maps that have different landscapes. To understand this phenomenon, consider an analogous situation; a snowboarder learns how to snowboard very well at a mountain ski resort. When he goes snowboarding in other ski resort, he cannot apply what he already learned from his first mountain to any new mountain. He has to start learning all over again. When you finished reading this scenario above, you know this is unlikely to happen in real life. This is because, if you have the skill to snowboard, no matter which ski resort you go to, you are still able to snowboard.

In addition, we should not learn how to act based on the actions that are specified for a particular map. What we should do is learn how to act from a set of actions that are independent from any maps.

Our model abstracts actions from units so that they are at a higher level than the actual actions the units can take. The actual actions of units include moving to some location, attacking another unit or building, patrolling between two locations, and standing ground in one location. We call these actions problem-domain actions because are actions that are given by the problem domain and can be directly executed in the domain. However, using the problem-domain actions would lead to an explosion in the size of the Q-tables. High-level actions are actions that are conceptual actions and group problem-domain actions. High-level actions can be independent from any particular map and may be effective to reuse in other maps. However, Wargus cannot execute high-level actions. Therefore there is a module called *High-level Action to Problem Domain Actions Converters (HapDAC)* to convert high-level actions into problem domain actions that Wargus can understand (se Figure 8.2,). Table 8-1 shows all possible high-level actions for each class. To convert from a high-level action to problem-domain actions, HAPDAC might need more information from the environment. Therefore, in Figure 8.2, HAPDAC also receives the current state S that is observed from the environment as its input. Table 8-1 shows all possible high-actions for each class.

The following are the principles that HAPDAC uses for mapping high-level actions into problem-domain actions. Let U_a denotes the set of actor units and U_r denotes the set of recipient units or target locations.

- (a) Using one-to-one mapping: for every recipient unit of U_r we assign at most one actor unit of U_a , if $|U_a| < |U_r|$.
- (b) Using onto mapping: for every recipient unit of U_r we assign at least one actor unit of U_a , if $|U_a| > |U_r|$.
- (c) Using bijective mapping (one-to-one and onto): for every recipient unit of U_r we assign one actor unit of U_a , if $|U_a| = |U_r|$.

In our current work, HAPDAC uses a simple modulo function from U_a to U_r without considering any other factors such as distance between an actor unit and a recipient unit.

Table 8-1: All possible high-level actions for each Wargus class.

Class	Actions
Peasant-Builder	<ul style="list-style-type: none"> • build a farm • build a barrack • build a town hall • build a lumber mill • build a black smith • build a stable • build a church • build a Gryphon aviary
Town-Hall Keep Castle	<ul style="list-style-type: none"> • train a peasant • Upgrade to keep (when it is a town-hall.)

Class	Actions
	<ul style="list-style-type: none"> • Upgrade to castle (when it is a keep.)
Black Smith	<ul style="list-style-type: none"> • upgrade sword level 1 • upgrade sword level 2 • upgrade human shield level 1 • upgrade human shield level 2 • upgrade ballista level 1 • upgrade ballista level 2
Lumber Mill	<ul style="list-style-type: none"> • upgrade arrow level 1 • upgrade arrow level 2 • Elven ranger training • ranger scouting • research longbow • ranger marksmanship
Church	<ul style="list-style-type: none"> • upgrades knights to paladins • research healing • research exorcism
Barrack	<ul style="list-style-type: none"> • train a footman • train an Elven archer/ranger • train a knight/paladin • train a ballista
Footman Archer Ranger Knight Paladin Ballista Gryphon Rider	<ul style="list-style-type: none"> • 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 • and attack all enemy's units that have their range of attacking 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 HP less than those of us) • attack all enemy's units (no matter what kind) • Break walls to make path to enemy

The list of all possible actions for each class mentioned previously is not the list that CLASS_{QL} will choose from. It is true that the AI agent can order a peasant to build a farm anytime. However, without enough resources or under some conditions, after the agent ordered a peasant to build a farm, the Wargus game denial or ignore the command and this makes the agent lose its turn for nothing. Therefore, in each point of time, the agent will filter all possible actions to a new set of valid actions that are available for the current state to make sure no matter what an action that the agent picked, that action is always valid.

8.1.6 Analysis of CLASS_{QL}

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

$$\pi_i(s) = \underset{a}{\text{ArgMax}}\{M_i(s, a) | a \in A_i\}$$

That is $\pi_i(s)$ picks the action a that has the maximum value $M_i(s, a)$ (π_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 $1 - \epsilon$, where ϵ is probability of random action in ϵ -greedy policy). However, to guarantee that optimal policies are learned, it will from sometimes pick a random action (with a probability ϵ). Assume that each agent i has learned an optimal policy π_i . That is,

when following the policy π_j , 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 π_1, \dots, π_n where each π_k maximizes the returns for class k , then $\pi = (\pi_1, \dots, \pi_n)$ is an optimal policy in $S \times (A_1 \times A_2 \times \dots \times A_n)$ (where $S = \bigcup_{1 \leq k \leq n} S_k$ as defined in the previous section). This means that agents will coordinate despite the fact that each agent 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 8.3 shows a typical timeline at the beginning of the game after CLASS_{QL} 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; the decision by the town hall agent to create the second peasant enables the peasant agent to order this peasant to produce the barracks.

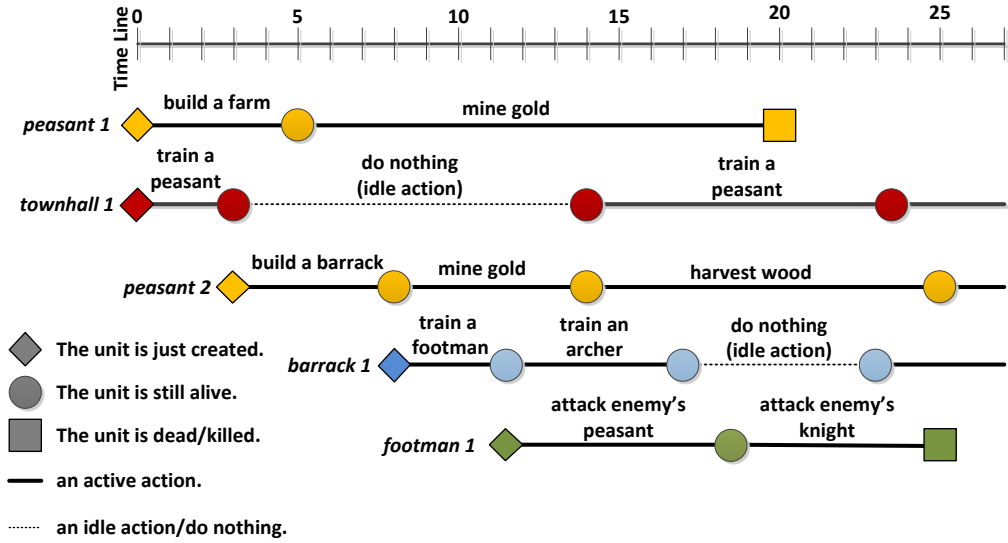


Figure 8.3: An example of a timeline at the beginning of the Wargus game.

The CLASS_{QL} agents require $|S_1 \times \mathcal{A}_1| + \dots + |S_n \times \mathcal{A}_n|$ space (i.e., adding the memory requirements of each individual agent α_k). In contrast, an agent reasoning with the combined states and actions would require $|S \times \mathcal{A}|$ space. Under the assumption that $\forall(i, j)[i \neq j \rightarrow \mathcal{A}_i \cap \mathcal{A}_j = \{\} \wedge S_i \cap S_j = \{\}]$ hold, then the following inequality holds:

$$|S \times \mathcal{A}| \geq |S_1 \times \mathcal{A}_1| + \dots + |S_n \times \mathcal{A}_n|,$$

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 type can take are typically disjoint from the actions of units of a different type. The following table summarizes some of the savings for these assumptions:

Table 8-2: Space saved by CLASS_{QL} compared to a conventional RL agent.

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

8.2 GDA-C: Case-Based Goal-Driven Coordination of Multiple Learning Agents

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 or control only a small set of decision-making tasks within a larger hard-coded system that plays the full game.

To address this limitation, GDA-C was introduced. GDA-C is 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, who 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 $|S_k| \times |\mathcal{A}_k|$, 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 $|S_1||\mathcal{A}_1| + \dots + |S_n||\mathcal{A}_n|$. This is a substantial reduction in memory requirements compared to a system that must reason with a space of size $|S||\mathcal{A}|$, where $S = \bigcup_{1 \leq i < n} S_i$ and $\mathcal{A} = \bigcup_{1 \leq i < n} \mathcal{A}_i$ (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.

The claim is 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 an empirical evaluation using the Wargus RTS environment was conducted to compare the performance of GDA-C versus CLASS_{QL}, an ablation of GDA-C where the RL agents coordinate by sharing only the same reward function. First, GDA-C and CLASS_{QL} were compared 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. Their performances were compared in direct competitions. The main findings are:

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

8.2.1 Multi-Agent Setting

The task focusing on is to control a set Γ of agents $\alpha_1, \dots, \alpha_n$, where each belongs to one class c_k in $\mathcal{C} = \{c_1, c_2, \dots, c_n\}$. 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., $S = \bigcup_{1 \leq k \leq n} S_k$). 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 $\{(a_1, p_1), \dots, (a_n, p_n)\}$, 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 π_1, \dots, π_n where each π_k maximizes the returns for class k , then $\pi = (\pi_1, \dots, \pi_n)$ is an optimal policy. As in the next section, 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 $\pi^* = (\pi_1, \dots, \pi_n)$ where each π_i must reason on all states and actions.

Q-learning was used 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. This baseline system as CLASS_{QL} was referred to because each Q-learning (QL) agent controls a class of units in Wargus.

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

It is now the time to discuss how Case-Based Reasoning techniques are used in GDA-C to manage goals on top of CLASS_{QL}. Figure 8.4 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 π , a *trajectory* is a sequence of states $\langle s_0, \dots, s_m \rangle$ visited when following π from the starting state s_0 . Any such state in this trajectory is a goal that can be achieved by executing π . The policy is assigned the last state in a trajectory as its goal. The case base *Policies* is a collection

of pairs (g, π_g) , where π_g is a policy that should be used when pursuing goal g . GDA-C stores such pairs as it encounters them.

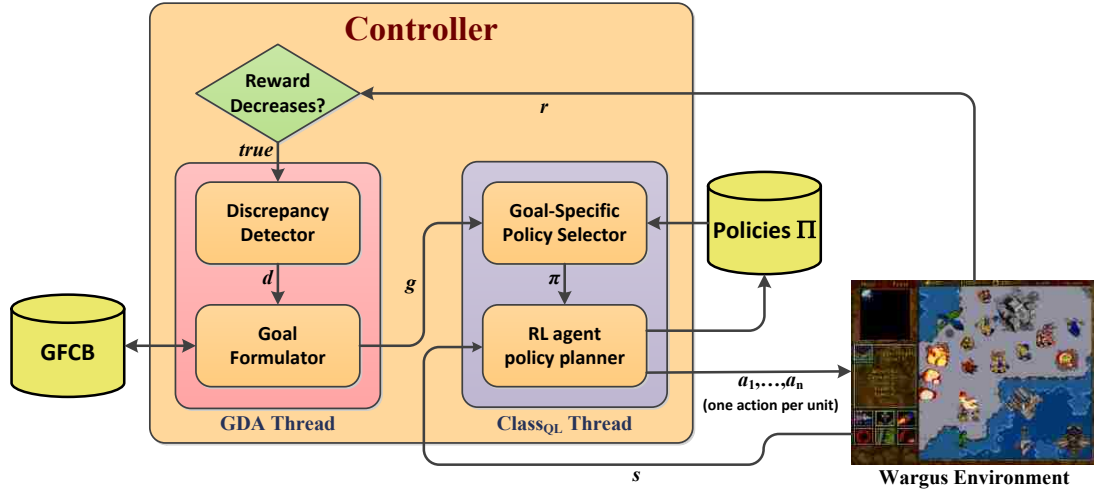


Figure 8.4: Information flow in GDA-C.

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 $\{(g_1, v_1), \dots, (g_m, v_m)\}$, 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 & 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.

Now, let us talk about the formal definitions for the GDA process. Assuming a state is represented as a vector $s = (v_1, \dots, v_n)$ 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 $v'_i \not\leq v_i$, where *expectation* $e = (v'_1, \dots, v'_n)$ and *previous state* $s = (v_1, \dots, v_n)$. An optimistic expectation *implicitly* was used in our algorithm. That is, if the previous state is $s = (v_1, \dots, v_n)$ and, after executing an action, a current state $s' = (v'_1, \dots, v'_n)$ is reached such that, for some k , $v'_k < v_k$ holds, then a *discrepancy* occurs. A discrepancy is represented as a vector of Boolean values $d = (b_1, \dots, b_n)$, where b_k is *true* iff $v'_k < v_k$ holds. Basically, the agent expects that actions will not decrease the features' values. In later section, the 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.

8.2.3 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.

Algorithm 7 GDA-C algorithm

```

GDA-C ( $\Delta, \Pi, \text{GFCB}, \mathcal{C}, \mathcal{A}, \varepsilon, g_0$ ) =
1:  $s' \leftarrow \text{GETSTATE}()$ ;  $d' \leftarrow \text{CALCULATEDISCREPANCY}(s', s')$ ;  $\pi \leftarrow \Pi(g_0)$ ;
    $g \leftarrow g_0$ 
2: //----- GDA thread -----
3: while episode continues
4:    $s \leftarrow \text{GETSTATE}()$ 
5:    $\text{WAIT}(\Delta)$ 
6:    $r \leftarrow U(s) - U(s')$            //  $s'$  is the prior state
7:   if  $r < 0$  then
8:      $d \leftarrow \text{CALCULATEDISCREPANCY}(s', s)$ 
9:      $\text{GFCB} \leftarrow \text{Q-LEARNINGUPDATE}(\text{GFCB}, d', g, d, r)$ 
10:     $g \leftarrow \text{GET}(\text{GFCB}, d, \varepsilon)$     //  $\varepsilon$ -greedy selection
11:     $\pi \leftarrow \Pi(g)$ 
12:     $s' \leftarrow s$ ;  $d' \leftarrow d$ 
13: //----- CLASSQL thread -----
14: while episode continues
15:    $s \leftarrow \text{GETSTATE}()$ 
16:   parallel for each class  $c \in \mathcal{C}$  // this loop controls agent  $\alpha_c$ 
17:      $s_c \leftarrow \text{GETCLASSSTATE}(c, s)$ 
18:      $\mathcal{A}_c \leftarrow \text{GETCLASSACTIONS}(\mathcal{A}, c)$ ;  $A \leftarrow \text{GETVALIDACTIONS}(\mathcal{A}_c, s_c)$ 
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:          $s'_u \leftarrow s_c$ ;  $a'_u \leftarrow \text{do-nothing}$ 
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:          $\pi_c \leftarrow \text{Q-LEARNINGUPDATE}(\pi_c, s'_u, a'_u, s_c, r_u)$ 
26:          $a \leftarrow \text{GETACTION}(\pi_c, \varepsilon, s_c, A)$ 
27:          $\text{EXECUTEACTION}(a)$ 
28:          $s'_u \leftarrow s_c$ ;  $a'_u \leftarrow a$ 
29:   return  $\Pi, \text{GFCB}$ 

```

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 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 $\pi = (\pi_1, \dots, \pi_n)$ 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.

CHAPTER 9

EXPERIMENTAL EVALUATION

Success is not determined by the outcome. The outcome is the result of having already decided that you are successful to begin with.

— T.F. Hodge, From Within I Rise: Spiritual Triumph over Death and Conscious Encounters with “The Divine Presence”

9.1 The description of problem domains used for experiments

My research mainly focuses on building AI systems that have the ability to adapt themselves to new environments by examining their own knowledge. Experiment on complex environments such as real-time strategy games makes the research challenging. In most RTS games, the environment is non-deterministic; that is, actions have multiple possible outcomes. It is also very adversarial; that is, agents are opposing other agents. Finally, movements are asynchronous; a player doesn't wait for other players to make their moves. I used two games DOM and Wargus as my problem domains for the experiments.

9.1.1 DOM: Domination game

Domination games are played in a turn-based environment in which two teams compete to control specific locations called domination points. Teams are composed of k bots. The player's actions are k -tuples (l_1, \dots, l_k) indicating the domination location l_i to which each bot b_i is assigned. A player captures a location by simply moving a bot to it. In other words, each time a bot on team t passes over a domination point, that point will belong to t . Team t receives one point for every 5 seconds that it owns a domination point. Teams compete to be the first to earn a predefined number of points. Domination games have been used in a variety of combat games, including first-person shooters such as Unreal Tournament and online role-playing games such as World of Warcraft.

Domination games are popular because they reward team effort rather than individual performance. No awards are given for killing an opponent team's bot, which respawns immediately in a location selected randomly from a set of map locations, and then continues to play. Killing such bots might be beneficial in some circumstances, such as killing a bot before he can capture a location, but the most important factor influencing the outcome of the game is the strategy employed. An example strategy is to control half plus one of the domination locations. A location is captured for a team whenever a bot in that team moves on top of the location and within the next 5 game ticks no bot from another team moves on top of that location. Figure 9.1 displays an example DOM game map with five domination locations.

Bots begin the game and respawn with 10 health points. Enemy encounters (between bots on opposing teams) are handled by a simulated combat consisting of successive die rolls, each of which makes the bots lose some number of health points. The die roll is modified so that the odds of reducing the opponent health points increase with the number of friendly bots in the vicinity. Combat finishes when the first bot health points decreases to 0 (i.e., the bot dies). Once combat is over, the death bot is respawned from a spawn point owned by its team in the next game tick. Spawn point ownership is directly related to domination point ownership, if a team owns a given domination point the surrounding spawn points also belong to that team.

