12-2012

Generalizing Agent Plans and Behaviors with Automated Staged Observation in The Real-Time Strategy Game Starcraft

Zackary A. Gill
University of Texas-Pan American

GENERALIZING AGENT PLANS AND BEHAVIORS WITH AUTOMATED STAGED OBSERVATION IN THE REAL-TIME STRAGEGY GAME STARCRAFT

A Thesis

by

ZACKARY A. GILL

Submitted to the Graduate School of the
University of Texas-Pan American
In partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2012

Major Subject: Computer Science

GENERALIZING AGENT PLANS AND BEHAVIORS WITH AUTOMATED STAGED OBSERVATION IN THE REAL-TIME STRAGEGY GAME STARCRAFT

A Thesis
by
ZACKARY A. GILL

COMMITTEE MEMBERS

Dr. Emmett Tomai
Chair of Committee

Dr. Richard Fowler
Committee Member

Dr. Robert Schweller
Committee Member

December 2012

ABSTRACT

Gill, Zackary, Generalizing Agent Plans and Behaviors with Automated Staged Observation in the real-time strategy game StarCraft. Master of Science (MS), December, 2012, 48 pp., 7 tables, 13 figures, references, 21 titles.

In this thesis we investigate the processes involved in learning to play a game. It was inspired by two observations about how human players learn to play. First, learning the domain is intertwined with goal pursuit. Second, games are designed to ramp up in complexity, walking players through a gradual cycle of acquiring, refining, and generalizing knowledge about the domain. This approach does not rely on traces of expert play. We created an integrated planning, learning and execution system that uses StarCraft as its domain. The planning module creates command/event groupings based on the data received. Observations of unit behavior are collected during execution and returned to the learning module which tests the generalization hypothesizes. The planner uses those test results to generate events that will pursue the goal and facilitate learning the domain. We demonstrate that this approach can efficiently learn the subtle traits of commands through multiple scenarios.

DEDICATION

The competition of this thesis and my masters degree in total would not have been possible without the morale support from my family. Without them I would have never been able to finish this work. My father, Jonathan Gill, my mother, Kelly Gill, and my brother, Joshua Gill, kept me going through all of the stress and pressure. Thank you so very much.

TABLE OF CONTENTS

Page

ABSTRACT…… iii

DEDICATION…….……... iv

TABLE OF CONTENTS…… v

LIST OF TABLES…….. vii

LIST OF FIGURES..…….. viii

CHAPTER I. INTRODUCTION…… 1

Problem Statement..…… 2

Related Work..…… 4

CHAPTER II. SYSTEM DESCRIPTION……..…… 15

Technologies Used…….……. 16

Terms…… 17

Functionality.…….…….. 18

Algorithm Description……..…….…….. 19

CHAPTER III. EXPERIMENT..……..... 28

Hypotheses…….…… 28

Experiment Setup…… 29

Result……..…… 32

Conclusion……..…… 39

Future Work…..……... 40

REFERENCES……... 44

APPENDIX A……... 46

BIOGRAPHICAL SKETCH……... 48

LIST OF TABLES

Page

Table 1: Trial 1…… 33

Table 2: Trial 2……..……... 34

Table 3: Trial 3…… 35

Table 4: Trial 4…… 36

Table 5: Trial 5…… 36

Table 6: Test 1……..…… 37

Table 7: Test 3…….. 38

LIST OF FIGURES

Page

Figure 1: Flowchart of Prodigy's System…… 12

Figure 2: All parts of the system – Learner, AI Agent, Chaoslauncher, and StarCraft….. 16

Figure 3: Outline of the system created in this thesis…… 18

Figure 4: System Flowchart…….. 19

Figure 5: Sample goal given to Learner…….…... 19

Figure 6: Scenario 1…..…….…... 30

Figure 7: Scenario 2…….…….…... 30

Figure 8: Scenario 3……..…….…... 31

Figure 9: Scenario 4……..……..... 31

Figure 10: Scenario 5…….……... 31

Figure 11: Scenario 6…….…... 32

Figure 12: Move's results from Trial 2……..……... 34

Figure 13: Visual representation of what Distance means……..……... 47

CHAPTER I

INTRODUCTION

Games play a prominent role in human society, often as simplified abstractions of real-world activities. The human ability to transfer knowledge and skill between games and life provides a powerful view into how learning applies across both small, formal systems and complex, ill-defined human experience. Artificial Intelligence has a long history of working in game domains, and more recent digital games provide challenges to understand increasingly dynamic, uncertain, complex systems. In recent years, there has been a good deal of interest in real-time strategy (RTS) games as a domain for planning, strategy learning and transfer learning. These games present well-defined, simplified action contexts over very large state spaces with durative multi-agent interactions in real-time, complex conditional outcomes, and the need to work at multiple levels of abstraction.

Learning in RTS games has focused on statistical learning how to win the game over higher levels of abstraction such as what units to build or what pre-created strategy to apply. Because they are very popular games, they have well-understood strategies and tactics that can be leveraged and/or learned. Experts in the domain are relatively easy to come by, and there is a wealth of existing text and other resources describing and analyzing the various games. Prior projects have used these resources to model human explanatory communication (Metoyer et al., 2010), as well as to script tactics and strategies as hierarchical plan steps. It has been shown that

these steps can be effectively applied and learned (Aha et al., 2005, Ng et al., 2009, Molineaux et al., 2008) and transferred to novel situations (Sharma et al, 2007).

In this work, we investigate the processes involved in learning to play a game. We are inspired by two observations about how human players learn to play. First, learning the domain is intertwined with goal pursuit. Human players predominantly learn to play by actually playing and the low cost of failure encourages exploratory gameplay. Second, games are designed to ramp up in complexity, walking players through a gradual cycle of acquiring, refining, generalizing and abstracting knowledge about the domain. We explore the feasibility of learning a hierarchical, causal-link planning domain with conditional outcomes by observing agent behaviors while pursuing in-game goals. This approach follows the learning-by-doing paradigm, but does not rely on prior traces of expert play. We propose an integrated planning, learning and execution system. The planning module creates command/event groups based on the operators it has learned, ceding control to the execution module to try out new commands when it gets stuck. Observations of unit behavior are collected during execution and returned to the learning module, which hypothesizes generalizations and creates test cases for them. The planner uses those test requests to generate command/event groupings that will simultaneously pursue the goal and facilitate learning the domain. We demonstrate that this approach can efficiently learn generalized commands that have conditional outcomes through staged observation and demonstrate the learned knowledge by testing in multiple scenarios.

Problem Statement

Within the realm of artificial intelligence many different methods for teaching a machine to learn exist. Many of them involve having the machine learn the desired problem directly. This

is done by immediately giving it the final problem and letting the learning algorithm attempt to solve the problem. The issue with this method of learning is that it may require many test iterations which can make the problem take a long time to complete. Also, if the test space is large the amount of possible test iterations can border on the infinite. This makes it desirable for a different approach for learning those types of problems.

In this thesis we use something called ‘staged observation’ to learn. A system using staged observation does not attempt to immediately tackle a desired learning problem. Instead it first learns multiple simple aspects of the problem so that when the actual problem is loaded to be tested it only has to learn the way that the previously acquired knowledge is combined. This allows knowledge learned to be built on top of each other; which is similar to the quote by Isaac Newton “If I have seen further it is by standing on the shoulders of giants” (Isaac, 2012) – in that way the previously learned knowledge learned is stood upon by newly learned knowledge. Because of these ‘stages’ of learning exist it is necessary to ensure that the lower level knowledge is accurate otherwise it is possible that the final knowledge learned may be inaccurate or could take a long time to generate a good result. This is similar to building a skyscraper, the lower levels must be stable otherwise the whole thing could come crashing down. In order for this to work it is important that the correct base testing scenarios are handpicked. In order to do this it is necessary to fully understand the testing scenarios in advance. This handpicking is exactly what is done in tutorial scenarios for Humans. This requires knowledge of how both the learner operates and also what will be generated by that learner in multitudes of situations. This handpicking of initial test cases allows the true nature of the commands to show themselves which makes the command be able to be used effectively and efficiently in the final problem. However, once the true nature of the command is found it is then possible to use them on cases

that they were not handpicked. Although for more complex strategy the staged observation would still be required because the generated plans from the commands will need to be fully observed – just like the commands. This style of learning follows the way humans learn. For example, when trying to learn how to integrate in mathematics one does not just sit a baby down in front of an integration problem. Instead the person first learns how mathematical operators work then fractions, decimals, algebra, trigonometry, and then calculus. Once all of those have been mastered that knowledge can be put to use with a little bit of new information and the problem dealing with integration can be tackled.

Related Work

There are a multitude of examples of learning using independent agents, learning by observation, using a Real Time Scenario (RTS) game as the learning domain, etc. In order to more fully understand the domain of research relating to the desired research project a massive amount of related research papers were investigated. Following is a small sample of works that this thesis was inspired by. The final paper (Learning Planning Operators by Observation and Practice) is the work that most inspired this thesis and is the most similar.

