University of Alberta

SEMI-AUTOMATED GAMEPLAY ANALYSIS FOR ROLE-PLAYING GAMES

by

Jonathan Newton ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**

Department of Computing Science

Edmonton, Alberta
Fall 2005

Your file Votre référence
ISBN: 0-494-09246-7
Our file Notre référence
ISBN: 0-494-09246-7

Abstract

This thesis presents a tool, the Game Analyzer, which tests video games rapidly and provides useful feedback about the game's behavior to the game developer, who can then make adjustments to the game as desired. With the help of user-defined abstract states and actions, the Game Analyzer builds a state-transition model for a user-defined scenario within the game. This state-transition model defines the set of possible policies (gameplay strategies). The Game Analyzer then evaluates a large sample of the policies. These results can then be manipulated and visualized to help the game developer fine tune the scenarios played in the game and the game engine itself. This research was applied mainly to Role-Playing Games but could be applied to other domains.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Games have been around for a very long time. They are an important part of every culture and almost everyone plays some sort of game. Many games have been recreated on the computer. Computer games allow a person to play a multi-player game alone with the computer's artificial intelligence taking the place of all other players.

The computer game industry is now a multi-billion dollar industry worldwide. Computer game companies are highly competitive and are on the forefront of cutting-edge technology. It is the computer game industry that pushes computer manufacturers to develop new hardware. New games are produced with the expectation that the next generation of graphics cards and CPUs will be able to run them smoothly.

The goal of this research is to help game developers create new and exciting video games. Video games combine cutting-edge graphics, music and sounds with advanced algorithms, scripts, artificial intelligence and graphical user interface design. This means that video games are the combined effort of many different people in many different fields. Techniques used to create video games keep changing as the technology advances.

Unfortunately in this world of continuously advancing technologies some things get left behind and always stay the same. One of these things is how game developers test video games. Game testers, game programmers and game developers interact and try to improve their game. They balance the game's difficulty level and play time. If the game is too hard, people quit and they never buy the sequel. If the game is too easy, people find it boring. If the game is too short, people feel cheated out of their money. And if the game is too long, people get bored and move on. These factors affect game sales not only of the current title but of future titles from the same company. If a game company is reputed to produce bad games, people will stop buying games from this company.

To help game developers create video games this research will concentrate on helping game developers test their game in a more efficient manner. Testing video games is a challenging task since each game is different and each developer has different game analysis requirements. Since it would be difficult for a computer program to learn the style and preference of each designer, a testing suite should be versatile in order to cater to each individual designer.

The research in the thesis specifically considers testing of Role-Playing Games, a subset of computer games, as a target genre. The Game Analyzer takes the role of the human player, automatically generating the inputs the human would normally generate, and measures important outcome variables such as the time taken to complete the scenario. The result of this research, a tool called the Game Analyzer, can improve the performance of a human game tester thousands of times over. A game tester armed with the Game Analyzer can gather the results of thousands of hours of regular testing in minutes.

To speed up testing the Game Analyzer is applied to small isolated parts of a game, called scenarios. The resulting data can be used to answer a wide variety of questions about the scenario. The types of questions being answered by the Game Analyzer's data vary depending on the type of game being played. It should be noted that the emphasis of this research is placed on obtaining the data that can then be interpreted by the game developer. The tools actually used to interpret the data are not the main focus and as such only a primitive visualization tool was implemented. With this visualization tool and the data provided by the Game Analyzer, many questions can be answered, but the data itself contains the answer to many more questions.

1.1 Game Genres

The Game Analyzer was made to help test Role-Playing Games but it is general enough to be adapted to other types of games. This section describes some of the types of video games that exist today.

1.1.1 Role-Playing Games

Role-Playing Games (RPGs) are quite new when compared to games like chess and go. RPGs take their roots from table top war games and the book “The Lord of the Rings”. These two elements found themselves together in the late 1960's and early 1970's when Dungeons and Dragons was born.

With the pen and paper version of RPGs, there are several players and one Game Master. The Game Master guides his fellow players in an adventure through strange and fantastic lands. The players each control a single hero through the story laid out by the Game Master. These heroes are described on a piece of paper. A hero has a set of numbers describing his strengths and weaknesses as well as a number of hit points. Each time a hero is hit, damage is subtracted from his hit points. When his hit points reach zero, the hero dies. Actions are done simply by describing what the hero is doing. There is an infinity of actions available to the player, but a few specific actions are governed by dice. If a player wanted to swim across a lake or pick up a stone, the Game Master could simply indicate that his action was successful. But if the player wanted to strike a goblin with his axe, dice would be rolled to determine the success of such an action.

Computer RPGs, like most computer games, eliminate the need for other players. These RPGs are mostly single player games where the Game Master and the other players are replaced by the computer. This setup greatly limits the number of actions a player can choose from. Since computers cannot improvise, a scenario that was not programmed to allow a player to swim across the river would not let the player choose that action. Dice are also replaced by the computer. When a player performs an action which would normally entail the rolling of dice, the computer generates random numbers and displays these dice rolls for the player and acts upon their results.

A typical computer RPG can be divided into small scenarios. Between each scenario the player is free to do things such as heal his wounds, buy weapons and armor, or a variety of other actions. An example of such a scenario would be as follows. The hero is spending the night in a castle when it is attacked by goblins. The hero runs to the great hall and discovers two goblins trying to detach a golden statue from the wall. Once the encounter with the two goblins is done the hero will no longer be in direct peril and could spend some time doing any number of other things such as going to his room to rest. Since the hero is given much liberty as to how to proceed the scenario will be considered concluded after this encounter.

