Learning from Demonstration for Goal-Driven Autonomy

Ben G. Weber
Expressive Intelligence Studio
UC Santa Cruz
bweber@soe.ucsc.edu

Michael Mateas
Expressive Intelligence Studio
UC Santa Cruz
michaelm@soe.ucsc.edu

Arnav Jhala
Computational Cinematics Studio
UC Santa Cruz
jhala@soe.ucsc.edu

Abstract

Goal-driven autonomy (GDA) is a conceptual model for creating an autonomous agent that monitors a set of expectations during plan execution, detects when discrepancies occur, builds explanations for the cause of failures, and formulates new goals to pursue when planning failures arise. While this framework enables the development of agents that can operate in complex and dynamic environments, implementing the logic for each of the subtasks in the model requires substantial domain engineering. We present a method using case-based reasoning and intent recognition in order to build GDA agents that learn from demonstrations. Our approach reduces the amount of domain engineering necessary to implement GDA agents and learns expectations, explanations, and goals from expert demonstrations. We have applied this approach to build an agent for the real-time strategy game StarCraft. Our results show that integrating the GDA conceptual model into the agent greatly improves its win rate.

Introduction

One of the goals of AI is to create agents capable of operating in complex domains, which require performing actions at multiple scales and reacting to exogenous events in real-time. Goal-driven autonomy (GDA) is a model that works towards this goal by building reflective agents that reason about the goals they try to achieve (Molineaux, Klenk, and Aha 2010; Muñoz-Avila et al. 2010b). The model is used to create agents that monitor their own execution in order to determine which goals to pursue, identify when to select new goals, and explain why new goals should be pursued.

A GDA agent integrates subtasks that generate expectations for plans, detect discrepancies, generate explanations, and formulate goals. One of the challenges in applying the model is that a comprehensive GDA agent requires substantial domain knowledge in order to implement these processes. For sufficiently complex domains, it is impractical for a domain expert to specify expectations for every action, explanations for every discrepancy, and goals to pursue for all possible states (Jaidee, Muñoz-Avila, and Aha 2011b). Integrating learning reduces the amount of domain engineering necessary to build GDA agents. We introduce a method for building GDA agents that learn from expert demonstrations. Our system represents world state, explanations, and goals as feature vectors, which can be retrieved from gameplay examples. Rather than requiring a domain expert to specify expectations, explanations, and goals, our approach learns how to generate these objects from demonstrations. We use case-based goal formulation (Weber, Mateas, and Jhala 2010b) to select goals for the agent to pursue and anticipate the actions of adversaries. The components are integrated within a reactive planner that manages the active goals of the agent.

We apply learning from demonstration for GDA to an agent that plays the real-time strategy (RTS) game StarCraft. Using the GDA model enables the agent to emulate a subset of the cognitive processes necessary for RTS gameplay. RTS games are an excellent domain for AI research because they contain many real-world properties (Buro 2003), enable investigation of integrative AI approaches (Laird and VanLent 2001), and provide huge data sets available for analysis in the form of professional player replays (Weber and Mateas 2009). We extract examples from professional replays enabling the agent to estimate hidden game state, anticipate opponent actions, and select goals to pursue. Our results show that integrating these processes into the agent using the GDA model greatly improves the gameplay performance of the system, while reducing the domain engineering task.

Goal-Driven Autonomy

The goal-driven autonomy conceptual model provides a framework for creating agents capable of responding to unanticipated failures during plan execution in complex, dynamic environments (Molineaux, Klenk, and Aha 2010). It is motivated by Cox's claim that agents should reason about their goals in order to continuously operate with independence (Cox 2007). The conceptual model specifies subtasks that enable an agent to detect, reason about, and respond to unanticipated events. However, it makes no commitment to specific algorithms.