Learning Models of Intelligent Agents

The focus of this paper (Learning Models, 1996) is the interaction between multiple intelligent agents. In the real world intelligent agents must frequently interact with each other which creates the need for them to react and adapt to how they interact with each other. The way the paper observes that interaction is by treating the interaction as a repeated two player game. Inside this thesis we also interaction between multiple intelligent agents (the units within

StarCraft – which is explained a later paragraph). This is the case because each of the units have their own built in behavior other than what they are commanded to do. In the research paper the agent has a model of what the opponent has done in the past and it uses that in order to predict what the opponent will do in the future. This is similar to what our learner does – it creates effects based on observed behaviors. Within the research paper there are two types of examples that can be brought to the model: supporting or counter example. When a counter example occurs the current model is modified to allow for the counter example. The name of this algorithm is US-L* (Unsupervised L* algorithm). The algorithm can also learn by observing its opponent's games vs. other opponents.

Markov games as a framework for multi-agent reinforcement learning

This paper (Markov Games, 1994) uses Markov games to demonstrate learning when "exactly two agents with diametrically opposed goals share an environment." The reasoning behind this approach to learning is because as the paper (correctly) states, "no agent lives in a vacuum." It uses "a Q-learning-like algorithm" (called minimax-Q) "for finding optimal policies." This idea of pitting two agents against each other is exactly what we are trying to do. In StarCraft, the overarching goal is to destroy all the opponents. Our learning algorithm starts at low levels (for example, moving a unit), then moves up to higher level goals (staged observation). Just as the paper states, our learning and testing is being done verses an opponent.

Learning Models of Other Agents Using Influence Diagrams

(Influence Diagrams, 2000) "provides a framework that an agent can use to learn the models of other agents in a multi-agent system based on observed behavior." When doing a task

it retains a number of models and each of the models has a probability assigned to them. Also, it has the ability to modify current models to (Influence Diagrams, 2000) "better account for the observed behaviors." It knows if a plan needs to be modified by the probability that is assigned. The modification includes a parameter refinement process. One situation where this system was tested was in an Anti-air Defense Domain in which incoming missiles must be shot down by two interceptor aircraft that are out of communication with each other. In the world of StarCraft each unit can be thought of as an agent. This thesis's goal of learning that passing through an enemy unit's line of fire is generally bad because the unit could be damaged/killed is similar. Hence, one could say that our agent also learns the behavior of other agents. Also, like the before mentioned system, our system includes a plan refinement system called generalizing. Once we have observed multiple examples of the same command those are then generalized together to remove unnecessary preconditions and superfluous effect.

Multi-agent Reinforcement Learning: Independent vs. Cooperative Agents

The point of (Multi-Agent Reinforcement, 1997) is to show that cooperation between agents will allow the agents to be more effective than having strictly independent agents. Throughout the course of the paper it is shown that cooperation does speed up joint tasks. However, it was also learned that the information that was shared between agents must be prudently used otherwise it can interfere with learning. This was shown using a Q-learning algorithm. This thesis's learner has some similar traits - it doesn't learn cooperation like in the paper, but it does learn by getting information from multiple sources (units) during the course of a game. In essence, it is a single learner taking in information from multiple agents' actions.

Learning Hierarchical Performance Knowledge by Observation

KnoMic (Learning Hierarchical, 1999) is a real-time "learning observation system [that] extracts knowledge from [an] observation of an expert performing a task and generalizes this knowledge into rules that an agent can use do perform the same task." It is "designed to work in complex domains with real time behavior requirements." The testing domain for the system was in Quake II. This thesis learns similarly, not by observing an expert agent but instead by trying out a command and observing and interpreting the results it receives from StarCraft. Also, like KnoMic, the learner works in real time due to fact that our testing is done entirely in StarCraft which is an RTS game.

An Extensible Action Architecture for Planning in Complex Domains

(Extensible Action, 2011) deals with balancing remaining resources (e.g. fuel) while doing a task (such as driving a vehicle). It uses what it calls 'ARC's (Action Component Relations) which are part of an "expressive model of actions containing variables and relationships between those variables." Another feature of this model is it has iterative repair. Our project uses the idea of iterative repair. After we have generated effects for a command and a command is tried out again, the old effects and preconditions are generalized against the new effects and preconditions which will remove unneeded preconditions and superfluous effects (in essence it 'repairs' them).

Decomposition and Causality in Partial Order Planning

DPOCL (Decomposition, 1994) is a "partial-order causal link planner that includes action decomposition." The whole idea of plan decomposition is to break a high level idea down into

smaller and computable pieces. The whole idea of staged observation follows the same line of thought – break a problem down and learn the lower level basic commands like ‘move’ and work its way up to higher level concepts.

A Probabilistic Planner for the Combat Power Management Problem

The CORALS (Probabilistic Planner, 2008) system is a probabilistic planner. It was designed for the Combat Power Management problem. It works by generating “a set of local plans, one for each target, then merged them by searching the space of global plans.” CORALS also has an “iterative plan repair process” that is the result of merging plans. The system is designed to make choices in a similar fashion as to how a human would reason. The plan merging process is similar to how the generalization algorithm works in our system. Generalizing is where we take multiple commands with effects and preconditions and they are generalized together with the result being the elimination of incorrect preconditions.

Nonlinear Planning with Parallel Resource Allocation

NOLIMIT (Nonlinear Planning, 1990) is a nonlinear problem solver that “generates a partially ordered solution from a totally ordered one.” It does this by analyzing “the dependencies among plan steps.” This system exploits parallelism in the plan steps so that it runs multiple non-dependent steps in parallel from the partially ordered solution. One of the ways the creators of NOLIMIT tested the algorithm was by having agents come together to help push a box. Our algorithm has a similar feature and yet it works in the opposite way. We generate plans from observations and which have effects (post-conditions), we do not create them from existing

plans. Because we know what effect a specific plan will have we can combine plans together to generate larger and more complex plans.

Learning Opponent Strategies through First Order Induction

The goal of this paper (Learning Opponent, 2011) was to learn opponents' strategies in games by using first order inductive learning (FOIL). After seeing a trace of an opponent's strategy, it converts that trace into a rule. It then uses those rules in game to recognize opponents' plans in order to defeat them. The way FOIL method generates rules is similar to how we do in our learner. We initiate a command in StarCraft then observe what happened at each frame. Then we generate plans, which are similar to their rules, from the results.

Integrated Learning for Goal-Driven Autonomy

LGDA (Integrated Learning, 2011) is a goal-driven learn algorithm that learns the domain knowledge that is required. For the learning portion of the project, it uses case-based reasoning (CBR) and reinforcement learning (RL). In their testing it performed well against other GDA's that had expertly crafted domain knowledge. Similarly in our system, our learner and agent know nothing innately – it only knows what it can observe what they see. That forces our learner to observe and learn about the domain. It does this by trying out possible commands and seeing what effect that command has. Those results are packaged into command/event groupings that the learner/agent can use in the future.

Transfer Learning in Real-Time Strategy Games Using Hybrid CBR/RL

(Transfer Learning, 2007) uses case-based reason and reinforcement learning in tandem inside an RTS game. The location of learning for CARL was in MadRTS. Its goal was to be able to learn in one scenario and to be able to use that learned information in another scenario. This is another feature that is in this thesis. In fact, that method of learning is the basis of staged observation learning. Once our learner has acquired knowledge from small test scenarios, that knowledge is then used in other tests that were not learned on inside StarCraft. Since we package each of those bits of knowledge about each command into command/event groups, we can accurately select commands to use once larger and more complex situations arise to be tested.

RETALIATE: Learning Winning Policies in First-Person Shooter Games

RETALLIATE (Retaliate, 2007) is an "online reinforcement learning algorithm" that is designed to learn and win in team FPS games. It is able to play against any BOT whose behavior is unknown but fixed. Its performance was slow in the beginning of "domination" game but as the game advanced it was able to overcome the slow start and defeat the opponents. Likewise, our learner will be able to take any bit of knowledge that it learned from previous knowledge acquisitions and apply them to a new scenario.

Case-Based Goal Formulation

The EISBot (Case-Based, 2010) was designed for the purpose of creating a case-based technique for "formulating new goals… using a library of examples." It was created to work and learn in StarCraft. It had a 14% winning rate verses human players on a ladder ranked server. (This low percentage was attributed to the competitive and skilled nature of that server). The way

the EISBot plans is similar to ours in that when our learner receives a goal it checks to see if there are any plans available that can satisfy part of the goal. If there are none it tries commands in the world then packages those commands into plans if they accomplish a desired goal. Another part of this research that we used is the JNIBWAPI that was created for the EISBot (described later).

Learning to Win: Case-Based Plan Selection in a Real-Time Strategy Game

CaT (Learning to Win, 2005) used case-based reasoning and applied it to WARGUS (an RTS game) in order to not just learn the game but to win. In attempting to win an RTS game, one needs to not just be able to defeat one specific opponent, but to be able to defeat many opponents with different strategies. That is exactly what approach CaT took. CaT's concept is generally along the lines of what we wish to do. When our learner/agent acquires knowledge, that knowledge is packaged into plans that are not based on a specific opponent or scenario. Because that knowledge is multi-purpose it can (theoretically) be applied against any opponent within our domain of StarCraft.

