Project Number: SCH-1102

LEARNING FROM DEMONSTRATION IN A GAME ENVIRONMENT

A Major Qualifying Project Report
submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by

Zachary Card
Michael Miranda

Date: April 26, 2012

Approved:
Professor Sonia Chernova, Major Advisor
Professor David Finkel, Co-Advisor

Table of contents:

Abstract

Darmok 2 is a Case-Based Reasoning AI designed to learn from demonstration. Using the Brood War API and the Java Native Interface, we adapted Darmok 2 to work with the Real-Time Strategy game StarCraft: Brood War. Adapting Darmok 2 to a robust competitive strategy game allowed us to demonstrate its learning capabilities, assess its performance and limitations, and suggest improvements to its architecture.

I. Introduction

Learning from demonstration is the process of a student learning by watching a task be performed. It is a common method used by teachers to help their students understand a task. In artificial intelligence and robotics it is a method used to train an artificial agent in a task without directly coding instructions for that task. This can be useful both for teaching agents tasks quickly, as well as for agents that will be required to learn a variety of tasks specific to their area of deployment. Examples of this include things as varied as a social robot that will be marketed across multiple cultures, to artificial intelligence in games. As a popular area of research, it is important to assess the usefulness of this method in an environment not intended for research if in the future it is to be used for regular interaction with artificial agents.

Darmok 2 (D2) is an open source system developed by Georgia Tech to assemble real-time case based reasoning artificial intelligence modules from xml recordings of games. D2 converts an XML representation of a game, called a trace, into a knowledge base that the system uses to select actions in a chosen environment (Ontanon et al.). Building plans from these traces is a method of learning from demonstration. At the start of this project, the only games that D2 had been tested on were created specifically for use with D2, although they were designed based on popular games. This limited the conclusions that could be derived from tests in this environment.

D2 is an expansion of a prior project called Darmok that formed the basis of D2's planning structure. D2 added automatic goal inferences while observing traces and simulation of plans for performance assessment into the older system, which required the player to annotate each trace of gameplay. The D2 system addresses the difficulty of case based reasoning in a real time environment. Case based reasoning is difficult in such an environment because in the time that passes while developing a plan, the plan may become invalid. D2 is able to adapt its plans in real time to account for

portions of its plan becoming invalid using an online case based planning cycle, a standard case based reasoning cycle with an execution step that checks the world state before executing a portion of a plan.

StarCraft: Brood War is a popular real-time strategy game published in 1998 that has become a well-known competitive measure of real-time strategy skills of both human players (Cohen) and artificial intelligence (Expressive Intelligence Studio). In spite of the release of a newer iteration of the game, StarCraft II, StarCraft: Brood War remains one of the best supported environments for artificial intelligence research, with A.I. hook-ins for both external and internal A.I. Control, and a wide variety of existing A.I.s of varying tiers of capability.

We have adapted D2 to generate A.I.'s for StarCraft Brood War to take advantage of D2's ability learn from any XML documented environment. In addition, we have evaluated its capabilities with respect to accuracy of style emulation when learning from both human and non-human players, appropriateness of plan choice and execution, and performance against human players and other A.I.s. D2 was selected for this project because it is an open source system designed to be used in a wide variety of environments. By adapting D2 to a more competitive, publicly known and expansive environment we will better be able to measure its capabilities, assess weaknesses, and understand ways in which it may be improved.

In Chapter 2, we outline the basics of case-based reasoning, as well as background info about the Darmok engines and StarCraft: Brood War. In Chapter 3 we describe how we went about the process of adapting Darmok 2 to StarCraft. In Chapter 4, we discuss the results of testing the Darmok 2 engine with the new environment. In Chapter 5, we describe ways that the engine could be used or expanded beyond the scope of this project. In Chapter 6, we discuss and explain our conclusions from this project.

II. Background

1. Case-Based Reasoning and Case-Based Planning

Case-Based Reasoning (CBR) is a method of problem solving in which stored solutions to remembered problems are retrieved and reused for current problems(Bartsch-Sporl). Unlike generative problem solving algorithms which attempt to systematically solve a problem by exploring solution space starting with a clean slate, CBR algorithms start from a concrete solution to a similar problem and search from there. CBR has attracted interest due to its similarity to human problem solving using likeness between two situations, and by extension, human capability to develop a formal framework in domains which lack such a framework (L. Spalazzi).