The GDA model extends Nau's (2007) model of online planning by identifying specific subtasks within the controller of an agent, as shown in Figure 1. The controller interacts with an execution environment that provides observa-

Copyright © 2012, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

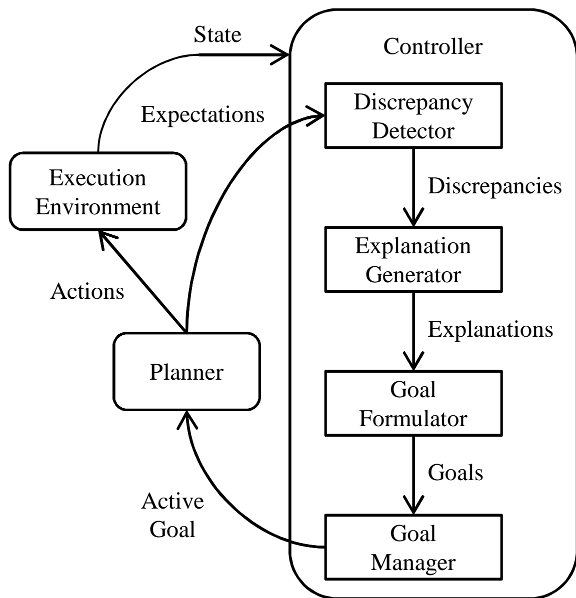

Figure 1: A GDA agent interacts with a planner and execution environment and incorporates a controller with subtasks for monitoring plan execution.

tions of world state and a planner that generates sequences of actions to achieve the agent's current active goal. In order to identify when planning failures occur, a GDA agent requires the planning component to generate an expectation of world state after executing each action in the execution environment.

A GDA agent starts by passing an initial goal to the planner. The planner generates a plan consisting of a set of actions, a, and expectations, p. As actions are performed in the execution environment, the discrepancy detector checks if the resulting world state, s, matches the expected world state. When a discrepancy is detected between the expected and actual world states, the agent creates a discrepancy, d, which is passed to the explanation generator. Given a discrepancy, the explanation generator builds an explanation, e, of why the failure occurred and passes it to the goal formulator. The goal formulator takes an explanation and formulates a goal, g, in response to the explanation. The goal is then passed to the goal manager, which is responsible for selecting the agent's active goal.

The model has been used to build autonomous agents for naval strategy simulations (Molineaux, Klenk, and Aha 2010), first-person shooters (Muñoz-Avila et al. 2010b; Muñoz-Avila and Aha 2010), and real-time strategy games (Jaidee, Muñoz-Avila, and Aha 2011a). In previous work we applied the GDA model to StarCraft, and hand-authored rules for generating expectations, explanations, and goals (Weber, Mateas, and Jhala 2010a). These approaches require a domain expert to specify expectations for every action, explanations for every discrepancy, and goals for every explanation. To reduce the amount of domain engineering required to author GDA agents, Muñoz-Avila et al. (2010a) use case-based reasoning to predict expected states and formulate goals in response to discrepancies. Our approach generates expectations, explanations and goals as well, but learns a case library by directly capturing examples from professional demonstrations rather than online learning. Another difference in our approach is that intent recognition is used to generate explanations of future world state rather than generating explanations of the current world state.

Previous work on goal-driven autonomy has explored case-based reasoning for learning explanations and goals. Jaidee et al. (2011b) used reinforcement learning to automatically acquire goal selection knowledge, but showed that the learned model was less effective than an expert-specified model. In this approach the case library is defined manually and the system learns expected states and values for goals online. This differs from our approach that populates a case library by extracting examples from gameplay demonstrations in order to reduce the domain engineering task.

Case-Based Goal Formulation

Case-based goal formulation is a method for performing goal formulation using a collection of demonstrations (Weber, Mateas, and Jhala 2010b). It is similar to sequential instance-based learning (Epstein and Shih 1998), but selects goals rather than actions. We use case-based goal formulation to select goals for an agent to pursue and to model the actions of opponents in adversarial domains. It is defined as follows:

> The agent's goal state, g, is a vector computed by retrieving the most similar case, q, to the world state, s, and adding the difference between q and its future state, q', which is the state after n actions have been applied to q.

Formally:

$$q = \underset{c \in L}{\operatorname{argmin}}\ distance(s, c)$$

$$g = s + (q' - q)$$

where g is a numerical vector representing the goal state, c is a case in the case library, L, and the distance function may be a domain independent or domain specific distance metric. We refer to the number of actions, n, needed to achieve the generated goal state as the *planning window size*.

Trace Algorithm

The Trace algorithm implements case-based goal formulation using traces of world state. A *trace* is a list of tuples containing world state and actions, and is a single episode demonstrating how to perform a task in a domain. For example, a game replay is a trace that contains the current game state and player actions executed in each game frame.