Making Robot Learning Controllable: A Case Study in Robot Navigation

A case study was done on RoLL (Making Robot, 2005) which is a "robot control language that allows explicit representations of learning problems." That language was "applied to learning robot navigation tasks" which in this case was robot soccer. This project has some similarities to ours because like the robots, the units in the StarCraft game (such as a Marine) have their own behavior after being given a command. One aspect of this thesis is where it shows the units learning the difference between move, attack-move, and attack. The inclusion of an

enemy unit will show the independent behavior of our commanded unit and make the difference between those three commands apparent.

Learning Planning Operators by Observation and Practice

This extension of prodigy (Learning Planning, 1994) learns preconditions and the effects of operators by observing expert agents. Their learner starts off with no knowledge about either the preconditions or effects. During the course of learning some incorrect operators and preconditions are generated but that is not an issue because their system "deals explicitly with incomplete and incorrect operators." Our project most closely resembles this prodigy system. Their system operates by having an expert agent (usually a human) do an action 100% correctly while observing. That observation is composed of the: 'name' of operator being executed, state before the operator is executed (called 'pre-state'), state after the operator is executed (called 'post-state'), and the 'delta-state' which is the difference between pre and post states. An 'episode' is what is termed as a sequence of observations. Those episodes are then sent to the learning module. Inside the module the preconditions and effects are generated. It creates a directed graph to link goals or operators to preconditions. Next the new or refined operators are sent to the planning module. The received operators may be incorrect or

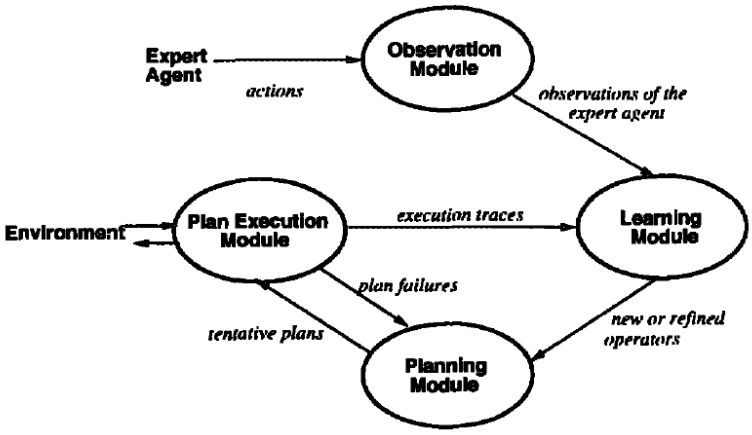

Figure 1.1: Flowchart of Prodigy's system (Learning Planning, 1994)

incomplete because the preconditions could be over specific or the effects may be incomplete. The planning module uses plan repair when the execution fails due to preconditions. It generates a plan fragment to try to verify the validity of one precondition at a time. The tentative plans are sent to the execution module where they are tested in the environment. If it succeeds the trace is sent to the learning module where it sees if it can refine it. If it fails it is sent back to the planning module where it tries to figure out what went wrong and how to fix it. Prodigy refines the operators by first negating an operator then executing the plan. The result of that plan is then treated as an observation. Each of the operators are treated as independent from each other. Following are some of the major differences between our project and prodigy:

- Prodigy has an expert agent (such as a human) do an action that is then observed; whereas we have our *Learner* randomly try out commands in an attempt to figure out what each of them do in order to satisfy a given goal.
- Prodigy requires that everything in each state is observable. This is partially true, except in the world of StarCraft the Fog of War is always enabled so that only an area around the unit that was given the command is observed – due to this perfect information about the entire world is not available.
- In Prodigy there are no conditional effects. We do have conditional effects and they are identified when generating the preconditions for the plans.
- Within the learning module Prodigy checks to see if what was learned was learned for the first time. Ours allows a command to be tried again only if there were no plans available to satisfy a precondition. This is allowed because we have conditional effects – those conditional behaviors may not surface until a command has been attempted a few times.

- Prodigy tries to figure out what preconditions are bogus through practice by removing preconditions. Ours only modifies preconditions if the plan fails or if a previous version of the command was learned and the effects are different.

CHAPTER II

SYSTEM DESCRIPTION

Within the system described in this thesis the planning operators are learned from observation sequences in an incremental fashion. The learning method applies when the domains can be modeled with discrete actions, the states are observable, and the description language for representing the states and the operators (i.e. the set of predicates) is known. At any point of time during learning, any learned operator always has all the correct preconditions, but it may also have some unnecessary preconditions. Also, we relax the assumption that all the plans generated by the planner are correct. The reason that those plans may be incorrect is because they may be incomplete or missing preconditions – and also from effects not having been discovered.

This work makes the following assumptions:

- The operators and the states are deterministic. If the same operator with the same bindings is applied in the same states, the resulting states are the same.
- Noise-free sensors. There are no errors in the states
- The preconditions of the operators are conjunctive predicates
- Everything in the state is observable

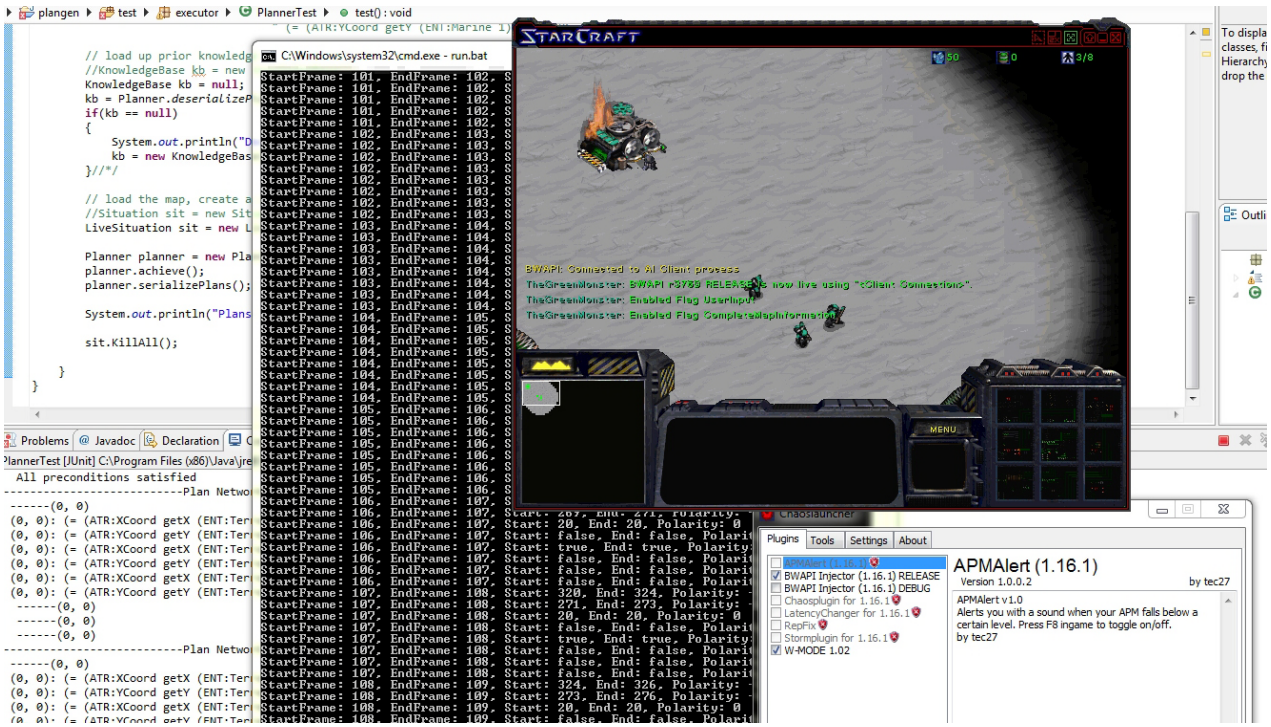

Figure 1.2: All parts of the system – Learner, AI Agent, Chaoslauncher, and StarCraft

Technologies Used

Within this system there are many different technologies used together in order to make this system function. We use StarCraft as the domain for testing our the learning algorithms. StarCraft is a science fiction RTS game developed by Blizzard in 1998. The Brood War expansion that we use was released in late 1998. StarCraft is still considered one of the best balanced asymmetric RTS games of all time. Because of how balanced the gameplay is it is still played in competitions and also used in artificial intelligence research. The next technology used to create the system is the BroodWar API (BWAPI) which is an API that was created in C++. That API was specifically developed to allow researchers and hobbyists to create their own artificial intelligences for StarCraft's Brood War expansion. When a program written with this API is compiled a DLL is created which interfaces with the Chaoslauncher. The Chaoslauncher

is a program that was developed for allowing plugins for StarCraft. Initially the Chaoslauncher was limited to cheats and minor tweaks – such as adding more information to the user interface. With the development of BWAPI the Chaoslauncher became useful for researchers. The ability to allow plugins is what allows the BWAPI to inject code into StarCraft which is how this thesis's AI is able to control units and able to see what is going on inside an instance of StarCraft. Since we wished to write this code in Java we used a ported version of the BWAPI which is called the JNIBWAPI. This version uses JNI in order to communicate with the C++ DLL which then in turn communicates with StarCraft. The JNIBWAPI was initially created for the EIS bot (Case-Based, 2010). The publicly available release is still in an alpha state so the creators of this thesis had to upgrade some of the functionality in order for it to fully operate – which included modifying the C++ DLL that was included and the part that connected to the Java half. The next program is the MPQ editor which is used to decompile StarCraft maps. The final program used is StarEdit. This is a command line StarCraft map (de)compiler that we use each time a map needs to be saved or loaded, or if some information about a unit (such as its hit points) needs to be changed.