DOM is a good testbed for testing algorithms that integrate planning and execution because domination actions are non-deterministic; if a bot is told to go to a domination location the outcome is uncertain because the bot may be killed along the way. Domination games are also adversarial; two or more teams compete to control the domination points. Finally, domination games are imperfect information games; a team only knows the locations of those opponent bots that are within the range of view of one of the team's own bots.

9.1.2 Wargus

Wargus is a modification of Warcraft2, a commercial video game originally created by Blizzard Entertainment. It runs under the Stratagus engine, a free cross-platform real-time strategy game engine used to build other games. The original version of Warcraft2 was built on 1995 and required to run on DOS mode. The

Stratagus engine allows users to play Warcraft2 under operating systems not supported by the original Warcraft2 engine such as Windows. In addition, it also allows users to play over the internet.

In Wargus, the race of each character (unit) can be either humans or orcs. Generally, ability of human units and orc units are fairly balance. Humans and orcs units are composed of three main types: land, naval and air units. There are 28 types of units: 14 types of human units and 14 types of orc units. There is only one type of civilian units for both races; Peasant for humans and Peon for orcs. There are three main tasks for each civilian unit: building a new structure, repairing a damaged structure, and harvesting resources. There are three kinds of resources in Wargus: gold, wood, and oil. To harvest gold, a civilian must walk to a gold mine and carry the gold back to its own camp. As per to harvest wood, a civilian walk to a tree, cut and bring the wood back to the camp. However, harvesting oil is not a task for civilians because they are land units and it is more complicate than harvesting wood and gold. To harvesting oil, we have to send an oil tanker (a naval unit) to an oil rig to pump oil and carry it back to the shipyard. If there is no oil rig, we have to build it first by sending an oil tanker to find an oil patch and build a rig over the patch.

About structures in Wargus, there are two main kinds of structures for both races: land-based and sea-based structures. Each structure type has its own duty. For example, barracks product military units, aviaries create aircraft, town halls train civilians.

9.2 Empirical Evaluation of GDA-HTNbots

As we mentioned in Section 4.2, to prove the claim that GDA increases system's performance, an experiment on DOM games was conducted by playing the GDA-HTNbots system versus a set of opponents. The explanation of the behavior of each adversary is shown in Table 9-1.

Table 9-1: The adversaries in DOM game and their descriptions

Adversaries	Description	Difficulty
Dom1 Hugger	Sends all agents to domination location 0.	Trivial
First Half of Dom Locations	Sends an agent to the first half + 1 domination location. Extra agents patrol between the 2 locations.	Easy
Second Half of Dom Locations	Sends an agent to the second half + 1 domination locations. Extra agents patrol between the two locations.	Easy
Each Agent to One Dom	Each agent is assigned to a different Dom location and remains there for the entire game.	Medium
Smart Opportunistic	Sends agents to each Dom location the team doesn't own. And, if it is possible, it will send multiple agents to each un-owned location.	Hard
Greedy Distance	Each turn the agents are assigned to the closest domination location they do not own.	Hard

By comparing the performances of both GDA-HTNbots and HTNbots, HTNbots performs well versus several hard-coded opponents. Thus, HTNbots should provide a good baseline for the system's evaluation. However, we expected GDA-HTNbots

would outperform HTNbots for opponents whose behaviors motivate the dynamic formulation of new goals.

The performance of these systems was recorded and compared versus the same set of hard-coded opponents. Our performance metric is the difference in the score between the system and opponent while playing DOM, divided by the system's score.

Both systems were run against each of the six opponents summarized in Table 9-1. The first three were the same used to test HTNbots, which was found to perform well on them. Hence, these are challenging DOM opponents for testing whether GDA enhancements can improve HTNbots' performance. The final three opponents were created in subsequent studies of HTNbots to test reinforcement learning and case-based reasoning algorithms. Among these, the final two opponents were found to be particularly difficult to beat. In summary, these opponents form a challenging and varied testbed to measure the utility of GDA-HTNbots.

The experimental setup was as follows: Both systems were tested versus each of these opponents on the map shown in Figure 9.1. This is the same map that was used in the previously mentioned experiments. Each game was run three times to account for the randomness introduced by non-deterministic game behaviors.

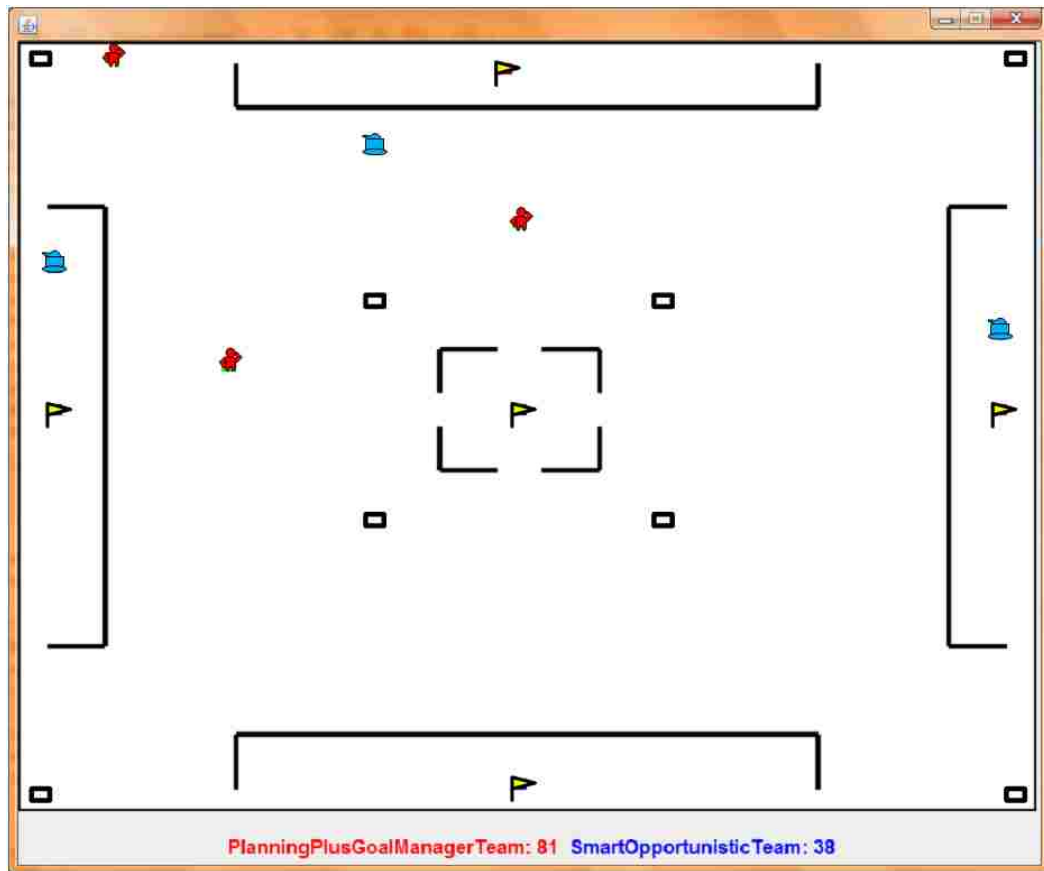


Figure 9.1: An example DOM game map with five domination locations (yellow flags), where small rectangles identify the respawning locations for the agents and the remaining two types of icons denote each player's agents.

The results are shown in Table 9-2, where each row displays the normalized over three average difference in scores (computed games) versus each opponent. It also shows the average scores for each player. The same experiment was repeated with a second map and obtained results consistent with the ones discussed here. The limited number of trials in this pilot study prevents us from computing statistical significance.

Table 9-2: Average Percent Normalized Difference in Game AI System vs. Opponent Scores (with average scores in parentheses).

Adversaries	HTNbots	GDA-HTNbots
Dom1 Hugger	81.2% (20002 vs. 3759)	80.9% (20001 vs. 3822)
First Half of Dom Locations	47.6% (20001 vs. 10485)	42.0% (20001 vs. 11605)
Second Half of Dom Locations	58.4% (20003 vs. 8318)	12.5% (20001 vs. 17503)
Each Agent to One Dom	49.0% (20001 vs. 10206)	40.6% (20002 vs. 11882)
Smart Opportunistic	-19.4% (16113 vs. 20001)	-4.8% (19048 vs. 20001)
Greedy Distance	-17.0% (16605 vs. 20001)	0.4% (19614 vs. 19534)

The results can be summarized as follows: Against difficult opponents (the final two opponents in Table 9-1), GDA-HTNbots outperforms HTNbots. Against easy opponents (the first four listed in Table 9-1) HTNbots outperforms GDA-HTNbots. Game-play records were examined to investigate why this occurred, and concluded that the initial strategy chosen by HTNbots is frequently sufficient to win the game. For example, the Dom1 Hugger (opponent) team sends all agents to one location. It is easy for HTNbots to immediately generate a winning plan against this strategy and start winning from the outset. Indeed, in situations where the goals should not be changed, this implementation of GDA should not be used.

The more difficult opponents reason about the distance between the agent locations and the domination locations as part of their strategy. These strategies are particularly effective versus HTNbots and GDA-HTNbots, which encode their knowledge symbolically without metric information. Indeed, the two hard opponents soundly defeat HTNbots. The advantage of using a specialized component to reason about goals becomes apparent in this study. By tracking which domination locations the opponent is trying to control and which goal was used to generate the current plan, GDA-HTNbots can react quickly to the opponent's strategy. This allowed GDA-HTNbots to outperform the Greedy Distance opponent (which outperformed HTNbots) and almost perform as well as the Smart Opportunistic opponent.

9.3 Empirical Evaluation of CB-GDA

We describe an empirical study of CB-GDA on the task of winning games defined using a complex gaming environment (DOM). Our study revealed that, for this task, CB-GDA outperforms a rule-based variant of GDA when executed against a variety of opponents. CB-GDA also outperforms a nonGDA replanning agent against the most difficult of these opponents and performs similarly against the easier ones. In direct matches, CB-GDA defeats both the rulebased GDA system and the non-GDA replanner.

An exploratory investigation was performed to assess the performance of CB-GDA. The claim of CB-GDA is that the case-based approach to GDA can outperform

the previous rule-based approach (GDA-HTNbots) and a non-GDA replanning system (HTNbots) in playing DOM games. To assess this hypothesis a variety of fixed strategy opponents were used as benchmarks, as shown in Table 9-1. These opponents are displayed in order of increasing difficulty.

The performance of these systems was recorded and compared against the same set of hard-coded opponents in games where 20,000 points are needed to win and square maps of size 70 x 70 tiles. The opponents above were taken from course projects and previous research using the DOM game and do not employ CBR or learning. Opponents are named after the strategy they employ. For example, Dom 1 Hugger sends all of its teammates to the first domination point in the map. The performance metric is defined by the difference in the score between the system and opponent while playing DOM, divided by the system's score. The experimental setup tested these systems against each of these opponents on the map used in the experiments of GDA-HTNbots. Each game was run three times to account for the randomness introduced by non-deterministic game behaviors. Each bot follows the same finite state machine. Thus, the difference of results is due to the strategy pursued by each team rather than by the individual bot's performance.

The results are shown in Table 9-3, where each row displays the normalized average difference in scores (computed over three games) against each opponent. It also shows the average scores for each player. The results for HTNbots and GDA-HTNbots are the same as reported in, while the results for CB-GDA are new. The same experiment was repeated with a second map and obtained results consistent with

the ones presented in Table 9-3 except for the results against Greedy, for which inconclusive results were obtained due to some path-finding issues.

Table 9-3: Average Percent Normalized Difference in the Game AI System vs. Opponent Scores (with average Scores in parentheses)

Opponent Team (controls enemies)	Game AI System (controls friendly forces)		
	HTNbots	HTNbots-GDA	CB-GDA
Dom1 Hugger	81.2% (20,002 vs. 3,759)	80.9% (20,001 vs. 3,822)	81.0% (20,001 vs. 3,809)
First Half Of Dom Points	47.6% (20,001 vs. 10,485)	42.0% (20,001 vs. 11,605)	45.0% (20,000 vs. 10,998)
Second Half Of Dom Points	58.4% (20,003 vs. 8,318)	12.5% (20,001 vs. 17,503)	46.3% (20,001 vs. 10,739)
Each Agent to One Dom	49.0% (20,001 vs. 10,206)	40.6% (20,002 vs. 11,882)	45.4% (20,001 vs. 10,914)
Greedy Distance	-17.0% (16,605 vs. 20,001)	0.4% (19,614 vs. 19,534)	17.57% (20,001 vs. 16,486)
Smart Opportunistic	-19.4% (16,113 vs. 20,001)	-4.8% (19,048 vs. 20,001)	12.32% (20,000 vs. 17,537)

Table 9-4: Average percent normalized difference in the game AI system vs. opponent scores (with average scores in parentheses) with statistical significance.

Opponent	CB-GDA – Map 1	CB-GDA – Map 2
Dom 1 Hugger	80.8% (20003 vs. 3834)	78.5% (20003 vs. 4298)
	81.2% (20001 vs. 3756)	78.0% (20000 vs. 4396)
	80.7% (20001 vs. 3857)	77.9% (20003 vs. 4424)
	81.6% (20002 vs. 3685)	77.9% (20000 vs. 4438)
	81.0% (20003 vs. 3802)	78.0% (20000 vs. 4382)
Significance	3.78E-11	1.92E-11
First Half of Dom Points	46.0% (20000 vs. 10781)	53.1% (20000 vs. 9375)
	45.8% (20001 vs. 10836)	56.7% (20002 vs. 8660)
	44.9% (20001 vs. 11021)	54.6% (20002 vs. 9089)
	46.1% (20000 vs. 10786)	52.0% (20001 vs. 9603)
	43.4% (20001 vs. 11322)	53.7% (20001 vs. 9254)

Opponent	CB-GDA – Map 1	CB-GDA – Map 2
Significance	4.98E-08	1.38E-07
Second Half of Dom Points	45.6% (20002 vs. 10889) 47.2% (20002 vs. 10560) 44.1% (20001 vs. 11188) 45.1% (20000 vs. 10987) 45.8% (20000 vs. 10849)	60.6% (20000 vs. 7884) 61.7% (20000 vs. 7657) 61.7% (20000 vs. 7651) 61.0% (20001 vs. 7797) 60.8% (20002 vs. 7848)
Significance	4.78E-08	7.19E-10
Each Agent to One Dom	46.1% (20001 vs. 10788) 46.2% (20000 vs. 10762) 44.7% (20002 vs. 11064) 44.6% (20000 vs. 11077) 47.6% (20002 vs. 10481)	54.9% (20002 vs. 9019) 53.7% (20002 vs. 9252) 56.8% (20001 vs. 8642) 55.4% (20000 vs. 8910) 57.7% (20002 vs. 8469)
Significance	6.34E-08	7.08E-08
Greedy Distance	6.4% (20001 vs. 18725) 8.3% (20001 vs. 18342) 5.0% (20000 vs. 18999) 9.0% (20001 vs. 18157) 12.7% (20001 vs. 17451)	95.6% (20003 vs. 883) 92.7% (20002 vs. 1453) 64.6% (20004 vs. 7086) 94.9% (20004 vs. 1023) 98.0% (20004 vs. 404)
Significance	1.64E-03	6.80E-05
Smart Opportunistic	4.5% (20000 vs. 19102) 11.5% (20000 vs. 17693) 11.5% (20000 vs. 17693) 10.6% (20000 vs. 17878) 13.4% (20009 vs. 17333)	13.4% (20001 vs. 17318) 13.9% (20001 vs. 17220) 1.0% (20001 vs. 19799) 10.7% (20002 vs. 17858) 12.0% (20003 vs. 17594)
Significance	1.23E-03	1.28E-03

In more detail, the results of additional tests here designed to determine whether the performance differences between CB-GDA and the opponent team strategies are statistically significant. Table 9-4 displays the results of playing 10 games over two

maps (5 games per map) against the hard-coded opponents. The difference in score between the opponents was tested using the Student's t-test. For the significance value p of each opponent, the constraint $p < 0.05$ holds. Hence, the score difference is statistically significant.

For deeper understanding, CB-GDA was ran against the two dynamic opponents (i.e., HTNbots and GDA-HTNbots) to compete directly using the same setup as reported for generating Table 9-3. As shown in Table 9-5, CB-GDA easily outperformed the other two dynamic opponents. Again, this study was repeated with a second map and obtained results consistent with the ones presented in Table 9-5.

Table 9-5: Average Percent Normalized Difference for the Dynamic Game AI Systems vs. CB-GDA Scores (with average scores in parentheses)

Opponent Team	CB-GDA's Performance
HTNbots	8.1% (20,000 vs. 18,379)
GDA-HTNbots	23.9% (20,000 vs. 15,215)

9.4 Empirical Evaluation of the LGDA

We introduced LGDA in Section 6.2. LGDA is a goal-driven autonomy agent that automatically acquires state expectation and goal selection knowledge. In this section, we will investigate the performance of LGDA agent versus other agent using different methods.

9.4.1 Experimental Setup

We used the task of winning DOM games to investigate two hypotheses: (H1) LGDA can learn to perform as well as a non-learning GDA agent that employs expert knowledge, and (H2) LGDA can significantly outperform agents that use only RL or only CBR, respectively.

We also used six hand-coded adversaries as baselines as described in Table 9-1 except the new adversary called *Priority*, which prefers to send bots to those location with highest priority first. The domination locations owned by opponents are highest priority, un-owned domination locations are lower priority and finally, domination locations held by our team are lowest priority. Briefly, these adversaries pursue a unique goal g_i to play DOM. Their behavior is approximately modeled using a policy π_i . That is, while the first three adversaries (Dom1Hugger, First Half of Dom Locations, and Second Half of Dom Locations) are easy to defeat, the latter three (Smart Opportunistic, Each Bot to One Dom Location, and Priority) cannot be perfectly represented as policies based on our models for S and A because they reason

about the proximity of bots to locations. Proximal information is not represented by any of the four agents we tested. Thus, the latter three adversaries pose difficult challenges for the agents.