The algorithm utilizes a case representation where each case is a numerical feature vector that describes the goal state at a specific time. The algorithm can determine the actions performed between different time steps by analyzing the difference between feature vectors. Cases from a trace are indexed using the time step feature. This enables efficient lookup of q' once a case, q, has been selected. Assuming that

the retrieved case occurred at time t in the trace, $q\prime$ is defined by the world state at time $t+n$. Since the algorithm uses a feature vector representation, g can be computed as follows:

$$g(x) = s(x) + (q_{t+n}(x) - q_t(x))$$

where x is a feature in the case representation.

Learning GDA Subtasks

We present an approach for learning expectations, explanations and goals for a GDA agent from demonstration. Our approach focuses on adversarial domains, in which planning failures or exogenous events occur due to the actions of other agents. We use two case libraries: the adversary library, AL, provides examples for adversary actions and is used for intent recognition by performing goal formulation on the adversary's state, while the goal library, GL, is used to select goals for the agent to pursue. By using case-based goal formulation to retrieve goals from the adversarial case library, we can build predictions of the future actions of an adversary. We refer to the number of actions, m, that our retrieval mechanism looks ahead at the adversary's actions as the *adversary planning window*.

Our approach utilizes a numerical feature vector representation to describe world state. The representation includes features for describing the agent's world state and a single adversary's world state. By performing actions in the execution environment, an agent can create or destroy objects in the execution environment. A non-zero feature value describes the existence of a specific object type in the environment.

Building Expectations

We define an expectation, p, as an anticipated change in a single world state feature caused by an adversary's actions. Each expectation represents an anticipated change in a single adversarial feature at a specific time. Specifically, expectations are beliefs that the adversary will perform an action that introduces a new object type, x into the execution environment at a specific time, τ:

$$p(x, \tau) : s_t(x) > 0, t \geq \tau$$

where x is a feature that describes the adversary's state. When an expectation is generated, the agent anticipates the presence of object type x in the execution environment after time τ.

In order to build expectations, the system retrieves the most similar case from the adversary case library and identifies times in which new objects are introduced into the environment. Our system triggers expectation generation when a new active goal is selected by the goal manager. A set of expectations, π, is generated as follows:

$$q = \underset{c \in AL}{\operatorname{argmin}} \; distance(s, c)$$
$$\pi = \{p(x, \tau) : \tau = \min_v [q_v(x) > 0, v > t]$$
$$\mid \forall x, q_t(x) = 0, q_{t+m}(x) > 0\}$$

The retrieval process occurs at time t with a look-ahead window of size m. It generates an expectation for each feature describing an adversarial object type, x, in the retrieved case, q, which changes from a zero value to a positive value between time t and time $t+m$. The time selected for the expectation is the smallest time step, τ, where the feature has a positive value.

Detecting Discrepancies

We define a discrepancy, d, as an expectation that is not met. This situation occurs when an adversary does not perform actions causing an anticipated state change. Our system creates a discrepancy when the following conditions are met:

$$p(x, \tau), s_t(x) = 0, t \geq \tau + \delta$$

where δ is a discrepancy period, which provides a buffer period for observing state changes. This situation occurs when an object of type x has not been observed in the environment before time $\tau + \delta$. Because StarCraft enforces partial observability through a *fog-of-war*, it is possible for our system to incorrectly detect discrepancies due to insufficient vision of the opponent state.

Generating Explanations

Our system generates explanations when the active goal completes or a discrepancy is detected. It creates explanations by applying goal formulation to the adversary, using the adversary case library. We define an explanation, e, as a prediction of future world state resulting from the execution of an adversary's actions. Our system uses explanations to attempt to identify how the adversary will change world state. This definition differs from previous work in GDA, which uses explanations in order to identify why discrepancies occur using current world state (Molineaux, Klenk, and Aha 2010). Both our approach and previous work use explanations as input to the goal formulation subtask. We use case-based goal formulation to create explanations as follows:

$$q = \underset{c \in AL}{\operatorname{argmin}} \; distance(s, c)$$
$$e = s + (q_{t+m} - q_t)$$

where AL is the adversary goal library, m is the number of adversary actions to look ahead, and e is an anticipated future world state. Our system uses explanations in order to incorporate predictions about adversarial actions into the goal formulation process.