Terms

The system described in this thesis can be split up into two major components. The first component is what we call the *Learner*. The *Learner* is the portion of our program that does the learning and analyzing of the information received from the data from StarCraft. This portion of the code was written in Java and contains approximately ninety percent of the code. Another other nine percent of code is what we call the *AI Agent*. The *AI Agent* is the second component; it receives commands from the *Learner* via a socket and executes those commands in an instance

of StarCraft. This program has the ability to have many instances open in order to connect to StarCraft. It is also written in Java and is integrated with the JNIBWAPI which connects to the BWAPI DLL which in turn communicates with StarCraft through the Chaoslauncher (Figure 1.3). This program can also be queried to learn about the state of the world within StarCraft. The remaining one percent of the code is split between the BWAPI DLL and the various batch files that were/are generated for executing the system.

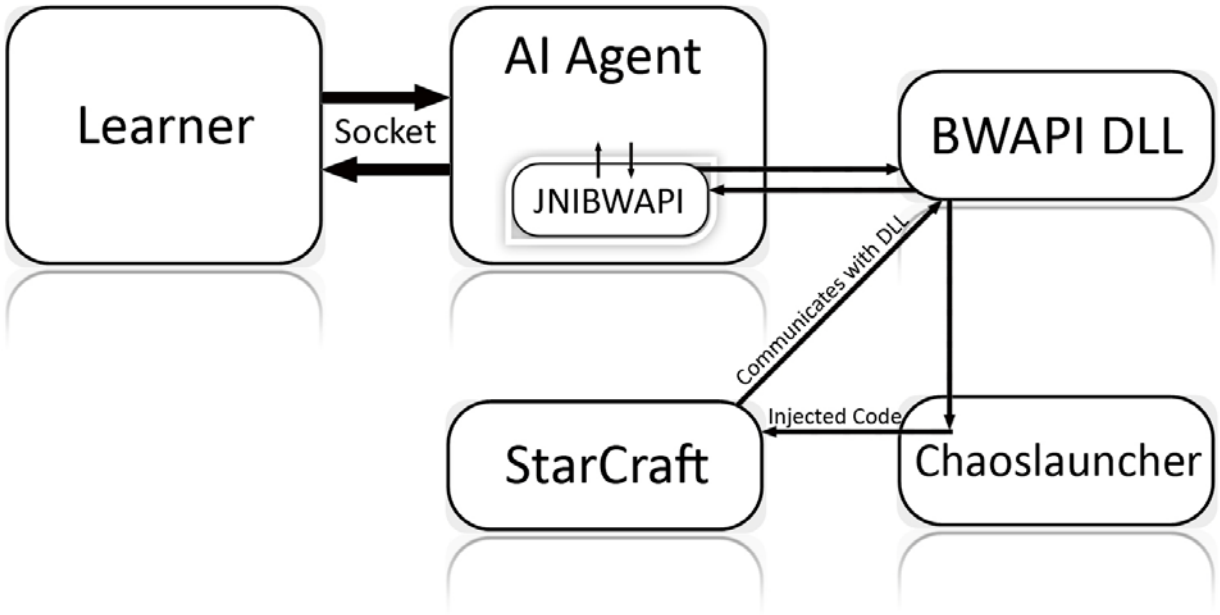

Figure 1.3: Outline of the system created in this thesis

Functionality

The *AI Agent* executes commands that the *Learner* sends to it. It also sends back information about the environment in which the command was executed. The *Chaos Launcher* is a program that allows code to be injected into StarCraft. Without it the AI Agent could not execute commands or get information from StarCraft. The Learner uses a Socket to communicate with the AI Agent. A socket is required because two way communication needs to be available

when executing commands within multiple StarCraft instances. Not only can the Learner send commands to the AI Agent, but it can also inquire about the state of the game in general. For instance it can ask for the information about all the buildings that your ally owns. This type of query is allowed so that when executing plans that require any unit of a specific type to accomplish a goal, the learner will know what units are available.

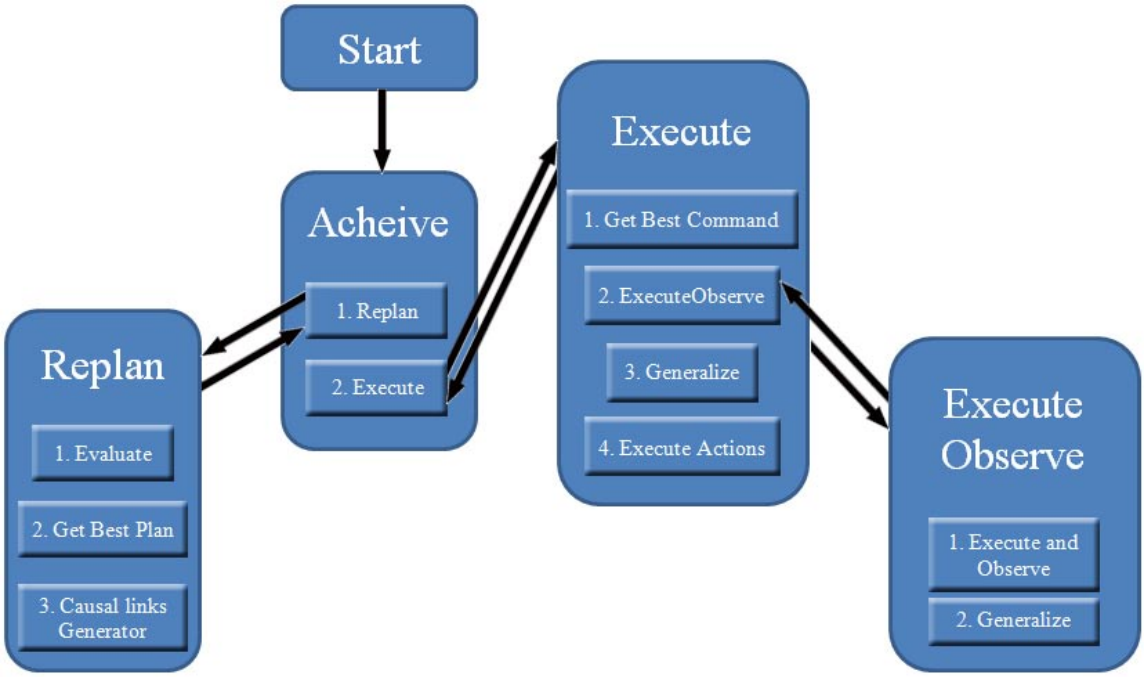

Figure 1.4: System Flowchart

Algorithm Description

First the *Learner* saves the goal that it has been given for future comparisons. The goal is composed of a set of relations; Figure 1.5 shows a learning goal. In this example the goal means that Terran Marine with ID three needs to be at location 370 on the x-axis. The same marine with the same ID also

```
"(and " +
" (= (ATR:XCoord getX (ENT:Terran_Marine 3)) (VAL:XCoord 370))" +
" (= (ATR:YCoord getY (ENT:Terran_Marine 3)) (VAL:YCoord 300))" +
" (= (ATR:XCoord getX (ENT:Terran_Marine 1)) (VAL:XCoord 299))" +
" (= (ATR:YCoord getY (ENT:Terran_Marine 1)) (VAL:YCoord 257))" +
" (= (ATR:XCoord getX (ENT:Terran_Marine 2)) (VAL:XCoord 399))" +
" (= (ATR:YCoord getY (ENT:Terran_Marine 2)) (VAL:YCoord 357)) )" );
```

Figure 1.5: Sample goal given to Learner

needs to be at 300 on the y-axis. Terran Marine of ID 1 needs to be at x-axis 299, etc... Each of these relations is considered a precondition of the goal and therefore must be satisfied in order for the goal to be completely satisfied. Next it loads in all the saved information from an .ini file that contains path info for the Chaoslauncher, AI Agent, initial map to test on, StarCraft, and the exact names of those programs that are needed to run. The Chaoslauncher has a .ini file that is loaded into the Learner and which is then modified in order to allow the correct map to be loaded. If there are any other relevant settings, they are then changed. Now that all the settings are correct the AI Agent, Chaoslauncher, and an instance of StarCraft are then launched. The Learner saves the process id's of the AI Agent, Chaoslauncher, and StarCraft instance so that at the end of the execution it can terminate their processes. They are saved by using a generated batch file that gets all of the processes, saves them to a file, and then loads them into the Learner. The batch file and the txt file that contained the process id's are then deleted. Next the program checks to see if any previously created plans have been saved to file. If there have been, they are deserialized and are loaded into the knowledge base. The map and goal are then packaged into what is called the '*situation*' and is saved with any of the previously created knowledge (which is inside the knowledge base) in the Planner. At this point the *Learner* will now try to achieve the goal that has been given.

