The Future of Game AI: Learning to Use Human Tactics

by

Neil M. Dowgun

B.S. Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2008

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTERS OF ENGINEERING IN COMPUTER SCIENCE

AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
SEPTEMBER 2009

© 2009 Neil M. Dowgun. All rights reserved. The author hereby grants to MIT permission to
reproduce and to distribute publicly paper and electronic copies of this thesis document in
whole or in part in any medium now known or hereafter created.

Signature of Author

Department of Electrical Engineering and Computer Science
August 20, 2009

Certified by

Philip Tan
US Executive Director of the Singapore-MIT GAMBIT Game Lab
Thesis Supervisor

Accepted by

Dr. Christopher J. Terman
Professor of Computer Science
Chairman, Committee for Graduate Students
The Future of Game AI: Learning to Use Human Tactics

by

Neil M. Dowgun

Submitted to the Department of Electrical Engineering and Computer Science
On August 21, 2009 in partial fulfillment of the
requirements for the Degree of Master of Engineering in
Computer Science

Abstract

Many video games require artificial players that can act as replacements for human players. With today’s constant increases in the complexity of games and the prevalence of user-created content, creating an AI that is sufficiently adaptable is becoming more challenging than ever. This thesis proposes a system which takes data from games played between humans, and from that data learns their most effective tactics. In this way, it may be possible to create an AI whose potential is limited only by the human imagination.

The prototype that is described in this paper is made up of four components: a Log Interpreter and an Evaluator that process the data gathered from games played by humans; an Organizer that utilizes machine learning to glean useful information from the processed data; and a decision-making bot that uses this information while playing the game. The prototype is implemented using the Torque Game Engine and is applied to a game called TenXion, developed at the Singapore-MIT GAMBIT Game Lab.

The implementation that was used for this project is merely the foundation for an efficient, robust system that can accomplish the goal of learning from human tactics. A large portion of this thesis is dedicated to the description of lessons that were learned over the course of the prototype’s development and ideas for future extensions, including a map-processing module and better methods of restricting the bot to feasible strategies.

Thesis Supervisor: Philip Tan
Title: US Executive Director of the Singapore-MIT GAMBIT Game Lab
Acknowledgements

Thank you Mom and Dad, for your infallible support and for being the best examples of responsibility and thoughtfulness a boy could ever ask for.

Thank you Philip Tan, for giving me a chance to do exactly what I always wanted to do.

Thank you Andrew Haydn Grant, for seeing me through the times that no one else could.
# Table of Contents

1. Introduction ....................................................... 6  
   1) The Progress of the Video Game Industry 
   2) How AI can Keep Up 
   3) Steps Taken in This Paper 
   4) Current Progress 
   5) Related Work 

2. Design Principles ............................................... 12  
   1) Build upon standard practices 
   2) Separate ‘Decision Making’ from ‘Problem Solving’ 
   3) Do most of the work before the game starts 
   4) Know the character of the data 
   5) Be mindful of different users 

3. System Structure ................................................... 18  
   1) The System as a Whole 
   2) Data Types 
   3) The Runtime AI 
   4) The Log Interpreter 
   5) The Evaluator and Organizer 

4. Implementation Details ........................................ 26  
   1) TenXion 
   2) Representing the State of the Game 
   3) Tree-Based AI 
   4) Logging and Log Interpretation 
   5) State Evaluation 
   6) Action Evaluation 
   7) Decision Making based on Maximum Benefit 
   8) Levels of Interaction 

5. Results and Conclusions ....................................... 38  
   1) Results Overview 
   2) More Behavior Tree restrictions can combat a lack of examples 
   3) A learning Interpreter can make Action recognition more reliable
4) Action evaluation needs a new paradigm
5) Using subjective States can prevent cheating
6) More information can be gained by pre-processing levels
7) More complex relationships can be learned with Neural Nets

6. Contributions .......................................................... 45

7. Glossary ................................................................. 46

8. Bibliography ............................................................ 48

9. Appendices .............................................................. 50
   1) Log Files
   2) Tree Structure Files
   3) Action Vector Files
1 Introduction

1.1 The Progress of the Video Game Industry

For some members of the video game industry, designing the artificial intelligence that controls non-player characters is similar to level design—the goal is to create just enough of a challenge, but not so much that the player gives up hope of success. However, this is not the goal for all games—for years, games that can be played multiplayer such as chess or Street Fighter have had to create enemies that are believable simulations of other human players. Brilliant minds have created artificial players that can dominate in these highly-constrained domains, and so the field of game AI has never needed to incorporate much learning into its solutions. However, what happens when the domains we are designing artificial players for becomes less predictable?

Two trends are making the job of the AI programmer more and more difficult. One is the increase in user-created content. User-created maps have already become prevalent in shooters, and are now spreading to other genres, such as racing and platformers. If players want to test their custom maps or play them with bots, the AI programmer’s artificial players may need to act in situations that he never dreamed of. Soon level editing won’t be the programmer’s only problem—he may have to write an AI that can accomplish mission types that were not included in the original game, or that uses the capabilities of characters that were invented after the code has been shipped. How does one write an AI when he does not know what it will be trying to accomplish, let alone where it will be doing it?

The other big trend in video games is increased complexity. Many genres of games are incorporating in-depth simulations of physical, social, and economic systems, and past a certain level of complexity the results of these simulations cannot be predicted at all [Smith]. Consequently, the optimal strategies for these games are also impossible to determine in advance.

---

1 For clarification of terms that may be unfamiliar to some readers or that I have given specific definitions for the purposes of this paper, consult the Glossary.
There are numerous advantages to using simulated systems in games that suggest the industry will be seeing more and more of them. First, a thorough simulation can create consistency throughout a game, so that once the player has been playing for a while they know how the game is going to respond to their actions. This comforts players, and makes it easier for them to immerse themselves in the game’s world. Simulations also can make the job of the Level Designer easier. A physics engine can allow the designer to define all objects in a standard way, rather than having to define unique interactions for each type of object. Lastly, simulations take advantage of a property called “emergence” that creates a vast amount of possibilities for the players to explore using only a finite set of rules [Smith]. Emergence allows designers to add hours to the game-play experience with decreasing amounts of design work.

As simulations become more complex and the potential scenarios produced by their emergent properties become harder to foresee, the likelihood increases that an AI will run into a situation that it will have no idea how to handle.

1.2 How AI can Keep Up

To me, the solution to both the problem of complexity and the problem of user-created content is the same: the games industry needs a system that will allow artificial players to emulate the tactics of human players.

Why copy human players? Well, even when we (humans) find ourselves in completely alien situations, our brains generate ideas for possible solutions. An artificial player with the adaptability of a human, even if it did not always come up with an efficient strategy, would at least find one that would make sense to the humans playing with it. This is hugely important if we are trying to use that AI as a stand-in for a person. An AI that uses a completely counter-intuitive strategy will break the illusion of playing against a human being.

Of course, we might be able to design a program that generates ideas for us, but it is a safe bet that given enough time, some human player will come up with the best strategies anyway. Even for games that the AI programmers think they understand well, it is likely that some human player will eventually master all their virtual opponents. Fighting games are a good example of this, as they have some exceedingly challenging AI, but the ultimate test is
always against tournament level players. The ancient game of Go is another arena in which human intuition and experience has always bested all automated strategies.

Another advantage of being trained on human experience is that an AI that experiments with different tactics by itself would need to spend quite a bit of time finding any that work well. During this time, the player would most likely notice the AI acting odd or incompetently, and would have less fun playing the game. Projects that have tried to learn to play real-time games using Genetic Algorithms or other methods of automated experimentation have held this as a major concern [Stanley]. However, if all the AI has to do is observe human players playing against each other, then it will be ready to play the first time it is called upon, and the illusion of the game world remains intact.

The genetic algorithms approach to learning also potentially has the problem of convergence [Stanley]. That is, all the AI players may eventually end up using exactly the same solution. Of course, there are ways to avoid this result, such as Speciation, but when we are learning from human players there is a surprisingly simple solution. All we need to do in order to generate bots with different strategies is to train them on the histories of different players.

### 1.3 Steps Taken in this Paper

I have created a prototype of a system that automatically generates bots influenced by games played by human players. In order to create this bot generator from scratch, it was necessary to work through several steps: choosing design principles, designing the system, implementing the system, and experimentation to determine what could be improved. This thesis is roughly organized in the same fashion, as the choices made in each step will help you understand the repercussions discussed in subsequent chapters.

The first step of the design process was choosing goals for the overall system. Based upon the status quo of the game industry and the particular goals of this project, I settled upon a set of principles to guide the design and implementation of this system, as well as future endeavors along these lines.