Many questions about such scenarios can be answered by the data from the Game Analyzer. It will be shown that the data provided by the Game Analyzer can answer questions like:

- How long will this scenario last?
- How much damage will the hero take?
- How much difference is there between different ways of playing the scenario?
- What happens if certain parameters of the scenario are changed?
- How good is a particular subset of policies?
- Which policies yield the best/worst results?
- Which policies yield similar results?

1.1.2 Other Genres of Games

Real-Time Strategy (RTS) Games are closer to table top war games than RPGs. The player controls a group of units in the construction of a base and the destruction of his opponents. As the name suggests this game runs in real-time as opposed to turn based games like chess. RTS games are not the main focus of my work but are included to show the diversity of the Game Analyzer.

First Person Shooter (FPS) Games are games where the player controls a single character from the "first person" point of view. The "first person" view can be described as looking through the character's eyes. In these games the player must make his way through scenarios while shooting at opponents. These games are mostly based on reflexes and hand-eye coordination but there are some games of this type that rely on stealth aspects. The stealth aspect of this type of game is tested in the experiments in this thesis.

Other genres of games exist but the Game Analyzer has not been tested on these games.

1.2 Overview of the Approach

The Game Analyzer is a semi-automated game testing device. For the Game Analyzer to test a specific game the user must link the Game Analyzer to the game itself. The user must first implement an interface layer based on the interface specifications provided by the Game Analyzer. Then additional work may be needed on the game engine itself to allow the interface layer to provide information to the Game Analyzer. Once this is completed the Game Analyzer will be able to control the game through the interface layer and extract any pertinent information. The user must then create scenarios for the Game Analyzer to analyze and define state and action abstractions to facilitate game controls for the Game Analyzer.

Once this is done the Game Analyzer will be ready to play the game. The user can then select scenarios in the game and the Game Analyzer will test them. The Game Analyzer will play each scenario repeatedly to collect data about the game. The data collected can be anything about the game the user desires. While playing the game, the Game Analyzer will try all combination of actions possible including the stupidest sequences of actions as well as the best. This will give the developers a view of every play style for their game and not just the play style of the few game testers they employ.

Using the Game Analyzer is a one person operation. A single game tester could set up the scenario in a matter of hours and then feed it to the Game Analyzer to test it. The Game Analyzer would then condense hours of testing into a few minutes. The visualization tool used in this document is only an example of what could be done using the data generated by the Game Analyzer. The data could be used in clustering algorithms, statistical calculations, prediction models and more advanced visualization tools. These visualizations and computations can then be used by the game developer to fine tune his game.

1.3 Research Contributions

- This research provides a novel approach to automated software testing.
  - This research is meant to provide output that helps the game designer decide how to tune or adjust the program.
  - The state representation used is non-Markovian and can therefore differentiate between identical states based on their past history.
- This research culminated in a generic system that can be used in a variety of computer games.
  - The research in this document is based on Role-Playing Games but the system can be adapted to other types of games.
  - The system can provide output based on any parameters that can be measured in the video game being tested.
  - The system can accumulate data on any number of parameters.
- This document provides results on realistic experiments.
  - The experiments were run on a simple game engine which implements the combat rules of the Advanced Dungeons and Dragons Role-Playing Game.
  - The scenarios presented appear in some games currently on the market.

1.4 Thesis Overview

In Chapter 2 the Game Analyzer and all of its components are explained in detail including algorithms and data structures used in the Game Analyzer. Chapter 4 explains how the Game Analyzer was tested, presents the results of applying the Game Analyzer to various scenarios and evaluates the overall functionality of the Game Analyzer. In Chapter 5 work related to this research is discussed. Finally Chapter 6 presents the limitations of the research and reviews its achievements.

Chapter 2

The Workings of the Game Analyzer

Balancing a game is a challenging task. This research do not propose to solve this problem automatically but merely to help developers with this task. The way this is currently done involves play testing the game repeatedly and tweaking certain game parameters to modify the game slightly. The approach of this research is to automate a portion of this process. Instead of having a person evaluate a scenario after playing it a few times the computer automatically plays the part of the human in the scenario thousands of times and the person can then evaluate these results.

2.1 Policies

In playing a scenario, the human player, or the Game Analyzer playing the part of the human, will be faced with numerous decisions about which action to take in various circumstances. A policy represents a specific set of action choices.

Formally, A policy is a collection of state-action pairs. A state is a set of observations about the world. In the RPG example where the hero finds the goblins the state could contain information about the health of the goblins and the hero, the location of everyone in the room as well as what weapons everyone is using. In a policy there is a single action selected for each possible state. Using this policy the computer knows exactly which action to perform in any given state.

In principle, the results obtained from the Game Analyzer should represent the set of all possible policies. The number of policies in a policy space with X states and Y actions could be as large as Y^X. Returning to the RPG example described previously assume that by keeping track of the location and health of all the participants there are 100 possible states. This is not an overly large number of states since keeping track of a single monster on a 10x10 grid would yield 100 possible states. Assuming that at each state the hero has a choice of two actions on average, this example has a possibility of 100 states and on average 2 actions available from each state. This means that there are 2^{100} possible policies, a number greater than 10^{30}. For any non-trivial scenario, the policy space will be extremely large, and therefore, in practice, it must be sampled or abstracted to be very small.