Replan

Once the *learner* attempts to achieve the goal there is a continuous loop that exists. While the method 'replan' is not true it calls 'execute'. Inside the replan method it first checks to see if there are any blocked preconditions and if the preconditions on the goal have been satisfied. Since goal conditions are also considered preconditions this function can be called recursively in

order to satisfy each of the preconditions. If a plan is found that accomplishes some of the goals, the preconditions of that discovered plan are checked to see if they are currently satisfied. If they are not the replan method it is recursively called in order to see if there is a plan that can satisfy the precondition. When a plan is found that has satisfied preconditions that plan is saved to be executed. To see if there is a plan that satisfies a precondition the 'getPlansForEffect' function is called. That function returns all of the commands that have events/effects that will satisfy the precondition in question. Next the AI agent is queried for the current state of the world. That state of the world is then compared to the preconditions that are in the events that were retrieved, and the information about how many preconditions of the failure events and success events are tallied. The percentage of these are then multiplied by each other to find which command has the highest possibility of success. The command that has the highest probability value then assigns values to all of the attributes that are inside the command from the effects. Next a function is called to see if any of the other preconditions will be satisfied by the command that is about to be executed. This helps to minimize the number of commands that need to be executed for all of the goal's preconditions to be satisfied. If an optimal solution to the number of commands needed (aka the lowest number of commands needed to be executed) to accomplish the goal was desired, each of the preconditions would need to be looked at and compared with each other and the goal's preconditions. Because of the recursive nature of this test the number of possibilities is exponential. This optimal plan problem has similarities to the "finding optimal change" problem. Although this would be a nice feature to include, due to the scale and scope of the testing this feature was not implemented. Even if the least number of plans could be found that would satisfy the goal's preconditions each of the plan's preconditions would also need to be taken into consideration. If this learner was trying to do something complex such as play a full match

verses an opponent having some pseudo optimal algorithm for selecting the commands in order to minimized the number needed to be executed may (hypothesis) improve the performance. The commands are then added into the tentative command list. After all of these previous steps are taken, if there are still some blocked preconditions this is where the preconditions are solved recursively. The preconditions have causal links to each of the plans that are selected.

Execution

After the replanning is done the learner goes to its execution phase. If there are any preconditions that are still blocked the learner selects a command from a list of all possible commands. If possible, it selects one that has not been executed yet so that something new may be learned. If there are any matches between what attributes the command has and what the precondition to satisfy has (such as the precondition's XCoord and the command's XCoord) they are copied into the command. Otherwise the values are randomly generated – in the case where the command requires a target unit, the Learner first queries the AI Agent to see what units are available. The AI Agent can only see what units are outside of the fog of war (within its line of sight). All of those units are sent back to the Learner and it randomly selects one from that list. However, if a command (such as move) is selected and the command has a field that specifies a target unit, the coordinates of that unit are used. That command is then sent over the socket to the AI Agent.

The AI Agent then parses the command that was received to determine which one it received. Using Java's reflect.Method library it then executes the command. A unit observer is initialized which observes the unit that the command was given to and any other relevant units. The state of that unit is saved on each frame of the game updating and other relevant information

about what is occurring inside StarCraft is saved. The frame number is also saved so that the temporal order can be seen. Other enemy units that are nearby are also observed in order for complex behaviors to be discovered in the future. The reason for saving is because sometimes the action that a unit performs is reliant on what other units are in its vicinity. For example during the execution of the attack move command if there are no units in the area the learner will assume that the command does the same thing as the move command. However, if there are enemy units within range when this command is executed the unit will move to attack. This example shows why it is important for other relevant units to be observed during the execution of a command. Once the unit has reached a stable state then the information that was recorded on each frame is sent in one chunk back to the Learner across the socket. As mentioned, that information contains the frame number and all of the attributes that are necessary for the observation. This always includes the Unit ID and Unit Type and also if the observed unit is still alive.

Generate

Once the learner has received the information from the AI Agent it is then saved in a knowledge bank for future easy access. Now a plan is generated from that information. First the holds are found. Holds are the attributes that did not change during the course of the execution of the command. Those attributes are safely eliminated because if they did not change during the execution of the command it is safe to assume that the command did not affect that attribute. Next the effects are generated. The effects are generated by saving the final value for every attribute that changed from the initial state of the command until the final state. These effects are what is looked at for satisfying a goal or precondition. Next the preconditions are generated by

getting the initial value for each of the attributes. If the agent unit died during the execution of the command the effects in events are marked as fail conditions. A fail condition indicates that during the execution of the indicated command, with these preconditions, the command resulted in the death of the agent unit. There are also effects included in these fail conditions, but they are not used by the computer.

Only one copy of each command is stored in the knowledge base at a time. Inside that command there are events. Events are composed of preconditions, effects, holds, entities, fail/success flag, and the probability of success. Once this has occurred the raw plan is then dumped into an event bank that stores everything for future use. If the fail/success flag is true it marks that preconditions on that event cause the plan to fail. This is important to have when choosing which command to select. The probability of success is changed after each execution of a command during the generalization phase.

Generalize

Now that the plan has been generated from the data from the AI Agent, it is time for it to be generalized. Generalizing consists of six parts. First is garbage precondition removal. If the current event is marked as fail, the preconditions are compared to all of the others of the same command type that were stored in the event bank. Using logical conjunctions it figures out if a precondition is garbage or not. For example, if the new event's precondition of some type has a value of 100, and two other success events that were saved have values 100 and 110. This precondition would be considered garbage because the command succeeded and failed no matter the value. Similar logic is applied to the other possible cases.

The second component is where the newly created event has its effects examined. If one of the events is of the same type but has a different value (or one is literal and the other is variable) the current state in StarCraft is saved and the effects are tested, but the command is given different values. If the command is to move, the x and/or y location in the command is changed and then tested to determine which is the correct effect. When testing out a command, the current state of StarCraft is saved so that after the execution of the command it can be reverted back to the original state. After the result of that test is done the AI Agent and the StarCraft windows are closed. Now the Learner modifies a StarCraft scenario file. Everything about the units are included – the positions, hit points, location… of the units are taken from the saved version of StarCraft that was just created. This is done through hex editing a .chk file that was inside the .scx file (.scx is StarCraft's scenario format). Next the .chk file is saved back into the .scx file by using a command line program called StarEdit and then the .scx file is copied into the map folder of StarCraft. Once that is finished a new instance of the AI Agent, Chaoslauncher, and StarCraft are opened which essentially reverts the state of the world within StarCraft to where it was before the test was performed. Whichever effect that succeeded is kept and the other one is deleted. In our experience in every instance of this occurring during testing the variable effect was always kept.

In the third case the effects and preconditions of the new event (success/fail are done separately) are now compared against each of the events that were created in the past. In this instance there are 16 possibilities: both events have equal effects, event A's effects are a subset of event B's effects (vise versa), the effects are disjointed (AB, BC). Replace the word 'effect' with 'precondition' and four cases are created. Since each event has effects and preconditions,

that four and four are multiplied which makes there be 16 possibilities. In each instance logical conjunctions are used to remove/keep/split the events.

Fourth, the fail preconditions are compared against each other in order to further simplify the failure conditions. Next (sixth) the new and old events are once again compared against each other and this time the *Learner* attempts to find a StarCraft map that has all of the required preconditions to satisfy both of them. The effects of the same type are then tested on that map in order to find out which ones are accurate. The previous map is restored after each of these tests.

Finally the new event's preconditions and effects are looped over and all of the duplicates are removed. This happens because in the previous generalizing stages it is possible that the same effect/precondition can be added to the event. Also if any of the success events do not have any effects they are deleted and if any of the fail events do not have any preconditions they too are deleted. If there are any duplicate events they are also deleted, once again this is possible due to the method of removing/inserting events into the knowledge base. Immediately following this the event is added to the knowledge base so that it can be used for future testing and generalizing against.

Execute Again

Now all of the command/event groups that were saved to be executed in the future from the replan method are executed in StarCraft. The replan method is then called again and the cycle continues. On a side note, even when a command is executed that the outcome is known the information gathered from the execution is still sent to the generate, generalize sections. If it is relevant it is saved in the knowledge base. As stated in the replan section if at this point all of the preconditions for the goal have been accomplished there is no need to continue running the test

because the goal has been satisfied. The plans that are stored in the knowledge base are now serialized to a file using Java's Serializable class. That file can be used in subsequent tests or can be saved for later tests. Because the test is done the AI Agent, Chaoslauncher, and StarCraft instances are no longer needed. So the Process ID's for the AI Agent, Chaoslauncher, and StarCraft are retrieved and the processes are terminated. The socket that has been running for communicating from the Learner to the AI Agent is also terminated.

CHAPTER III

EXPERIMENT

Hypotheses

The main point of the system that was implemented in this thesis was to test the idea of 'staged observation'. A system using staged observation does not attempt to immediately tackle a desired learning problem. Instead it first learns multiple simple aspects of the problem so that when the actual problem is loaded for testing it only has to learn the way that the previously acquired knowledge should be combined. This allows knowledge learned to be built on top of previously learned knowledge. This requires knowledge of how both the learner operates and also what will be generated by that learner in multitudes of situations. This handpicking of initial test cases allows the true nature of the commands to show themselves which makes the command be able to be used effectively and efficiently in the final problem. However, once the true nature of the command is found it is then possible to use them on cases that were not handpicked.

In particular, this thesis puts forth the hypothesis that having staged tests will create an overall learning increase greater than what would have occurred if the final complex learning test was performed without the staged tests. We believe this to be an accurate hypothesis to test because this is the same way that humans learn. For example, when trying to learn how to integrate in mathematics one does not just sit a baby down in front of an integration problem. Instead the person first learns how mathematical operators work then fractions, decimals,

algebra, trigonometry, and then calculus. Once all of those have been mastered that knowledge can be put to use with a little bit of new information and the problem dealing with integration can be tackled.