We compared LGDA versus the following agents: Retaliate, which performs Q-learning, the ablation Random GDA (RGDA), which replaces LGDA’s ϵ -greedy goal selection procedure with a random selection procedure, and CB-GDA, a non-learning CBR agent whose case bases were manually crafted by a domain expert (Muñoz-Avila, H.; Aha, D.W.; Jaidee, U.; Carter, E., 2010). It includes two case bases, whose mappings are:

$$\text{PCB: } G \times S \times A \rightarrow S, \text{ and MCB: } G \times D \rightarrow G$$

PCB records the expected state for each (goal, state, action) tuple, and MCB records the preferred goal to formulate for each (goal, discrepancy) pair. All agents (CB-GDA, LGDA, Retaliate, and RGDA) use the same model for S and A . The learning agents (the latter three) use the same utility function U . The definitions for S , A , and U are given in Section 6.3.

Games are won by the first team to reach 2000 points on Figure 9.1’s map. There were 8 bots per team, which is typical (i.e., there are usually more team members than domination locations). Scores were averaged over 10 games. In our first study, we indirectly compared the agents by testing them against the six hard-coded adversaries using a leave-one-out cross-validation (LOOCV) method: we trained each learning agent versus five adversaries, using four repetitions per starting state, and tested it

against the remaining adversary. The second study addresses our hypotheses: it directly compares LGDA versus the other agents. We trained each learning agent versus the six adversaries. LGDA received as input the policies for the six adversaries but not the policies for the other agents. We recorded results before and after each training repetition of LGDA versus each of the three agents, continuing until their relative performance stabilized. Knowledge learned during testing was flushed between games. Our metric is state utility, as defined in Section 6.3.

9.4.2 Results

Experiment 1 (Table 9-6): CB-GDA recorded the best performance among the agents; it outperformed all of the adversaries, although barely so versus *Each Bot to One Dom*, which is the strongest of the hard-coded adversaries. This adversary maintains at least one bot in each location. LGDA outperformed five of the opponents, losing only to *Each Bot to One Dom*. In contrast, Retaliate and RGDA performed poorly versus all three of the difficult adversaries. This provides initial evidence that LGDA performs comparatively well compared to its ablations but it is outperformed by CB-GDA. Our next experiment provides strong support for these observations.

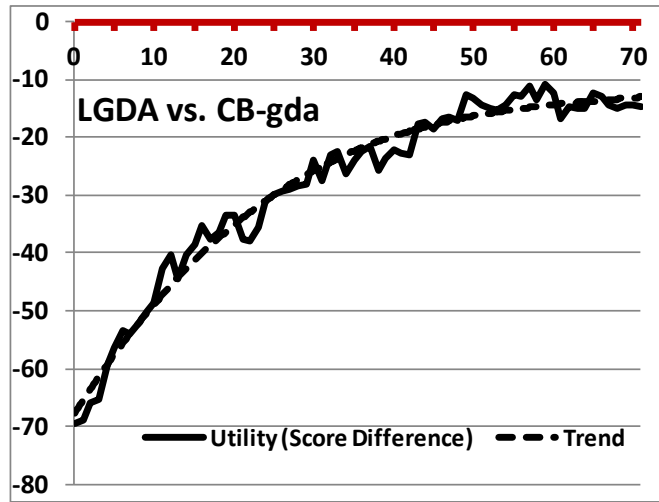
Experiment 2 (Figure 9.2): LGDA is outperformed by CB-GDA. The mean of the underlying distribution for their relative utility values after training was -14.6 ± 2.6 at the 95% confidence level. Thus, H1 is not supported, though LGDA's final

performance is fairly close. This is not too surprising, given that CB-GDA’s case bases were manually encoded by a domain expert.

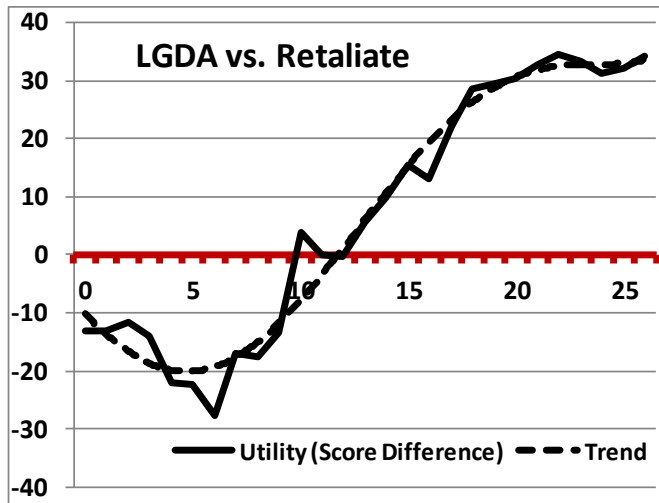
Table 9-6: Average Utility Results from Experiment 1.

Adversary	CB-GDA	Retaliate	RGDA	LGDA
Dom1Hugger	77.38	74.26	71.36	61.38
First Half of Dom Points	75.47	58.88	74.23	64.91
Second Half of Dom Points	65.36	65.79	66.28	63.03
Smart Opportunistic	54.85	-10.62	-36.59	45.27
Each Bot To One Dom	0.46	-47.13	-68.48	-50.11
Priority	45.14	-6.37	-45.28	23.08
Learning methods	None	RL	CBR	CBR & RL

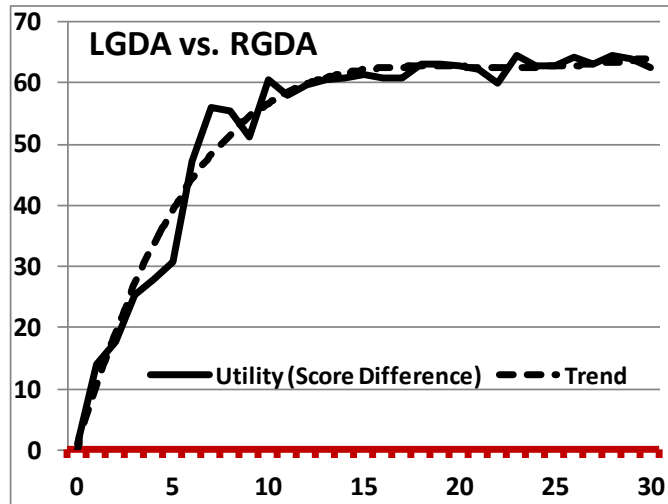
LGDA is initially outperformed by Retaliate because Retaliate quickly converges to an action that on average works well versus the adversaries. In contrast, LGDA needs to learn expectations and best goals to pursue when discrepancies occur. This results in a slower learning process in part because the interdependency between expectations and discrepancies. Over time we see that it pays off; LGDA eventually outperforms Retaliate. LGDA outperforms RGDA from the outset. Versus Retaliate, the same analysis reveals a mean of 34.3 ± 4.1 at the 95% confidence level, and the mean (at this level) versus RGDA was 62.6 ± 2.4 . Thus, these results strongly support H2.



(a)



(b)



(c)

Figure 9.2: Results from Experiment 2: Average learning curves for comparing LGDA DOM performance vs. non-learning and ablated agents. The trend lines were generated using a polynomial fit to the raw curves.

9.5 Empirical Evaluation of the GRL

We examined the task of winning two adversarial games to investigate the following hypothesis: GRL can significantly outperform a standard RL agent that learns *only* policies (i.e., Retaliate (Smith, et al., 2007), which uses Q-learning) and an ablated GDA agent that does *not* learn policies (i.e., LGDA (Jaidee, U.; Munoz-Avila, H.; Aha, D.W., 2011), which is given policies representing an opponent’s strategies and their goals, and learns only expectations and goal formulation knowledge). In our study, all three learning agents use the same models for states, actions, and rewards.

9.5.1 Scenarios

The adversarial games we use are Wargus and DOM. Both are two-player real-time video games: players make asynchronous moves. They exhibit the characteristics that we want to explore in this paper: there doesn't seem to be a universally good strategy for these games. Instead, they exhibit the "rock-paper-scissors" behavior whereby any strategy can be countered. LGDA have demonstrated good performance in DOM and Wargus (Jaidee, et al., 2011; Jaidee, et al., 2011a) while Retaliate has demonstrated good performance in DOM (Smith, et al., 2007), so they are good baselines for testing GRL.

We used two maps in our Wargus experiments. The first is a medium-sized map with 64×64 cells and 8 units per player, while the second uses the largest feasible map (128×128 cells) and 32 units per player. We set the games' score limits to be 200 and 1000 points, respectively. In our experiments, we used five hand-coded opponents that order all units of the same type to attack a single type of the agent's units. For example, they might assign knights to attack archers. These opponents differ in their attack order. In testing, no single opponent outperformed all the others. We used these built-in opponents to train the three agents (i.e., Retaliate, LGDA and GRL).

The second domain is DOM game as explained in Section 9.1.1. In our experiment, we use a map with five domination locations and eight bots per team. In addition, we used the same six hand-coded opponents in DOM we previously used in (Jaidee, et al., 2011), where we used a variety of fixed strategies such as the "half plus

one adversary”, which attempts to control a majority of locations by sending bots to them whenever they are owned by the competing agent. Another strategy, called “smart opportunistic”, sends a different bot to each domination location the team does not own. Among these six adversaries, there are two that are better than all the others, two that are middling performers, and the last two are defeated by all the others.

9.5.2 Protocol and Results

Agents played N episodes, where $N = 20$ for Wargus and $N = 340$ and 2000 for DOM. The difference in the number N of runs between Wargus and Dom is due to the fact that running DOM games is much quicker. During each training episode, each agent played each of the M built-in opponents once ($M = 5$ for Wargus and $M = 6$ for DOM). During training, the agents GRL, LGDA and Retaliate are learning. We tested GRL against Retaliate and LGDA after each training episode. Because both DOM and Wargus are highly stochastic, games during testing were repeated 10 times. Any knowledge learned during a game in the testing phase was removed after the game ends. Thus, the only knowledge affecting the performance of the agents when competing versus one another was learned during training and any knowledge learned online within that particular game episode.

Figure 9.3 and Figure 9.4 summarize the average results. The x -axis plots the number of training episodes, while the y -axis plots the average utility (i.e., score difference of GRL versus another agent).

Experiment 1 (Wargus): In most Wargus episodes (Figure 9.3), GRL clearly outperformed the other agents, although LGDA sometimes defeated GRL in the medium-size map (Figure 9.3b). Nevertheless in all cases the differences are statistically significant ($p < 0.001$), as determined by a two-tailed Student's t -Test on the utility scores of GRL versus the scores of another agent (i.e., Retaliate or LGDA). Hence our hypothesis is supported for Wargus, and we can draw three conclusions:

1. There is either no universally good strategy for these games or none can be found by Q-learning even after a large number of episodes.
2. GRL outperformed the Q-learning agent. This highlights the importance for using case-based approaches to learn and reason about expectations, goal formulation knowledge, and goal-specific policies in domains where no universally-best strategy can be elicited by RL.
3. GRL outperformed the LGDA agent. This highlights the importance of identifying new goals and using CBR to learn and reuse goal-specific cases.

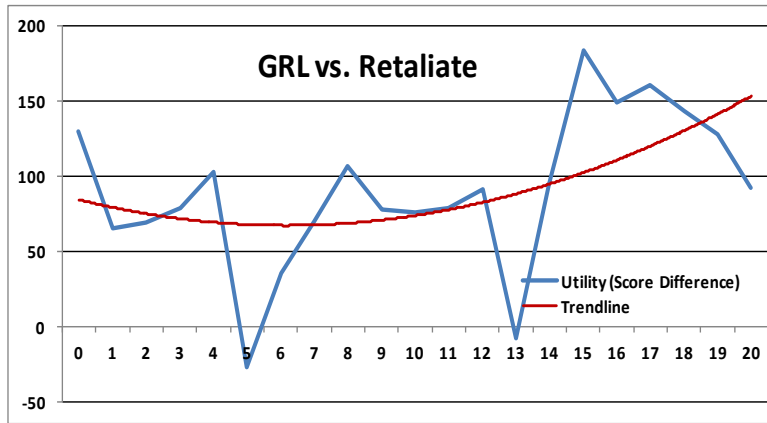
We were surprised that GRL outperformed LGDA after only a few episodes because GRL begins with no goals and no policies. In contrast, LGDA begins with policies representing the built-in opponents' strategies and goals for these policies. Upon inspection we found that the opponents' strategies cause their units to form choke points while trying to reach the units they intended to attack. As a result, few units, mostly ranged attack units, actually were effective. Without knowledge about

expectations and goals, LGDA rotates among the various opponents' strategies. As mentioned, these end up being ineffectual because it frequently results in choke points. GRL instead initially performs random actions that, on average, cause more of their own units to damage opponent units, which explains the relative results of the first few episodes.

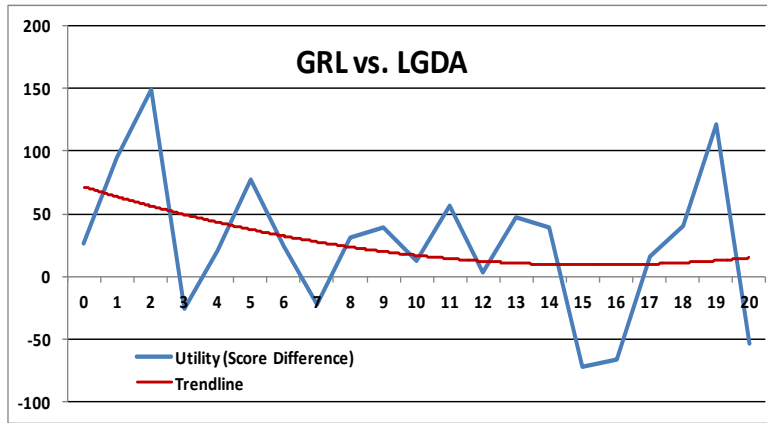
We also investigated why, despite its overall good performance, GRL will occasionally lose games to the opponents in the medium-sized map (e.g., in round 13 versus Retaliate (Figure 9.3a) and round 20 versus LGDA (Figure 9.3b)). We found that for this map the score limit was frequently reached even though both teams had several units left. That is, the maximum point threshold was set too low for the number and types of units in the scenario (i.e., killing high-value units such as knights are worth many points, and the game ends sooner when any such unit is killed). This caused high variation in the results because, after a while, several units from both sides will have few health points. In this situation, after a few of these units die the game terminates because the point limit is reached. As a result, depending on the random factor that determines which unit attacks succeeded, units from either side die while others remain with few health points. However, points are only awarded for deaths, and not for low health points. This caused the variance in the results. This was not a factor in the large map because the number of points was set sufficiently high and, although there is fluctuation; GRL did not lose a game on average (Figure 9.3c and Figure 9.3d).

Experiment 2 (DOM): Figure 9.4 summarizes the results with DOM games. In all cases GRL clearly outperformed the other agents, although initially both Retaliate and LGDA outperformed GRL. This is to be expected; GRL initially has no knowledge of which goals to pursue nor how to achieve them. Nevertheless in all cases the difference is statistically significant ($p < 0.001$) across the entire curves, as determined by a two-tailed Student's t-Test for comparing the utility scores of GRL versus those of the other two learning agents). This also supports our hypothesis and allows us draw the same conclusions as mentioned above for Experiment 1.

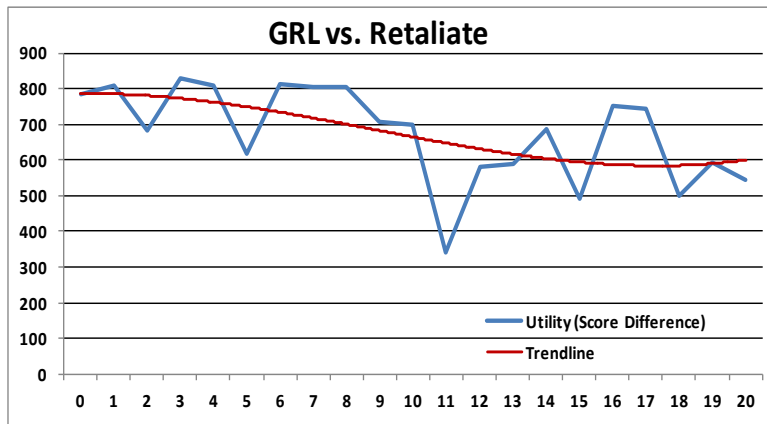
We investigated why it took so many episodes for GRL to start winning versus Retaliate and LGDA in the DOM game compared to Wargus. This occurred because the state model used by the agents forms a DAG for Wargus, meaning that a state is never visited more than once. As a result, for Wargus, we define the new goal to be the final state (whereas for DOM this is defined as the most frequently visited state). In contrast, the same state can be visited multiple times in DOM. Thus, multiple goals were frequently learned per DOM episode, resulting in many more goals being learned overall. Hence, Π grows faster in the DOM rather than in the Wargus experiments during the initial training episodes. This in turn increases the number of episodes needed to learn useful goal formulation knowledge and good policies. Thus, it takes longer for GRL to outperform the other agents in DOM scenarios.