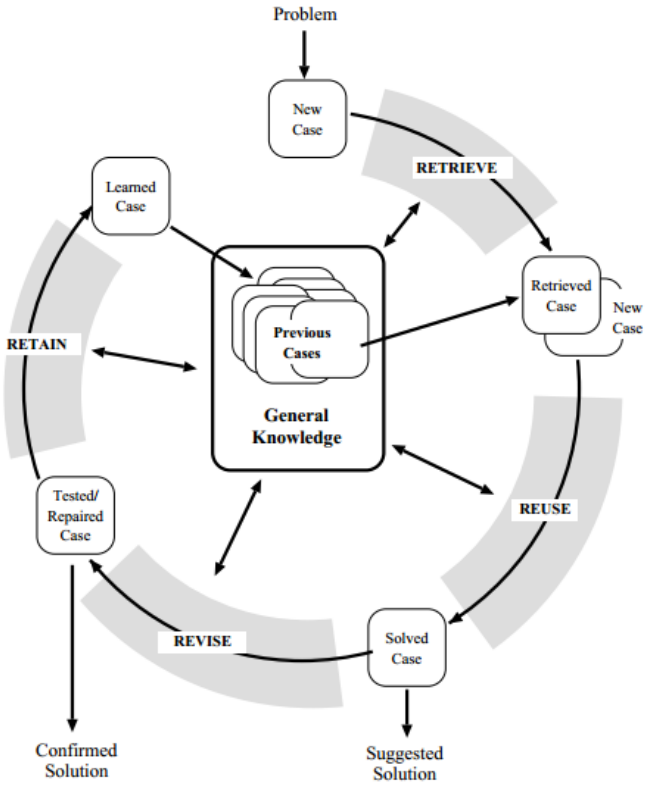

Image from Bartsch-Sporl et al, *Case-Based Reasoning - Survey and Future Directions*

Case-Based Planning is an application of CBR in which each case stores a sequence of actions or plan for an domain with an existing framework for actions, preconditions, and postconditions. True case-based planning records both failed and successful plans. Since learning can be viewed as a problem with actions, preconditions, and postconditions, this makes CBP useful for learning applications. Results conflict regarding the efficiency of case-based planners vs. generative planners (L. Spalazzi).

2. Darmok and Darmok II

Darmok was a case-based planner that worked with a java version of Warcraft II called “Wargus” (Ontañon et al). Darmok learned from recordings of Wargus being played. Darmok's recordings, called “traces”, consisted of actions, timestamps, and the game state when that action was performed. The entity generating the trace would then need to manually indicate what the goals of those actions were. While assessed as no better at winning a match than the game's built in AI, AIs generated by Darmok performed very differently showing good long-term strategy, but poor short term. The built in AI did the reverse.

Darmok II was an improvement over Darmok in that it similarly learns from traces, but can infer goals, making generation less time consuming and also allowing it to generate traces from entities that are incapable of annotating a trace, such as another AI (Ontañon et al). D2 is a general system that is designed to be adapted to a wide variety of tasks. In order for D2 to be able to generate an AI for a given environment, the actions and objects that exist in the game must be described in an XML document (referred to as a “domain”), and the static environment must consist of a cell-based grid, also documented in XML. Tests which operate on game data (called “Sensors”) must be made to analyze the world-state during both AI generation, and at run-time. The set of goals D2 can use for inference

must be created as well, where each goal is a test which is true in some desired state. One of these goals must be the victory condition for the game, and D2 must be able to generate a trace of the environment from the start of an episode until the end-goal is achieved.

When running an AI built from traces, D1 and D2 both use case-based planning to execute a plan. A plan is first selected, from the set created during training, based on which plan's original environment is most similar to the current situation. The AI then takes the series of actions in the plan, for each action adapting the action's original parameters (what unit to use, where to perform the action, etc.) to the current game environment. If the start conditions for an action is met the action is executed. The AI waits until either the success or failure conditions for the action have been met, at which point it evaluates the game state and decides on its next action.

3. Starcraft: Brood War and the Brood War API

We chose to use StarCraft Brood War for expanding Darmok 2 because of its reputation as a robust, competitive real-time strategy environment. In each game the player starts with 5 noncombat units for collecting resources and building structures. Once enough resources have been collected, the player may construct more workers, and also allocate workers to begin building other structures. Other structures can build combat units, research combat utilities or advantages, or participate in combat themselves. Which buildings are constructed, the order they are constructed in, and the timing can all vary greatly, though the early portions of the game are often very similar due to the standard starting state. Adding to the desirable traits of the environment is diversity of viable strategies. When properly executed, both short-term rush tactics and long-term plans can win. Lastly, adaptation of strategies can also play a major role. While early game it is important to have a strong force, late game it becomes important to be able to rapidly change army composition and strategy to counter or outmaneuver an

opponent.