The second step was to outline how the system would work at a high level. The system is divided into several parts, and while the purpose of each of them is easily described, their
exact functionality is undefined without the interfaces between them. Therefore, it was necessary to decide upon the details of their interaction and how the whole system would come together before designing any individual piece.

After the design was completed, the system had to be implemented in order to be sure it would work in practice. Although my implementation underwent retooling several times, it is only the first iteration in creating a successful example of this technology. There are always unanticipated complications in the design of a system that need an implementation to bring them to light, and I hope future endeavors can learn from the challenges that this one faced.

Some of the shortcomings of my implementation did not become apparent until the entire system had been coded and I tried it out with different sets of data. This experimentation was critical in identifying pitfalls that should be considered prior to designing similar systems in the future.

1.4 Current Progress

The prototype works with the game TenXion, a 3rd-person shooter developed by the Singapore-MIT GAMBIT game lab that pits one player against multiple enemies controlled by other people [TenXion]. This game was ideal to work with because it already included a comprehensive way of recording everything that happened in a round of play, and because the goals for the enemy players were fairly simple to analyze.

The prototype successfully interprets logs created by playing TenXion and creates bots that choose the "best" action to perform in any situation, even if the system has never encountered the exact set of circumstances before. Unfortunately, the definition of the "best" action is dependent upon many values that can be tuned by the AI designer, and which produce unexpected results if not perfectly calibrated. Therefore the use of this prototype to create bots that act out preconceived strategies is not very practical, and I have made some suggestions for how this might be remedied in the future.

---

TenXion © 2007 MIT & MDA. All Rights Reserved. Section 4.1 of this paper contains a more in-depth description of the game.
1.5 Related Work

I believe it is important to define the goal and methodology of this project in terms of previous work, so that it is clear what new challenges are being tackled. First of all, the desired system will use a specialized method of Reinforcement Learning to train artificial players. Reinforcement Learning means the player monitors its own actions and gradually learns which ones lead to better performance [Jehiel]. The interesting change in this project is that I am monitoring the actions of other players, but they are other players who are filling the same role as the artificial player that is being trained. Therefore, I am still trying to learn what actions lead to success in this role. By contrast, the Fictitious Player Modeling method monitors the actions of players who are filling the role of an adversary, and attempts to form predictions of what they are going to do [Jehiel]. To use Fictitious Player Modeling would require the extra step of determining what the best reaction is to each of the adversary’s tactics.

To my knowledge there has never been a system designed with the specific aim of learning the best way to play a real-time game through the observation of human players. It is notable, however, that there has been an increasing interest in creating AI bots that play similar to human players. There have been several research projects [Zanetti][Geisler] and even commercial efforts such as Microsoft’s Forza Motorsports [Cowling] and TruSoft’s Artificial Contender [Champandard07] that have already attempted to copy the playing styles of people. The difference between their goal and mine is that I would like to create an artificial player that amalgamates and chooses only the best strategies of a large sample of players, rather than using all the tactics (good and bad) of a single expert. The applications differ as well – my system will be better built for adapting to new environments and missions, while theirs will be more convincing as stand-ins for particular players (for instance, if you wanted to practice playing against an AI that plays like a friend of yours).

Of course, there have been attempts to build AIs that learn to play games as well as they can before. This is basically my goal, but the methodology is different. For real-time video games, most learning AIs have depended upon experimentation on the part of the program. A particularly interesting project was the rtNeat learner used in the NERO game, which used a genetic algorithm to explore different tactics and which was given feedback on its performance
by the user [Stanley]. This sort of experimentation is fine if the users are somehow involved
with and entertained by it, but I am attempting to add a learning AI to games without adding
new modes of play to them.
2 Design Principles

In this chapter I explain the principles that I used as guides in designing my automated bot generator. Some of these principles were discovered midway through the design process, some of them were discovered during my first attempt at an implementation. In the future, I hope that these principles can help guide anyone who is looking to improve and expand upon the goals of this system.

2.1 Build upon standard practices

In the past, those who have written AI for games have not been focused on making truly adaptive AI. After all, as long as the game world was defined by the programmers, all that the AI needed to adapt to was the tactics of the human players. This adaptation was traditionally done in a highly constrained manner, where the game designers would pre-define more behaviors than the player would actually see, and rig cases for the AI to switch between them based upon the current circumstances [Champandard09]. It is clear that such a solution will not be able to scale up as games become harder to predict and the community tries to produce more complex AI [Lent].

However, there is no reason to try to completely revolutionize the traditional approaches to the problem. Not only would it be easier for the games industry to adapt to a learning AI that is somewhat similar to their current approaches, but I found the traditional methods of game AI to be easily malleable and ready to be adapted to incorporate learning. Some theories of how this could be done were already proposed by folks in the industry [Champandard09], and there has already been some research showing examples of how it can be accomplished [Zanetti].

The global standard of bot AI is the Hierarchical State Machine (HSM); a state machine that is capable of defining goals on many levels. For each state in given level of the HSM, there is a goal that needs to be accomplished, and a lower level State Machine is executed until the AI succeeds or fails in achieving that goal [Isla]. Depending on the outcome, the machine may
move to another state on the higher level. Damien Isla [Isla] and Alex Champandard [ChampandardBT] point out that this architecture can be generalized into a Behavior Tree, which is essentially an HSM that uses some form of standardized logic to choose the next state to enter, rather than custom code. This is perfect for building a malleable AI that is mainly controlled by past experience rather than what the AI designer predicts will be good decisions.

![Behavior Tree Diagram](image)

**Figure 1:** The Behavior DAG (referred to in this paper as a Behavior Tree) of the Halo 2 AI [Isla].

### 2.2 Separate “Decision Making” from “Problem Solving”

Although the overall goal of this project is to learn the best way to approach a game, there are certain sub-problems for which there already exist very good (in some cases provably
optimal) solutions. Path-finding may be the best example of these problems. The NERO project showed that it was possible for bots to use reinforcement learning to evolve the ability to navigate mazes [Stanley], yet it is still hard to imagine a way of getting around any user-created map that would be better than using the A* algorithm [Hart].

Some sub-problems have just turned out to be extremely difficult to solve through learning. This is usually because certain actions, such as jumping or firing a gun are infrequent enough that it may always seem best to not do them [Stanley][Zanetti]. Presumably such low-level actions are used to accomplish some goal – in the case of jumping, maybe the goal is “Evasion.” It makes sense, when making an intelligent AI, to recognize when the goal is appropriate rather than when the action is. When it is clear that it is appropriate to pursue a goal then the AI can choose how it goes about it, and the best way to do it may be with custom code.

In short, I was determined not to get carried away with my learning mechanism and handicap the AI by making it learn basic operations that it should be able to do smoothly at the outset. The line has to be drawn between “Problem Solving” (accomplishing a small task) and “Decision Making” (choosing a task to tackle). Any problem where the correct approach depends on the runtime circumstances may actually involve a decision that could be made easier by looking at past experience. However, problems like “how do I get through this door?” usually only have one solution in the realm of shooters.

One of the problems that I found with applying the work of Geisler and Zanetti and Rhalibi to my project is that their neural nets do not make decisions about goals or actions, but are more influenced by the situation. Variable such as “Number of Enemies Around” and “Direction of Nearest Enemy” each influence multiple actions at once, such as whether to strafe left or right and whether to shoot or not [Geisler][Zanetti]. Looking at a set of outputs in the form of a complex action, it is difficult to say why the bot is taking that action, except through a mathematical formula. The “Decision Making” paradigm used in my prototype makes it clear at any given time what the bot is trying to accomplish, so that the system should be easier to debug and tinker with.
2.3 Do Most of the Work Before the Game Starts

Video game programmers often face challenges centered on performance [Isla][Lent]. Keeping a simulated world fluid and responsive can be extremely difficult, and breaking the user experience even for a split-second constitutes a bug. Therefore, a system that is meant to be an extension of a game should be as lightweight as possible, so as not to contribute to the horde of calculations that are already being done every second by the game engine. A system that runs efficiently and does not hog resources when the rest of the game engine needs them is more likely to be deemed useful by the games industry [Lent].

Learning can take a long time, especially since the more data you are able to give a learning algorithm as input, the more effective you expect it to be. It is obvious that many, many different situations are possible when playing a real-time game, so I expected to need a lot of data to feed to the system. Therefore, I needed a way to process massive amounts of data and still create an AI that runs in real time. This was a concern for the NERO project as well, as they wished to create bots that could do “In-Game Learning” [Stanley].

Fortunately, if enough data is gathered from previous games, it is not necessary to mimic behaviors that players have exhibited earlier in the game that is currently running, as that will only be a relatively small sample. Therefore, processing time can be kept to a minimum by doing most of the learning “offline.” My prototype makes use of this observation by batching all the information from previous games and processing it while games are NOT running. This does not mean that I use “Out-of-Game Learning” as defined in [Stanley], because this learning happens after the game has been finalized and shipped, but just not during play.