Formulating Goals

We define a goal, g, as a future world state for the agent to achieve. The input to the goal formulation component is the current world state and an explanation, e, which contains the anticipated future world state of the adversary. Goal formulation is triggered when the agent's active goal completes or a discrepancy is detected. Our system formulates goals as follows:

$$q = \underset{c \in GL}{\operatorname{argmin}} \; distance(e, c)$$
$$g = s + (q_{t+n} - q_t)$$

where GL is the goal case library and the input to the case retrieval similarity metric is an explanation of future world state. The output of the formulation process is a goal that is passed to the goal manager, which is responsible for selecting the agent's active goal.

Application to RTS Games

We applied goal-driven autonomy to build an agent for the RTS game StarCraft[1]. Real-time strategy games have many-real world properties including partial observability and an enormous decision complexity (Aha, Molineaux, and Ponsen 2005). StarCraft is an excellent domain for learning from demonstration, because there are a huge number of professional replays available for analysis.

Our agent is an extension of McCoy and Mateas' (2008) integrated agent framework. It is implemented in the ABL reactive planning language (Mateas and Stern 2002), which is similar to BDI architectures such as PRS (Rao and Georgeff 1995). We use ABL to implement goal management, because it provides mechanisms for supporting real-time plan execution and monitoring. Our system decomposes StarCraft gameplay into subproblems, and includes managers for handling each of these tasks. These subproblems include strategy selection, production, income management, tactics and reconnaissance. We use GDA to implement the strategy manager in the agent, which is responsible for selecting the agent's high-level strategy. The strategy manager selects which units to produce, structures to construct, and upgrades to research.

One of our goals in using the GDA model is to emulate a subset of the cognitive processes necessary for RTS gameplay (Weber, Mateas, and Jhala 2011). In StarCraft, players take the role of a commander in a military scenario and are given the objective of destroying all opponents. Performing well in StarCraft requires building anticipations of opponent actions, formulating plans to counter opponents, scouting the map in order uncover opponent state, and selecting a new strategy when the current strategy is no longer effective. Many of these processes map to subtasks in the GDA model.

Case Representation

Our case representation is a feature vector that tracks the number of units and buildings that each player has produced. For each player, there is a feature for each unit and building type, and the value of each feature is the number of the type that have been produced since the start of the game.

We collected thousands of professional-level replays from community websites and converted them to our case representation. The professional replays are the source of our case library. Replays were converted from Blizzard's proprietary binary format into text logs of game actions using a third-party tool. A subset of an example trace is shown in Table 1. An initial case, q_1, is generated with all values set to zero, except for the worker unit type. It is set to 4, because the player begins with 4 worker units. A new case is generated for each action that trains a unit or produces a building. The value of a new case is initially set to the value of the previous case, then the feature corresponding to the train or build action is incremented by one. Considering a subset of the features (# Workers, # Depots, # Barracks, # Marines), the example trace would produce the following cases:

Table 1: An example trace showing when a player performed build and train actions.

Frame	Player	Action
100	1	Train Worker
300	1	Build Depot
500	1	Train Worker
700	1	Build Barracks
900	1	Train Marine

$$q_1 =< 4, 0, 0, 0 >$$
$$q_2 =< 5, 0, 0, 0 >$$
$$q_3 =< 5, 1, 0, 0 >$$
$$q_4 =< 6, 1, 0, 0 >$$
$$q_5 =< 6, 1, 1, 0 >$$
$$q_6 =< 6, 1, 1, 1 >$$

The case library consists of 1,831 traces and 244,459 cases. Our approach differs from previous work, because cases are extracted from gameplay traces rather than defined manually (Muñoz-Avila et al. 2010a).

Our system uses the same case library contents for both the adversary case library, AL, and the goal library, GL. When retrieving cases using the adversary case library, AL, the *distance* function uses a subset of features, which describe adversary state. This feature subset is used by the expectation builder and explanation generator. The system generates expectations for features that correspond to changes in adversary state, and explanations of future state are computed based on changes in adversary state. When retrieving cases using the goal library, GL, the complete feature set is used by the *distance* function. During goal retrieval, the system computes the goal state based on changes in agent state, while ignoring adversary state.

Integrating GDA

Our agent starts a game by selecting an initial strategic goal, known as the build order, which becomes the agent's active goal. In our current implementation, the initial goal is provided by a domain expert. The active goal is passed to the goal manager, which is implemented in the ABL reactive planning language (Weber, Mateas, and Jhala 2010a). The different components in the system communicate using the reactive planner's working memory as a blackboard (Hayes-Roth 1985). The reactive planner is responsible for performing the actions in the game necessary to achieve the goal state. Plans are generated by the Trace algorithm, which retrieves actions directly from traces. Our approach differs from previous GDA implementations, because the same component is used for both goal formulation and plan generation.