The domain that this thesis tested in is the RTS game StarCraft. It is used because it is easy to limit the scope so that the results from a test can be easily observed. There are two different situations that have been created into StarCraft scenarios that will determine whether the learning was successfully accomplished. Part of the goal of this thesis is for the given goals to be successfully accomplished. Within StarCraft we have narrowed the number of possible commands down to three: move, attack-move, and attack. Each of these commands have similarities in how they operate, but each have some effects that are unique to their own command. Because staged observation requires lower level testing to be done before the final tests, we have created five different low level base scenarios for testing the commands. As mentioned in the problem statement, it is necessary that those base trials be selected carefully so that the base knowledge will be able to be learned. Because of this, the writers of this thesis crafted the tests so that the results of each of these test cases would cause the true nature of the commands to be shown. This type of careful selection the same as what is done in selecting tutorial levels that humans use to learn. Note, each of these tests were created before the actual generalization algorithm was devised, this ensures that the tests were not created to cater to the learning process and give a favorable outcome.

Experiment Setup

Even though StarCraft has three different teams to choose from (Terran, Protos, and Zerg), we chose to test with just the Terran because the other two have some unique unit

mechanics. Those mechanics would only serve to complicate the learning process and so are not included for simplification of testing purposes. The learner is only given information that a human who is playing StarCraft could find out, aka only information available in the UI is available. This gives the learner legitimacy in that it is not cheating in the game by having perfect information. For our tests we have limited the number of possible commands inside StarCraft for a unit to three. Those three commands are: move, attack-move, and attack. Each of these commands have specific instances where they work and do not work. Our initial test cases where they learn were designed to allow each of these commands a chance to operate properly before being used on more complex tests. It is important to note that the ordering of these basic staged tests was selected so that the maximum information is learned and also so that information will have the correct preconditions and effects.

Experiments to be Conducted

Here are the basic staging tests that were conducted initially to give the planner something to learn on before the complex tests were attempted. The first Trial uses Scenario 1 (Figure 1.6); this is where a unit is tasked to move from point A to some point B. Both the move command and the attack-move command work in this trial and they will both have the same effect. Because there is no enemy unit present the attack move cannot be tested. The second Trial uses Scenario 2 (Figure 1.7) where the goal is to move a unit from point A to point B with a weak enemy in the middle. Only the attack-move command works in this instance because the unit will stop to destroy the enemy unit and then continue on to the desired

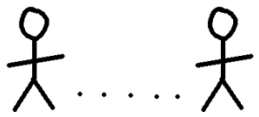

Figure 1.6: Scenario 1

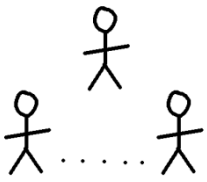

Figure 1.7: Scenario 2

location. The move command causes the agent unit to be destroyed by the enemy that is in the middle and the attack command does not apply because applying that command to the enemy unit will not satisfy the goal state. The third Trial uses Scenario 3 (Figure 1.8) and requires a unit to move from point A to point B with a strong tower in the middle. Only the move command works in this instance because the only way to survive is to run past the tower.

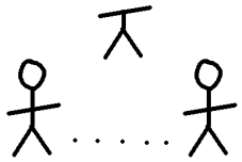

Figure 1.8: Scenario 3

The attack-move command causes the agent unit to stop and attack the tower, but the tower will not be destroyed before the agent unit dies. The attack command will not work here again just like the previous trial. The fourth Trial uses Scenario 4 (Figure 1.9), which is where a unit has to kill an enemy unit. Each of the three commands work in this instance because all of them will put the agent unit close enough to the enemy to be able to attack it. The fifth Trial uses Scenario 5 (Figure 1.10) which is a combination of the second and fourth scenarios. In it a unit is tasked to attack an enemy unit and there is a weak enemy that is between them. Only the attack-move command will work because just like the second trial the agent unit will die unless the enemy unit that is in the middle is destroyed. Attack fails in this instance because for this to work the attack command will have to be carried out twice, once against the enemy in the middle and then finally against the target unit.

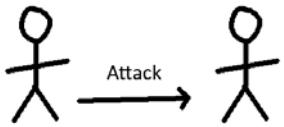

Figure 1.9: Scenario 4

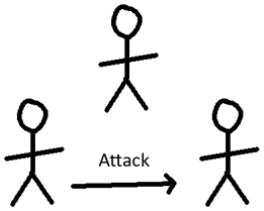

Figure 1.10: Scenario 5

The first Test uses Scenario 5 (Figure 1.10) which is also used in the fifth trial. This test was included because it shows that the information learned from the trials can be used on the trials on which the information was learned. Also, it shows that because it is a trial that was handpicked, all of the information learned from it is accurate, unlike with some other scenarios.

Here are the complex tests that were conducted to determine if the staged observation method works. The second Test uses a scenario (Figure 1.11) where an agent marine is trying to kill a weak target unit without caring if the agent unit dies in the process. The target unit is surrounded by two other enemy marines who are between the agent unit and the target unit. In this instance only the attack command will work as desired.

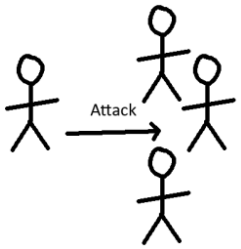

Figure 1.11: Scenario 6

If the move command is used then the agent unit will be destroyed before it gets to the indicated x/y location. The success of this does not depend on where the unit moves to because the strong enemy units will draw its fire as soon as it stops. This makes this command the wrong one to choose for satisfying the goal of this test. If attack-move is used then the agent unit will attack the marines that are defending the target unit and the agent unit will be destroyed. When the attack command is used the unit will go straight to the target unit and kill it, and the goal will have been satisfied.

Result

The results of the trials that were explained in the Experimental Setup section are summarized into tables. Following is an explanation of labels on the columns on the tables. Note: because the failure event's effects don't matter they are not included in the total.

- *Number of Attempts:* The number of times the command was executed
- *Success?:* If the command succeeded in satisfying the goal
- *# of Fail Events*: The total number of failure events
- *# of Success Events*: The total number of success events that are stored in the knowledge base

- *# of New Effects:* The total number of new effects learned before generalizing
- *Total # of Effects:* The final number of effects after generalizing the new events against the old
- *# of Preconditions:* The total number of preconditions after generalizing with the old events
- *# of New Preconditions:* Number of new preconditions before generalizing with the old events
- *Success Rates:* The number of times successfully executed divided by the total number of times executed

Preliminary Trials

When Trial 1 starts, it has no information about the domain saved. After it is finished that information is taken into Trial 2 and learned upon. Then that knowledge that was learned in Trial 2 is then learned on top of during Trial 3. This continues through Trial 5.

Table 1.1: Trial 1

	# of Attempts	Success?	# of Success Events	# of Fail Events	# of New Effects
Move	2	Yes	1	0	2
Attack-Move	2	Yes	1	0	2
Attack	N/A	N/A	N/A	N/A	N/A

	# of Effects	# of Preconditions	# of New Preconditions	Success Rate
Move	2	0	0	100%
Attack-Move	2	0	0	100%
Attack	N/A	N/A	N/A	N/A

Trial 1 (Table 1.1) was picked first because it allows the basic principle of moving to be learned by both move and attack-move. This is important because all of the future trials require this to be known. Also, to learn that both the X and Y coordinates are affected requires that the

commands be run two times. If that was done while some unit was attacking it would drastically complicate the ability to figure out what preconditions applied to what effects. The reason that there are no preconditions on the commands is because there is nothing to test them against because there are no fail events yet.

Table 1.2: Trial 2

	# of Attempts	Success?	# of Success Events	# of Fail Events	# of New Effects
Move	2	No	1	1	0
Attack-Move	2	Yes	1	0	3
Attack	N/A	N/A	N/A	N/A	N/A

	# of Effects	# of Preconditions	# of New Preconditions	Success Rate
Move	2	4	4	50%
Attack-Move	5	0	0	100%
Attack	N/A	N/A	N/A	N/A

Trial 2 (Table 1.2) exposes the weakness of the move command. That weakness causes the unit to be killed by an enemy unit that is in the path to the desired X Y coordinates. The trial also shows that the attack-move command can also attack, not just move. The command causes any enemy unit that is between itself and the final desired location to be attacked.