2.1.1 Incomplete Policies

To reduce the policy space incomplete policies were developed. Throughout the document any reference made to policies is actually referring to incomplete policies. These policies are incomplete since they do not contain an action for every possible state. They only contain state-action pairs for the states which are reachable. In the previous RPG example if the hero chooses to turn around and leave the room before the goblins saw him, he would never reach a state where he is fighting these goblins. Therefore any policy which opts to run away from the goblins at the start state does not need a state-action pair for any of the states involving combat.

2.2 Scenario Description

All the scenarios in this thesis use a generic Role-Playing Game (RPG) game engine. Most of the scenarios are set in a fantasy world. The following format will be used to describe all the scenarios including those in Chapter 4.

Actors:

The actors section describes who takes part in the scenario. Also described is the scripted behavior of computer controlled opponents as well as the actions available to the hero.

Setup:

The setup portion of the scenario description indicates how the scenario begins. It will also describe what causes the hero to choose a new action.

End States:

The end states portion of the scenario description lists the possible end states for the scenario.

Abstract State Variables:

Abstract state variables are binary variables that summarize key aspects of the current game state. See Section 2.4.1 for more discussion.

2.2.1 Example Scenario : The Troll and the Ogre

Actors:

- The hero can attack an adjacent monster or he can move to a monster of his choice if he is not currently adjacent to a monster.

- The troll has large claws with which he can attack the hero. The troll will always move toward the hero and attack him when possible.
- The ogre has a giant axe which deals massive damage. The ogre will always move toward the hero and attack him when possible.

Setup:

In this scenario the hero, the troll and the ogre are in an empty room. The hero is closer to the ogre but it is possible to dash by the ogre to reach the troll. Once the hero chooses an action he cannot choose a new action until a change of state occurs. The state changes when the hero becomes adjacent to a new monster or when a monster dies. This setup can be seen in Figure 2.1.[1]

End States:

1) Both the troll and the ogre are dead.

2) The hero is dead.

Abstract State Variables:

Hero is alive, Troll is alive, Ogre is alive, Troll is in range, Ogre is in range.

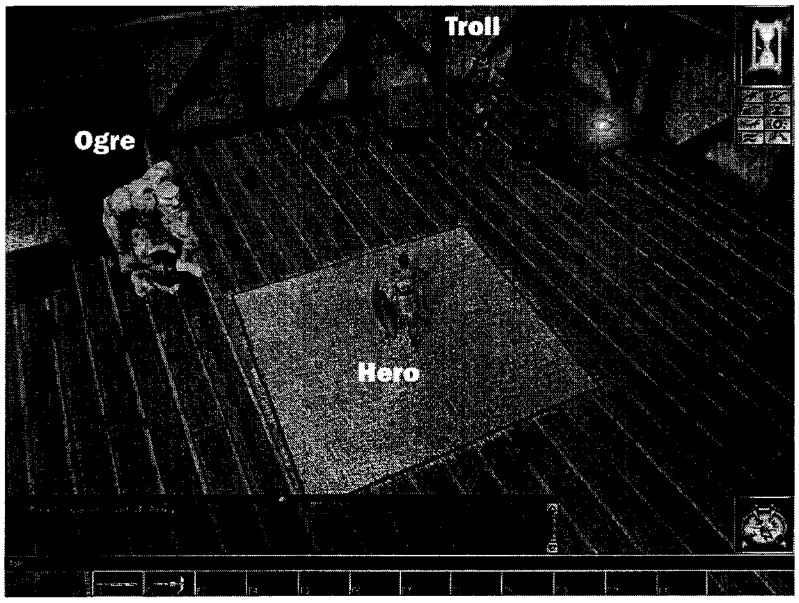


Figure 2.1: Scenario: The Troll and the Ogre

[1]The screenshots used throughout the document are created with the Aurora toolset and Neverwinter Nights ©Bioware Inc.

2.3 Game Analyzer

The result of this research is a Game Analyzer which plays out user-defined scenarios repeatedly. From a set of predefined actions the Game Analyzer discovers and evaluates many of the possible policies. The Analyzer then displays results, as specified by the user, meant to help evaluate the balance and challenge of the game. The Game Analyzer is made up of two parts, the model builder and the policy evaluator, and interacts with several other software components: the simulator, the interface layer, and the visualization tool. Each of these parts plays an essential role in the creation of the final output. The diagram presented in Figure 2.2 shows the composition of the Game Analyzer and is briefly described in this section. A detailed description of each part will be given in Chapter 3.