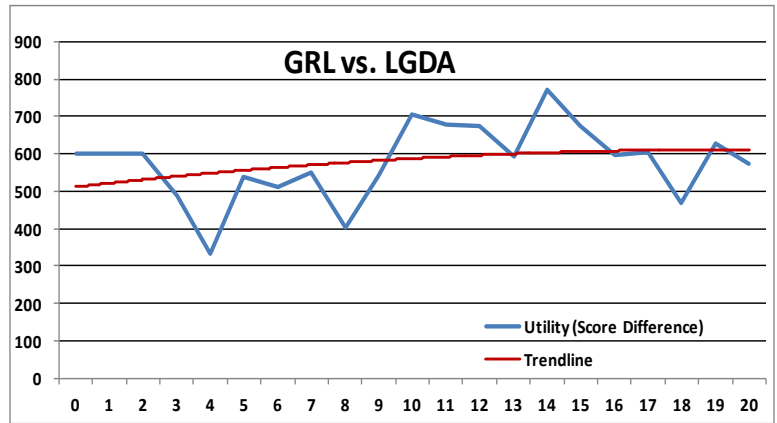
(a)



(b)

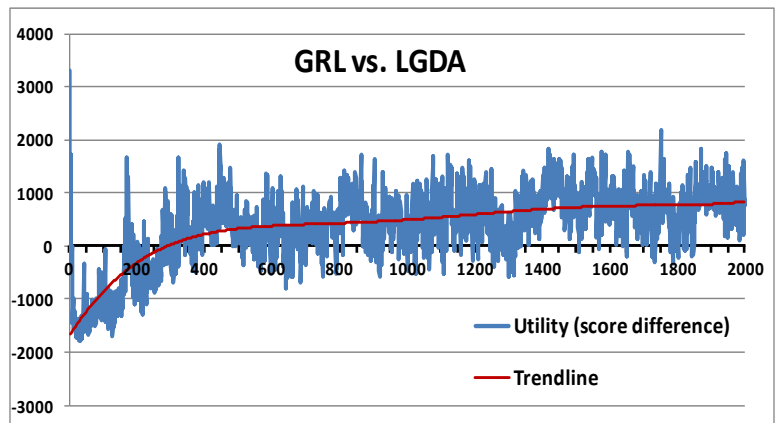


(c)

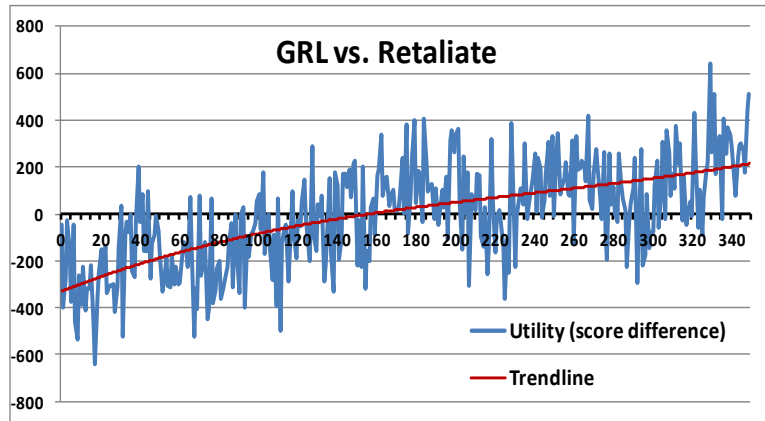


(d)

Figure 9.3: The results of the Wargus experiments: GRL vs. Retaliator (a) and vs. LGDA (b) on the medium map, and GRL vs. Retaliator (c) and vs. LGDA (d) on the large map. The x-axis plots the number of training episodes, while the y-axis plots the average utility (i.e., score difference of GRL versus another agent).



(a)



(b)

Figure 9.4: Results from the DOM experiments: (a) GRL vs. Retaliate and (b) GRL vs. LGDA. The x-axis plots the number of training episodes, while the y-axis plots the average utility (i.e., score difference of GRL versus another agent).

We also investigated why it took so many more episodes for GRL to outperform LGDA compared to Retaliate. Namely, it took around 150 episodes for Retaliate compared to almost 300 for LGDA. This was caused by the two strong hand-coded adversaries, which LGDA was able to leverage. This also explains why, in the first episode, GRL loses to Retaliate by approximately 350 points whereas it loses to LGDA by approximately 1600 points.

9.6 Empirical Evaluation of the CLASS_{QL}

We conducted two experiments for CLASS_{QL}: the first experiment (see Section 9.6.1) and the second experiment (see Section 9.6.2).

9.6.1 Experiment #1

The first experiment, we tested CLASS_{QL} on a small 32×32-cell map versus five adversaries.

9.6.1.1 *Experimental Setup*

At the first turn of each game, both teams start with only one peasant/peon, one town hall/great hall, and a gold mine near them. We have five adversaries: land-attack, SR, KR, SC1 and SC2 for training and testing our algorithm. These adversaries come with the Warcraft distribution and have been used in machine learning experiments before.

These 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 offensive/defensive actions with research. It builds only land units.
- *Soldier's Rush* (SR): This attempts to overwhelm the opponent with cheap military units early in the game.

- *Knight's Rush* (KR): This strategy attempts to quickly research advanced technologies, and launch large attacks with the strongest units in the game (knights for humans and ogres for orcs) as soon as they are available.
- *Student Scripts* (SC1 & SC2): These strategies are the top two competitors created by students for a classroom tournament.

We trained and tested CLASS_{QL} by using *leave-one-out training* as the model of our experiment processes. We remove from the training set the adversary that we want to compete against. For example, if we want to experiment CLASS_{QL} versus SC1, the set of adversaries that we use for training is {land-attack, SR, KR, SC2}.



Figure 9.5: The screen capture of the small map from Wargus.

All experiments were performed on the 32×32 tile map shown in Figure 9.5. This is considered a small map in Wargus. Each competitor starts in one side of the

forest that divides the map into two parts. We added this forest to give time to opponents to build their armies. Otherwise, CLASS_{QL} was learning a very efficient soldier rush and defeating all opponents including SR very early in the game.

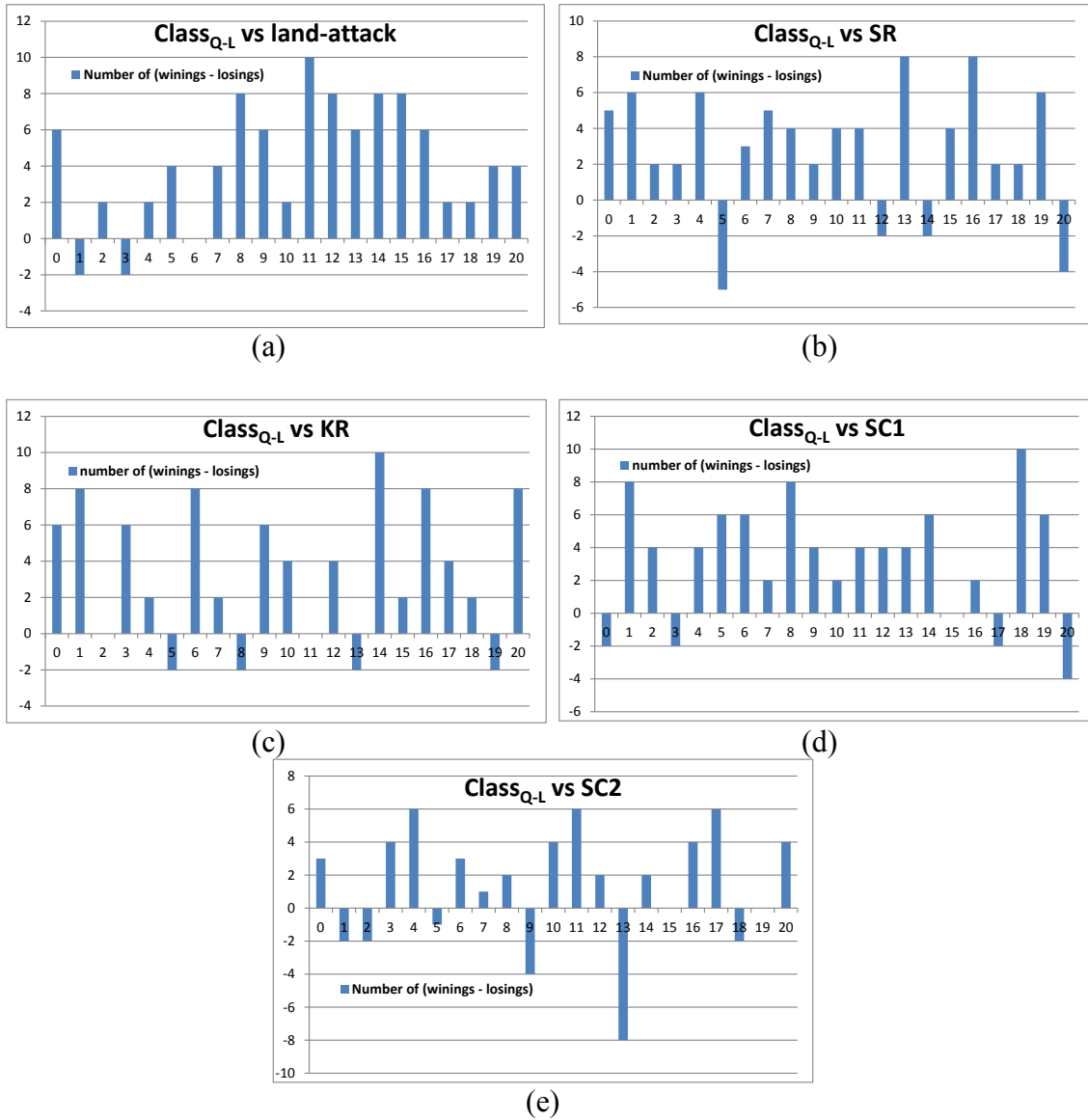


Figure 9.6: The results of the experiments #1 from Wargus: Class_{QL} vs. (a) Land-Attack, (b) SR, (c) KR, (d) SC1 and (e) SC2 respectively.

9.6.1.2 Results

Our performance metric is: $\text{wins}(\text{CLASS}_{\text{QL}}) - \text{wins}(t)$, where $\text{wins}(t)$ is the number of wins for team $t \in \{\text{land attack, SR, KR, SC1, SC2}\}$. First we match CLASS_{QL} against each opponent with no training ($m = 0$). Then we play against each opponent after one round of training using leave-one-out training ($m = 1$). We repeat this until $m = 20$. We repeat each match 10 times and compute the average metric. So the total number of games played in this experiment is $21 \times 10 = 210$ games. Including training, the total number of games run in this experiment is $210 \times 3 = 630$ games.

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 the performance is better. In fact, the performance should be worse than the previous case because it takes longer time to win.

Overall the performance of CLASS_{QL} is better than that of the adversaries (see Figure 9.6). The x -axis shows the results after x number of iterations training in the leave-one-out setting. The first bar is $x = 0$ and the last bar is $x = 20$.

9.6.2 Experiment #2

We conducted the experiments for CLASS_{QL} on a small, medium and large Wargus maps whose sizes are 32×32 , 64×64 , and 128×128 cells, respectively with the fog-of-war mode turned off. 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 building), 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 parts. As for the same reason mentioned in Section 9.6.1.1, 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” attack), which will end the game when the opponents have produced only a few units or buildings.

9.6.2.1 Experimental Setup

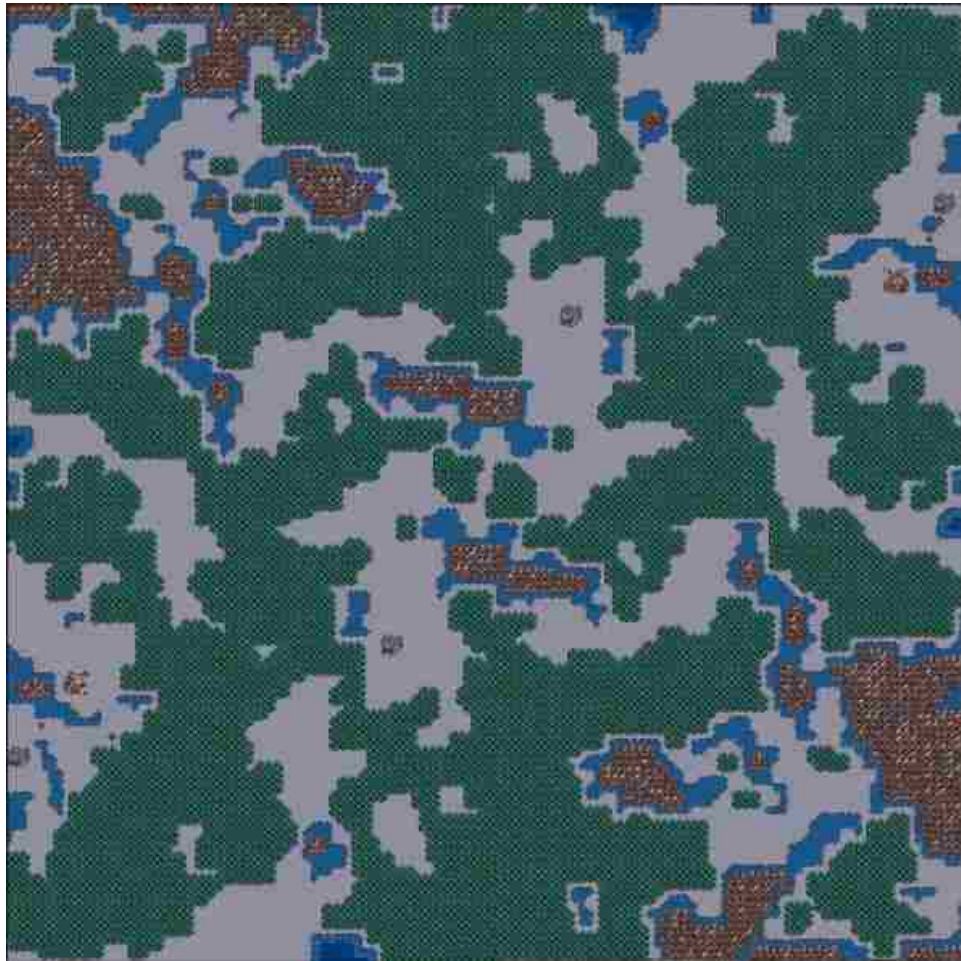
We compared the performance of CLASS_{QL} against Wargus’ built-in AI. The built-in AI in Wargus is quite good; it provides a challenging game to an average human player. We use five adversaries (defined in Section 9.6.1.1) to train the algorithm and test with the Wargus’ built-in AI. The built-in AI is capable of

defeating average players and is a stronger player than the 5 adversaries used for training.

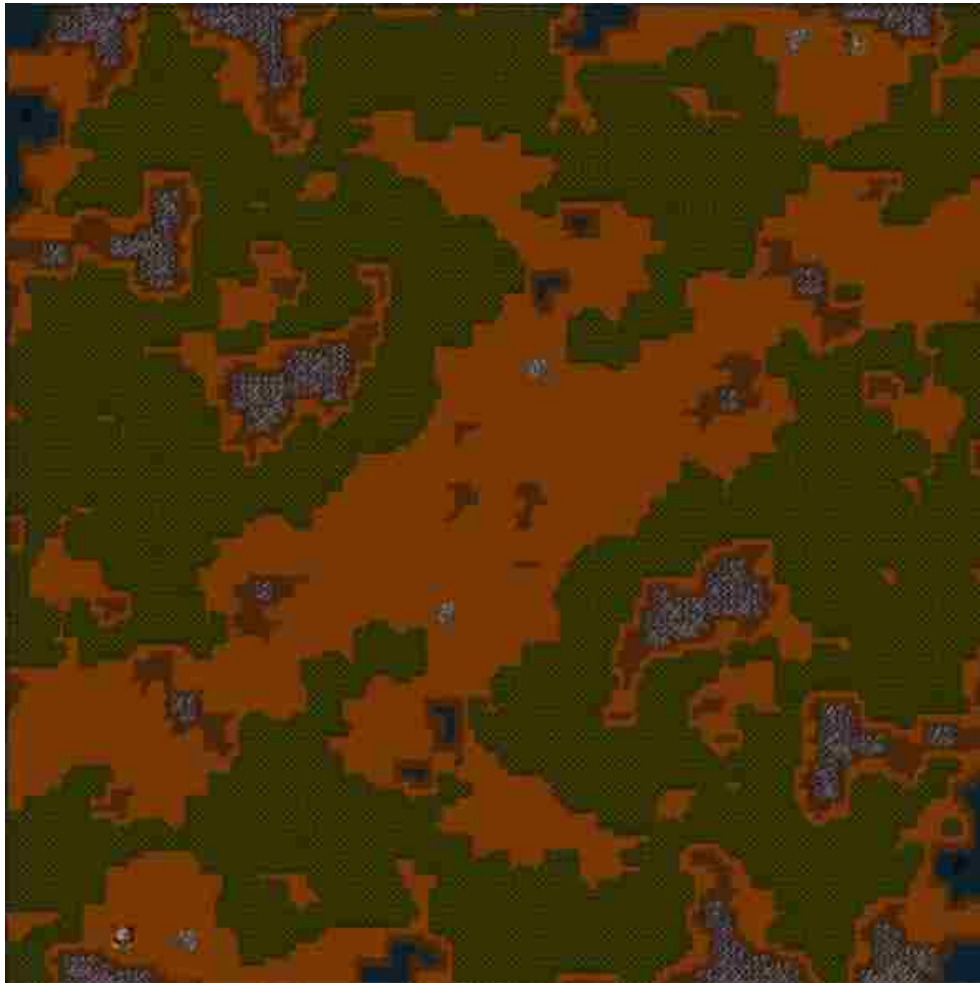
The performance metric that we used for generating the results of the experiment is $\text{wins}(\text{CLASS}_{\text{QL}}) - \text{wins}(\text{built-in})$.