After the system learns what happened in the past, it needs to reduce what it learned to a point where it can be read and used in real time without any performance issues. This is why I used the simplest representations possible for data that was passed in to the part of the system that is running while the game is being played.

2.4 Know the Character of the Data

When it comes to implementing a learning algorithm, it is important to consider the type of data that is being used as input and what the goal is of the learning process. For
instance, when classifying data, the process is supposed to take a potentially infinite set of inputs and match each one with one of a finite set of outputs. My problem is slightly different – I am taking an infinite set of possible inputs (all the states the game might ever be in) and producing a ranking of a finite set of outputs (the actions that the AI can perform).

The way the game state is represented is closely tied to what inputs will be available for the system’s learning algorithm, since it can only learn from data that it actually keeps track of! In my experience with implementing this system, I have found that all variables in the game state should either be treated as continuous values, or be broken up into sets of Boolean values. For instance, the amount of health a player has is a continuous variable, as is the distance between two players. Whether a player is waiting to spawn is a Boolean value – either they are waiting to spawn or they are not.

There are variables that do not appear to fit this approach at first. For example, the type of weapon that a player is using might be represented as an enumerated set of possible values. But how could you define when to take a particular action, based solely upon the current weapon being used? The logic would invariably be done with ANDs and ORs, at least at the most basic level – the enumerated set of options would eventually be converted into a set of Booleans when making decisions anyway.

It is quite possible that the entire state of the game could be reduced to a bit vector, and most of the important continuous values could be included as well by creating variables such as “isHealth>20”. In some previous research projects, such as [Geisler], the entire state has been represented with binary variables. This was made possible by dividing some continuous variables into chunks, as described above. This idea has already been accepted in the industry – [Isla] suggests using a bit vector of variables to make a quick check of whether an Action is useful at the current time (i.e. “Jump” would not be useful if the character is sitting in a vehicle).

The advantage of using only Booleans is that it would allow us to use certain AI techniques, such as Decision Trees, that only work with Boolean variables. The problem is that this restricts what the bot might learn, as the divisions imposed upon the continuous variables may be rather arbitrary. For this reason I have attempted to allow for both Boolean sets and
continuous variables as inputs, even though it might be more efficient or more effective to reduce everything to Booleans.

2.5 Be mindful of different users

There is a growing level of interest in letting non-programmers design and build AI. There are already game companies out there who write their AI through scripts, either because they want their AI to be easily adjustable or because they want designers (who are more responsible for the feel of the game) to have control of what the AI is doing. Even games that let players influence the behavior of the AI characters have become popular [Cowling]. The question when designing a new system is who should be responsible for changing which parts of it?

First of all, I did not want the players to be responsible for anything. Unlike games like Black & White or NERO, the goal of this system is not to let the player tailor an AI to their liking. Instead, one of the goals of this project is to let the players worry less about the AI when generating their own content – it is not really meant for players to create clones of themselves. Although that is a potential use, I still want the player to be able to do that without leaving the game world – they should be able to customize their AI just by having fun playing the game.

Secondly, I did not want the Engine Programmers to have to worry about how the game is going to work. There are many tools and engines out there that can be utilized in vast, eclectic assortments of games, and I wanted this system to be adjustable in the same manner. Consequently, there are some pieces of the system that are designed solely for the Engine Programmers, making it general enough to work with any game.

As for the code that is specific to the game, I decided to try and divide it into two pieces – that which would be written with the rest of the game, and that which could be tinkered with and adjusted long after the game code is stable. The former code acts as the constraints upon the AI, and is responsible for making sure the AI only performs reasonable actions. The adjustable parts allow the designers to explore the set of reasonable actions until the system is working the way they want it to.
3 System Structure

3.1 The System as a Whole

The structure of my system owes much to the system of [Zanetti], which proved to be quite successful. In my system, the “recording” is done by writing logs that are read by a Log Interpreter. The Training System in [Zanetti] has been divided into an Evaluator and Organizer because I am tackling the extra task of determining the benefit of game states. That means there are four major parts of the system I have designed, but it should be pointed out that none of them could be designed completely without working out some of the details of the other parts. This is because the four parts – the Log Interpreter, the Evaluator, the Organizer and the Runtime AI – must provide interfaces that the AI designer can tinker with, and they all must be able to accept the same design changes.

There are also interfaces between the four parts that dictate what each of them will be able to do. I had to determine what sort of data would be passed from the Organizer to the Runtime AI before I could determine how the Organizer would calculate that data. The type of data also dictates what the Runtime AI will be able to do. On one hand, if too much information is passed to it, it may not be able to process it all given the time constraints of the game. Conversely, if not enough data is given to the Runtime AI, it will not be able to make intelligent, informed decisions.

The system works like this (see Figure 2 for a graphical representation):

- The Log Interpreter receives a batch of log files to read through.
- The Log Interpreter makes a timeline for each game, consisting of States that the game was in and Actions that were taken by the player being tracked.
- The Evaluator takes the timeline and determines how beneficial each Action was in retrospect.
- The Organizer takes the timelines, with Values now associated with each instance of each Action, and turns that information into data that will help the Runtime AI decide if that Action should be used in a particular situation.
- While the Game is running, it continuously asks the Runtime AI for what Action it would like to perform, and updates the Whiteboard to reflect the current State of the Game.
- The Runtime AI returns whatever Action is calculated to be the most beneficial, given the current State of the Game.
- The Game makes the AI player perform whatever Action was chosen.
Figure 2: The High-Level Structure of the System, and the types of data that are passed between modules.
3.2 Data Types

To allow AI designers to tailor the system to their game and to tinker with it afterwards, the types of data that are used need to be flexible but well-defined, as more than one module will be working with each type. The following data types are necessary no matter how the system is implemented, although exactly what data is going to be put into them depends on the game.

Logs – The game must produce records in the same format that the Log Interpreter is expecting them. Logs are collections of changes in the State, and may also contain records of some Actions.

State – The variables present in the State while the game is running must all be present in the State that the Log Interpreter generates. Since the Game generates Logs, and they are in the same format that is passed to the Log Interpreter, the easiest way to guarantee this is to put the same code in both the Log Interpreter and the State Whiteboard. Both of these modules combine a Log entry (which indicates a change in State) with the previous State, and produce a new State.

Actions – The Runtime AI needs to know what Actions it is allowed to perform, and the Log Interpreter needs to know what Actions it is looking for. These must be the same set of Actions, or else the system will waste time on Actions that it cannot use or it will not have the necessary data for Actions that it could have used.

The only interface that is heavily dependent upon the implementation of the system is the one between the offline modules and the Runtime AI. What information are we going to give the Runtime AI to aid its decision (in Figure 2, this is described generally as the Qualifications of Actions)? For the purposes of this prototype, I used the simplest solution possible: a vector that can be multiplied by the State of the game to produce an estimate of the current benefit of an Action (referred to from now on as an Action Vector). This decision was spurred mostly by the desire to do as much work as possible before the game started running, but was mediated by the need to take all the features of the game state into account. There are
certainly other possible forms that this data could have taken, some of which are discussed in chapter 5.

3.3 The Runtime AI

The job of the Runtime AI is to fit into the game seamlessly, so that all the game has to do is call an Update function periodically and the bots will do something intelligent. When it is called, the Runtime AI should take into account at least the current State of the game, although a notion of what has happened in the game in the past would help too. The Runtime AI has to output some Action, which will affect the game world somehow.

The simplest approach to this problem might be to have a collection of ALL the Actions that can be taken, and pick one at any given moment depending on what the state of the game is. Perhaps this could serve as a beginner-level AI in a Turn-Based game, but there’s a reason why game AI doesn’t traditionally follow this model. Some Actions only make sense to try when paired with other Actions. This is why the HSM has become a staple of the game industry’s AI. It allows the AI to group together Actions into tactics, and to define plans and follow them until they are either successful or total failures.

By using an HSM for the AI, the system is able to maintain some information on what has happened in the past, but it doesn’t have to actually process that information multiple times. Instead, our past decisions are encoded in our current place in the HSM.

There are two ways to use a HSM – either each node can use custom code to decide which of its children to run; or the children can compete for the opportunity. I recommend the second approach because it is more general and much more scalable, allowing designers to define hundreds of Actions quickly and easily without coding logic for each one [Isla]. Plus, if every node in the HSM makes choices in the same general way, it allows the learning mechanism to have control over all of them.