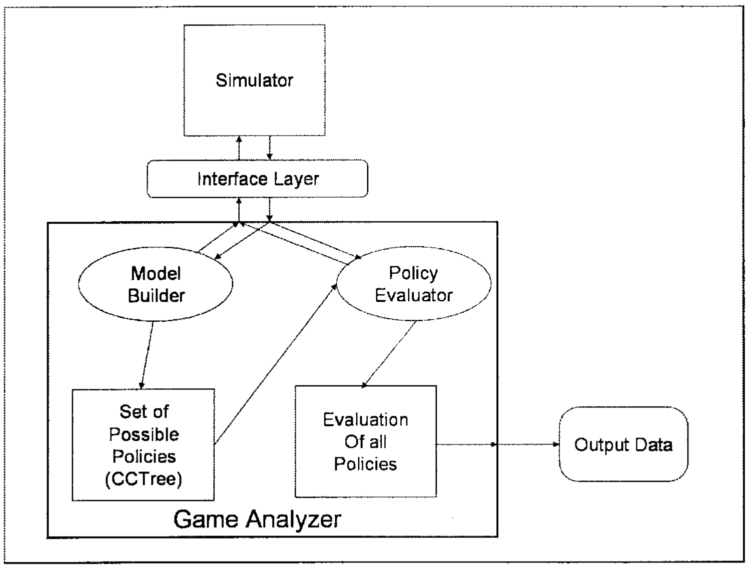


Figure 2.2: Overview of the Game Analyzer

The Simulator is the actual game engine, modified to hook up with the Interface Layer. It is provided by the game developers and used by the Game Analyzer as a black box. To speed up the analysis the graphics and time delays are disabled leaving a streamlined engine on which to run scenarios.

The Interface Layer is meant to ease communications between the rest of the Game Analyzer and the Simulator. Since no two games are exactly alike a wide variety of Simulators can be expected. The Interface Layer provides all the necessary functions to bridge the gap between the Simulator and both the Model Builder and the Policy Evaluator. It is used to convert the raw data from the simulator to a more manageable form as well as provide control over the simulations. Research on interfaces between game simulators and artificial intelligence systems is currently being done in projects such as TIELT[7]. Unlike TIELT the main focus of my work is not the interface layer.

The purpose of the Model Builder is to build the set of policies that can be evaluated for the scenario that is currently being analyzed. The Model Builder, using the Simulator, plays the scenario repeatedly and discovers the policies available to the player. This set of policies is represented by a data structure called a Choice/Chance Tree (CCTree). As the simulation progresses the Model Builder builds up the CCTree by storing each new policy in the CCTree. Once the Model Builder ceases to find new policies the CCTree is considered complete and the Model Builder hands off the CCTree to the Policy Evaluator.

The Policy Evaluator uses the set of policies obtained from the Model Builder to produce a data set to be reviewed by the user. Each policy is executed a number of times to measure the effects of the inherent randomness of most computer games. The execution of a policy could fail if a state that was not seen by the Model Builder is reached. In such a case the Game Analyzer returns to the Model Builder to expand the CCTree. Once all policies have been executed in this fashion the data gathering is complete. The resulting performance summary is stored in files. The measurements collected in the performance summary are specified by the game developer.

The output files contain a list of all the policies that have been evaluated. Each policy is associated with the data gathered for the policy during policy evaluation. These files can then be used to produce visualizations or further calculations that help the game designer assess the scenario. To demonstrate the usefulness of the data a visualization tool was created. The tool displays the average results for each policy. The pseudocode for the Game Analyzer is shown in Table 2.1.

```
Use the Model Builder to build the CCTree
While the CCTree is not fully evaluated do
    Select a Policy to evaluate
    Use the Policy Evaluator to evaluate the Policy
    If the evaluation fails
        Use the Model Builder to expand the CCTree at the failing Policy
    End If (Evaluation fails)
End While
Create the Policy List and Results Files
```

Table 2.1: Game Analyzer Algorithm

2.4 The Interface Layer

Purpose: To facilitate communications between the Game Analyzer and the Simulator.
Inputs: Abstract Actions or Control Commands
Outputs: Abstract States, Outcome Data, Scenario Complete Signal
Workings:

The Interface Layer converts data from the simulator into a form which can be used by the rest of the Game Analyzer and to control the scenarios on the simulator. It is also used to generate state abstractions, retrieve the outcome data from the simulator and determine if the scenario has ended or not. The interface layer queries and controls the simulator to achieve these goals. The interface layer must create the state abstractions from the real state in the simulator. The algorithm for the Interface Layer is provided in Table 2.2. This algorithm is applied when the Interface Layer receives an abstract action to execute. The Interface Layer can also execute control commands such as resetting the scenario to its initial state.

```
Convert abstract action into a sequence of real actions
While the abstract state does not change
    Apply one step of the sequence of real actions
    If the sequence is ended then restart from the beginning
    Accumulate performance data
    Convert the resulting real state into its abstract state
End While
Return the accumulated performance data
  and the new abstract state
```

Table 2.2: Interface Layer Algorithm

2.4.1 The Abstraction

To minimize the size of the policy space, the developer defines abstract states and actions for the Game Analyzer to use. An abstract action is a sequence of actions that can be performed in a given state. The actions are considered abstract because in most cases they will lead to a series of real actions which will be executed by the simulator to produce the desired action.To clarify abstract actions and abstract states the scenario in Section 2.2 will be used as an example.

Abstract actions are normally easy to define; they are something the hero should be able to do during a short scenario. Making the actions abstract, together with the fact that the hero can only choose a new action when the abstract state changes, eliminates many redundancies. Consider the following sequence: The hero first takes a step toward the troll and then a step toward the ogre followed by five steps toward the troll, finally reaching the troll. This sequence does not really differ from any other sequence of events that would lead the hero to the same resulting abstract state. Therefore by limiting actions to be executed by the simulator until a change of abstract state happens, micro management is eliminated. Eliminating micro management reduces the search space thus enabling more complicated scenarios to be analyzed.

Now that a set of abstract actions is defined the abstract state needs to be defined. Abstract states consist of binary variables that represent the validity of a set of statements. The abstract state differs from the real state since it contains very limited information. The real state keeps track of all sorts of details that are irrelevant to the abstract state. Details excluded from the abstract states are things like the exact locations of the actors. The abstract state keeps track of wether the actors are within each others weapons' ranges instead of their exact locations. Another example is the actual life points of the actors; the real state tracks these exactly but the abstract state considers only two values for life points, alive or dead.