(a)



(b)



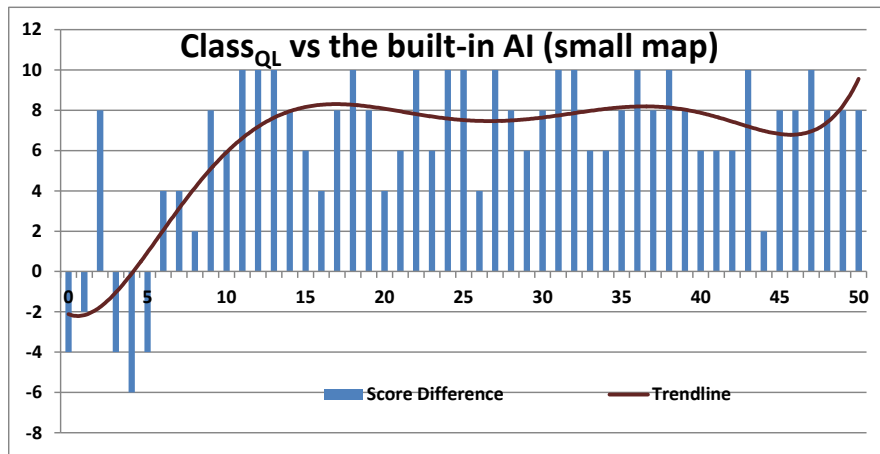
(c)

Figure 9.7: The detailed landscape of the (a) 1st, (b) 2nd, (c) 3rd large maps. The highlighted squares are the locations of both teams.

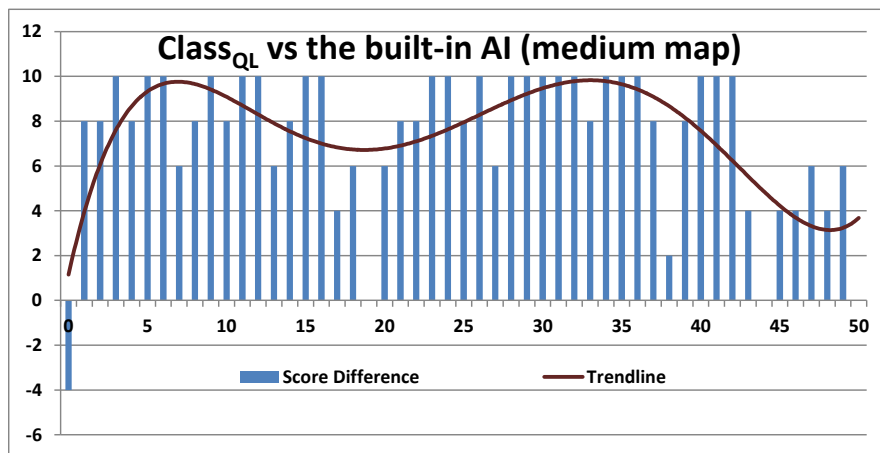
9.6.2.2 Results

Figure 9.8 shows the results of the experiments. The x -axis is the number m of training iterations and the y -axis is the performance metric. In all three maps, the build-in AI starts winning, which is not surprising since CLASS_{QL} has no training. After a few iterations CLASS_{QL} begins to outperform the built-in AI and continues outperform for the remaining iterations.

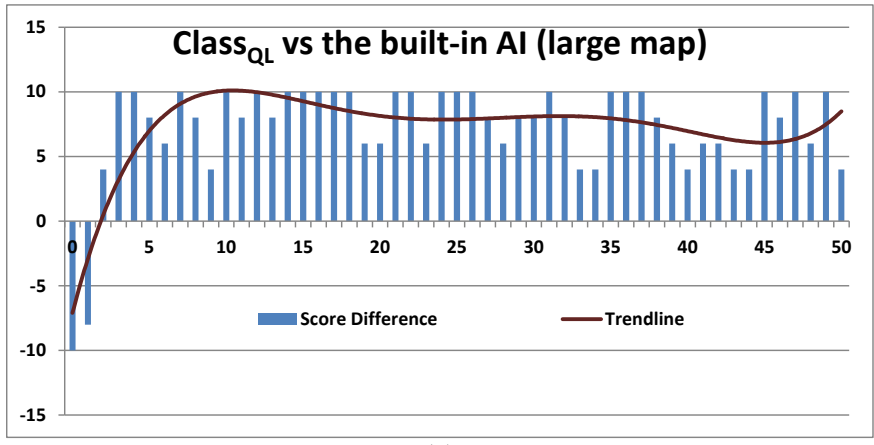
We also conducted experiments, where we tested the AI learned by CLASS_{QL} on one map after $m = 50$ iterations and tested it against the built-in AI in other two unseen maps without any additional training. We repeated this experiment for the AI learned in the small, medium and large maps. Figure 9.7 shows the original large map used for learning and the two other medium maps we used for testing. The results are shown in Table 9-7. In the original map used for training (column labeled 1st map), CLASS_{QL} 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).



(a)



(b)



(c)

Figure 9.8: The results of the experiments #2 from Wargus game: CLASS_{QL} vs the built-in AI on the (a) small, (b) medium and (c) large maps.

Table 9-7: The results of using the q-table that was trained with one scenario (the first landscape) and tested with other unseen scenarios (the second and the third landscape) on the small, medium and large maps.

scenarios map size	1 st landscape	2 nd landscape	3 rd landscape
Small	8	9	10
Medium	10	10	10
Large	10	10	10

9.7 Empirical Evaluation of the GDA-C

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. The details and landscapes of each map are shown in Figure 9.9. 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 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.

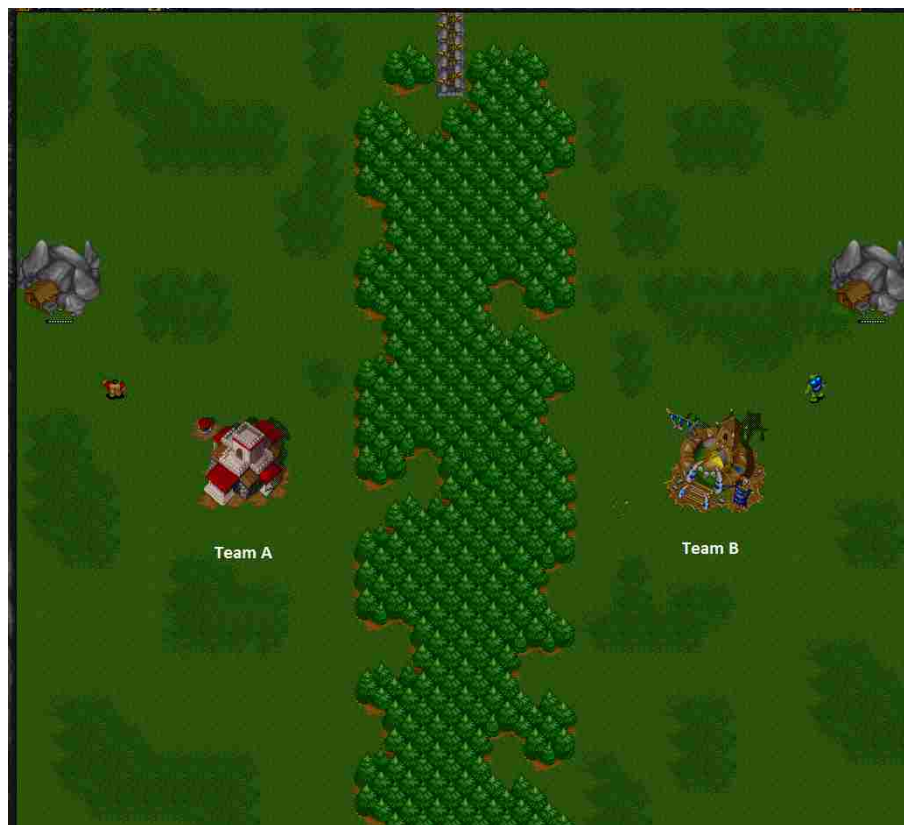
9.7.1 Experimental Setup

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 (Land Attack, Soldier's Rush, Knight's Rush, Student Scripts 1 and 2 as defined in Section 9.6.2.1) and the Wargus’ built-in AI to train and test each algorithm. These adversaries can construct any type of unit unless otherwise stated.

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 the experiment, we trained GDA-C and CLASSQL with all five adversaries and then tested them in combat against each other, where the performance metric is $\text{wins}(\text{GDA-C}) - \text{wins}(\text{CLASSQL})$, where $\text{wins}(A)$ is the number of wins for team A.

In the experiment, the matches pitting GDA-C versus CLASSQL took place after training them against each of the five adversaries for n games, where again $n = 0, 1, 2, \dots, N$. The total number N of games varied as indicated in the results. Table 9-8 shows the running times for the experiments.



(a)



(b)

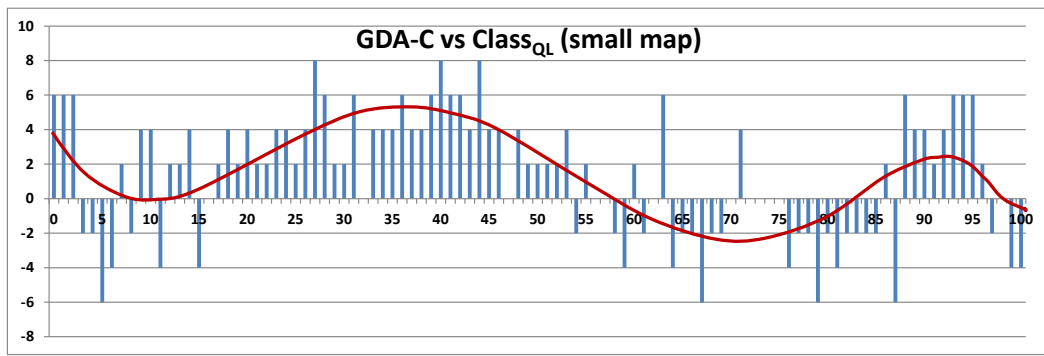


(c)

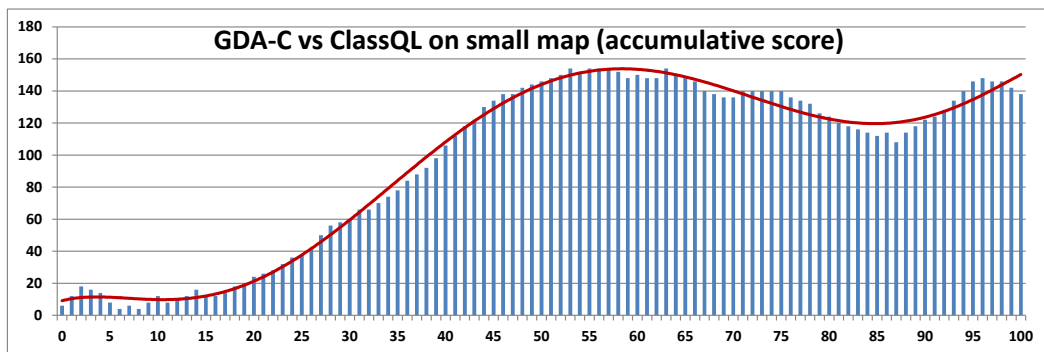
Figure 9.9: Landscapes and details of the small (a), medium (b), and large maps (c) that used for the experiment #1 and #2.

Table 9-8: The average time of running a game for both experiments

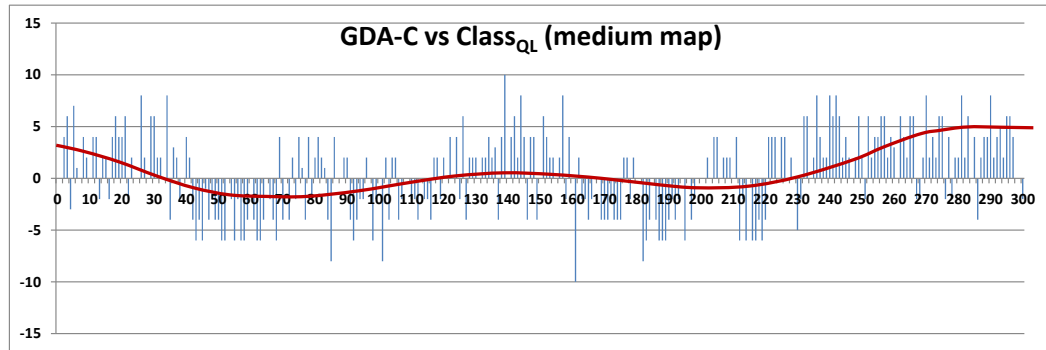
Map size	One game	Experiment 1	Experiment 2
small	31 sec	25 hours	38 hours
medium	3 min 27 sec	115 hours	172 hours
large	11 min 28 sec	191 hours	286 hours



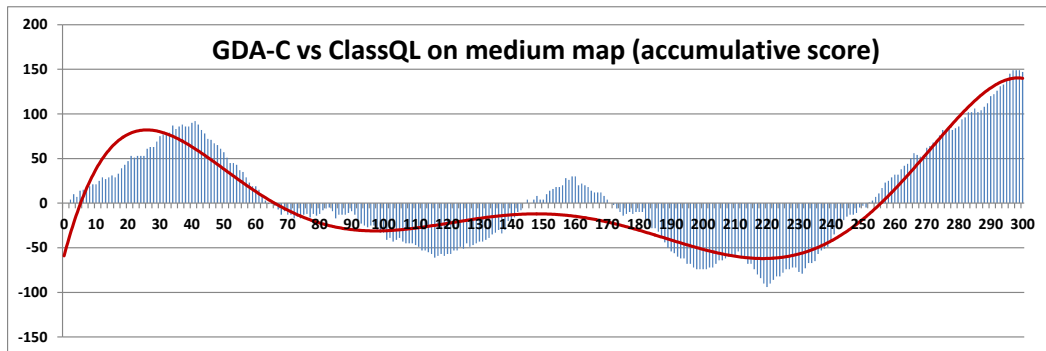
(a-1)



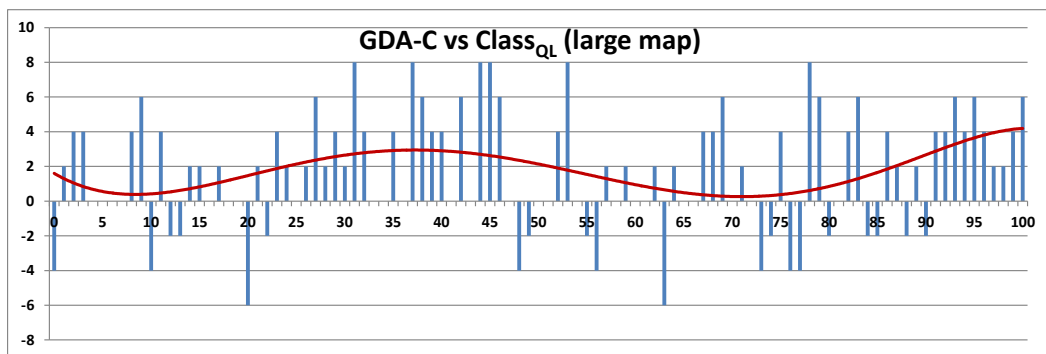
(a-2)



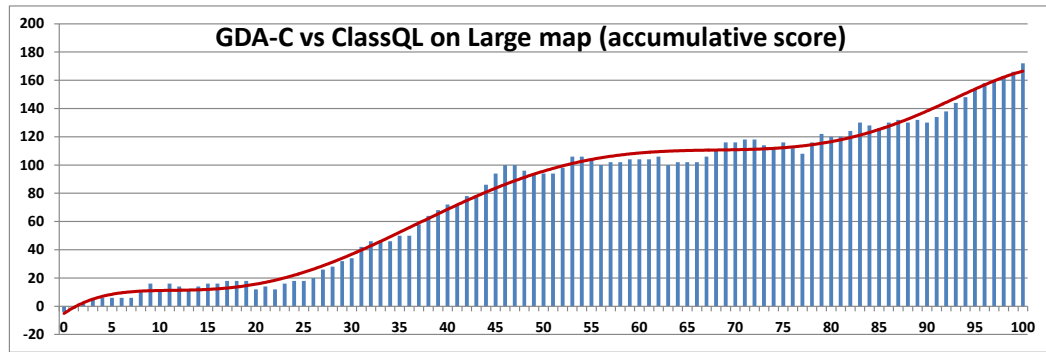
(b-1)



(b-2)



(c-1)



(c-2)

Figure 9.10: The results of Experiment: GDA-C versus CLASS_{QL} on the small (a), medium (b), and large maps (c). Figures (a-2), (b-2), and (c-2) show the results as accumulative score.

9.7.2 Results

Figure 9.10 display the results for the experiment. Each data point in the experiment is the average score difference of 10 tests, and the graphs display the results for the small, medium, and large maps. The curve is the trend line of the score difference for each data point. The x-axis refers to the training iteration number and the y-axis is the performance metric. The result from the experiment shows that after training for many rounds; eventually GDA-C outperforms CLASS_{QL} in all small, medium and large maps.

The difference between the geographies of different maps causes GDA-C agent to learn different strategies. For example, in the small map (Figure 9.9-a) there is just a forest that separates both teams' basecamps. The GDA-C agent learns to attack the enemy as quickly as possible. In this small map, it is rare that the GDA-C agent will produce high powerful units such as gryphon riders to attack the enemy's units.

Instead, it focuses on producing a lot of cheap level-entry military units to defeat the opponent. As for large map (Figure 9.9-c), there are forests and walls that separates the base camps of both teams. Also, because the paths on the map are zigzag and winding, it is take much longer time (compared to small and medium maps) for units to walk to the enemy's basecamp. As a result, in the large map, agents have a long time to produce units/structures. Therefore, GDA-C agent learns to produce very powerful air units such as gryphon riders to attack the enemy's base camp. As for the medium map, its geography looks like a hybrid of the geographies of the small map and the large map. Therefore, GDA-C agent needs more time to learn how to balance its plans; the length of time of each episode is neither sufficiently short enough for the plan of producing a lot of entry-level units nor long enough for the plan of producing very powerful military units. As a result, for the medium map (Figure 9.10-b), GDA-C agent needs more time to learn a balanced attack to outperform the opponent.