The main strategy we focused on for this project centered around the Zerg race which has a very strong early game setup called the “Zerg Rush”. The Zerg race is able to produce a lot of weak units called “zerglings” cheaply making for a fast attack. Properly executed, the zerglings will arrive and destroy any buildings before the opponent has adequate defenses. There are viable defenses against this strategy, but at low skill levels, it is both easy to execute and hard to defend against.

As an externally developed program, we needed a method to retrieve the game state from StarCraft: Brood War. To do this we used Brood War API (BWAPI), a C++ Library of hook-ins to StarCraft. The Brood War API is a 3rd party module which has been developed for allowing an external AI to interact with StarCraft: Brood War(Bwapi - An API). The API creates a number of functions which make developing an external AI for Brood War possible. The API exposes these functions to C++ code, but D2 was constructed in Java. To connect D2 to BWAPI we used an existing Java interface for the Brood War API, called JNIBWAPI, which uses the Java Native Interface to expose the majority of BWAPI functions to Java code (JNI Interface for BWAPI). We used both of these to allow our project to communicate and interact with StarCraft: Brood War.

III. Methodology

1. Engine Overview

The D2 Engine itself has three major preexisting parts: a domain generator which creates Java code based on a given XML domain, an AI trainer which creates a plan based on given traces, and an AI executor which runs the plan during the game by looking at the current game-state and outputting actions which the AI player should take.

The D2 engine must be added to in four major areas to be adapted to a new domain: Java code which converts the game state into D2's data format, Java code which converts D2's output when the AI is running into in-game commands, the tracer client which retrieves the game state from the game and records it in an XML file, and the AI client which takes the compiled traces and actually runs planning algorithms on them. Because the tracer client already retrieves the game state, we used the tracer code to provide game state to the AI client.

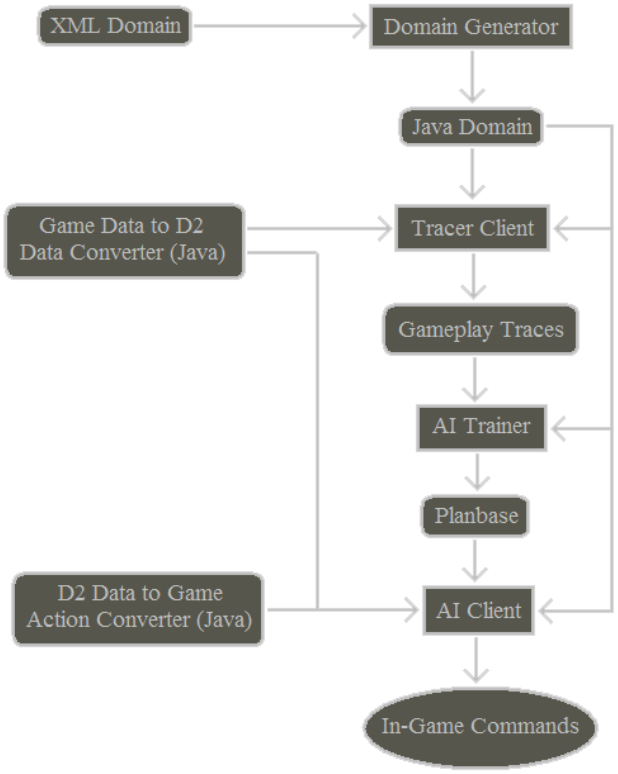

Figure 1. Flowchart for creating and running an AI using Darmok 2

A number of steps are required to build and run an AI using Darmok 2. First, we needed to write the XML domain, which describes all of the entities, actions, sensors and goals the AI will need to know about. This is passed to the Domain Generator, which generates Java code that represents all the data in the domain and loads it into Java objects when the program starts. The Java Domain code is used by all the other parts of the Darmok 2 system to understand the environment it is operating with.

After generating the Java Domain, we wrote the Java required to convert the StarCraft game state into the data format described in the Java Domain, and implemented our tracer client. Using the tracer client, we generated a number of XML traces by having a simple Java AI run our training strategy. These traces were passed to the AI Trainer, which created a set of plans based on the traces, and recorded them in an XML document, called a planbase.

We next wrote the Java code to take D2's output, interpret it, and send the desired actions to in-game units through the Brood War API. Using this code, we wrote our AI client, which loaded the previously generated XML planbase into a D2 AI. The AI client was able to play through a game by continuously feeding the AI the current StarCraft game state, and taking the actions indicated by the AI.