In the scenario actions will be executed until a change of abstract state occurs. The state abstraction needs to be defined with this in mind. Some abstract actions might yield a change of state directly while others will last for a longer period of time. A sample action that would change the state immediately might be changing weapons. An action like attacking the troll will not usually produce a change of state immediately. The hero will fight the troll until the state changes. The state may change as a direct result of the hero's actions or it might change because of the passage of time. In the example the hero might kill the troll which would result in a change of state or the ogre might become adjacent to the hero also resulting in a change of state.

The abstract state should be represented by a binary variables that define which abstract actions can be used in the state. In the example a variable representing the statement "I am next to the troll" could be used. If the variable is true then the action "attack the troll" can be used, if the variable is false then that action is unavailable. Other statements can deny the use of some actions. For example "The troll is dead" would deny the action "attack the troll" when the variable is true. Statements that deny actions normally take precedence. Thus if the hero is next to a dead goblin, he should be denied the use of the action "attack the goblin". A third type of statement can be used to prompt a change of action. The statement "I am almost dead" could be used to help guide the hero but does not in itself allow or deny any action. When all these true or false variables are combined the state abstraction is obtained. The Interface Layer can now obtain the answers to all these statements from the simulator and provide an abstract state.

Choosing the right level of state abstraction is a very important task. If the abstraction is too fine-grained, there will be an exponential blowup in the number of policies. On the other hand if the abstraction is too coarse-grained too many details will be abstracted out and important policies could be missed. It is easiest to start off with a coarse-grained abstraction that gives the Game Analyzer access to basic actions such as moving and attacking. This abstraction includes information about the hero and the monsters. The abstraction can then be expanded to include other information such as available spells, items or interesting waypoint locations. The abstraction can also be tuned to test particular aspects of the scenario. The abstraction could be coarse-grained to provide a feel for the range of possible outcomes or it could be fine-grained to analyze a distinct style of play.

2.4.2 The Action Definition Table

The Action Definition Table is used by the Interface Layer to convert abstract actions to simulator actions. The table is created by the designer of the scenario and lists which actions can be done and in which set of states they are allowed. The table is then used by the Interface Layer to relay the set of abstract actions currently allowed.

When an Abstract Action is received to be executed the Interface Layer uses the Action Definition Table to translate the abstract action into a series of simulator commands and controls the simulator accordingly.

2.5 Using the Game Analyzer

This section describes the overall process for using the Game Analyzer is used. This discussion will cover the software needed to use the Game Analyzer and the work that needs to be done by human participants in the process. Since Game Analyzer is completely game independent it will be considered as a black box for this discussion.

The overall process for using the Game Analyzer is shown in Figure 2.3. Since the Game Analyzer is semi-automated this process includes people who perform essential tasks. A group of programmers construct the game engine including the hooks needed for the Game Analyzer. The game designer then defines a scenario and the relevant action definition table. More programmers implement the Interface Layer with the guidance of this game designer. The Game Analyzer tests this scenario and produces output files. The output files are read by a visualization tool and displayed. A game tester then analyzes the data using the viewer and reports his findings to the game designer. The game designer can then apply changes to the scenario and action definition table or instruct the programmers to modify the Interface Layer or the game engine itself.

Three things must be created for the current implementation of the Game Analyzer to

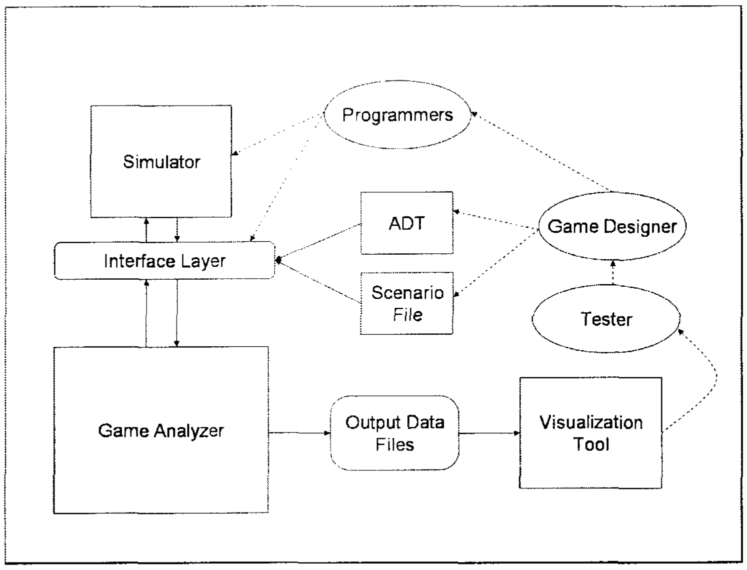


Figure 2.3: The Overall Process for Using the Game Analyzer

be used with a new game engine: the Interface Layer, the action definition table and the scenario. These will now be described in detail.

2.5.1 Defining a scenario

The scenario is created using the tools available for the particular game engine and should be in a format that is easy to load into the game engine. In Neverwinter Nights, for example, the scenario would be created using the Aurora Toolset, and would be in the format of a saved game. The scenario file for the game engine used in this document contains information about the monsters, the hero and the room. The details about the monsters include their location in the room, their hit points, what weapon they are using and their skill with that weapon. The details about the hero are almost indentical to that of the monsters, the only difference being that the hero might have more than one weapon. The details about the room include the size of the room, the location of walls and the location of any points of interest.