3.4 The Log Interpreter

The basic function of a Logging mechanism is to take what is currently going on in a running game and keep track of it. The Logger should at least keep track of what we have
defined as the State – anything the AI designer has deemed important. The Log Interpreter, which is part of the offline processing, has two tasks:

- To take the changes in State that were logged and produce a timeline of full States. If the Logger logs the entire state periodically, then this step can be skipped or simplified, but I would rather spend more time outside the game recreating the timeline than writing tons of data into logs during the game.

- To take a sequence of States and figure out what Actions occurred to move the game from one State to another. This is not a novel problem, nor an easy one. The RoboCup competition has a “Commentator” category that challenges teams to tackle this very problem: participants build AIs that recognize what is happening in RoboCup games and translate them into understandable play-by-play accounts [Lent].

Depending on the type of game, it may be feasible to Log actual Actions in addition to State changes, in which case the role of the Interpreter is really just to parse whatever form the log data is in and put it in a form that the Evaluator is expecting. Turn-Based games, for example, can have their actions reduced to log entries easily because they are well-defined. Using chess as an example, any action by a player can be encoded in a standardized, easily parsed format using Algebraic Chess Notation.

Certain real-time games may also be able to log their Actions if the Actions are small enough and well defined. For instance, a Real-Time Strategy (RTS) game might have Actions like “Start Construction of Building Type X” or “Attack Enemy Unit A with Unit B”. Because of the wide variety of commands available to the player in the RTS, just the fact that a particular one is used gives the system a good impression of what the player is trying to do. However, this is not the case for a shooter. Most of the time the player is giving the game a continuous stream of input via the mouse and keyboard, and it would be extremely difficult for the Logger or the Interpreter to break these inputs into discrete Actions. Instead, the Interpreter takes the changes in State that have been deemed significant enough to be logged, and infers what Actions could have led to those changes.
Arguably, the most difficult task of the Interpreter is to recognize the high-level Actions of the player, which I will refer to as “Tactics”. Tactics are any Actions that combine more than one Action, so they are any node in the Behavior Tree except the leaves. Conceptually, they represent what is going on in the player’s head and what goals he is trying to accomplish when he executes a set of lower-level Actions.

One possible way to circumvent the recognition of Tactics would be to just rank the Actions at the leaves of the tree and to perform a search through the tree for the best low-level Action. However, it is quite possible that the same Action exists in multiple branches of the tree. Also, it may be more effective to not circumvent tactics: Zanetti and Rhalibi found that recognizing plans that unfold over time (such as going to pick up a certain power-up) is easier than recognizing when to use low-level Actions such as firing a shot [Zanetti].

I decided to try and recognize the high-level Tactics of the player based upon what low-level Actions were found. The Log Interpreter can really be divided into two stages: the first creates a timeline of States and Leaf Actions, and the second stage repeatedly runs through the timeline of Actions, adding in higher and higher-level Tactics where they appear to be utilized.

3.5 The Evaluator and Organizer

Together, the job of the Evaluator and the Organizer is to take all the information that has been decrypted from the Logs and produce some valuable information for the Runtime AI. This is a very general task, but once the form that this information is going to take has been decided, it becomes easier to figure out how the Evaluator and Organizer should work.

In my system the job of the Evaluator/Organizer is to assign a numerical Value for any possible Action paired with any possible State, given only the State-Action pairings that have already been experienced. In essence, these modules need to produce a function that takes in a State and an Action, and returns a measurement of how good an idea it is to use that Action.

The Evaluator, by itself, can be seen as the prototype for this function. It defines what the Value was for each State-Action pair in the timelines, based upon what States came after that Action in the timeline. That is, an Action occurs at a certain time – the Evaluator looks at what happened after that occurrence, and returns a Value that sums up how much the player’s
fortune increased or decreased. If this was all the data that was ever collected and the Runtime AI happened to run into this exact State again, the expected benefit of this particular Action would be equal to the Value that the Evaluator returned.

Of course, the system should find more than one occurrence of each Action, and so the Evaluator produces a large collection of mappings of the form: \((\text{State}, \text{Action}) \rightarrow \text{Value}\). The job of the Organizer is to take all that data and find the most accurate \(\text{State} \rightarrow \text{Value}\) mapping for each Action. In my system this means producing a vector for each Action that, when multiplied by the current State, will give a good approximation of what the Value will be.
4 Implementation

4.1 TenXion

Before explaining the decisions made in how to implement the system, including all the parts that would normally be done by an AI designer, it is necessary to explain how the TenXion game works. Without any AI, TenXion puts one player in the role of a heroic robot, and pits him or her against multiple mutants, also controlled by human players. The goal of the Hero is to escape from the level, while the goal of the mutants is simply to kill the Hero. The mutants are all much weaker and/or slower than the Hero, but whenever they are killed the player controlling them gets to re-spawn as a new mutant. Although each character has only one way of attacking, they are each quite unique, and although there are no guns to pick up, there are power-ups that can be collected and doors that need to be blown up for the Hero to advance.

TenXion is written for the Torque Game Engine (TGE) [Torque], and is made entirely from Torque Script, the TGE-specific scripting language. Torque Script is good for quick development of ideas, as the scripts can be modified and run without being compiled first. However, because Torque Script is not compiled, it is more likely to contain errors when it is run, and it cannot be executed as quickly as engine code. Therefore, logic that is considered to be stable and unlikely to be modified should be written in code rather than script.

The Runtime AI that I created has some pieces that are written in C++ and embedded within the Torque Game Engine, so that any game using the engine can use the general system. I also wrote some game-specific pieces in Torque Script that are run only by TenXion. The Interpreter, Evaluator, and Organizer are all written in C++ for efficiency, and they are combined into their own application called the AI File Generator, rather than being inserted into the TGE.
Figure 3: Top – a screenshot of TenXion, from the perspective of the Hero. Bottom – the “Mutant Select” screen, displaying the characters that the adversaries of the Hero are allowed to choose from.
4.2 Representing the State of the Game

The State of the game is represented by a collection of Boolean and Decimal variables. Each variable is given a name that it can always be accessed by, rather than being maintained in a certain order. This makes it much easier to add or remove variables from the definition of the State.

Because I am looking for correlations between these variables and the Values that they generate, all types need to be converted into numerical values. Booleans are represented as -1 if false, and +1 if true. If a State variable would naturally be an enumerated set (such as which mutant character a player is using), it is represented as a set of Booleans. In fact, unless the increase in a Decimal variable actually means an increase in something in the game (and not just a change), that variable is reconfigured as Booleans.

A lot of the basic variables (especially the Decimal ones) don’t mean anything important on their own, such as the absolute direction a player is facing. However, there are predictable combinations of these variables that can be calculated fairly quickly on the fly, and which are actually useful. For example, the global coordinates of a player probably don’t matter nearly as much as the Boolean “is this Player in sight of an Enemy”. I refer to these special variables as Calculated State Variables, since they must be re-calculated every time one of the variables they depend upon changes.

A portion of the game’s State is dependent upon the particular level that is being played. My implementation was only tested on one level, but an important addition to a system like this would be a module that breaks levels into chunks, so that the State contains a set of Booleans of the form “Hero is in Area X”. For the purposes of my prototype, I faked the existence of such a module, defining the State by hand to contain a Boolean for each room defined in the level.

The State I used consists of:

- The global locations of each player.
- The direction each player is facing.
- The character that each non-hero player currently is (as a set of Booleans).
- The room that each character is currently in (as a set of Booleans).
- The number of power-ups each player has collected.
- The amount of damage each player has taken.
- The number of Panels destroyed (destroying three panels is necessary for the Hero to escape the level).
- The Calculated State Variables, which focused mostly on the relationship between the location of the Hero and the locations of the mutants. Examples: “Mutant in same room as Hero,” “Mutant facing Hero,” and “Hero facing Mutant.”

4.3 Tree-Based AI

As mentioned briefly in section 3.3, the Runtime AI is implemented with a “Behavior Tree,” which is a generalized form of an HSM. The Behavior Tree is made up of three types of Behaviors, and each Behavior is the implementation of an abstract Action that the bot can perform. The types of Behaviors are Sequences, Selectors, and Leaves. Sequences and Selectors are nodes in the tree that have children (they are implementations of Tactics), whereas Leaves are self-contained routines that run custom script when they are called upon (they are implementations of Leaf Actions). Since Leaves are the only Behaviors that actually do anything, the possibilities of the AI are constrained by the Leaves that are defined.

All Behaviors, when they are scheduled to run, are added to the head of a list of Behaviors that are currently running. When a Behavior is put on this list, a callback to its parent Behavior is registered too. When the Behavior terminates, it returns Success, Failure, or Error, and that result is reported to the parent.