2. XML Domain

The XML domain contains static representations of all the pieces of Brood War that Darmok 2 can make use of, and also contains the sensors that are built into each AI plan base by D2. Darmok 2 has four main types of XML constructs. Actions, Entities, Sensors, and Goals. Creating the XML domain was one of the most time consuming aspects of this project, and it went through numerous revisions as our understanding of the Darmok system evolved.

2.1. Actions

The first section of the XML domain documents Actions (Fig. 2). Actions are tasks that entities in-game can perform. While the actions are assigned by the player or AI, in StarCraft the actions are actually performed by the units. Each action contains a set of requirements that must be met to perform that action, and takes the parameters needed to give the command associated with the action through BWAPI.

```
<Action name="AttackUnit">
  <Parameter name="preFailureTime" type="INTEGER"/>
  <Parameter name="failureTime" type="INTEGER"/>
  <Parameter name="target" type="ENTITY_ID"/>
  <ValidCondition>
    <![CDATA[
    HasPath(Entity(,playerID,entityID),CoordinateAttribute(target,"coords"))
    ]]>
  </ValidCondition>
  <OnPreFailureCondition>
    preFailureTime = 800 + cycle
  </OnPreFailureCondition>
  <PreFailureCondition>
    Timer(preFailureTime)
  </PreFailureCondition>
  <PreCondition>
    EntityExists(Entity(,playerID,entityID))
  </PreCondition>
  <OnFailureCondition>
    failureTime = 800 + cycle
  </OnFailureCondition>
  <FailureCondition>
    UnitKilled(Entity(,player,entityID)) || Timer(failureTime)
  </FailureCondition>
  <SuccessCondition>
    UnitKilled(target)
  </SuccessCondition>
</Action>
```

Figure 2. Example XML for the "Attack Unit" action

Each action has a number of conditions that the AI uses to determine information about the state of the action. The various conditions consist of logical tests or simple operations, which can make use of sensors defined in the domain (or the basic sensors defined in the engine). An action's ValidCondition is used to check whether the given parameters are valid for the entity trying to perform the action when D2 is adapting parameters. The action's PreCondition is checked after parameter adaptation to determine whether the action can currently be performed. If the AI is waiting for PreConditions to be met, the PreFailureCondition is checked to determine if the AI should stop trying to perform the action. The OnPreFailureCondition segment is code that is run the first time the PreFailureCondition is checked (in this case it sets the duration of a timer which tells the AI to stop trying to do the action). Once the action has been started, the AI repeatedly checks the

SuccessCondition and FailureCondition to determine if the action has succeeded or failed. Much like the OnPreFailureCondition, the OnFailureCondition code segment is run the first time the FailureCondition is checked.

Writing the XML for actions was one of the first things we did when creating the domain, and was difficult in part because some in game commands were different depending on the parameters given, requiring multiple D2 actions to represent fully. Other times, multiple commands could be represented by a single D2 action, such as the commands to cancel different types of actions. Because there was not always a one-to-one correspondence of BWAPI commands to D2 actions, we had to be careful to design our domain code to adequately cover the different cases that could arise. In addition, we had to make changes to the different conditions for some actions later in the project, when we achieved a better awareness of how they are used by D2 during execution.

2.2. Entities

Another section of the domain describes the entities Darmok needs to interact with during game play (Fig. 3). Entities are divided into physical entities and nonphysical entities, with physical entities representing in-game units, and nonphysical entities representing players. Each entity has a set of features, which contains information about a unit or type of unit, and a list of actions that they are able to perform.

```
<Entity>
    <Name>Terran_Refinery</Name>
    <Super>Terran_Building</Super>
    <Features>
        <Feature>
            <Name>max_hitpoints</Name>
            <DefaultValue>750</DefaultValue>
        </Feature>
        <Feature>
            <Name>width</Name>
            <DefaultValue>1</DefaultValue>
        </Feature>
        <Feature>
            <Name>length</Name>
            <DefaultValue>1</DefaultValue>
        </Feature>
        <Feature>
            <Name>cost_minerals</Name>
            <DefaultValue>100</DefaultValue>
        </Feature>
        <Feature>
            <Name>cost_gas</Name>
            <DefaultValue>0</DefaultValue>
        </Feature>
        <Feature>
            <Name>build_time</Name>
            <DefaultValue>600</DefaultValue>
        </Feature>
    </Features>
    <Actions>
    </Actions>
</Entity>
```

Figure 3. Example XML for the “Terran Refinery” Entity

Units are representations of in-game units. We created further subcategories to represent special properties of ground, air, and building units of each race. Regardless of subcategory, all unit entities have a unit size, hit points, build time, gas cost, mineral cost, and supply cost. When recording data from traces and during gameplay, the entity data is used to determine walkablility, current hit points, and other features. Most important to D2 are the build time and resource costs. These costs form preconditions for building the units when an AI is being run.