[1]StarCraft and its expansion StarCraft: Brood War were developed by Blizzard Entertainment™.

The agent invokes the GDA subtask upon completion of the initial goal. During this process, the agent generates an explanation of the opponent's future game state and builds expectations of anticipated unit and building types. For example, the agent may expect the opponent to construct a *Starport*, and train a fleet of *Valkyrie* aircraft. This anticipated game state is passed to the goal formulator, which selects the next strategic goal for the agent to pursue. In this example, the agent would retrieve similar game situations in which air units are produced by the opponent and formulate a goal to counter air units. The retrieved goal state is then passed to the goal manager, becoming the current active goal, and the expectation of an opponent *Starport* is passed to the discrepancy detector.

After formulating a new goal and set of expectations, the agent enters a plan monitoring phase. During this process, the agent executes actions to pursue the active goal and checks for discrepancies. For example, if the agent observes an opponent *Starport*, then the expectation is validated and the agent continues to pursue its current goal. However, if the agent does not observe an opponent *Starport* after a discrepancy period has expired, then the expectation becomes invalid, triggering a discrepancy. Upon detecting a discrepancy or achieving the current goal, the agent invokes the explanation generation subtask.

There are three free parameters in our GDA implementation: the planning window size, n, specifies the size of plans resulting from goal formulation, the adversarial window size, m, specifies how far to look-ahead at opponent actions, and the discrepancy period, δ, specifies a buffer period for detecting discrepancies. In order to find suitable values for these parameters, we ran an ablation study in which GDA subtasks are individually integrated into the agent. We use a greedy search to select parameter values.

Evaluation

We evaluated our approach by integrating the GDA model in an agent that plays complete games of StarCraft. The GDA model was used for implementing the strategy manager, while the remaining managers in the agent remained unchanged. We ran an ablation study to demonstrate the importance of each GDA subtask in the agent.

The system was tested against the built-in AI of StarCraft and bots submitted to the AIIDE 2011 StarCraft AI Competition[2] including *Aiur*, *SPAR*, *Cromulent*, *Nova*, and *BTHAI*. The map pool consisted of six maps from the competition, which support two to four players and encourage a variety of play styles. For each trial in our experiments, we evaluated our agent against all permutations of bots and maps, resulting in 48 games. We utilize win ratio as a metric of agent performance.

Our ablation study evaluated four versions of the agent. In each experiment, we introduced an additional subtask of the GDA model into the agent. We evaluated the following configurations of the system:

- **Base Agent:** A base version of the agent with a fixed strategy goal and no GDA subtasks.

[2] http://www.StarCraftAICompetition.com

Table 2: Win rates from the goal formulation experiment show that the agent performed best with a planning window size of 15 actions.

Planning Window Size (n)	Win Ratio
0	0.73
1	0.79
5	0.88
10	0.85
15	0.91
20	0.88

- **Formulator:** A version of the agent that integrates the goal formulation subtask, with no look ahead or discrepancy detection.
- **Predictor:** A version of the agent that incorporates explanation generation and goal formulation, but no discrepancy detection.
- **GDA Agent:** A version of the agent implementing the complete GDA model.

We evaluated each version of the agent separately in order to select suitable values for GDA parameters and then compared the overall gameplay performance of each agent configuration.

In each experiment, we selected a different initial goal for the agent. We selected different initial goals in order to demonstrate improvements in the gameplay performance of the system using the GDA model. There are game situations in which using the GDA model does not improve the gameplay performance of the system, because the outcome of the game is determined before any of the GDA subtasks are invoked. This situation can arise in StarCraft, because the game enforces partial observability through the *fog-of-war* and the agent's initial goal may be dominated by the opponent's initial goal. In StarCraft, losing a game based solely on initial goal selection is referred to as a *build-order loss*. To prevent the agent's win rate from plateauing due to this aspect of RTS gameplay, we evaluated a variety of initial goals in our experiments. Because different initial goals are assigned to the agent, the win ratios reported in each experiment should not be directly compared to win ratios in other experiments. A direct comparison of the different agent configurations utilizing the same initial goal is presented in our fourth experiment.