Leaf Behaviors are the basic routines for which it is more efficient to use scripted logic than to evolve a strategy. Although some past work has been able to learn low-level Behaviors, often tasks like aiming and shooting are simply easier to script [Stanley][Zanetti]. Leaf Behaviors can be as simple as telling the AI to fire, or they can be as complex as finding the best path to follow to the desired destination. Because this is custom code, I wanted it to be specific to the game and easy to modify, so the Leaf Behaviors are written in Torque Script.
The Sequences and Selectors are automatically generated by the Runtime AI. All Sequences follow the same logic – they run each of their children in order, and if any of them fail, the whole Sequence is a failure. Selectors are the opposite – they choose one child Behavior to run based upon how beneficial it would be given the current situation. This is the part of the Runtime AI that is actually affected by the learning process: I use the Action Vectors produced by the Organizer to calculate the Value associated with running each child Action at the current time. If the chosen child fails, then the Selector tries the next best child. All the Selector needs is for one child to succeed; it only fails to complete its task if all of its children fail.

The Tree’s structure is defined by the AI designer in a “Tree Structure File” (see Appendix 9.2). All he has to do to define a Behavior is to list its name, its type, and its children. If the type is “Leaf” then the system assumes the designer has coded a routine of that name in script. Otherwise, the logic is handled automatically. It may be easier for some AI Designers to follow the Goal-Method paradigm. Looking at the Behavior Tree this way, each Selector is a Goal and the Runtime AI must choose between several Methods that might accomplish it. Each Sequence is a Method, and the system must accomplish several Goals to say that the Method has been completed as planned. Using this structure, the root of the tree is a special Selector, whose Goal is very generally to “win the game.” However, whenever a child of the root Selector succeeds, the Runtime AI simply clears the list of currently running Behaviors and tests the first layer of children again.

4.4 Logging and Log Interpretation

The log entries that TenXion creates fall into three categories:

- Actions, such as firing a weapon or picking up a power-up, which are logged whenever they happen. Not all Actions can be automatically logged, however.
- Events that change the State of the game, such as a player dying or entering a room.
- Periodic updates on the locations of all players

The Log Interpreter simulates a game and maintains a model of the current State, so that the log entries do not have to reiterate what the whole state is, just what has changed. As
discussed in section 3.2, the Interpreter and the Whiteboard have the same job – to turn changes into new States. For this purpose, I made a State-Change Interpreter which takes care of all writes to the State and has a function defined for each type of State-changing Log entry it might see. A copy of the State-Change Interpreter handles all the changes to the in-game Whiteboard, so that the interpretation is consistent.

The part of the Log Interpreter that is not replicated in the Runtime code is the Action interpretation. Some Actions are easy to recognize because they have their own log entries, but the majority of Actions in TenXion are continuous movements that are not explicitly logged. There are many ways to recognize when these Actions occurred, but all of them require knowledge of the possible Actions and what kind of changes would happen in the State for each one. The Interpreter might say that, for the Action “Approach Enemy,” the distance between the player and his enemy should be decreasing. If the Interpreter sees this happening in the log, it’s a sign that the player may have been deliberately approaching the enemy (see Appendix 9.1 for an example of how log files are read).

Inevitably, there are going to be State changes for which many Actions match. Note that since we’re not trying to copy what the player did, it doesn’t matter exactly what the original intent of the player was, just whether what he did ended up being beneficial. I could have made the Interpreter try to infer the player’s intent, but if the player approached his enemy and ended up killing him, then it was a good idea even if he didn’t realize he was approaching his enemy at the time.

The Action interpretation process is as follows: as the Interpreter reads through the log and changes the simulated State, it makes note of what Actions might have caused each change and what the probability is that each of those Actions were responsible. If the log entry itself is an Action instead of a State change, then the Interpreter notes that Action with probability = 1. Since the Actions that are not explicitly logged are often continuous, it is likely that they might cause several consecutive State changes. Therefore the probability for each Action increases based upon the length of the chain of changes that it could explain.

To learn the Tactics (high-level Actions) that the player was using, the Interpreter first looks at all the sequences of probable Actions and assumes that the Tactic that explains the
largest amassed probability is correct. Then it assumes the next most popular one is also correct, unless it only explains changes that have already been explained. If the second best Tactic overlaps with the one we already locked in, that is ok, because the Interpreter can never really be sure if it is correct, and I wanted it to recognize all likely explanations of what is happening in the game. For instance, in Figure 4, the most probable conclusions the Interpreter can draw are that the player used the Tactic that combines Actions A and B at the beginning of the game, and the Tactic that combines Actions A and D at the halfway point of the game. These two together do not explain what the player was trying to do for 10% of the game, so it is quite possible that he was using the Tactic that combines Actions B and C at that point. The Interpreter makes a note of this because it explains what the player was doing at a time for which there is no better explanation. Note that it is also possible that the player was using the A-B Tactic at the end of the game, but it is much more likely that he was using the A-D Tactic. Therefore the possibility of the A-B Tactic being used at the end of the game is dismissed.

Figure 4: An example of Tactic recognition. In this case, there are only three Tactics, and each one is made up of only two leaf Actions.
4.5 State Evaluation

To have the Evaluator determine whether Actions are “good” or “bad,” a designer or programmer needs to define some conversion from multi-faceted States to numeric Values. Fortunately, it is often easier to tell whether you are doing well in a game than whether you are doing well at a task in real life. Most games have certain variables in their State that strongly indicate whether you are doing well, but they may only change once or twice per round (for instance, have you passed this checkpoint? Have you killed this adversary?). There may also be State changes (such as damaging an enemy or progressing in the correct direction) that are less important but happen more frequently, and some of them may even be somewhat subjective (such as getting closer to the enemy). The way in which the State is evaluated is the most subjective part of this system, and is ideally the last part to be tuned, after everything else is shown to work correctly.

The two dimensions along which the Evaluator can be tuned are the Values of individual State variables and how long each Action is presumed to have an effect after it was started. For instance, if a player uses a certain Action at the very beginning of the game, is the final outcome really affected? Would he have definitely lost if he did something different? Conversely, if there is no immediate, significant change in the game after an Action is performed, was the Action completely inconsequential?

This depends upon the type of game you are working with. If you are making an AI for a fighting game, the immediate damage that is dealt by an Action is probably all that is important. However, for a strategy game, moving to grab a certain position early may be important but may not pay off until much later. In trying to determine the best time-span for a shooter, I have been relatively unsuccessful, as there are Actions with immediate consequences (such as firing a gun) and ones that definitely will not have any effect until later (such as picking a certain character when you spawn).

4.6 Action Evaluation

I used several different methods of combining State Values into Values for State-Action pairs, but they all worked in the same general way. In the timeline of States that the Interpreter
produced, an Action would be found at State X. To determine the Value of that State-Action pair, the Evaluator looked at the N States following State X, and whether the Value of each State was greater or less than when the Action occurred. I decided that reactions were most likely to happen immediately, and that changes in Value that happened later were more likely to be caused by some other Action. Therefore, the influence of each subsequent State’s Value on the Action’s Value decreases as the State becomes further away from the Action in the timeline.

The function for calculating the Value of an Action at time X was generally of this form:

$$\text{Value}(\text{Action}_x) = \sum \alpha^K \left( \text{Value}(\text{State}_{x+K}) - \text{Value}(\text{State}_x) \right)$$

where $\alpha$ is a constant between zero and one, and the summation is over $K = (1 \text{ up to } N)$. Additionally, I experimented with including the Value of the final State of the game as a way of measuring the long-term consequences of Actions, but this proved to be ineffective.

### 4.7 Decision Making based upon Maximum Benefit

Given that a player performed an Action in a certain situation and it led to good or bad outcomes, how is the system supposed to generalize this information? This is where a number of different AI techniques have been tried and have succeeded, and I’m sure that each of those techniques could be used within this system as well. The decision function, whatever it is, has to be able to work for States that are completely new, and it’s definitely possible that the Nearest Neighbor, Neural Net, or Decision Tree methods would work, as long as they can be calculated quickly enough during Runtime. The work of Geisler suggests that Neural Nets do the best job of dealing with this problem, as long as training time is not a constraint. However, for simplicity’s sake, I just wanted to insert some system that would pick the best Action (measured with the Values we calculated) for a given State.

My actual implementation pushes all the calculation into the preprocessing, so that all we have to do during Runtime is find the dot product of two vectors, one representing the current game State, and the other associated with the potential Action. The Action Vector should be the one that gives the closest approximation to observed Values when multiplied by
the observed States. To be more precise, the Action Vector is a Least Squares Estimate of whatever the real function is that maps States to benefit Values for that Action.