The process of writing the XML for entities was one of larger initial time investments we had to make. StarCraft is well documented, so once we decided on a format for the different entity types the

data to fill the entities in was easily available. However, StarCraft is a very large and complex game, featuring over 100 types of unit encountered in general play. As a result, it took a significant amount of time just to go through and translate each unit type into the appropriate XML.

2.3. Sensors

Sensors are functions that operate on data when D2 is analyzing game state (Fig. 4). They can use basic logical operations, or other sensors defined either in the domain or the D2 engine itself, and can return any basic data type usable in D2's logic. Sensors directly used in conditions usually return a Boolean value, based on some data in the game state or the results of calling some other sensor.

```
<Sensor name="SupplySensor" type="INTEGER">
  <Code>
    IntAttribute(Entity(Type("eisbot.proxy.starmok.entities.Player"), player), "supply")
  </Code>
</Sensor>
```

Figure 4. Example XML for the "Supply" Sensor

D2 uses the sensors we built to check the game state during planning and adaptation to determine if pre- and post- conditions for an action are met. This can be done to check if an action or goal is able to be completed from the current state or if an action or goal has been completed.

We decided on the sensors that would be required for StarCraft's domain while writing the action XML, and wrote dummy code for different sensors based on what information we would need to test different action conditions. Writing the sensor code also proved to be somewhat of a challenge, as D2 uses an engine specific syntax for performing operations or comparisons with sensors, which has some quirks although it does mostly follow general programming syntax convention. In addition to D2's syntax, it is also possible to have a sensor directly insert Java code as its evaluation function

when generating the Java Domain, which makes it possible to write more complex sensors than would otherwise be feasible. Of significant note is a discovery we made later in the project when training the AI, which is that sensors may need to be evaluated when training an AI, and can therefore be based only on the data available to D2 through the trace provided. Prior to finding this out, we had some sensors directly use Java code to easily access information available through BWAPI, which caused problems when initial training attempts started. This meant that we had to spend a non-trivial amount of time rewriting the sensors to only use trace data.

2.4. Goals

The last part of the XML domain consists of goals (Fig. 5). Most goals function like sensors, checking portions of the game-state to determine if the goal has been completed. D2 uses goals as a part of its planning structure, to infer what the player intended when performing various actions based on when different goals were met. The only goal that is actually required for D2 to function is the win-goal, which indicates to D2 when a desired end state of the game has been reached.

```
<Goal name="Have_Terran_Marine">
  <Code>
    HaveUnit(Type("eisbot.proxy.starmok.entities.Terran_Marine"))
  </Code>
</Goal>
```

Figure 5. Example XML for the "Have Terran Marine" Goal

For our purposes, we wrote a set of goals which were based on having different types of units. The set of goals was relatively simple, but allowed D2 to infer enough information to determine what was required to produce different types of units and develop the requisite base infrastructure.

3. Tracer Client

The tracer client is an executable program which runs in the background while another source, either a human player or another AI, sends input to StarCraft. The purpose of the tracer client is to record both the game state and actions the player takes throughout the game, then save that information so that it can later be used by Darmok to train an AI. Accomplishing this required us to connect D2 with the Brood War API and JNIBWAPI.

Darmok contains built in trace output code which can save a record of what happens during a game as an XML trace. However, this code requires that the data be in a format usable by Darmok, the Java code for which is created during domain generation. Because of this, we first needed to write code that could retrieve the game state using BWAPI, and store it in the data structures of the Java Domain. Once the data was converted into a format usable by Darmok, we were able to output it as an XML trace by using D2's built in tracer code. To make the process easier, we created a wrapper class which encapsulated the process of converting the game state and sending to the tracer when required. At that point, we were able to write AI client using BWAPI, which allowed for external input and did nothing but make the appropriate calls in order to record gameplay traces.