In the first experiment, we evaluated the formulator agent with various planning window sizes. We also evaluated the base agent, represented by the formulator agent with a planning window of size 0. Results from the experiment are shown in Table 2. The agent performed best with a planning window of size 15. With smaller window sizes the agent often retrieved duplicate production actions resulting in wasted resources. With larger window sizes, goal formulation is performed less frequently.

The second experiment evaluated the predictor agent with various look-ahead window sizes and a fixed planning win-

Table 3: Results from the opponent prediction experiment show that the agent had the highest win ratio when integrating a look-ahead window of 10 actions.

Look-Ahead Window Size (m)	Win Ratio
0	0.71
5	0.75
10	0.79
15	0.73
20	0.69

Table 4: Win rates from the discrepancy detection experiment show that the agent performed best with a discrepancy period of 30 seconds.

Discrepancy Period (δ)	Win Ratio
∞	0.81
60	0.83
30	0.92
15	0.81

dow size of 15. We also evaluated the formulator agent, represented by the predictor agent with a look-ahead window of size 0. Results from the experiment are shown in Table 3. The agent performed best with a look-ahead window size of 10. Incorporating predictions helped the agent prepare for opponent strategies. With smaller window sizes, the agent was unable to anticipate opponent actions in time to develop counter strategies, while large window sizes resulted in the agent employing counter strategies too early, resulting in wasted resources.

In the third experiment, we evaluated the complete GDA agent with various discrepancy period sizes, and planning window and look-ahead windows of size 15 and 10. We also evaluated the predictor agent, represented by the GDA agent with a discrepancy period of ∞. Results from the experiment are shown in Table 4. The agent performed best with a discrepancy period of 30 seconds. Discrepancy detection enabled the agent to respond to critical game situations, such as formulating goals to build detector units in order to reveal cloaked units. When using larger discrepancy periods, the agent rarely detected discrepancies that trigger new goals, while smaller discrepancy periods resulted in the same problems as small planning window sizes.

The fourth experiment evaluated the different configurations of the agent using the same initial goal, providing a direct comparison of the different agent configurations. Results from the experiment are shown in Table 5. The results show that each additional GDA subtask integrated into the agent improved the overall system performance and the agent has the highest win ratio when utilizing the complete GDA model.

The agent had varying success against each of the opponent agents. Win rates versus individual opponents during the fourth experiment are shown in Table 6. Five of the opponents were dominated by our agent, and all ablations of the system achieved 100% win rates versus these opponents. Against these opponents, the GDA subtasks for anticipating opponent actions and triggering goal formulation were not necessary to win. Against three of the opponents, incorporating additional GDA subtasks improved the performance of our agent. The most significant improvement was demonstrated versus Aiur with discrepancy detection enabled. However, the results versus the Blizzard Protoss AI show that each of the GDA subtasks improved performance.

Table 5: The fourth experiment evaluated all of the agent configurations using the same initial goal. The agent performed best when using the complete GDA model.

Agent	Win Ratio
Base	0.73
Formulation	0.77
Prediction	0.81
GDA	0.92

Table 6: Our agent had varying success against individual opponents in the fourth experiment. The base version performed worst, while the complete GDA agent improved in performance versus three of the opponent agents.

	Base	Form.	Pred.	GDA
Blizzard Protoss	0.14	0.33	0.50	0.83
Blizzard Terran	1.00	1.00	1.00	1.00
Blizzard Zerg	1.00	1.00	1.00	1.00
Aiur	0.00	0.00	0.00	0.50
SPAR	1.00	1.00	1.00	1.00
Cromulent	1.00	1.00	1.00	1.00
Nova	0.67	0.83	1.00	1.00
BTHAI	1.00	1.00	1.00	1.00
Overall	0.73	0.77	0.81	0.92

Conclusions and Future Work

The GDA conceptual model provides a framework for building autonomous agents capable of operating in complex environments while responding to planning failures and exogenous events (Molineaux, Klenk, and Aha 2010). One of the main challenges in implementing the model is the substantial amount of domain engineering necessary to specify expectations for every action, explanations for every discrepancy, and goals for every explanation.

We presented an approach for learning GDA subtasks from demonstration, reducing the authorial burden of creating GDA agents. Our system learns expectations, explanations, and goals from expert examples. We use case-based goal formulation to predict changes in world state caused by adversaries and to select goals for the agent to pursue. The agent formulates new goals when the current active goal is accomplished or a discrepancy is detected.