I start by constructing a matrix \( |G| \) containing all the game States during which a particular Action was used. \( G \) is \( m \times n \), where \( m \) is the number of variables in the State, and \( n \) is the number of different States encountered. There is also a vector \( v \) of size \( n \), whose ith entry is the Value produced when the Action in question was used with the ith State in \( G \). Ideally, I would like to solve for the vector \( a \) that satisfies \( G \times a = v \), but assuming that the system has collected a large set of data, \( a \) is likely to be over-constrained. That is why I calculate the Least Squares Estimate of \( a \) (denoted as \( a' \)), which is done by solving \( G^T \times G \times a' = G^T \times v \). As a bonus, \( G^T \times G \) produces an \( m \times m \) matrix, which is independent of the number of samples the system is being trained on.

It is often the case that, unless the system has seen a vast amount of data, there will be some variables which do not affect certain Actions. For instance, maybe the player never performed the “Fire” Action when in a certain room. If the game State contains a Boolean indicating whether the player is in that room, then it will always be False. When trying to solve for \( a' \), a column full of zeros in \( G \) may make it impossible, because \( G^T \times G \) may be a singular matrix. I use an algorithm that makes use of LUP decomposition (as described in [Cormen]) to solve for \( a' \), and a singular matrix may not be decomposable in this way. It is therefore necessary to eliminate all the variables in \( G \) that never change in all the instances where the Action in question has been used. This also usually reduces the size of \( G^T \times G \) drastically, so that the matrix solving algorithm does not take as long. After the Organizer has a solution for the reduced matrix, it simply adds zeros into the Action Vector for the inconsequential variables that do not affect it. The end result is an “Action Vector File” (see Appendix 9.3).

4.8 Levels of Interaction

One of the goals that I maintained while designing this prototype was to allow for different levels of interaction with the system. At the core of the system, each module is based on general code that can be used with any game. Each module (with the exception of the Organizer) also has slots into which game-specific code, scripts, and properly formatted text
files can be inserted. These scripts and pieces of code contain some sections that should remain stable throughout the development of the game, as they represent the boundaries of the game’s universe. For instance, the definition of how a Leaf Action works should not change drastically. However, there are also sections that can be tinkered with, such as which Leaf Actions are actually going to be included in the Behavior Tree.

In my prototype, I tried to keep it clear where the lines between these levels of interaction would be drawn. The ideal split of each module is summed up in Figure 5. In a perfect world all the Game-Specific code would be written in TorqueScript, but to make the development process more efficient I ended up writing some of that logic (specifically the Action interpreter code and the Calculated State Variable definitions) in C++.
<table>
<thead>
<tr>
<th></th>
<th>Log Interpreter</th>
<th>Evaluator</th>
<th>Organizer</th>
<th>Runtime AI</th>
<th>State Whiteboard</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tinkering Level</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Which Variables are</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>included in the State</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameters for Evaluation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Behavior Tree structure</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Game-Specific Level</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leaf Action definitions</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Leaf Action interpreter code</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logging code</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Calculated State Variable definitions</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td><strong>Engine Level</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Learning Method</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Format of Organizer output</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Sequencer + Selector Logic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Tactic Recognition function</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Basic Evaluation function</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Whiteboard functionality</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: The Different Levels of the System and Dependencies of the Modules: Although all the modules are defined in code that is not specific to any game, most of them are influence heavily by the character of the game.
5 Results and Conclusions

5.1 Results Overview

With the prototype fully functional, I took on the role of an AI Designer, experimenting with different Behavior Trees, Evaluation functions, and representations of the game’s State.

The first tests involved only the Runtime AI, and artificial Action Vectors that I wrote out by hand. By limiting the State that was being used to only a few variables, I found it possible to control the Runtime AI via these Action Vectors. However, this only proved that the Runtime AI was working as expected – with a game State of any reasonable size it would be much easier to write custom code to control the AI’s decision making, rather than using the Action Vector format.

Although the other parts of the system, the Interpreter, Evaluator, and Organizer, were thoroughly tested for correctness, the Action Vectors that they produced often caused unexpected behaviors, even when trained upon fairly simple log files. A few patterns emerged that suggested shortcomings in my basic implementation. This chapter focuses on these issues, and how future work related to this project could solve them.

5.2 More Behavior Tree Restrictions can Combat a Lack of Examples

Even when using large numbers of log files, it was found that some strategies were never used in a bad situation. This was first observed when the AI began to continuously commit suicide in order to change characters. The way the State was being evaluated, not having a character (i.e. being dead) carried a strongly negative influence. Therefore, any time a player committed suicide to change characters, there was massive dive in the value of the next State (when the player was temporarily dead), and the “Change Character” Action was found to be a bad idea. However, if a player chose a character when already dead and waiting to spawn, the Action would be considered positive, as it would cause the player to spawn immediately. The problem was that in the log files that were used, no player ever committed suicide just to
change characters. The Evaluator ended up seeing nothing but positive results from the “Change Character” Action, so much so that it would use it in any situation!

The lesson is that a complete lack of negative examples can lead to the AI exploring actions that no human being ever tried. Introducing some negative examples on purpose can help, so one idea is to create a feedback loop by giving the AI some of its own games to learn from. In fact, just by including in the list of training files one of the games when the AI committed suicide over and over, I was able to stop that undesirable behavior.

However, there is a much more difficult problem to fix: some Action-State combinations do not have negative examples because we will just never, ever see them. The one that occurred most in my experiments was the AI attempting to fire a weapon when it was already dead. The reason the system will never find these Actions in the log files is that they are not possible to do – therefore they shouldn’t be possible for the AI to choose either. Simply giving the AI some of its own experience as feedback will not work here – but a solution for this sort of problem is already in place in the game industry’s traditional AI.

[Isla] discusses the use of Behavior “Tags” which can be used as a check for relevancy that precedes the normal logic used to select Actions. The point is to quickly eliminate Actions from our choice that cannot be performed in the current State. Champandard has a similar solution for his Behavior Trees, which he calls “Preconditions.” These are small bits of logic that are checked before a Behavior is executed – if the Precondition is not met, the Behavior fails immediately, saving the system from trying to execute an Action that may be impossible at the current time [ChampandardBT].

Both of these paradigms would require some extra logic to be included in the engine-level definition of non-leaf Behaviors, and it might make them significantly harder for designers to define. Ultimately though, there is not going to be any way around this – the AI designers need more sophisticated ways of constraining the decision making of the AI to ensure that it is always making reasonable choices.
5.3 A Learning Interpreter can make Action Recognition more Reliable

While trying to get the AI to focus on specific Tactics, I found it difficult to get the system to recognize the desired Actions as positive. For example, I built one Behavior Tree with an “Approach Hero” Action, and ran several games where I made sure to only damage the Hero while I was approaching him. Surprisingly, when the AI was trained on these logs, the bot ended up being just as likely to choose the “Strafe” or “Retreat” Actions as the “Approach” one. It was found that the Interpreter was finding a lot more instances of “Approach” than were intended – whenever the Mutant was coming closer to the Hero it was judged to be an “Approach,” even if there were walls or obstacles in the way and the Mutant could not possibly know where the Hero was. The majority of these cases had no significant impact on the game, and so the “Approach” Action did not turn out to be as favorable as I had planned.

Now, in this simple case, I might have been able to fix the problem with the addition of a “Hero in Line-of-Sight” variable to the State, which would be checked by the Interpreter when deciding whether an “Approach” Action was really intended. However, this sort of addition of complexity has no limit – I could be building logic into the Log Interpreter for years and never have the perfect logic to recognize exactly when a certain Action was intended.

The most important lesson that I have learned from this project is that this system really consists of two separate problems that can both be aided by machine learning techniques. One is the recognition of what tactics a player in a real-time game was using, and the other is learning which of these tactics worked best. My system has only used learning to address the optimization of tactics, and the recognition part has been put together with a system that relies heavily on custom, game-specific code. This approach is not scalable, and since the ultimate goal is to reduce the strain on AI programmers, the recognition of tactics should be automated as much as possible. Based on the work of Zanetti and Rhalibi [Zanetti] and TruSoft’s Artificial Contender [Champandard07] I believe that machine learning could be helpful in building a better Log Interpreter.

One way to automate the recognition of Actions in the Logs would be to do some supervised training of the Action Interpreter. By running through a series of games where the AI only does one specific Action or records somewhere else exactly what Actions it is choosing, we
can get feedback on what the Interpreter should be outputting. Then we can use one of many established AI techniques to get an idea of what the State changes will really look like when a certain Action is used. This would put the pattern matching in the hands of the system, rather than the programmer, and would significantly reduce the amount of custom code he would have to write.

5.4 Action Evaluation needs a new Paradigm

As described in the section 4.6, the Value that is assigned to an Action depends upon the Value of the State when it was used and a certain number of States following its use. However, this implementation turns out to be both inefficient and relatively ineffective.

First of all, the time-span that the Evaluator takes into account is limited in length, as it quickly becomes the determining factor in the speed of the learner if it is too long. With a short time-span, many Actions will see no change at all in the States immediately following their use.