When the tracer client starts, it connects to StarCraft via BWAPI, and then starts D2's trace output code. Once the game begins, our converter code retrieves the info Darmok initially needs to start a trace. It builds a representation of the map by encoding each map tile as a different character based on which type of map tile it is. Stored with each map are the names of the players and the name of the map. Once the trace has been started, the client uses a function which is called by BWAPI each time the game updates to record game information as necessary. At set intervals and whenever a player takes an action, the game state and any actions taken are converted and recorded into the XML trace. To reduce the size of the trace, Darmok only actually records changes in game state. This is determined using a difference function between the previously recorded game state and the state about to be

recorded. Once the game is over, the winner of the game is recorded, and the trace is complete.

The most difficult and time consuming part of building the tracer client was creating the code to store the game state in D2's data format, which required understanding the way Darmok represents game data. We had to reconcile numerous differences between how D2 and the Brood War API store information, such as naming convention of unit types and the fact that StarCraft locations are pixel based, whereas D2 uses a tile based coordinate system. Another significant obstacle was learning how to use D2's built in tracer code, which was poorly documented. Once the back end code was complete, we programmed the client using BWAPI hooks to call the appropriate functions. The client did require tuning after completion, as we found that recording the game state too often could slow down the game.

4. AI Client

In addition to the tracer client, we also created an AI client which D2 uses to play the game. D2 needs access to the game state to make decisions at run time, so we based the AI client off of our tracer client, which already had the functionality for converting the game state and storing it in D2's data format. We added functionality to our tracer wrapper code to allow us to retrieve the game state once it had been converted. However, our client had to be able to use the output from D2 as well as provide input. D2 gives output as a list of actions that should be executed, in its own data format. To handle this, we added conversion code, which could take the action data and send a command through BWAPI based on each action object. The code looks at the action objects, and depending on the action's type extracts the parameters necessary for the appropriate command from the action's data. The code then sends the command that corresponds to the action via BWAPI, using the parameters retrieved. From there, we modified our client to make use of D2's execution code.

When the AI client starts, it has D2 load a planbase (generated based on previous traces) from a

file. At each update, the AI client retrieves the game state with the same functionality used by the tracer client, and then pulls the game state from the tracer and sends it as input to D2. The client then receives a list of any actions D2 wishes to perform, and uses the conversion code to send the actions to StarCraft via Brood War API. For testing purposes, we had D2 pause the game when it was processing, allowing us to see how long it took, and letting us compare its performance to that of StarCraft's built in AI without time constraints. We also tried running the AI in multi-threaded mode to allow the game to continue while it worked, but did not pursue this very far because of D2's performance issues (discussed in section IV).

Overall, the work done on the AI client consisted largely of expanding the functionality of the tracer client and conversion code. A significant amount of time was also spent developing an understanding of how D2's execution code functioned, as the code often required extensive analysis in order to be used properly. Over the course of developing and debugging the AI client, we also made modifications to previous work on the XML domain and converter code as our understanding of D2's operation improved.

5. Training and Testing D2

In order to train D2, we incorporated our tracer client into an existing AI module, which used the "zerg rush" strategy on the map "Astral Balance"(Fig. 6), and recorded each match using the tracer module. We collected several trace batches. One batch ran using a single starting location on the map, and another ran using several starting locations. After running the example module a number of times, we compiled the traces into a planbase using the D2 Trainer module.