CHAPTER 10

RELATED WORK

“Why make mistakes, learn from someone else’s experiences”

— AJ Kumar

10.1 Planning Methods and Their Disadvantages

Compared to Goal-Driven Autonomy

One of the most frequently cited quotes from Helmuth von Moltke, one of the greatest military strategists in history, is that “no plan survives contact with the enemy” (Moltke, 1993). That is, even the best laid plans need to be modified when executed because of:

- (a) The non-determinism in one’s own actions (i.e., actions might not have the intended outcome).
- (b) The intrinsic characteristics of adversarial environments (i.e., the opponent might execute unforeseen actions, or even one action among many possible choices).

- (c) Imperfect information about the world state (i.e., opponents might be only partially aware of what the other side is doing).

As a result, researchers have taken interest in planning that goes beyond the classic deliberative model. Under this classic model, the state of the world changes solely as a result of the agent executing its plan. So in a travel domain, for example, a plan may include an action to fill a car with enough gasoline to follow segments (A, B) and (B, C) to drive to location C from location A. The problem is that the dynamics of the environment might change (e.g., segment (B, C) might become unavailable due to some road damage). Several techniques have been investigated that respond to contingencies which may invalidate the current plan during execution.

Plan generation is the problem of generating a sequence of actions that transform an initial state into some desired state (Ghallab, M.; Nau, D.S.; Traverso, P., 2004). A considerable amount of research exists on relaxing the assumptions of classical planning. For example, *contingency planning* permits dynamic environments (Dearden, R.; Meuleau, N.; Ramakrishnan, S.; Smith, D.; Washington, R., 2003). Agents that use this approach create a plan that assumes the most likely results for each action, and generate contingency plans that, with the help of monitoring, are executed only if a plan execution failure occurs at some anticipatable point(s). In contingency planning, the agent plans in advance for plausible contingencies. In the travel example, the plan might include an alternative subplan should (B, C) becomes unavailable. One such subplan might call to fill up with more gasoline at location B and continue using the alternative, longer route (B, D), (D, C). A drawback of this

approach is that the number of alternative plans required might grow exponentially with the number of contingencies that need to be considered.

Another alternative suggested is *conformant planning* (Goldman, et al., 1996) instead generates plans that are guaranteed to succeed. These methods require the a priori identification of possible contingencies. For example, the plan might fill up with enough gasoline at B so that, even if it has to go back to B after attempting to cover the segment (B,C) , it can continue with the alternative route (B,D), (D,C). The drawback is that the plan might execute many unnecessary steps for contingencies that do not occur (such as obtaining additional gasoline while initially in location B). *Plan repair* methods (Fox, et al., 2006) instead adapt a plan's remaining actions whenever the state conditions required to execute the plan's next action are not satisfied. These agents cannot change their goals, while GDA agents instead dynamically reason about which goals they should achieve.

Another assumption of classical planning concerns the set of goals that the agent is trying to achieve. If no plan exists from the initial state that satisfies the given goals, then classical planning fails. *Partial satisfaction planning* relaxes this all-or-nothing constraint, and instead focuses on generating plans that achieve some "best" subset of goals (i.e., the plan that gives the maximum trade-off between total achieved goal utilities and total incurred action cost) (van den Briel, et al., 2004). While these approaches each relax an important assumption of classical planning, neither addresses how to respond to unexpected events that occur during execution. One straightforward solution is *incremental planning*, which plans for a fixed time horizon. After plan

execution, these planners then generate plans for the next horizon. This process iterates until the goal state is reached. Another approach is *dynamic replanning*, which monitors the plan's execution. If it is apparent that the plan will fail, the planner will replan from the current state. For example, HOTRiDE (Ayan, et al., 2007) employs this strategy for non-combatant evacuation planning. These approaches can also be combined. For example, CPEF (Myers, 1999) incrementally generates plans to achieve air superiority in military combat and replans when unexpected events occur during execution (e.g., a plane is shot down).

However, these approaches do not perform *goal formulation*; they continue trying to satisfy the current goal, regardless of whether their focus should dynamically shift towards another goal (due to unexpected events).

Fortunately, some other recent research has addressed this topic. For example, bestowed agents (Coddington, A.M.; Luck, M., 2003) with *motivations*, which formulate goals in response to thresholds on specific state variables (e.g., if a rover's battery charge falls below 50%, then a goal of full battery charge will be formulated (Meneguzzi, et al., 2007)). GDA can adopt an alternative rule-based approach whose antecedents can match to complex games states.

Research on game AI takes a different approach to goal formulation in which specific states lead directly to behaviors (i.e., sequences of actions). This approach is implemented using behavior trees, which are prioritized topological goal structures that have been used in HALO 2 and other high profile games (Chamandard, 2007).

Behavior trees, which are restricted to fully observable environments, require substantial domain engineering to anticipate all events. GDA can be applied to partially observable environments by using explanations that provide additional context for goal formulation.

GDA focuses on the meta-process of goal reasoning. Some goal reasoning planners relax the requirement that the plan must achieve all of its goals. For example, *over-subscription planners* attempt to satisfy only a maximal subset of the goals (van den Briel, et al., 2004).

As previously discussed in this section, the main drawback of planning methods is that, before plan execution, they require the a priori identification of possible contingencies. In DOM games, a plan would need to determine which domination points to control, which locations to send a team's bots, and identify alternative locations when this is not possible. An alternative to generating contingencies beforehand is performing plan repair. In plan repair, if a mismatch occurs during plan execution (i.e., between the conditions expected to be true to execute the next action and the actual world state), then the system must adapt the remaining actions to be executed in response to the changing circumstances (Fox, et al., 2006; Warfield, et al., 2007). The difference between plan repair and GDA is that plan repair agents retain their goals while GDA agents can reason about which goals should be satisfied. This also differentiates GDA from replanning agents, which execute a plan until an action becomes inapplicable. At this point, the replanning agent simply generates a

new plan from the current state to achieve its goals (Hoang, et al., 2005; Ayan, et al., 2007; Myers, 1999).

There has been some research related to reasoning with goals. Classical planning approaches attempt to achieve all assigned goals during problem solving (Ghallab, et al., 2004). Van den Briel et al. relax this requirement so that only a maximal subset of the goals must be satisfied (e.g., for situations where no plan exists that satisfies all the given goals) (van den Briel, et al., 2004). Unlike GDA, this approach does not add new goals as needed. Formulating new goals has been explored by Coddington and Luck, and then by Meneguzzi and Luck, among others (Coddington, et al., 2003; Meneguzzi, et al., 2007). They define motivations that track the status of some state variables (e.g., the gasoline level in a vehicle) during execution. If these values exceed a certain threshold (e.g., if the gasoline level falls below 30%), then the motivations are triggered to formulate new goals (e.g., fill the gas tank). In contrast, we investigate the first case-based approach for GDA, where goals are formulated by deriving inferences from the game state and the agent's expectations using case-based planning techniques.

10.2 Integrations of Case-Based Learning and Reinforcement Learning

As explained in Section 2.1, reinforcement learning (RL) is a learning system which learns how to map situations to actions so as to maximize a numerical reward. Also, as explained in Section 2.2, Case-Based Reasoning is the process of solving new problems based on the solutions of similar past problems. In this section we discuss related works to integrations of CBR and RL.

Several groups have studied integrations of CBR and RL. Bridge noted that these typically attempt to use the advantages of one to improve the other (Bridge, 2005). For example, RL has been used to help CBR solve problems in continuous environments (Ram, et al., 1997; Molineaux, et al., 2010) and to improve case retrieval (Juell, et al., 2003). Analogously, CBR has been used to speed up the RL process (Gabel, et al., 2007; Auslander, et al., 2008; Bianchi, et al., 2009) and to reduce RL's memory footprint (Dilts, et al., 2010). We instead integrate them to automatically acquire and reuse GDA knowledge.

There is substantial interest in integrating CBR and RL, as exemplified by Derek Bridge's ICCBR-05 invited talk on potential synergies between CBR and RL (Bridge, 2005), the SINS system that solves problems in continuous environments (Ram, et al., 1997), and CBRetaliante, which stores and retrieves Q-tables (Auslander, et al., 2008). Most previous contributions focused on improving the performance of an agent by

exploiting synergies among CBR and RL or by enhancing the CBR process by using RL (e.g., to improve similarity metrics). More recently, researchers have studied ways in which CBR can improve reinforcement learning. This includes reducing the memory requirements of RL (Dilts, et al., 2010), using cases as a heuristic to speed up the RL process (Bianchi, et al., 2009) and using cases to approximate state value functions in continuous spaces (Gabel, et al., 2005; Gabel, et al., 2007). Our GRL system falls in this latter category; it uses CBR to fine-tune strategies by exploiting the episodic knowledge captured in the cases while embedded in the RL cycle. In this context, GRL's novelty is that it automatically identifies goals, learns policies specific to those goals, learn expectations about the action's outcomes, and reasons when a discrepancy occurs.

10.3 Goal-Driven Autonomy Agents and Their Integration of Learning

Most research on GDA assumes that experts provide domain knowledge on what to expect when an action is executed and which goal should be achieved next if a state discrepancy arises. The two exceptions are work on learning goal selection knowledge. First, Weber et al. uses CBR for this task, but doesn't learn about expectations (Weber, et al., 2010). Their cases map discrepancies (between the current state and the goal the agent is trying to achieve) to new goals, which are represented as states, and their nearest neighbor algorithm compares the current state

with recorded cases to perform goal selection. LGDA system (Section 6.2) instead learns expectations, discrepancies, and goals. Furthermore, goals can be state abstractions (e.g., win the game) and LGDA could map a discrepancy to multiple goals. Second, Powell et al.'s active learner requires a user to indicate which goal to achieve next when discrepancies occur. In contrast, LGDA is fully automated (Powell, et al., 2011).

As mentioned in CHAPTER 3, GDA agents use a four-step strategy to respond competently to unexpected situations in their environment: (1) detect any discrepancy between the observed state and the expected state(s), (2) explain this discrepancy, (3) formulate a goal to resolve it (if needed), and (4) manage this new goal along with its pending goals (Molineaux, et al., 2010; Muñoz-Avila, et al., 2010). In step 3, these agents use a variety of models to formulate new goals. For example, INTRO (Cox, 2007) uses explanation patterns represented as *cause* \rightarrow *effect* rules such that, if a state is judged to be a discrepancy and it maps to the effects of a rule, then INTRO will select the negation of that rule's cause as its new goal. ARTUE (Molineaux, et al., 2010) uses rule-based reasoning for goal formulation and ranking (i.e., pending goals are maintained in a priority list). Its rules encode expert knowledge in a manner similar to Intro's rules, but ARTUE adds a more robust process by encoding planning dependencies in a truth-maintenance system. EISBot (Weber, et al., 2010) instead uses a case-based model to formulate goals, where a case $c_i = (c_{i,1}, \dots, c_{i,n})$ is an expert-provided sequence of states for accomplishing a task, and states are represented as a vector of numeric values. Given current state c , EISBot retrieves a most similar state

$c_{i,j}$ in its case base along with $c_{i,j+w}$, where w is the length of its planning window. It computes the difference $c_{i,j+w} - c_{(i,j)}$ and adds this to c to define its new goal. In contrast to these GDA agents, GRL *learns* its goal formulation knowledge.

T-ARTUE (Powell, et al., 2011) is an extension of ARTUE that interactively learns goal formulation knowledge; it can query the user to ask for new goals or confirm their formulation, and the user can provide feedback on these decisions. In contrast, GRL automatically learns goal formulation knowledge and new goals.

Agents can compute state expectations using action models (i.e., their preconditions and effects) and the current state. Bouguerra *et al.* use description logics to model and infer expectations after executing a plan, which is particularly useful for partially observable environments (Bouguerra, et al., 2008). For example, an agent might observe John entering a vehicle at a location A and the vehicle later arriving at location B , where its occupants departed. Given this, it could infer that John arrived at B . GDA agents vary in how they compute expectations, including using a model of abstract explanation patterns (Cox, 2007), or by defining discrepancy detectors to trigger when state expectations fail (Weber, et al., 2010). Unlike these (and most other) GDA agents, GRL *learns* its action models for computing state expectations. The only related agent is LGDA (Jaidee, et al., 2011), which learns action models it uses to compute expectations but it assumes that the policies and goals are given as input. In contrast, GRL identifies new goals, and learns and reuses goal-specific policies.

10.4 Learning Agents in Real-Time Strategy Games

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.

Before we begin our description of this analysis a clarification must be made: many of the works described are capable of playing the complete RTS games and hence perform the tasks by the 6 managers. Our point is that learning is that in those works 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 10-1. 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.

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.

Table 10-1: Categories of works versus managerial tasks (Scott, 2002)

	Build	Unit	Research	Resource	Combat	Civilization
A					✓	
B	✓	✓				
C	✓	✓	✓			
D	✓	✓		✓		
E				✓		
F	✓	✓	✓	✓	✓	✓

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. 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 (Weber et al., 2012; Jaidee et al., 2011). As we discussed earlier Weber et al. (2012) learning belongs to category B. Jaidee et al. (2011) manages goals for combat tasks so it belongs to category A. Given the variety of tasks, we could expect goal-driven autonomy works in the future to be capable of learning for all 6 managerial tasks.

CLASS_{QL} and GDA-C are this first systems that we are aware of that is capable of learning on 5 out of 6 managerial tasks (almost Category F). It follows ideas on micro-management in RTS games (e.g., (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).

10.5 Goal-Driven Autonomy Agents That Can Play RTS

Games

Weber reported on EISBot (Weber, et al., 2012), 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 where 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 generates instances of expectations by using a given model of abstract explanation patterns (Cox, 2007). Molineaux et al. use planning operators to define expectations (Molineaux, et al., 2011). Borrowing ideas from Weber et al. (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.⁴

⁴ 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.

CHAPTER 11

CONCLUSIONS

11.1 Final Remarks

Our research steps were incremental. We started with a narrow research focus and move to the more difficult issues later. In our first system, GDA-HTNbots, an extension of HTNbots in which the controller performs the four tasks of the GDA model. However, unlike HTNbots, GDA-HTNbots reasons about its goals, and can dynamically formulate which goal it should plan to satisfy. It controls plan generation in two ways: first, it determines when the planner must start working on a new goal. Second, it determines what goal the planner should attempt to satisfy. All the knowledge of GDA-HTNbots was given by user as its input in the form of HTN syntax.

Our second system, CB-GDA is the first GDA system with integration of case base reasoning. CB-GDA uses two case bases to dynamically generate goals. The first case base relates goals with expectations, while the latter's cases relate mismatches with (new) goals. All CB-GDA's knowledge was still given as its input, but the system knew how to maintain and reuse cases in the case bases.

Next, we developed a system called LGDA which is the first system to automatically learn state expectations. LGDA can learn two important components of GDA: (1) expectations to store in its Expectation Case Base and (2) new goals to store and reuse for its Goal Formulation Case Base. LGDA partially support our claim that we can create GDA agents that have the ability to acquire knowledge by themselves and reuse it. However, goals and policies are still needed to be given as the agent's input. So, we cannot say that LGDA agent is fully autonomous learning GDA agent.

Thereafter, we developed the system called GRL, the first GDA system capable of learning and reusing goal-specific policies. Additionally, GRL can learn most of the GDA's components. It can learn a Policy Case Base, an Expectation Case Base and a Goal Formulation Case Base. At this point, the answer of our research question is fulfilled. Although, we can create such agent, the environment that we experimented on was not complex enough. Therefore, we continued to investigate a scalable agent that can handle complex environments such as full RTS games that have a lot of factors to consider.

The state-action space of full RTS games is very large. GDA algorithms, including GRL, have not been designed for learning and acting on large state-action spaces. Thus, my next objective is to develop a GDA algorithm capable of learning and acting in domains with large state-action spaces. I investigated this matter by extracting the learning part from GDA and creating a learning agent called CLASS_{QL}. And, it will later be integrated back to GDA agent. CLASS_{QL} Divide the state and action space among cooperating learning agents. Each agent of CLASS_{QL} is equal to a

RL agent. Therefore, each agent has its own q-table. Each agent's unit has its own record of previous state, previous action, and previous reward for updating the q-table of its class. CLASS_{QL} agent can play a complete RTS game and perform better than hand-coded AI agents.

Finally, we merge both GDA and CLASS_{QL} together to create a system call GDA-C. GDA-C is a GDA agent that executes two threads in parallel to control several RL agents. GDA-C agent has the ability to learn knowledge by itself in complex environments such as a full RTS game. Moreover, CLASS_{QL} and GDA-C are the first learning agent and the first GDA agent that are capable of learning on 5 out of 6 managerial tasks (Table 10-1).

Our main research question in this dissertation is whether we can create GDA agents that are able to learn knowledge by themselves and reuse it. Our later systems, specifically GRL and GDA-C, demonstrate that we can indeed construct such agents. Furthermore, learning in GDA-C and GRL takes place in Wargus, which is a complex environment as per the definition in (Russell, et al., 2003). GDA-C and GRL are able to cope with each of the characteristics of complex environments for the following reasons:

- **Partially observable and multi-agent environments:** Partially observable means some information about the environment (e.g., the opponent team's resources) is hidden from the GDA agent. Because there are multiple agents, the environment might change independent of our own

agent's actions (i.e., the other agents change the environment). As a result of these characteristics of the environment, at some point the GDA agent might encounter unexpected situations (e.g., a peasant is sent to harvest gold but the peasant is killed near the town). Because GDA agents are able to react when discrepancies happen, then the Goal Formulator will suggest a new goal enabling the GDA agent to react to the unexpected situation (e.g., military unit are sent to attack enemy units near the town).

- **Stochastic environments:** Stochastic means that actions taken in the environment might have multiple outcomes (e.g., send footmen to attack enemy units near our camp might result in two outcomes: (1) enemy units near our camp are killed or (2) enemy units are still near camp). Our GDA agent is able to cope with these kinds of environments because it is learning policies (mappings from states that agent might encounter to possible actions it can execute in such states), which learn to cope with the multiple outcomes from past experience (e.g., send archers and footmen to attack units near camp when outcome (2) happens).