Using the final State of the game as a measure of long-term consequences also proved to be ineffective – it was either completely overshadowed by the other Values, or dominated them such that any Action in a game where the mutants won became extremely desirable. This essentially decoupled the Values of Actions from the situations in which they were used, and often led to just one Action being used over and over by the AI.

I believe the only solution is to find a completely new approach to Action Evaluation. Fundamentally, not all Actions are expected to yield results in the same amount of time, and there needs to be a way for the system to account for this. Also, it may be more appropriate to measure the lifespan of an Action in terms of when certain other Actions are used, not when there are changes in the State. For instance, when a player fires a weapon we expect to know immediately whether it was an effective Action, but when he chooses a certain character to use, shouldn’t everything that happens until he chooses a different character be considered consequences of that choice?
5.5 Using Subjective States can Prevent Cheating

The way that my system is set up, a player could righteously accuse the AI of “cheating” because it has access to too much information. This is actually a problem with a lot of the game AI on the market today [Lent]. The problem is, in my experiments I used a game State that included absolute positions of all players, and every AI that was created was based upon the entire game State.

A practice shared by Benjamin Geisler and the NERO project is that they only allow the AI to view part of the game State – the part a human player would be able to “sense” [Geisler][Stanley]. This is certainly justifiable because the humans we are basing the AI on only have partial information about the game when they were playing – they would not, for instance, know exactly where the enemy was if he was on the other side of a wall. Moving forward with this project it would be a good idea to constrain each Runtime AI to a special portion of the game’s State, based upon that AI’s perspective. This might also help to limit the complexity of the Interpreter, as the recognition of Actions will have significantly less information that it can be based upon.

5.6 More Information can be gained by Preprocessing Levels

One of the challenges faced in defining the State of the game was providing useful information on the location of players. Taking the exact coordinates of players can tell the system the relative distance between them, which could be potentially useful, but it is hard to distinguish what ranges of coordinates are important because of their global position. That is, we can assume that there are going to be some critical areas of any map in any shooter that give a player an advantage when they control them. It would be great if the system could figure out where these areas are, but recognizing clusters of these points is a completely different learning problem, that I have not even begun to address. It may be easier to discretize the map before playing on it, breaking it up into chunks so that the State can say that a player either is or is not in a given area.

In building my prototype I had to pretend as if such a system existed – in reality I hand-wrote the game State and included variables such as “Hero is in the Warehouse.” Obviously this
would be impossible to do if the system was dealing with a user-created map, but TenXion already had “areas” of its map defined, and it would have been irresponsible not to take advantage of that information.

There are often critical points on a map which should be obvious from the design of the level, such as where power-ups can be found, where weapons and players spawn, or where there is a button that can be pressed. Zanetti and Rhalibi used a network of these sorts of critical points to describe the movement of players more meaningfully [Zanetti]. Hopefully, the use of this sort of information could make it easier for the Log Interpreter to determine the ultimate goal of a player’s movements around the level.

I believe that it would be feasible to build a module that processes not only the critical positions on the map, but also incorporates the division of the map into areas. I claim this is feasible because the TenXion map was already divided into “areas,” with callbacks that triggered whenever a character entered or left one of them. Additionally, there are already open-source modules written for the Torque Game Engine which pre-process maps and automatically produce networks of waypoints that can be used for path-finding.

What my system needs is similar, but it needs to be hooked in to the definition of the game State, the logging mechanism, and the Interpreter. The difficulty lies in creating yet another type of file that the Log Interpreter and Whiteboard have to read when they are initialized – they already use a “Game State File” that lists the variables included in the State. Part of the Game State File would have to be generated automatically by the map-processing module.

5.7 More Complex Relationships can be Learned with Neural Nets

Using linear regression to find the best approximation of how Actions convert States to Values is certainly limited. Even if we were to fix all the previously mentioned shortcomings of the current system, it is quite possible that it would still miss some of the truly significant relationships between multiple variables in the State. For example, maybe a winning strategy is to perform a certain Action when one (and only one) of two variables happens to be true – otherwise the player might end up in an even worse situation. In my system it is hard to tell
whether those variables should have a positive or negative impact on whether we use that particular Action – we are using a linear function to model a non-linear relationship.

To recognize more complex relationships, the system needs to assign more than just a vector to each Action in the tree. If we continue to use both continuous and discrete data, we could upgrade from a vector to a Neural Net, into which the current state is passed as an input. Alternatively, if all important information is converted into Boolean variables, we might be able to turn each Selector into a Decision Tree.

According to Geisler, Neural Nets appear to be the most effective method of decision making for real-time games [Geisler]. The results of [Stanley] and [Zanetti] indicate that they are at least effective for learning to choose routes and move effectively in combat. Geisler found that Neural Nets manage significantly lower error rates than Decision Trees or Naïve-Bayes divisions, and the only cost is more time to train. This is not a concern for a system that does all of its training between rounds, while the game is not in progress.

Along the lines of a Neural Net, but using the vector method that I have already implemented, we could experiment with higher dimension variables, such as “amount of health” multiplied by “Is in same room as enemy.” The system could automatically generate a very, very long vector of these variables by just combining each one with each other one in different ways. Of course, this would cause a tremendous increase in the amount of time it takes to solve for the best vector, but I believe the increase in the amount of time it would take to process these vectors at runtime would not be prohibitive.
6 Contributions

In the course of this thesis, I have tried to do the following, so that we might move closer to the goal of a real-time game AI that is as adaptable as a human player:

- I defined the challenges that I believe the video game industry will soon be facing in the field of artificial intelligence.

- I proposed a solution to these problems, namely that we create artificial players with the ability to learn the best human tactics.

- I laid out a design for a system that could use logs from previous games to teach bots what tactics are the best to use in any situation.

- I implemented this system specifically for a game called TenXion, and found that it was possible to influence the behavior of the bots it trained by giving it logs from different players.

- I identified numerous ways in which the system could be improved upon in the future, so that we might take the next steps towards making this a standard practice in the games industry.
## 7 Glossary

**Action**  
A general term for any behavior that is defined for an AI player. An Action may be as simple as firing a weapon or as complicated as approaching an enemy, circling around him, firing at him, and retreating to find cover.

**Action Vector**  
A vector of decimal values associated with an Action. When multiplied by a game State, it is expected to return an approximation of the change in Value that using the Action will cause.

**Behavior**  
For the purposes of this paper, Behavior is synonymous with Action. The term comes from the Behavior Tree AI infrastructure.

**Bot**  
A character in a video game that is controlled by code and is meant to be a replacement for a human player. Also referred to as “an AI” or “an artificial player” in this paper.

**Calculated State Variable**  
A variable in the game State that is not explicitly changed in the log entries. The State-Change Interpreter has the responsibility of knowing which changes in other variables could affect these special variables and updating them when those changes occur.

**Game Engine**  
A software system that allows for rapid development of video games. The game engine usually provides general tools such networking, rendering 3D graphics, and collision detection, so that the game developer does not have to build these pieces from scratch.

**Leaf Action**  
An Action that is defined explicitly by custom code or script. It has no children in the Behavior Tree.

**Log**  
A chronological sequence of changes in a game’s State, usually stored in a text file. May also contain records of Actions.

**Runtime**  
Describes any code that is run while a player is using the game.

**Scripting Language**  
A programming language that does not have to be compiled before being run, and typically does not require the programmer to worry about issues such as memory management or the types of variables.