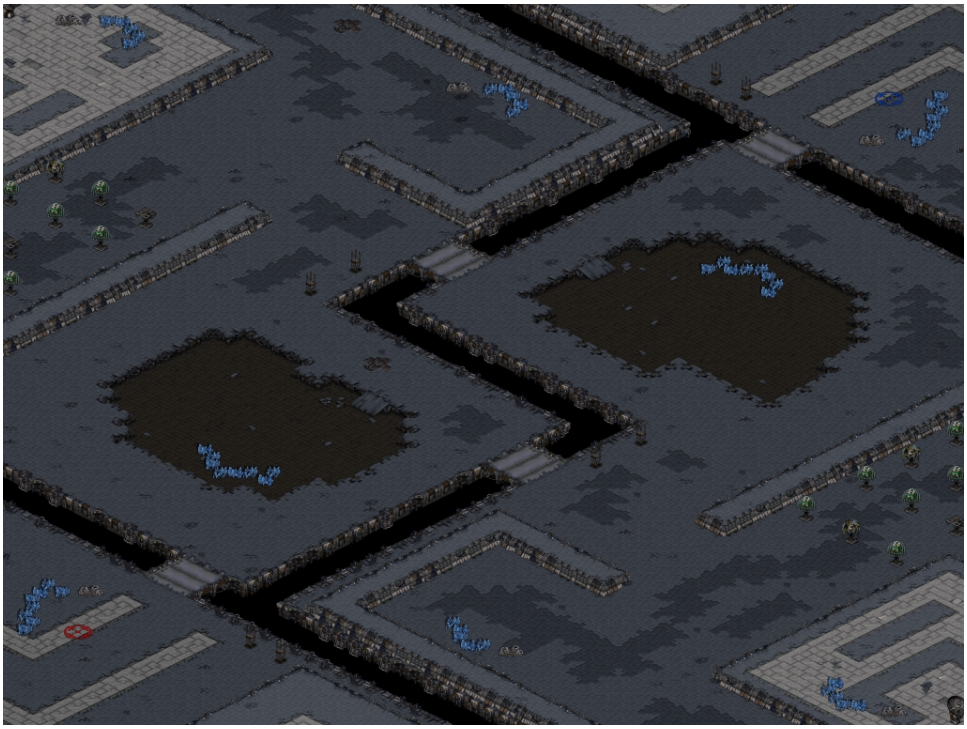

Figure 6. Astral Balance Game Map. Starting locations in upper right and lower left

Observing the D2 client play, there were significant pauses in activity, which we measured using time measurements taken manually while observing gameplay. During execution, we looked specifically for the order of actions issued to units, where the units executed the actions, and which units executed actions to determine accuracy of the plan D2 built.

IV. Results

Our tests on the completed system showed that it will successfully emulate the trace examples given when training the AI. In early tests this was made quite apparent when our traces were made only at one base on the map, and the resulting AI would almost always try to use that base, even when spawning at the other end of the map. Further tests that had examples from both map locations would usually mine from local mine locations, build when the appropriate resources were gathered, train the same units as in the trace case, and attack with them as demonstrated. However, D2's action adaptation is not perfect, and the AI would occasionally give commands that were not appropriate for the current context, even when provided with a variety of training examples. Overall, we found that D2 was generally capable of learning a course of action from the examples given, and carrying it out the plans it developed.

One major failing of the D2 engine that we have found is in its parameter adaptation for actions. The D2 engine can look at the parameters for an action (what unit did the action, where it was done, etc.), and tries to find the closest valid parameters in the current game. However, when adapting coordinates, the engine delays 10 to 105 seconds if the coordinate system is at the resolution of walkability data, which is needed to accurately assess where units can travel and requires 4 times as many map locations as data for where buildings can be built. The slowdown is likely because even the smallest StarCraft maps have dimensions of 512 by 384 when using walk coordinates, in contrast the low-resolution (in terms of map segments, not graphically) maps used by Darmok's test games, which had a maximum size of 128 by 128. Fixing this would likely require implementing domain-specific parameter adaptation, which is a non-trivial task, although the engine does allow for domain specific extension of its adaptation code. The time D2 requires when adapting actions made it ultimately unable to compete against the built in AI when time was a factor. When processing time was

unrestricted, it performed competitively against the built in AI and was able to sometimes win, depending on the strategy taken by its opponent. Whether or not the AI was able to defeat its opponent seemed to depend largely on whether the opponent was able to train combat units before the initial attack reached their base. If the opponent was able to create a meaningful defense, the initial attack would often fail and the D2 AI would eventually lose. It should be noted that because D2 tends to plan actions in series rather than in parallel, it was usually slower to build up and launch its attack than the AI we trained it with, leading to a lower probability of winning.

D2's focus on overall plan goals and order of action execution also caused issues with unit commands. While a command would always be issued to a unit if a valid unit to execute the command existed, it did not adequately distinguish between units that were already performing a task, such as gathering resources, and units that were not performing a task. While D2 can track which units are currently busy, the planning and execution mechanisms currently used by D2 don't seem to make much use of this, often preferring to have a sequence of actions that are performed to completion rather than performing multiple actions in parallel, some of which (such as resource gathering) may go on indefinitely. D2 also demonstrated a poor ability to micromanage its units, having problems adapting very specific combat actions to a new situation.

V. Future Work

The biggest initial challenge in this project was adapting the different components to work together. Each component (BWAPI, JNIBWAPI, and D2) had its own set of documentation, and there were numerous differences in naming convention and functionality between the different components. For example, BWAPI measures time differently than D2, requiring us to track game time in our Java code. Recording the traces too frequently also creates issues as rapid traces can slow down the game significantly due to the costs of converting the entire game state to D2's data format and recoding it to XML files.