The scenario file for "The Troll and the Ogre" consists of the actor definitions shown in Table 2.3 and the map shown in Table 2.4.

2.5.2 Creating the Action Definition Table

The Action Definition Table contains information indicating the conditions that must be true for an action to be permitted in a given abstract state. In "The Troll and the Ogre" a state has 5 binary variables; The hero is alive (HA), the troll is alive (TA), the ogre is alive (OA),

Name	HP	AC	TH	BD	RD	WS	WR	X	Y
Hero	140	13	5	0	10	6	1	2	4
Troll	20	10	2	1	5	8	1	5	1
Ogre	45	8	2	4	6	3	1	1	1

Table 2.3: Actor Definitions for "The Troll and the Ogre". Columns represent the following information: HP = hit points, AC = armor class, TH = bonus to hit, BD = base damage, RD = random damage, WS = weapon speed, WR = weapon range, X and Y = starting location X and Y coordinates.

X/Y	0	1	2	3	4	5	6
0	*	*	*	*	*	*	*
1	*						*
2	*						*
3	*						*
4	*	*		*	*	*	*

Table 2.4: Map for "The Troll and the Ogre". * represents a wall at that location.

the troll is in range (TR), the ogre is in range (OR). These binary variables can be seen in Table 2.5 where 1 means this variable must be true for this action to be allowed, 0 means that the variable must be false for the action to be allowed, and * means that this variable can be either true or false. The possible actions available to the hero are moving towards the troll (MT), moving towards the ogre (MO), attacking the troll (AT) and attacking the ogre (AO). The "MT" row in Table 2.5 indicates that the action of moving towards the troll should be allowed if the hero is alive, the troll is alive, and the hero is not in range of the troll. There is one special action, the end game (EG) action. This action is used to specify the conditions under which the scenario is considered finished. In this scenario there are two end game conditions, the last two rows in Table 2.5. It is possible for any action to have multiple conditions. If any one of the multiple conditions for an action is true then that action is allowed.

2.5.3 Adding Hooks to the Game Engine

For the Interface Layer to be able to communicate with the game engine hooks must be added to the game engine. These hooks enable the Interface Layer to control the game engine and collect information from the game engine. Since the Interface Layer is game dependent, the details of the hooks vary from game to game. The hooks needed for the current implementation of the Interface Layer are the following:

Action	HA	TA	OA	TR	OR
MT	1	1	*	0	*
MO	1	*	1	*	0
AT	1	1	*	1	*
AO	1	*	1	*	1
EG	0	*	*	*	*
EG	1	0	0	*	*

Table 2.5: Action Definition Table

LoadSavedGame(String SavedGame) The game engine finds the saved game file named SavedGame and loads it into memory.
Pause() This is used to pause the game so that the simulation does not continue while the Interface Layer or Game Analyzer are executing.
GetCurrentState() Returns a vector containing all the state variables needed to calculate the abstract state.
GetOutcome() Returns a vector containing all the game engine variables the game developer wants to evaluate.
Attack(Object Monster) Directs the hero to strike a blow with the currently equipped weapon at the target Monster.
MoveTo(int x, int y) Directs the hero to move towards location (x,y) on the map.
Equip(Object Weapon) Directs the hero to equip this particular Weapon.

2.5.4 Implementing the Interface Layer

The Interface Layer is meant to facilitate communication between the game engine and the Game Analyzer. The Interface Layer must implement specific methods that are called by the Game Analyzer and route these method calls to the underlying game engine. The methods used in the current Interface Layer implementation are the following:

init(String ADT, String Scenario) This method initializes the Interface Layer and loads the Action Definition Table into memory and records the Scenario file name. The specific file names are passed in as command line arguments when invoking the Game Analyzer.
resetGame() This method calls the LoadSavedGame hook in the game engine with the scenario file name to load the scenario file.

getState()
This method obtains a vector of state variables from the GetCurrentState hook and converts this information into an abstract state. This method then returns the current abstract state. Abstract states are represented as integers in the Game Analyzer.

getStartState()
This method returns the abstract start state for the scenario. The start state is computed when the scenario is loaded for the first time.

getActionDefTable()
This method returns the action definition table for use by the Game Analyzer.

isEndState(int someState)
The method returns true if someState is an end state. The Interface Layer compares the abstract state defined by someState to the end game conditions defined in the Action Definition Table to determine if it is an end state or not.

isGameOver()
This method calls isEndState passing in the result of getState() as a parameter. The result indicates if the current state of the game engine is an end state.

executeAction(int actionNumber)
This method converts the abstract action represented by actionNumber to an action sequence. This sequence is then executed repeatedly until a change of state occurs. The Game Analyzer assigns a unique integer to each abstract action. Each abstract action has a predefined action sequence hardcoded into the Interface Layer using the Attack, MoveTo and Equip hooks.

getNumStates()
This returns the number of possible states, typically $2^{StateLength}$.

getOutcomeVariables()
This method returns the vector of outcome variables for the current execution of the scenario obtained using the GetOutcome hook. These variables will be used to produce the final output of the Game Analyzer.

getPrintState(int state)
This method returns a descriptive text identifying the current state. The descriptive text is used to facilitate analysis of the policies. The text is a concatenation of the descriptors for the state variables in Table 2.5.

getPrintAction(int state, int action)
This method returns a descriptive text identifying the action. The descriptive text is used to facilitate analysis of the policies. The text is the action descriptors in Table 2.5.