**State**  
A collection of variables that describe a situation in a game.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tactic</td>
<td>A compound Action. For instance, the Tactic “Ambush” might be comprised of two Actions – “Hide” and “Attack.” These Actions are the children of the Tactic in the Behavior Tree.</td>
</tr>
<tr>
<td>Value</td>
<td>A number (positive or negative) indicating either how much a player would like to be in a certain State OR how much a player would like to use a certain Action when in a certain State.</td>
</tr>
<tr>
<td>Whiteboard</td>
<td>A structure in the game that holds the current State. It is updated only by the State-Change Interpreter, but can be read by any module at any time.</td>
</tr>
</tbody>
</table>
8 Bibliography


48


9 Appendices

9.1 Log Files

Below is a sample snippet of a Log File (with some lines omitted for brevity). From it the Log Interpreter will gather the following information:

- At time 423.007, both the Hero (player 1445) and a Mutant (player 1841) are in the “Warehouse” area. If the Calculated State Variable “Mutant in same area as Hero” was not already set to True, it is now.
- At time 423.035, the Mutant fires his weapon – the Interpreter assumes that the “Fire” Action was used by player 1841.
- At time 423.182, the Hero takes damage, changing the “Hero Damage” variable in the State.
- Between time 424.057 and 428.287 the Mutant has changed locations. His coordinate variables in the State change, as well as the “Distance between Mutant and Hero” Calculated State Variable.
- Since the distance between the Mutant and the Hero is increasing, the Interpreter postulates that the Mutant is using the “Retreat” Action.
- Between time 428.287 and 429.339, the distance between the Mutant and Hero continues to increase. To the Interpreter, this further increases the probability that the Mutant is using “Retreat.”
- At time 431.054, the Mutant commits suicide in order to change characters. Since he changes into “Mutant 2,” the Interpreter assumes that the Action used was “Change to Mutant Type 2.” The State changes significantly, as the Mutant takes severe damage, dies, and then respawns.

```
423.007 EVENT Player1841(PlayerBody4) TRIGGERTICK
VOLUMETRIGGER(Warehouse) LOCATION(51.4725 -134.861 7.45997)
DIRECTION(0.513823 -0.167071 -0.841471)

423.007 EVENT Player1445(Hero) TRIGGERTICK
VOLUMETRIGGER(Warehouse) LOCATION(53.8369 -135.688 7.45759)
DIRECTION(0.0594523 -0.973868 -0.219195)

423.035 ACTION Player1841(PlayerBody4) FIRE LOCATION(51.4725 -134.861 7.45997)

423.182 EVENT Player1445(Hero) TAKEDAMAGE DAMAGE(90)
TYPE(CrossbowBolt)
```
423.182 STATECHANGE Player1445(Hero) TOTALDAMAGE DAMAGE(376.775)

423.182 EVENT Player1445(Hero) TAKEDAMAGE DAMAGE(45)
   TYPE(Radius)

423.182 STATECHANGE Player1445(Hero) TOTALDAMAGE DAMAGE(421.775)

424.057 EVENT Player1841(PlayerBody4) TRIGGERTICK
   VOLUMETRIGGER(Warehouse) LOCATION(51.4725 -134.861 7.45997)
   DIRECTION(0.513823 -0.167071 -0.841471)

424.057 EVENT Player1445(Hero) TRIGGERTICK
   VOLUMETRIGGER(Warehouse) LOCATION(53.9338 -135.712 7.45804)
   DIRECTION(0.0594523 -0.973868 -0.219195)

428.287 EVENT Player1841(PlayerBody4) TRIGGERTICK
   VOLUMETRIGGER(Warehouse) LOCATION(49.4391 -136.308 7.46034)
   DIRECTION(-0.733386 -0.521891 -0.435631)

428.287 EVENT Player1445(Hero) TRIGGERTICK
   VOLUMETRIGGER(Warehouse) LOCATION(53.9338 -135.712 7.45804)
   DIRECTION(0.0594523 -0.973868 -0.219195)

429.339 EVENT Player1841(PlayerBody4) TRIGGERTICK
   VOLUMETRIGGER(Warehouse) LOCATION(46.1308 -138.662 7.46067)
   DIRECTION(-0.733386 -0.521891 -0.435631)

429.339 EVENT Player1445(Hero) TRIGGERTICK
   VOLUMETRIGGER(Warehouse) LOCATION(53.9338 -135.712 7.45804)
   DIRECTION(0.0594523 -0.973868 -0.219195)

431.054 ACTION Player1841(PlayerBody4) CHARACTERSELECT Mutant2

431.054 EVENT Player1841(PlayerBody4) TAKEDAMAGE DAMAGE(10000)
   TYPE(ChangeTeam)

431.054 STATECHANGE Player1841(PlayerBody4) TOTALDAMAGE
   DAMAGE(400)

431.054 EVENT Player1841(PlayerBody4) DEATH KILLER(Object0)
9.2 Tree Structure Files

Below is a small example Tree Structure File. The names of the different Mutant types are “Slick,” “Furioso,” and “Tiny,” and there is a specific Action for changing to each Character. Each Leaf-type Action must be implemented in Torque Script.

SampleAIFile
---
Root Selector ChangeCharacter HuntRobot GunFight Retreat
ChangeCharacter Selector ChangeToSlick ChangeToFurioso
   ChangeToTiny
GunFight Sequence CircleRobot Fire
HuntRobot Sequence ApproachRobot Fire
ApproachRobot Leaf
CircleRobot Leaf
Fire Leaf
Retreat Leaf
ChangeToSlick Leaf
ChangeToFurioso Leaf
ChangeToTiny Leaf
###

9.3 Action Vector Files

Action Vector Files are automatically generated by the offline modules. Below is an example of an Action Vector File created from a Game State File with 44 variables and the Tree Structure File specified in Appendix 9.2. Note that some variables in the game’s State did not seem to change in any of the games, as they did not affect any of the Actions and will always line up with zeros not matter which Action they are being evaluated against. Also noteworthy is that some Actions were not used in any of the log files that were seen, such as changing to the Tiny or Furioso characters. By default, these Actions are set to be completely neutral – they are neither good nor bad for the AI to use in any situation.
ApproachRobot 0 0 0 -17.9873 0 0 -0.871175 0 -9.86917 -1.69334 0
-8.54583 0 -17.9873 0 0.742003 0 -23.1634 0 -1.30784 0 -13.557
-0.027867 1.03283 -0.155883 -0.0224873 0.371795 -0.0300925 -
14.4321

ChangeCharacter 0 0 0 0 0 0 -77.397 0 -98.7264 -11.8358 0 -
72.5183 0 0 0 0 -77.397 0 -98.7264 0 -59.6554 0 -72.5183 0 0 0
0 0 0 0.0413779 27.1699 1.12391 0 11.9769 -0.329747 -0.093344
9.38655 0.534055 -0.107987 7.00423 2.0578 -160.066

ChangeToFurioso 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

ChangeToSlick 0 0 0 0 0 -42.7983 0 -55.9967 -17.0446 0 -
29.2592 0 0 0 0 -42.7983 0 -55.9967 0 -28.1073 0 -47.8495 0 0
111.365 -111.365 0 0 0.043719 20.256 0.000890967 0 0 4.78146
-0.0941274 7.95592 -0.0691103 -0.077983 2.82478
0.0261204 -70.5654

ChangeToTiny 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

CircleRobot 0 0 0 -16.4695 0 -13.0645 -0.56026 0 0 -4.23844 0
0.603816 0 -16.4695 0 -13.0645 0 -0.56026 0 0 0 0 0 7.60835 0 0
0 0 0 0 0.0074558 37.3895 0 0 0 37.3895 0.209507 -0.0401619
0.505713 0.718948 -0.0285286 1.24028 -0.0833172 -15.7402

Fire 0 0 0 -18.4927 0 -54.3071 -14.1077 -31.7934 -34.5848 -
1.38006 0 -26.1509 0 -18.4927 0 -44.2103 0 -24.2044 0 -34.5848 0
-1.38006 0 -26.1509 0 0 0 0 0 0 0.0121277 12.0509 -0.305688 0 0
8.68834 0.714541 -0.0427954 1.62667 0.480766 -0.0438769 2.37747
0.804893 -36.9167

GunFight 0 0 0 2.28411 0 -49.0096 -9.35409 -18.2214 0 -6.17827 0
-28.9065 0 2.28411 0 -49.0096 0 -9.35409 0 0 0 -1.134 0 -33.9507
0 0 0 0 0 0.0159628 28.4761 3.60421 0 0 8.53688 0.854357 -
0.0578816 1.79499 -0.750054 -0.0483289 2.4578 -0.0193824 -
33.4234
HuntRobot 0 0 0 -42.1138 0 0 16.725 0 -37.1062 0.964252 0 -19.8807 0 -42.1138 0 -27.7688 0 17.3567 0 -37.1062 0 16.5229 0 -35.4393 0 0 0 0 0 0 0.0107985 16.3705 0 0 0 4.03088 -0.421981 -0.0440471 1.25218 -0.666164 -0.0459058 2.47901 -0.130404 -36.1829

Retreat 0 0 0 -11.9392 0 -14.8094 11.9002 -10.3715 -16.1977 5.09942 0 0 0 0 -24.2174 0 18.4485 -13.8531 -16.1977 0 9.3214 0 -13.3015 0 0 0 0 0 0.00168679 7.8644 3.45018 0 0 -3.88117 -0.200723 -0.00419714 0.277277 -0.758881 -0.00415813 0.885542 -0.0373013 -10.4764

Root 0 0 0 -13.6905 0 -15.3307 9.12292 -13.5664 -16.0019 8.53875 0 0 0 0 -26.4303 0 16.764 -13.5664 -16.0019 0 15.0572 0 -16.7505 0 0 0 0 0 0.00173005 10.3979 3.16115 0 0 -8.28588 -0.116726 -0.00268451 0.245889 -0.691684 -0.00208526 1.21991 -0.0661256 -12.79

###