We used the GDA model to implement strategy selection in an agent that plays complete games of StarCraft. Our ablation study showed that integrating additional GDA subtasks into the agent improved the win ratio of the agent, and the system performed best when implementing the complete GDA model. Additionally, the GDA model provided an interface for integrating case-based reasoning and reactive planning in a real-time agent.

In future work, we will evaluate the agent versus human opponents. One of the limitations of our ablation study was that it was restricted to a small sample of opponents. By incorporating humans into the evaluation process, we can test the agent versus a much larger player base. This analysis will enable evaluation of the agent while reducing the bias of individual opponents.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Number IIS-1018954. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

Aha, D. W.; Molineaux, M.; and Ponsen, M. 2005. Learning to Win: Case-Based Plan Selection in a Real-Time Strategy Game. In *Proceedings of ICCBR*, 5–20. Springer.

Buro, M. 2003. Real-Time Strategy Games: A New AI Research Challenge. In *Proceedings of IJCAI*, 1534–1535.

Cox, M. 2007. Perpetual Self-Aware Cognitive Agents. *AI Magazine* 28(1):32–45.

Epstein, S., and Shih, J. 1998. Sequential Instance-Based Learning. In Mercer, R., and Neufeld, E., eds., *Lecture Notes in Computer Science*, volume 1418. Springer. 442–454.

Hayes-Roth, B. 1985. A Blackboard Architecture for Control. *Artificial Intelligence* 26(3):251–321.

Jaidee, U.; Muñoz-Avila, H.; and Aha, D. W. 2011a. Case-Based Learning in Goal-Driven Autonomy Agents for Real-Time Strategy Combat Tasks. In *Proceedings of the ICCBR Workshop on Computer Games*, 43–52.

Jaidee, U.; Muñoz-Avila, H.; and Aha, D. W. 2011b. Integrated Learning for Goal-Driven Autonomy. In *Proceedings of IJCAI*, 2450–2455.

Laird, J., and VanLent, M. 2001. Human-Level AI's Killer Application: Interactive Computer Games. *AI magazine* 22(2):15–25.

Mateas, M., and Stern, A. 2002. A Behavior Language for Story-Based Believable Agents. *IEEE Intelligent Systems* 17(4):39–47.

McCoy, J., and Mateas, M. 2008. An Integrated Agent for Playing Real-Time Strategy Games. In *Proceedings of AAAI*, 1313–1318. AAAI Press.

Molineaux, M.; Klenk, M.; and Aha, D. W. 2010. Goal-Driven Autonomy in a Navy Strategy Simulation. In *Proceedings of AAAI*, 1548–1553. AAAI Press.

Muñoz-Avila, H., and Aha, D. W. 2010. A Case Study of Goal-Driven Autonomy in Domination Games. In *Proceedings of the AAAI Workshop on Goal-Directed Autonomy*. AAAI Press.

Muñoz-Avila, H.; Aha, D. W.; Jaidee, U.; and Carter, E. 2010a. Goal-Driven Autonomy with Case-Based Reasoning. In *Proceedings of ICCBR*, 228–241.

Muñoz-Avila, H.; Aha, D. W.; Jaidee, U.; Klenk, M.; and Molineaux, M. 2010b. Applying Goal Driven Autonomy to a Team Shooter Game. In *Proceedings of FLAIRS*, 465–470. AAAI Press.

Nau, D. S. 2007. Current Trends in Automated Planning. *AI magazine* 28(4):43.

Rao, A., and Georgeff, M. 1995. BDI Agents: From Theory to Practice. In *Proceedings of ICMAS*, 312–319.

Weber, B. G., and Mateas, M. 2009. A Data Mining Approach to Strategy Prediction. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 140–147. IEEE Press.

Weber, B. G.; Mateas, M.; and Jhala, A. 2010a. Applying Goal-Driven Autonomy to StarCraft. In *Proceedings of AIIDE*, 101–106. AAAI Press.

Weber, B. G.; Mateas, M.; and Jhala, A. 2010b. Case-Based Goal Formulation. In *Proceedings of the AAAI Workshop on Goal-Directed Autonomy*. AAAI Press.

Weber, B. G.; Mateas, M.; and Jhala, A. 2011. Building Human-Level AI for Real-Time Strategy Games. In *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems*, 329–336. AAAI Press.