Chapter 3

Inside the Game Analyzer

This chapter gives a detailed description of the Game Analyzer's parts and their algorithms.

3.1 The Model Builder

Purpose: To create a model of the policy space to obtain the set of possible policies.
Inputs: Abstract States, Outcome Data, Scenario Complete Signal
Output: Choice/Chance Tree (Representing the Policy Space)
Workings:

Shown below is the generalized pseudo-code of the Model Builder. The Choice/Chance Tree (CCTree) is a data structure used to store the policy space. A state transition is an "Abstract State $\longrightarrow$ Abstract Action $\longrightarrow$ New Abstract State" triplet. The CCTree is considered incomplete if the state transitions have not been explored fully. The CCTree will be explained in more detail later.

```
While the Choice/Chance Tree is incomplete
  Get the Interface Layer to reset the scenario
  While the Scenario Complete Signal is not received
    Choose next action
    Get the Interface Layer to execute the action
    Add Abstract State obtained from Interface Layer to CCTree
  End While
End While
```

Table 3.1: Model Builder Algorithm

The Model Builder does two tasks. First it will build the CCTree used to evaluate the scenario and second it will expand that tree as needed by the Policy Evaluator. When the Model Builder first constructs the CCTree it tries to expand it as much as possible. The Model Builder will continue to expand the CCTree until it has reached every leaf node in

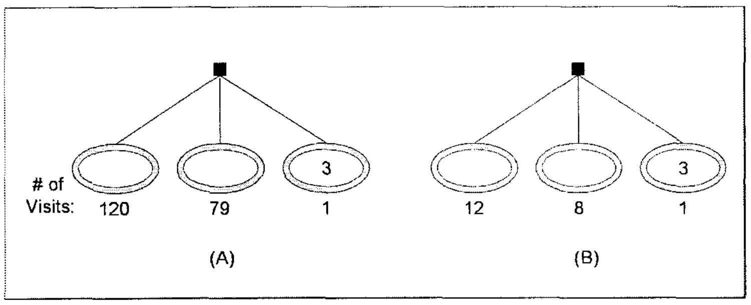


Figure 3.1: Extremely Unlikely Event

the tree a minimum number of times (the default number of times is 3). The actual number of times it will visit each leaf node can be set by the developer. It is possible for the Model Builder to ignore some leaf nodes that do not meet the required number of visits if these leaf nodes are on a branch that is deemed extremely unlikely (by default the threshold is set to 1%). For example, in both parts of Figure 3.1 all the ovals are leaf nodes, children of the black square node. The number below a node indicates how many times it has been visited. Leaf node 3 has been visited once in both diagrams. In diagram (B) leaf node 3 represents aproximately 5% of the visits and is a viable node. In diagram (A) leaf node 3 accounts for less than 1% of the visits, is below the threshold and is labeled as an extremely unlikely event. The threshold at which a branch is deemed extremely unlikely can be adjusted by the developer. Once the Model Builder considers the CCTree complete, it sends it off to the Policy Evaluator. While evaluating the CCTree the Policy Evaluator might discover new situations in the scenario. If the new situation is important enough, the Policy Evaluator will instruct the Model Builder to expand the tree (see Section 3.3.6).

The task of building the CCTree will be explained here. The second task of expanding the CCTree at the request of the Policy Evaluator will be explained after the Policy Evaluator.

3.2 The Choice/Chance Tree

The Choice/Chance Tree (CCTree) stores the abstract states encountered while executing the scenario. The series of abstract states accumulated and the transitions between these states become the policy space.

The CCTree is a tree data structure with special characteristics. The odd depth nodes are called choice nodes since the game analyzer gets to choose a particular action at these nodes. The even depth nodes are called chance nodes. The action selected at the choice node is executed by the game engine. The chance nodes represent this execution and the resulting state of this execution is added to the tree. From this resulting state, the game analyzer decides upon another action. Because the scenarios are typically probabilistic there are multiple possible resulting states. These multiple resulting states are all children of the same chance node. Figure 3.2 is the tree built by the Model Builder for the sample scenario "The Troll and the Ogre". The ovals represent choice nodes and are labeled with their state. The small squares are chance nodes. Notice that some chance nodes have multiple children representing the multiple possible resulting abstract states. The figure will be discussed in more detail in the next section.

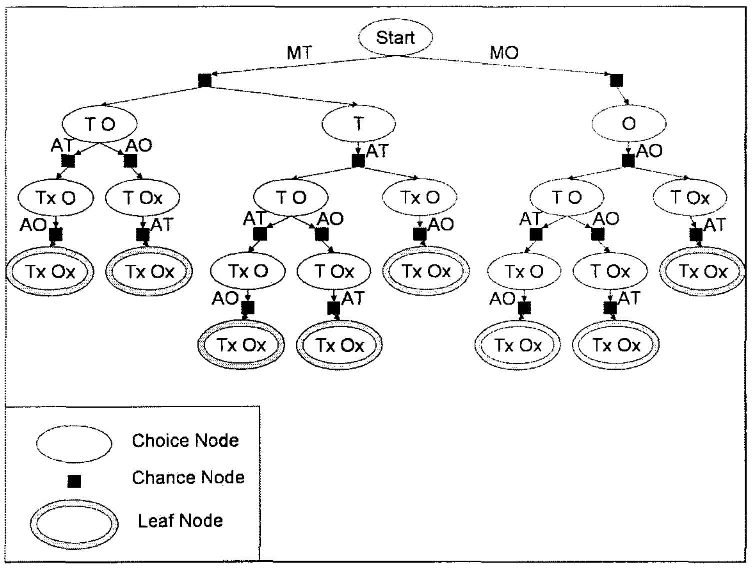


Figure 3.2: An Example CCTree

Each choice node is an abstract state in which certain possible actions can be done. The abstract state itself represents a wide range of real states. The abstract state generated from the real state contains all the information needed for the CCTree and the node created from the abstract state becomes a permanent part of the CCTree.