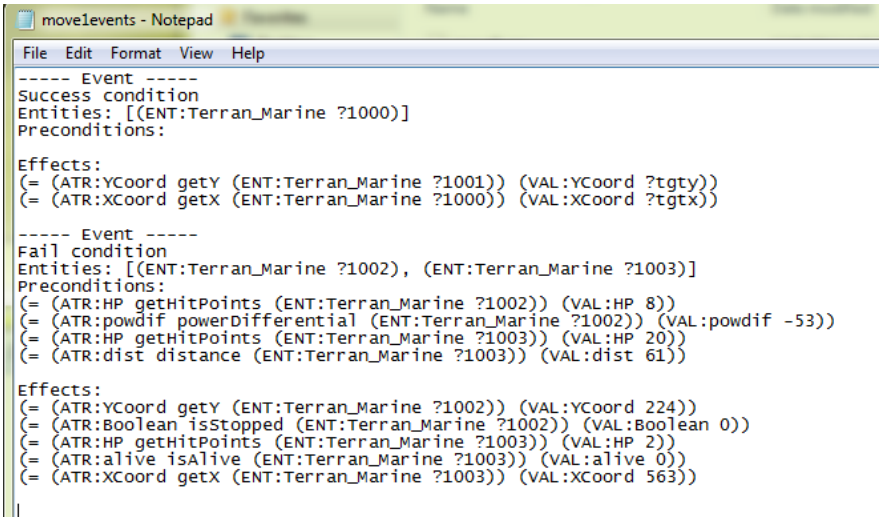

Figure 1.12 Move's results from Trial 2

Figure 1.12 is the output from the second trial. As you can see there is one Success event and one Fail event. The success event was learned in Trial 1. In it there are no preconditions and there are two effects. These effects state that the effect of the command is that the unit moves to the target X and Y positions. The fail event has four preconditions that indicate that an enemy unit starts with 8 HP, the power difference between the agent unit and the enemies is -53, the agent's HP is 20, and that the distance from the enemy in question is 61. This sample is to demonstrate how the events within the program appear. The information that Trial 3 learns is learned on top of this previously acquired knowledge.

Table 1.3: Trial 3

	# of Attempts	Success?	# of Success Events	# of Fail Events	# of New Effects
Move	2	Yes	2	1	3
Attack-Move	2	No	1	1	0
Attack	N/A	N/A	N/A	N/A	N/A

	# of Effects	# of Preconditions	# of New Preconditions	Success Rate
Move	5	8	4	66%
Attack-Move	5	6	6	66%
Attack	N/A	N/A	N/A	N/A

Here (Table 1.3) the move command learns that it is possible to move past a strong enemy that has a higher power differential than it does. The attack-move command learns that attacking an enemy with a larger power differential is a bad idea because it will lose. The attack command has still not been tested because there is no way for it to go to specific coordinates, this makes the command simpler than the other two.

Table 1.4: Trial 4					
	# of Attempts	Success?	# of Success Events	# of Fail Events	# of New Effects
Move	2	Yes	2	2	0
Attack-Move	2	Yes	2	2	5
Attack	2	Yes	1	0	5
	# of Effects	# of Preconditions	# of New Preconditions	Success Rate	
Move	5	10	2	75%	
Attack-Move	10	13	7	75%	
Attack	5	0	0	100%	

In this trial (Table 1.4) the move command figures out that if unit moves to an enemy's location the enemy will die if its health was low. It also determined that the enemy died due to the power differential. This condensed information was determined through generalizing the old events with the newly created one. The attack-move command generates the same effects that were generated in previous trials. Attack was used for the first time so all of the preconditions are eliminated just like what was done for the other two commands.

Table 1.5: Trial 5					
	# of Attempts	Success?	# of Success Events	# of Fail Events	# of New Effects
Move	2	No	2	4	0
Attack-Move	2	Yes	2	3	0
Attack	2	No	1	1	0
	# of Effects	# of Preconditions	# of New Preconditions	Success Rate	
Move	5	14	4	60%	
Attack-Move	10	12	-1	80%	
Attack	5	1	1	50%	

The move command (Table 1.5) generates another fail condition that contains the health of two enemies, the power differential, and distance. Through generalizing, attack-move doesn't gain any preconditions, instead it eliminates one of the previously generated preconditions. The attack command creates a fail event that has only the precondition power differential.

Final Tests with Staged Observation

Test 1 and 2 use the knowledge that was acquired from Trial 5 for each of the commands. The command is selected that the previously given formula chooses in order to see if the information that was learned is correct. The tests are done not to learn, but to show that the information that the system learned is accurate and that it can use the knowledge from the staged testing and properly choose which to use in a situation. The first test is a repeat of Trial 5 in order to determine if the system is able to correctly select the command to use on a trial that that was used for learning purposes. The system is not given any clues that this trial had been run previously. The second test is a new situation for which the staged trials were created. This test helps to ensure that the information learned is able to be transferred to other problems, not just the ones where learning was performed.

Table 1.6: Test 1

	# of Attempts	Success?	# of Success Events	# of Fail Events	# of New Effects
Move	-	-	-	-	-
Attack-Move	1	Yes	2	3	0
Attack	-	-	-	-	-

	# of Effects	# of Preconditions	# of New Preconditions	Success Rate
Move	-	-	-	60%
Attack-Move	10	12	0	83%
Attack	-	-	-	50%

Because this trial had already been run there was no new information generated. The only thing that changed was the success rate. This was expected because this scenario has already been tested. All the preconditions and effects there were generated are generalized out because they have already been seen.

In test one (Table 1.6) the system would choose the attack-move command because of the formula – percentage of satisfied preconditions times the percentage of unsatisfied failure preconditions. As seen earlier, this is the correct choice because this is the only one out of the three commands that satisfies the goal. That choice shows that the information learned in the previous trials was enough to ensure that the system could correctly identify which command to use in scenarios that it already encountered. If staged trials were not implemented for this test the system would have a 33% chance of correctly guessing the correct command.

For Test 2 once again we are using the knowledge that was retrieved from all of the trials combined (Trials 1 – 5). In Test 2 the system chooses the attack command based on the previously mentioned formula. Even though the most sensible choice would be the attack-move command, given that it has the highest success rate and attack has the lowest, the correct choice was made. Without any learned knowledge once again there was a 33% chance that the correct command would have been chosen. Also, if the move or attack-move command had been chosen (without any prior knowledge) it would have taken two attempts to get the unit to move to the correct location and there would have been no way to recognize how the power differential affects the survivability of the unit.

Extra Test

Table 1.7: Test 3

	# of Attempts	Success?	# of Success Events	# of Fail Events	# of New Effects
Move	-	N, N, Y	2	2	3
	# of Effects	# of Preconditions	# of New Preconditions	Success Rate	
Move	3	24	24	33%	

In order to show that the order of learning matters, we did a test with just the move command and ran trials 1, 2, and 3 in a different order (Table 1.7). During one of the trials where the command should have succeeded it instead failed. This was due to the agent unit being next to the strong enemy unit for too long which caused it to die. It is important to note, however, that valid information was learned. One of the success events has no preconditions and only has one effect in which it can move to the target enemy location. The other events have valid information but they have many extra preconditions added on.

Conclusion

As shown, this system is able to correctly choose the command that will cause the given goal to be satisfied after using designed staged trials. As seen in the two tests the probability of randomly correctly guessing which command to use is 33%. Obviously, if the system had not been limited to only three commands then the probability of correctly guessing what command would have satisfied the given goal for the final test would be significantly reduced. Also note that the formula did not include the probability of success. This is the case because in a case such as this the number of test cases is limited to the ones that where new information will be learned. This purposeful setup makes that success probability be artificial and not necessarily representative of the actual success probability if the command were given to multiple 'random' scenarios.

It is important to note that the order of the commands being given makes a significant difference (as shown in the extra trial case). Without using an order for learning that was created intelligently the learning would take longer. This is due to the complexity of domain. In the tests that were done the number of possible units was reduced to two and only three commands were

allowed and ten unit attributes were available. This is drastically less complex than what is handled in StarCraft normally. There are 42 possible commands in StarCraft and well over 100 units with unique abilities and each unit has more than 15 different statistics that can be queried. This complexity is trivial when compared to the real world – and is much greater than what was actually used. Even so, the system needs those staged trials in order for the information to be learned efficiently. Even with the dramatic limiting that was done to the StarCraft domain it floundered when given complex problems without learning through staged observation. Although it did flounder, after multiple trials were executed the inaccuracy of the command's preconditions/effects was slowly fixed – but still not as efficiently as intelligently selected staged trials.

One point to note is that this system has only been shown and is guaranteed to work with certain commands (the ones used in these trials). As mentioned there is a multitude of different commands within StarCraft that have different specifications of what they can do. Including these in the trials would create a need for more complex algorithms for deciding which command to select. Features not implemented due to them being outside the scope of this thesis are discussed in the next section: Future Work.

Future Work

There were many different intriguing aspects of learning that we wished to try out with this system. Sadly they fell outside of the scope of the test due to necessary constraints. Some of the most interesting ideas for future work are 'testing with the built in StarCraft tutorials', 'allowing the computer to dynamically modify maps that it tests on', 'learning by watching a replay', 'winning a match in StarCraft', and 'learning by reading text tutorials'.

Testing with StarCraft Tutorial

Inside StarCraft there is a series of tutorial scenarios that were created so that human players could learn how to play the game. They cover everything about the game starting with how to move a unit, gathering supplies, building different types of buildings, creating units, fighting, and doing specific missions. One of the first ideas that were conceived of when designing this system is that it would be intriguing to have it learn by only playing those tutorial scenarios. Those tutorial scenarios are set up very similar to how our staged testing works. Knowledge is slowly built up so that the higher level knowledge is able to be learned by combining already learned information. The creators of this thesis thought it would be fascinating to see a comparison between the system and a human player both learning to play using those tutorial scenarios. There were many questions devised such as: could the computer learn quicker? Would it need to do each of these tutorial scenarios multiple times? Would there need to be some scenario that bridged the gap between the learning in two consecutive scenarios? If so, does that show how easily humans transfer knowledge from one situation to another? Would the efficiency that the system learned be similar to a human who had never seen this game before?