The following is a summary of the scientific contributions of this dissertation to the state-of-the-art in integrated learning for goal-driven autonomy research:

- First integration of Case Based Reasoning (CBR) and Goal-Driven Autonomy.
- First GDA system to automatically learn state expectations.

- First GDA system capable of learning and reusing goal-specific policies.
- First learning agent that can learn multiple real-time strategy games managerial tasks.

11.2 Future Work

Our goal in this research was to investigate GDA systems that can learn knowledge by themselves. However, there are many potential research directions. We now discuss some of these directions.

1. Create a system that is able to autonomously learn about explanation of failures. Even though LGDA, GRL and GDA-C can learn knowledge about goal selection when a failure occurred, none of our systems can learn explanations. Some of our systems know how to use predefined explanations, but they cannot learn new ones for themselves. Our systems learn new goal based on statistic techniques but they can't learn new goals based on the explanations.
2. Experiment on other problems domains. The problem domains that we used for the experiments in the dissertation are complex real-time strategy games. Indeed GDA-C is the first GDA system that learns many managerial tasks. Our systems are general that could be used in any problem domain. The simple way to describe what kind of problems that is possible to use our

algorithms is to understand the target problem domain and answer those two questions about it: (1) can we give the collection of actions and states and observe how our actions affect the environment, and (2) whether or not we can provide a numerical signal value that can be a decent indicator to tell how good or bad is a particular situation. If the answer to these two questions is affirmative researchers could use our learning techniques for a GDA system acting in this problem domain. Examples of such problem domains include using robots for manufacturing. Another example is logistics tasks involving the delivery of multiple products.

3. Agents may learn a set of high-level actions. High-level actions used in CLASS_{QL} and GDA-C algorithms are highly effective in term of space saving and reusability of learned knowledge. However, high-level actions are designed by an expert of the specific problem domain. Therefore, it would be beneficial if agents can learn high-level actions themselves. Notice that the problem domain actions that are translated from a high-level action must have at least one factor with the same value or the same range of values. This would include the same unit-type, attacking-range, abilities of attacking, to name just a few. We can collect sequences of states, actions and time and later store them in a database. This can be used later by some algorithm to find related factors of actions from the database that are performed at approximately the same time. Learning a set of high-level actions for each agent can be another starting point for future research.

4. Agents learn how to change a new goal before a failure occurs. All of our systems are based on the assumption that the goal should change when a discrepancy occurs. But, changing a new goal after a failure occurs may be too late to recover from negative consequences. The adverse score that has already been accumulated can affect the final score at the end of the game. Goal Driven Autonomy always waits until an undesirable situation happens and then later tries to change it. It is more advantageous to detect failures before they happen. We can store a sequence of states and also note any states that GDA decided to use in changing the goal. Then, we can use this sequence to track down some states that occur before the failures. Another possible method, using the reward progress that we observe in the environment, if one notes that progress is going downward for some length of time or reaches a threshold value, GDA can suggest a new goal to pursue.

5. Developers may improve the module called *High-level Action to Problem-Domain Actions Converters (HAPDAC)* in the GDA-C and CLASS_{QL} architectures. Currently, the method that HAPDAC uses to perform unit job assignment is just a simple modulus assignment. The performance might increase if we can improve an HAPDAC's mapping method. For example, taking the distance between an actor unit and a recipient unit (or a target location) into account to minimize the total distance and time that the high-level action need to be perform as multiple problem-domain actions. As per another interesting factor that we can take it into account to develop a new

mapping function is using the remaining health points of actor units comparing to those of recipient units. For example, let assume units in a set of actors and units in a set of recipients are the same type at the same level of upgrading. Also, assume both sets have the same number of units; let's say 4. The list of remain health points of the actors is (8, 20, 11, 17) and those of the recipients is (9, 22, 14, 18). If we just map them as is, the result is (-1, -2, -3, -1). In other words, all the units in the set of actors might be dead after the actions are performed. However, if we resort the list of recipients as (22, 18, 9, 14), the result will be (-14, +2, +2, +3). In other words, just only one unit in the set of actors might be dead after the action is performed. Research about mapping methods for HAPDAC would be an interesting future research topic that could improve the performance of GDA-C and CLASS_{QL}.

6. Research on hierarchical agents. Agents in CLASS_{QL} or GDA-C make their own decision of the next actions that they will execute without explicitly coordinating with one another. There might be advantages, if we can apply a hierarchical model organizing the CLASS_{QL} and GDA-C agents. For example, blacksmiths, lumber mills, and churches are agents that improve units and structures by performing research actions. Thus, we could build an agent, named *research agent*, to coordinates the research actions among these 3 agents. If the blacksmith and the lumber mills agents plan to perform research actions that require more resources than the team can supply, the research agent will prioritize those actions.

7. Research on promoting units and structures as agents. In CLASS_{QL} and GDA-C, agents task their units/structures with activities to perform. For this new research, we believe it is better for units and structures to make their own decision. For example, we could have a high-level agent named *task manager* agent that manages which of the team's tasks T_s should be performed in the current situation s . We could borrow from the ideas of *joint intention* (Cohen, P. R. & Levesque, H. J., 1991; Levesque, et al., 1990), so the units or structures perform these tasks together as a subteam. Under this perspective, a unit or structure is allowed to be a participant in several joint intentions. However, in this approach, units/structures need to communicate to other units/structures and task manager. To do this, we can borrow some ideas for teamwork coordination as in *STEAM* (Tambe, 1997).

BIBLIOGRAPHY

- Aamodt A. and Plaza E.** Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches [Book Section] // AICom - Artificial Intelligence Communications, Vol. 7: 1. - [s.l.] : IOS Press, 1994.
- Aha, D. W.; Molineaux, M.; Ponsen, M.;** Learning to win: Case-based plan selection in a real-time strategy game [Conference] // Proceedings of the International Conference on Case-based Reasoning (ICCBR). - [s.l.] : Springer, 2005.
- Auslander, B.; Lee-Urban, S.; Hogg, C.; Muñoz-Avila, H.** Recognizing the enemy: Combining reinforcement learning with strategy selection using casebased reasoning [Conference] // Proceedings of the Ninth European Conference on Case-Based Reasoning. - Trier, Germany : Springer, 2008. - pp. 59-73.
- Ayan, N.F.; Kuter, U.; Yaman, F.; Goldman, R.** HOTRiDE: Hierarchical ordered task replanning in dynamic environments [Conference] // In 3rd ICAPS Workshop on Planning and Plan Execution for Real-World Systems. - Providence, RI : [s.n.], 2007.
- Bianchi, R.; Ros, R.; Lopez de Mantaras, R.** Improving reinforcement learning by using case-based heuristics [Conference] // Proceedings of the Eighth International Conference on CBR. - Seattle, WA : Springer, 2009. - pp. 75-89.

- Blizzard Entertainment** Warcraft II: Battle.net Edition Manual [Book]. - 1999.
- Bouguerra A., Karlsson L. and Saffiotti A.** Monitoring the execution of robot plans using semantic knowledge [Article] // Robotics and Autonomous Systems, 56(11). - 2008. - pp. 942-954.
- Bridge D.** The virtue of reward: Performance, reinforcement and discovery in case-based reasoning [Conference] // Proceedings of the Sixth International Conference on Case-Based Reasoning. - Chicago, IL : Springer, 2005.
- Bridge D.** The virtue of reward: Performance, reinforcement and discovery in case-based reasoning [Conference]. - Chicago, IL : Springer, 2005.
- Champanard A.** Behavior trees for next-gen game AI [Conference] // In Proceedings of the Game Developers Conference. - Lyon, France : [s.n.], 2007.
- Coddington, A.M.; Luck, M.** Towards motivation-based plan evaluation [Conference] // Proceedings of the Sixteenth International FLAIRS Conference. - Miami Beach, FL : AAAI Press, 2003. - pp. 298-302.
- Cohen, P. R.; Levesque, H. J.;** Confirmation and joint action [Conference] // In Proceedings of the International Joint Conference on Artificial Intelligence. - 1991.
- Cox M.T. and Veloso M.M.** Goal transformations in continuous planning [Conference] // Proceedings of the Fall Symposium on Distributed. - Menlo Park, CA : AAAI Press, 1998. - pp. 23-30.
- Cox M.T.** Perpetual self-aware cognitive agents [Article] // AI Magazine, 28(1). - 2007. - pp. 32-45.

Dearden, R.; Meuleau, N.; Ramakrishnan, S.; Smith, D.; Washington, R.

Incremental contingency planning [Conference] // In ICAPS-03: Proceedings of the Workshop on Planning under Uncertainty and Incomplete Information. - Trento, Italy : [s.n.], 2003. - pp. 38-47.

Dereszynski, E.; Hostetler, J.; Fern, A.; Hoang, T. D.; Udarbe, M.; Learning

probabilistic behavior models in real-time strategy games [Conference] // Proceedings of the conference on AI and Interactive Digital Entertainment (AIIDE). - [s.l.] : AAAI Press, 2011.

Dilts M. and Muñoz-Avila H. Reducing the memory footprint of temporal difference

learning over finitely many states by using case-based generalization [Conference] // Proceedings of the Eighteenth International Conference on Case-Based Reasoning. - Alessandria, Italy : Springer, 2010. - pp. 81-95.

Erol K., Hendler J. and Nau D. S. HTN planning: Complexity and expressivity

[Conference] // National Conference on Artificial Intelligence. - 1994.

Fox, M.; Gerevini, A.; Long, D.; Serina, I. Plan stability: Replanning versus plan

repair [Conference] // Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling. - Cumbria, UK : AAAI Press, 2006. - pp. 212-221.

Gabel T. and Riedmiller M. CBR for state value function approximation in

reinforcement learning [Conference] // Proceedings of the Sixth International Conference on Case-Based Reasoning. - Chicago, USA : Springer, 2005. - pp. 206–220.

Gabel, T.; Riedmiller, M. An analysis of casebased value function approximation by approximating state transition graphs [Conference] // Proceedings of the Seventh International Conference on Case-Based Reasoning. - Belfast, Northern Ireland : Springer, 2007. - pp. 344-358.

Ghallab, M.; Nau, D.S.; Traverso, P. Automated planning: Theory and practice [Book]. - San Mateo, CA : Morgan Kaufmann, 2004.

Goldman R.P. and Boddy M.S. Expressive planning and explicit knowledge [Conference] // Proceedings of the Third International Conference on Artificial Intelligence Planning Systems. - Edinburgh, Scotland : AAAI Press, 1996. - pp. 110-117.

Hierarchical task network [Online] // Wikipedia. - June 13, 2013. - July 10, 2013. - http://en.wikipedia.org/wiki/Hierarchical_task_network.

Hoang, H.; Lee-Urban, S.; Muñoz-Avila, H. Hierarchical plan representations for encoding strategic game AI [Conference] // Proceedings of the First Conference on Artificial Intelligence and Interactive Digital Entertainment. - Marina del Rey, CA : AAAI Press, 2005. - pp. 63-68.

Hsieh, J.L.; Sun, C.T.; Building a player strategy model by analyzing replays of real-time strategy games [Conference] // IEEE International Joint Conference on Neural Networks. - 2008.

Jaidee, U.; Munoz-Avila, H. Modeling Unit Classes as Agents in Real-Time Strategy Games [Conference] // Proceedings of the Ninth Annual AAAI Conference on

Artificial Intelligence and Interactive Digital Entertainment. - Boston, MA : [s.n.], 2013.

Jaidee, U.; Munoz-Avila, H.; Aha, D.W. Case-based goal-driven coordination of multiple learning agents [Conference] // Proceedings of the Twenty-First International Conference on Case-Based Reasoning. - Saratoga Springs, NY : Springer, 2013. - (Nominee: Best Paper Award).

Jaidee, U.; Munoz-Avila, H.; Aha, D.W. Case-Based Learning in Goal-Driven Autonomy Agents for Real-Time Strategy Combat Tasks [Conference] // In Proceedings of the ICCBR-11 workshop on Case-Based Reasoning for Computer Games. - Greenwich, London : [s.n.], 2011a.

Jaidee, U.; Munoz-Avila, H.; Aha, D.W. Integrated learning for goal-driven autonomy [Conference] // In Proceedings of the Twenty-Second International Conference on Artificial Intelligence. - Barcelona, Spain : AAAI Press, 2011.

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

Juell, P.; Paulson, P. Using reinforcement learning for similarity assessment in case-based systems [Article] // IEEE Intelligent Systems, 18(4). - 2003. - pp. 60-67.

Klenk M., Molineaux M. and Aha D.W. Planning in dynamic environments: Extending HTNs with nonlinear continuous effects [Conference]. - Atlanta, GA : AAAI Press, 2010.

- Lenz, M.; Bartsch-Spörl, B.; Burkhard, H.-D.; Wess, S.** Case-Based Reasoning Technology [Book]. - Berlin : Springer, 1998.
- Levesque, H. J.; Cohen, P. R.; Nunes, J.;** On acting together [Conference] // In Proceedings of the National Conference on Artificial Intelligence. - Menlo Park, Calif : AAAI press, 1990.
- Lopez de Mántaras, R.; McSherry, D.; Bridge, D.; Leake, D.; Smyth, B.; Craw, S.; Faltings, B.; Maher, M. L.; Cox, M. T.; Forbus, K.; Keane, M.; Aamodt, A.; Watson, I.** Retrieval, reuse and retention in case-based reasoning [Book Section] // Knowledge Engineering Review. - [s.l.] : Cambridge University Press, 2005. - Vol. 20(3).
- Marthi, B.; Russell, S.; Latham, D.; Guestrin, C.;** Concurrent hierarchical reinforcement learning [Conference] // In Proceedings of the 20th national conference on Artificial intelligence (AAAI-05). - [s.l.] : AAAI Press, 2005.
- Meneguzzi, F.R.; Luck, M.** Motivations as an abstraction of meta-level reasoning [Conference] // Proceedings of the Fifth International Central and Eastern European Conference on Multi-Agent Systems. - Leipzig, Germany : Springer, 2007. - pp. 204-214.
- Molineaux M., Kuter U. and Klenk M.** What just happened? Explaining the past in planning and execution [Conference] // Explanation-Aware Computing: Papers from the IJCAI Workshop. - Barcelona, Spain : [s.n.], 2011.

- Molineaux, M.; Klenk, M.; Aha, D.W.** Goal driven autonomy in a Navy strategy simulation [Conference] // In Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence. - Atlanta, GA : AAAI Press, 2010.
- Moltke H.K.B.G. von. Militarische werke** Moltke on the art of war: Selected writings [Book] / ed. Hughes D.J.. - Novato, CA : Presidio Press, 1993.
- Muñoz-Avila, H.; Aha, D.W.; Jaidee, U.; Carter, E.;** Goal directed autonomy with case-based reasoning [Conference] // Proceedings of the Eighteenth International Conference on Case-Based Reasoning. - Alessandria, Italy : Springer, 2010. - pp. 228-241.
- Munoz-Avila, H.; Aha, D.W.; Jaidee, U.; Klenk, M.; Molineaux, M.** Applying goal directed autonomy to a team shooter game. [Conference] // the Twenty-Third Florida Artificial Intelligence Research Society Conference. - Daytona Beach, FL : AAAI Press, 2010. - pp. 465-470.
- Myers K.L.** CPEF: A continuous planning and execution framework [Article] // AI Magazine, 20(4). - 1999. - pp. 63-69.
- Nau D.S.** Current trends in automated planning [Article] // AI Magazine. - 2007. - pp. 43-58.
- One Crafty Mother** Praying for the Burn [Online] // One Crafty Mother. - April 5, 2010. - June 8, 2013. -
http://www.onecraftymother.com/2010_04_01_archive.html.
- Othman, N.; Decraene, J.; Cai, W.; Hu, N.; Gouillard, A.;** Simulation-based optimization of StarCraft tactical AI through evolutionary computation

- [Conference] // Proceedings of IEEE Symposium on Computational Intelligence and Games (CIG). - 2012.
- Perez A. U.** Multi-Reactive Planning for Real-Time Strategy Games [Report] : MS Thesis / Universitat Autònoma de Barcelona. - 2011.
- Ponsen, M.; Munoz-Avila, H.; Spronk, P.; Aha, D.** Automatically generating game tactics with evolutionary learning [Article] // AI Magazine. - [s.l.] : AAAI Press, 2006.
- Powell J., Molineaux. M. and Aha D.W.** Active and interactive discovery of goal selection knowledge [Conference] // To appear in Proceedings of the Twentieth Fourth Conference of the Florida AI Research Society. - West Palm Beach, FL : AAAI Press, 2011.
- Radnitzky Gerard and Bartley William Warren** Evolutionary Epistemology, Rationality, and the Sociology of Knowledge [Book]. - [s.l.] : Open Court Publishing Company, 1993. - pp. 94-108.
- Ram, A.; Santamaria, J.C.** Continuous casebased reasoning [Journal] // Artificial Intelligence, 90(1-2). - 1997. - pp. 25-77.
- Richter Michael M.** Similarity [Journal] // Case-Based Reasoning for Signals and Imaging / ed. Perner Petra. - [s.l.] : Springer Verlag, 2007. - pp. 25-90.
- Rørmark R.** Thanatos - A learning RTS game AI [Report] : MS Thesis / University of Oslo. - 2009.
- Russell, S.; Norvig, P.** Artificial Intelligence [Book]. - [s.l.] : Pearson Education, Inc., 2003. - pp. 1-5.

Scoring and Ranking [Online] // battle.net. - August 7, 2013. -

<http://classic.battle.net/war2/basic/score.shtml>.

Scott B. Architecting an RTS AI [Book Section] // AI Game Programming Wisdom / ed. Robin Steve. - [s.l.] : Charles River Media, 2002.