From our experience running the AI, we see a few ways in which it can be improved. When searching through the map for build or movement coordinates, such as when ordering a unit to a specific location, or finding valid locations to build, the map is searched square by square. These squares come in three resolutions: individual pixels, walk tiles(8x8 pixels), and build tiles(32X32 pixels). Smaller resolutions are needed for finer unit control, which is key for more advanced strategies. D2 uses a search algorithm that checks all of the cells in the map to determine walkability before determining a destination. If the smaller resolutions are used, the game must pause for a rather large amount of time in order to execute essential and common commands. To improve this, we recommend using a localized, search starting from the unit the command is being issued to and extending outward. We attempted to reduce the area checked during coordinate adaptation, however we found that this method was often unable to find a result suitable for the current game environment. Incorporating a less expensive means of adapting coordinates without an unacceptable drop in the quality of the results would require significant effort, and was not among the tasks we could accomplish due to the time constraints of this project.

While the delays that occurred when we ran D2 were large enough that human error was not a

factor when measuring processing time, for improvements in measurement accuracy we recommend the use of timers embedded in D2. Embedding timers in the various planning and adaptation functions used by D2 could allow a better analysis of exactly which operations are most problematic, and could be useful in later efforts to optimize D2.

Due to the methods D2 uses to issue commands, checking which units are busy and making effective use of the information would require changes to D2's execution structure. Since commands are issued based on similarity to a plan, D2 cannot effectively distinguish between owned units with the same properties with efficient processing time. While D2 can track which units are currently performing a task, we found that because D2 prefers to perform actions in series, it was not possible to have D2 permanently set aside a unit assigned to some task of indefinite duration while still having it continue its plan. One possible solution for solving both problems with unit allocation and micromanagement would be to have another, lower level, domain specific AI. This AI would be used to specify which units should receive commands, and handle the details of situational combat actions.

VI. Conclusions

Learning from demonstration is an important problem to solve both for making AI and robots that can be taught without direct knowledge of coding systems and design, and also for understanding how humans learn from demonstration. To assess the current performance of research oriented learning from demonstration programs in games we used the Georgia Tech Darmok 2 and selected an independent test environment, StarCraft: Brood War. Adapting D2 to StarCraft required constructing a representation of the StarCraft environment in XML, and using two APIs, BWAPI and JNIBWAPI, to provide data which D2 could use to assess the current game state, and issue commands to the game. We constructed a Tracer Client to record demonstration games from StarCraft, and an AI Client to run D2's compiled AIs in StarCraft. All of these pieces working together allowed us to run demonstrations of a simple strategy, compile the traces of those demonstrations into an AI, and run the AI in an actual game of StarCraft against a computer opponent.

We determined from these results that D2 is not competitive in StarCraft: Brood War given its current state. The delays from D2 examining the game state at run-time are too large for it to be effective in the chosen environment. However, with optimization of D2's coordinate and entity adaptation mechanisms, we believe these delays can be significantly reduced. Finally, the actions taken in the plans executed by D2 reflected the proper order and general positioning necessary for successful game play. This shows that transition to an independent environment did not compromise the viability of the learning process, only the execution. As better methods of plan storage and access are implemented, this method of learning from demonstration will become more viable outside of research domains.

References

1. Cohen, Patricia. "Princeton Goes International for StarCraft E-Competition - NYTimes.com." *The New York Times - Breaking News, World News & Multimedia*. 15 Oct. 2011. Web. 15 Oct. 2011. <http://www.nytimes.com/2009/04/12/sports/othersports/12star.html?_r=1>.
2. Ontanon Et Al. *On-Line Case-Based Planning*. Georgia Institute of Technology. Web. 3 May 2011.
3. Ontanon Et Al. *Learning from Human Demonstrations for Real-Time Case-Based Planning*. Georgia Institute of Technology, 2009. Web. 3 May 2011.
4. "StarCraft AI Competition | Expressive Intelligence Studio." *Expressive Intelligence Studio | Games and Playable Media @ UCSC*. Web. 15 Oct. 2011. <http://eis.ucsc.edu/StarCraftAICompetition>.
5. L. Spalazzi *A Survey on Case-Based Planning*. University of Ancona. Web.18 October 2011.
6. *Bwapi - An API for Interacting with Starcraft: Broodwar (1.16.1)*. Web. 14 Oct. 2011. <http://code.google.com/p/bwapi/>
7. "JNI-BWAPI." *Jnibwapi - JNI Interface for BWAPI*. Web. 14 Oct. 2011. <http://code.google.com/p/jnibwapi/>.
8. Bartsch-Sporl, Brigitte, Mario Lenz, and Andre Hubner. *Case-Based Reasoning - Survey and Future Directions (1999)*. 5th German Biennial Conference on Knowledge-Based Systems, 1999.