Dynamically Modify Scenarios

Inside our system we implemented the ability to modify StarCraft scenarios. This was created in order to allow the system to save a scenario, test out a command, and then be able to revert the scenario back to what it was before the test occurred. In doing research we had never seen a project where the system was actually able to modify the scenario on which the testing would occur. This gave us the idea to allow the system to dynamically modify a map during

testing so that it could learn using a specific scenario, not multiple hardcoded scenarios. For example, if the goal was to have a Marine kill an enemy Marine the system would have the ability to modify every aspect of the scenario, including the number of hitpoints that the enemy Marine had. The system then would immediately be able to see that the number of hitpoints directly impacts the ability to eliminate an enemy. Or if the goal was to kill a strong enemy the system could give itself more soldiers to attack the enemy.

Watching a Saved Game

Another way that we wished to make the system learn is by watching replays of real games played by professionals. In theory this would make the learning quicker because you would not be observing a command that was randomly given. Instead, each of the commands and strategies that were executed had some purpose to them. This would make each command executed be able to be put into a plan which would eliminate incorrect attempts at executing commands. This method of learning is more like how Prodigy (Learning Planning, 1994) worked – which could allow direct comparisons against the Prodigy system.

Natural Language Learning

There are many text tutorials available online for humans to learn different strategies for playing StarCraft. This gives us the opportunity to have an alternate method of learning different strategies to play StarCraft. One goal was to read in the tutorials and have them be parsed and be transformed into plans. This would allow the system to have actual professional strategies for learning. The issue with this type of test is that there would need to be some sort of natural language processing, which is outside the scope of this thesis.

Full Game Support – Win as a Goal

One of the most compelling ideas for a learning AI such as this is to have it learn to play an entire StarCraft match. The direction that this thesis project took was based on the intension for this to happen. In the future this project would slowly be extended in order to allow more complex strategies to be developed by the system. This then would demonstrate that the system has the ability to not only learn low level information, but that same learning algorithm is capable of learning complex strategies. In other words it would show that the system is able to be scaled up.

Other Goals

The leaner does not account for inequalities (<, >). If this was included then fewer cases would need to be saved for variables such as powerDifferential and distance. Instead of comparing to a fixed value there would be a range.

Inside the program the ability to change maps is already implemented and it is used for testing out effects on different maps. The next step along those lines would be to give the system the ability to test out preconditions by dynamically altering the map and therefore altering the preconditions. This would allow preconditions to be eliminated more swiftly.

REFERENCES

Aha, David W., Matthew Molineaux, and Marc Ponsen. "Learning to Win: Case-Based Plan Selection in a Real-Time Strategy Game." Sixth International Conference on Case-Based Reasoning (2005): 5-20. Print.

Benaskeur, Adber Rezak. "A Probabilistic Planner for the Combat Power Management Problem." ICAPS (2008). Print.

Bindiganavale, Rama, William Schuler, Jan M. Allbeck, Norman I. Badler, Aravind K. Joshi, and Martha Palmer. "Dynamically Altering Agent Behaviors Using Natural Language Instructions." In Autonomous Agents (2000). Print.

Carmel, David, and Shaul Markovitch. "Learning Models of Intelligent Agents." AAAI-96 (1996). Print.

Genter, Katie, Santiago Ontanon, and Ashwin Ram. "Learning Opponent Strategies through First Order Induction." 24th International FLAIRS Conference on Artificial Intelligence (2011). Print.

Jaidee, Ulit, Hector Munoz-Avila, and David W. Aha. "Integrated Learning for Goal-Driven Autonomy." IJCAI (2011). Print.

Kirsch, Alexandra, Michael Schweitzer, and Michael Beetz. "Making Robot Learning Controllable: A Case Study in Robot Navigation." (2005). Print.

Kumar, Amit, and Alexander Nareyek. "An Extensible Action Architecture for Planning in Complex Domains." AAAI (2011). Print.

Lent, Michael Van, and John Laird. "Learning Hierarchical Performance Knowledge by Observation." 16th International Conference on Machine Learning (1999): 229-38. Print.

Littman, Michael L. "Markov Games as a Framework for Multi-agent Reinforcement Learning." 11th International Conference on Machine Learning (1994): 157-63. Print.

Metoyer, R., Simone Stumpf, Christoph Neumann, Jonathan Dodge, Jill Cao, Aaron Schnabel. Explaining How to Play Real-Time Strategy Games. Knowledge-Based Systems Journal 23(4) pp. 295-301. 2010

Molineaux, Matthew, David W. Aha, and Phlip Moore. "Learning Continuous Action Models in a Real-Time Strategy Environment." Twenty-First Annual Conference of the Florida Artificial Intelligence Research Society (2008). Print.

Moncur, Michael. "Isaac Newton Quotes." The Quotations Page. N.p., n.d. Web. 9 Nov 2012. <http://www.quotationspage.com/quotes/Isaac_Newton/>.

Sharma, Manu, Michael Holmes, Juan Santamaria, Arya Irani, Charles Isbell, and Ashwin Ram. "Transfer Learning in Real-Time Strategy Games Using Hybrid CBR/RL." Twentieth International Joint Conference on Artificial Intelligence (2007). Print.

Smith, Megan, Stephen Lee-Urban, and Hector Munoz-Avila. "RETALIATE: Learning Winning Policies in First-Person Shooter Games." IAI (2007). Print.

Suryadi, Dicky, and Piotr J. Gmytrasiewicz. "Learning Models of Other Agents Using Influence Diagrams." AAAI Technical Report (2000). Print.

Tan, Ming. "Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents." In Readings in Agents (1997): 487-94. Print.

Veloso, Manuela M., M. Alicia Perez, and Jamie G. Carbonell. "Nonlinear Planning with Parallel Resource Allocation." DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control (1990): 207-12. Print.

Wang, Xuemei, Herbert A. Simon, Jill Fain Lehman, and Douglas H. Fisher. "Learning Planning Operators by Observation and Practice." Second International Conference on AI Planning Systems (1994): 335-40. Print.

Weber, Ben G., Michael Mateas, and Arnav Jhala. "Case-Based Goal Formulation." AAAI 2010 Workshop on Goal-Driven Autonomy (2010). Print.

Young, Michael R., Martha E. Pollack, and Johanna D. Moore. "Decomposition and Causality in Partial Order Planning." Second International Conference on AI and Planning Systems (1994). Print.

APPENDIX A

APPENDIX A

DISTANCE CALCULATION

In this thesis distance was mentioned, which is inside the events and is an attribute that is calculated within the AI Agent. Following is the formula and the explanation diagram.

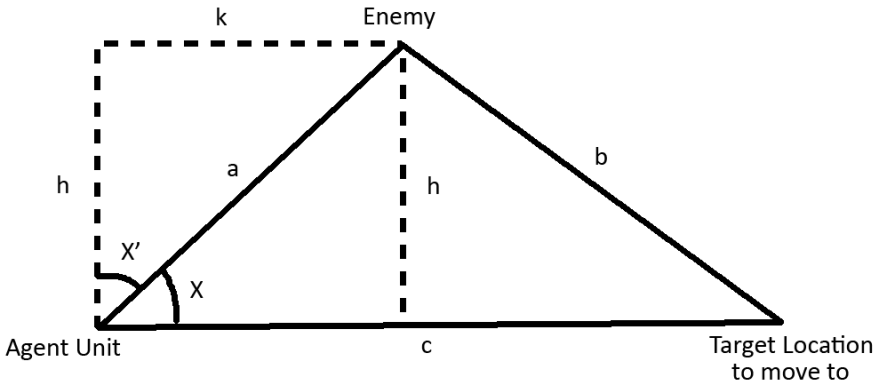

Figure 1.13: Visual representation of what Distance means

$$h = a \times \sqrt{1 - \frac{a^2+c^2-b^2}{2ac}}$$

BIOGRAPHICAL SKETCH

When Zackary Gill was fourteen he decided to take his enjoyment of modding games and attempt to create games himself. This led him to use the GameMaker game engine to create simple games and helped him develop an interest in game mechanics and learn how games are developed. It also made him realize that he wanted to work in that field of study. Zackary graduated high school at the age of fifteen and attended Liberty University at the age of sixteen. He then transferred to the University of Texas-Pan American and subsequently has received two degrees from the University of Texas-Pan American. The degrees are bachelor's degrees in Computer Science and Mathematics. Immediately following his graduation he decided to continue his schooling and pursue a Masters degree in Computer Science again at the University of Texas-Pan American because he felt like "there is so much more to learn". In the summer after his first semester he was contacted by Dr. Tomai to be his research assistant creating a learning AI for StarCraft. This project continued and eventually became Zackary's thesis.

Throughout his education Zackary Gill continued to develop his passion for creating games. In several undergraduate classes he created games to meet the required projects. For the senior project of his bachelor's degree in Computer Science he implemented the game called Cribbage that included three different artificial intelligences for a human to challenge. He is currently planning out several games to create for IOS/Android devices and in his spare time is part of a development team that is creating a free game called Movie Battles 3 – which based on a mod called Movie Battles 2.