Sharma, M.; Holmes, M.; Santamaria, J.; Irani, A.; Isbell, C.; Ram, A.; Transfer learning in real-time strategy games using hybrid CBR/RL [Conference] // In Proceedings of the 20th international joint conference on Artificial intelligence (IJCAI-07). - [s.l.] : Morgan Kaufmann Publishers Inc., 2007.

Si, J.; Barto, A.; Powell, W.; Wunsch, D. Reinforcement Learning in Large, High-Dimensional State Spaces [Book Section] // Handbook of Learning and Approximate Dynamic Programming. - [s.l.] : Wiley, 2012.

Smith M., Lee-Urban S. and Muñoz-Avila H. RETALIATE: Learning winning policies in first-person shooter games. [Conference] // Proceedings of the Nineteenth Innovative Applications of AI Conference. - Vancouver, British Columbia, Canada : AAAI Press, 2007. - pp. 1801-1806.

Spronck P. Dynamic Scripting [Book Section] // AI Game Programming Wisdom 3 / ed. Rabin Steve. - Hingham, MA. : Charles River Media, 2006.

Sutton R. and Barto S. Reinforcement Learning [Book]. - [s.l.] : MIT Press, 1998.

Synnaeve, G.; Bessière, P.; A Bayesian Model for RTS Units Control applied to StarCraft [Conference] // IEEE Conference on Computational Intelligence and Games. - Seoul, South Korea : [s.n.], 2011.

- Tambe M.** Towards Flexible Teamwork [Journal] // Journal of Artificial Intelligence Research. - [s.l.] : Morgan Kaufmann, 1997. - Vol. 7. - pp. 83-124.
- van den Briel, M.; Sanchez Nigenda, R.; Do, M.B.; Kambhampati, S.** Effective approaches for partial satisfaction (over-subscription) planning [Conference] // Proceedings of the Nineteenth National Conference on Artificial Intelligence. - San Jose, CA : AAAI Press, 2004. - pp. 562-569.
- Warcraft II structures [Online] // WoWWiki. - July 23, 2013. - http://www.wowwiki.com/Warcraft_II_structures.
- Warcraft II units [Online] // WoWWiki. - July 21, 2013. - http://www.wowwiki.com/Warcraft_II_units.
- Warfield, I.; Hogg, C.; Lee-Urban, S.; Munoz-Avila, H.** Adaptation of hierarchical task network plans [Conference] // Proceedings of the Twentieth Flairs International Conference. - Key West, FL : AAAI Press, 2007. - pp. 429-434.
- Wargus [Online]. - January 2012. - <http://wargus.sourceforge.net/>.
- Watkins C. J. C. H.** Learning from Delayed Rewards [Journal] // Ph.D. thesis, Cambridge University. - 1989.
- Weber B., Mateas M. and Jhala A.** Applying goal-driven autonomy to StarCraft [Conference] // In Proceedings of the Sixth Conference on Artificial Intelligence and Interactive Digital Entertainment. - Stanford, CA : AAAI Press, 2010.
- Weber, B.; Mateas, M.;** A data mining approach to strategy prediction [Conference] // Proceedings of IEEE Symposium on Computational Intelligence and Games (CIG). - 2009.

- Weber, B.; Mateas, M.; Jhala, A.** Learning from demonstration for goal-driven autonomy [Conference] // In Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence. - Toronto (Ontario), Canada : AAAI Press, 2012.
- Weber, B.; Mateas, M.; Jhala, A.** Reactive Planning Idioms for Multi-Scale Game AI [Conference] // IEEE Conference on Computational Intelligence and Games (CIG 2010). - 2010.
- Wender, S.; Watson, I.** Applying reinforcement learning to small-scale combat in the real-time strategy game starcraft:Broodwar [Conference] // Proceedings of IEEE Symposium on Computational Intelligence and Games (CIG). - 2012.
- Young, J.; Hawes, N.** Evolutionary learning of goal priorities in a real-time strategy game [Conference] // Proceedings of the Conference on Artificial Intelligence and Interactive Digital Entertainment. - Stanford, CA : AAAI Press, 2012.





APPENDIX A

WARGUS UNITS AND STRUCTURES





A.1. Wargus Units


Units in Wargus can be categorized into three main types: land, naval and air units. Human and Orc units are all produced at different structures and are essentially quite balanced in their abilities. Table A-1 below shows the properties of each unit in both races. The table shows only units that are used in the experiments of this dissertation.



Table A-1: List of Wargus Units and their properties in both Human (left column) and Orc (right column) (Blizzard Entertainment, 1999) (WoWWiki1)

Peasant		Peon	
			
Race	Human	Race	Orc
Type	Land	Type	Land
Unit		Unit	
Statistics		Statistics	
Hit Points	30	Hit Points	30
Armor	0	Armor	0
Speed	10	Speed	10
Production		Production	
Gold	400	Gold	400
Lumber	0	Lumber	0
Food	1	Food	1
Produced at	Town	Produced at	Great
Hall		Hall	



Peasant		Peon	
Build time	45	Build time	45
seconds		seconds	
Combat		Combat	
Basic Damage	2-9	Basic Damage	2-9
Piercing Damage	2	Piercing Damage	2
Range	1	Range	1
Upgradeability		Upgradeability	
Upgrades Into	cannot be upgraded	Upgrades Into	cannot be upgraded





Footman		Grunt	
 		 	
Race	Human	Race	Orc
Type	Land	Type	Land
Unit		Unit	
Statistics		Statistics	
Hit Points	60	Hit Points	60
Armor	2	Armor	2
Speed	10	Speed	10
Production		Production	
Gold	400	Gold	400
Lumber	0	Lumber	0
Food	1	Food	1
Produced at	(Human)	Produced at	(Orc)
Barracks		Barracks	
Build time	60	Build time	60
seconds		seconds	
Combat		Combat	
Basic Damage	6	Basic Damage	6
Piercing Damage	3	Piercing Damage	3
Range	1	Range	1
Upgradeability		Upgradeability	
Upgrades Into	cannot be upgraded	Upgrades Into	cannot be upgraded

Elven Archer		Troll Axethrower	
	 		 
Race	Human	Race	Orc
Type	Land	Type	Land
Unit		Unit	
Statistics		Statistics	
Hit Points	40	Hit Points	40
Armor	2	Armor	2
Speed	10	Speed	10
Production		Production	
Gold	500	Gold	500
Lumber	50	Lumber	50
Food	1	Food	1
Produced at	(Human) Barracks	Produced at	(Orc) Barracks
Build time	70	Build time	70
seconds		seconds	
Combat		Combat	
Basic Damage	3-9	Basic Damage	3-9
Piercing Damage	6	Piercing Damage	6
Range	4	Range	4
Upgradeability		Upgradeability	
Upgrades Into	Elven Ranger	Upgrades Into	Elven Ranger

Knight		Ogre	
	 		 
Race	Human	Race	Orc
Type	Land	Type	Land
Unit		Unit	
Statistics		Statistics	
Hit Points	90	Hit Points	90
Armor	4	Armor	4
Speed	13	Speed	13
Production		Production	
Gold	800	Gold	800
Lumber	100	Lumber	100
Food	1	Food	1

Produced at	(Human) Barracks	Produced at	(Orc) Barracks
Build time	90	Build time	90
seconds		seconds	
Combat		Combat	
Basic Damage	2-12	Basic Damage	2-12
Piercing Damage	4	Piercing Damage	4
Range	1	Range	1
Upgradeability		Upgradeability	
Upgrades Into	Paladin	Upgrades Into	Ogre Mage

Ballista		Catapult	
			
Race	Human	Race	Orc
Type	Land	Type	Land
Unit		Unit	
Statistics		Statistics	
Hit Points	110	Hit Points	110
Armor	0	Armor	0
Speed	5	Speed	5
Production		Production	
Gold	900	Gold	900
Lumber	300	Lumber	300
Food	1	Food	1
Produced at	(Human)	Produced at	(Human)
Barracks		Barracks	
Build time	250	Build time	250
seconds		seconds	
Combat		Combat	
Basic Damage	25-80	Basic Damage	25-80
Piercing Damage	0	Piercing Damage	0
Range	8	Range	8
Upgradeability		Upgradeability	
Upgrades Into	cannot be upgraded	Upgrades Into	cannot be upgraded

Gryphon Rider		Dragon	
			
Race	Human	Race	Orc
Type	Air Unit	Type	Air Unit
Statistics		Statistics	
Hit Points	100	Hit Points	100
Armor	0	Armor	0
Speed	14	Speed	14
Production		Production	
Gold	2500	Gold	2500
Lumber	0	Lumber	0
Food	1	Food	1
Produced at	Aviary	Produced at	Dragon Roost
Build time seconds	250	Build time seconds	250
Combat		Combat	
Basic Damage	8-16	Basic Damage	8-16
Piercing Damage	0	Piercing Damage	0
Range	4	Range	4
Upgradeability		Upgradeability	
Upgrades Into	cannot be upgraded	Upgrades Into	cannot be upgraded


A.2. Wargus Structures



Wargus structures are land-based and sea-based, capable of training military units, aircraft, and sea-faring vehicles.

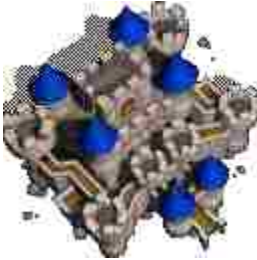

All human structures can be created by a peasant while all orc structures can be created by a peon. Table A-2 below shows the properties of each structure in both races. The table shows only structures that are used in the experiments of this dissertation.

Other than human and orc structures, there are some structures that are neutral. However, gold mine is the only one neutral structure that is used in the experiments of this dissertation.



Table A-2: List of Wargus Structures and their properties in both Human (left column) and Orc (right column) (Blizzard Entertainment, 1999) (WoWWiki2)



Town Hall		Great Hall	
			
Hit Points	1200	Hit Points	1200
Production		Production	
Gold	1200	Gold	1200
Lumber	800	Lumber	800
Oil	0	Oil	0
Upgradeability		Upgradeability	
Upgrades to	Keep	Upgrades to	Stronghold
Training Ability		Training Ability	
Peasant		Peon	



Keep	Stronghold
	
Hit Points 1400	Hit Points 1400
Prerequisite (Human) Barracks	Prerequisite (Orc) Barracks
Production	Production
Gold 2000	Gold 2000
Lumber 1000	Lumber 1000
Oil 200	Oil 200
Upgradeability	Upgradeability
Upgrades to Castle	Upgrades to Fortress
Training Ability	Training Ability
Peasant	Peon
Allowance	Allowance
Stables, Gnomish Inventor	Ogre Mound, Goblin Alchemist



Castle	Fortress
	
Hit Points 1600	Hit Points 1600
Prerequisite (Human) Barracks, (Human) Blacksmith, Elven Lumber Mill, Stables	Prerequisite (Orc) Barracks, (Orc) Blacksmith, Troll Lumber Mill, Ogre Mound
Production	Production
Gold 2500	Gold 2500
Lumber 1200	Lumber 1200
Oil 500	Oil 500
Training Ability	Training Ability



Castle	Fortress
Peasant Allowance Gryphon Aviary, Mage Tower, Church	Peon Allowance Dragon Roost, Temple of the Damned, Altar of Storms

Chicken Farm	Pig Farm
	
Hit Points 400	Hit Points 400
Production	Production
Gold 500	Gold 500
Lumber 250	Lumber 250
Oil 0	Oil 0
Ability Feeding 4 Units	Ability Feeding 4 Units



(Human) Barracks	(Orc) Barracks
	
Hit Points 800	Hit Points 800
Production	Production
Gold 700	Gold 700
Lumber 400	Lumber 400
Oil 0	Oil 0
Training Ability Footman, Elven Archer/Ranger, Knight/Paladin, Ballista	Training Ability Grunt, Troll Axethrower/Berserker, Ogre/Ogre-Mage, Catapult



(Human) Blacksmith		(Orc) Blacksmith	
			
Hit Points	775	Hit Points	775
Production		Production	
Gold	800	Gold	800
Lumber	450	Lumber	450
Oil	100	Oil	100
Research Ability		Research Ability	
Weapons 1, Weapons 2, Armor 1, Armor 2		Weapons 1, Weapons 2, Armor 1, Armor 2	

Elven Lumber Mill		Troll Lumber Mill	
			
Hit Points	600	Hit Points	600
Production		Production	
Gold	600	Gold	600
Lumber	450	Lumber	450
Oil	0	Oil	0
Research Ability		Research Ability	
Arrows 1, Arrows 2, Ranger, Ranger Scouting, Longbow, Ranger Masksmanship		Throwing Axes 1, Throwing Axes 2, Troll Berserker, Berserker Scouting, Lighter Axes, Troll Regeneration	

Stables		Ogre Mound	
			
Hit Points	500	Hit Points	500

Stables		Ogre Mound	
Prerequisite	Keep	Prerequisite	Stronghold
Production		Production	
Gold	1000	Gold	1000
Lumber	300	Lumber	300
Oil	0	Oil	0
Allowance		Allowance	
Knights / Paladins		Ogres / Ogre-Mages	

Gryphon Aviary		Dragon Roost	
			
Hit Points	500	Hit Points	500
Prerequisite	Castle	Prerequisite	Fortress
Production		Production	
Gold	1000	Gold	1000
Lumber	400	Lumber	400
Oil	0	Oil	0
Training Ability		Training Ability	
Gryphon Rider		Dragon	

Church		Altar of Storms	
			
Hit Points	700	Hit Points	700
Prerequisite	Castle	Prerequisite	Fortress
Production		Production	
Gold	900	Gold	900
Lumber	500	Lumber	500
Oil	0	Oil	0

Research Ability
Paladin, Healing, Exorcism

Research Ability
Ogre-Mage, Bloodlust, Runes

Gold Mine



Hit Points 25500

APPENDIX B

SCORING IN WARGUS

To earn points in Wargus, units in a team have to kill units or structures of its enemy teams. The total point score of team T is based on the numbers and types of enemy units that team T has killed. Points gained from killing specific units are shown in Table B-1. There is one type of the score that does not show in the table. Namely, if one team wins, the winning team will earn an additional 500 points.

Table B-1: Points earned from killing specific units or structures (Sco13).

Units / Structures	Score	Units / Structures	Score
Tower	95	Wall	1
Critter	1	Farm	100
Peasant/Peon	30	Lumber mill	150
Flying Machine/Zeppelin	40	Runestone	150
Tanker	40	Barracks	160
Footman/Grunt	50	Oil Rig	160
Transport	50	Blacksmith	170
Archer/Axe Thrower	60	Shipyard	170
Ranger/Berserker	70	Foundry	200
Dwarves/Sappers	100	Guard Tower	200

Units / Structures	Score	Units / Structures	Score
Knight/Ogre	100	Refinery	200
Ballista/Catapult	100	Town Hall	200
Mage/Death Knight	100	Stables/Ogre Mound	210
Demon	100	Inventor/Alchemist	230
Paladin/Ogre Mage	110	Church/Altar	240
Legendary Hero	120	Wizard's Tower/Temple	240
Submarine/Turtle	120	Cannon Tower	250
Destroyer	150	Aviary/Roost	280
Gryphon/Dragon	150	Keep/Stronghold	600
Battleship/Juggernaut	300	Castle/Fortress	1500

CURRICULUM VITAE

Ulit Jaidee was born on July 6, 1975 to Mr. Kit Jaidee and Mrs. Jongdee Boonyupakorn in Bang Phlad, Bangkok, Thailand. In 1997, he earned a B.Sc. in Applied Computer Science (Second Class Honors) from King Mongkut's Institute of Technology North Bangkok. He has worked for the Thai government as a Lecturer at Department of Applied Computer Science and Information Technology, Faculty of Applied Science, King Mongkut's Institute of Technology North Bangkok (KMUTNB), the same place where he earned his bachelor degree. In 2003, he earned a M.Eng. in Computer Engineering (KMUTNB scholarship) from King Mongkut's University of Technology Thonburi. In 2008, he earned a full scholarship from Ministry of Science and Technology to pursue his Ph.D. in Computer Engineering at Lehigh University.

List of Publication

- Jaidee, U., Munoz-Avila, H., & Aha, D.W. (2013). Case-based goal-driven coordination of multiple learning agents. *Proceedings of the Twenty-First International Conference on Case-Based Reasoning*. Saratoga Springs, NY: Springer.

- ❑ Jaidee, U., Munoz-Avila, H. (2013). Modeling Unit Classes as Agents in Real-Time Strategy Games. *Proceedings of the Ninth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Boston, MA.
- ❑ Jaidee, U., Munoz-Avila, H., & Aha, D.W. (2012). Learning and reusing goal-specific policies for goal-driven autonomy. *Proceedings of the Twentieth International Conference on Case-Based Reasoning* (pp. 182-195). Lyon, France: Springer.
- ❑ Jaidee, U., Munoz-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.
- ❑ Jaidee, U., Munoz-Avila, H., & Aha, D.W. (2011). Case-based learning in goal-driven agents for real-time strategy combat tasks. In M.W. Floyd & A.A. Sánchez-Ruiz (Eds.) *Case-Based Reasoning in Computer Games: Papers from the Nineteen International Conferences on Case-Based Reasoning Workshop*. U. Greenwich: London, UK.
- ❑ Munoz-Avila, H., Jaidee, U., Aha, D.W., & Carter, E. (2010). Goal directed autonomy with case-based reasoning. *Proceedings of the Eighteenth International Conference on Case-Based Reasoning* (pp. 228-241). Alessandria, Italy: Springer.
- ❑ Munoz-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.

- Jaidee, U., Madarasmi, S. (2004). A Coarse-and-Fine Approach to Stereo Matching, *Proceedings of the International Technical Conference on Circuits/Systems, Computers and Communications*, Sendai, Japan.