The chance nodes on the other hand do not store any information from the simulation. They are simply a connection between an action taken and all the states resulting from taking this action. The chance nodes are meant to represent the randomness that can occur from taking an action. With the existence of chance nodes the action "move to the troll" can be executed from a choice node and the result is a single chance node. The chance event then happens and the chance node selects from its children the correct choice node resulting from this chance event.

Since the CCTree stores only abstract states and not actual states the scenario cannot start from a random node, it must start from the initial state. Each path through the CCTree is the result of one or more executions of the scenario from start to finish.

3.2.1 A CCTree Building Example

The sample scenario "The Troll and the Ogre" will be used to demonstrate how to build a CCTree.

To build a CCTree the Model Builder starts with an empty tree. The root node is generated using the abstract state of the start position for the scenario. For the examples that follow the abstract state stored in the choice nodes will be represented by the following symbols:

T will indicate that the hero is adjacent to the Troll

O will indicate that the hero is adjacent to the Ogre

Tx will indicate that the Troll is dead

Ox will indicate that the Ogre is dead

At the start of the scenario neither of the monsters are dead or adjacent to the hero. Therefore the start state shall simply be labeled "Start". The Model Builder then consults the Action Definition Table in the Interface Layer to obtain a list of possible actions from this start state. The edges leading out from choice nodes will be labeled with the abstract action taken. In the scenario the hero has a choice of four actions:

MT will indicate the action "Move to the troll"

MO will indicate the action "Move to the ogre"

AT will indicate the action "Attack the troll"

OT will indicate the action "Attack the ogre"

The hero's actions are limited by not allowing an attack to be attempted if the monster is not in range and not allowing him to move if a monster is in range.

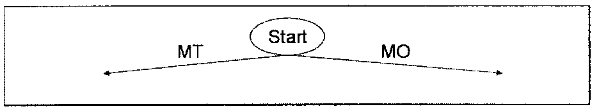


Figure 3.3: The Starting Node

To start the scenario the simulator is reset to the starting conditions shown in Figure 3.3 and the Model Builder selects an action. In the example the Model Builder will opt to move the hero toward the troll.

The Model Builder creates and adds a chance node as a child to the root node. This chance node is now attached to the action "Move toward troll" in Figure 3.4 and each time this action is taken from the start state this node will be reached.

Once the chance node is built the simulator is commanded to move the hero toward the

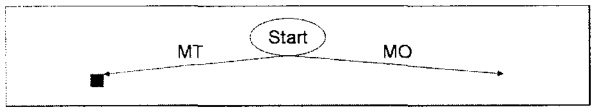


Figure 3.4: Adding a Chance Node

troll. Meanwhile the troll and ogre follow their own logic and move toward the hero. The hero and monsters will continue to move toward each other until the Interface Layer signals a change in abstract state. At this time only three things can cause a change in abstract state: the hero comes in range of the troll (toward whom he is moving), the hero comes in range of the ogre, or the hero comes in range of both simultaneously. In the example it is assumed that the hero comes in range of both the troll and the ogre simultaneously.

Once the Interface Layer detects a change in abstract state the simulation is paused. The Model Builder proceeds to generate a choice node from the abstract state supplied by the Interface Layer. The new choice node is added as a child to the chance node in Figure 3.5 and the cycle begins anew.

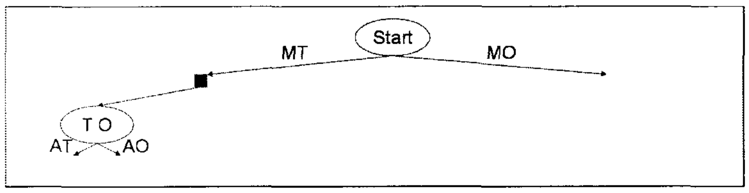


Figure 3.5: Adding a Choice Node

The Model Builder consults the action list, picks one, adds a chance node and the simulation takes over until a change in abstract state is detected. This process is repeated until an end of scenario abstract state is reached. In the example it is assumed that the following events occur. The hero moves toward the troll and encounters both monsters simultaneously. The hero attacks the troll and continues until the troll dies. The hero now opts to kill the ogre and succeeds. An end of scenario state has been reached, the hero has conquered the monsters and can make his way to the exit. The resulting branch is shown in Figure 3.6.

After reaching an end of scenario state, the Model Builder creates a choice node from the end of scenario abstract state and adds it to the chance node. This choice node has no actions listed since it is an end of scenario node and is therefore considered a leaf node.

The Model Builder then returns to the root node and the simulator is reset to the start state. The scenario restarts again with the Model Builder making different choices until it has taken all the actions available at the root node once. The tree in Figure 3.7 shows one complete path for each action available at the start state.

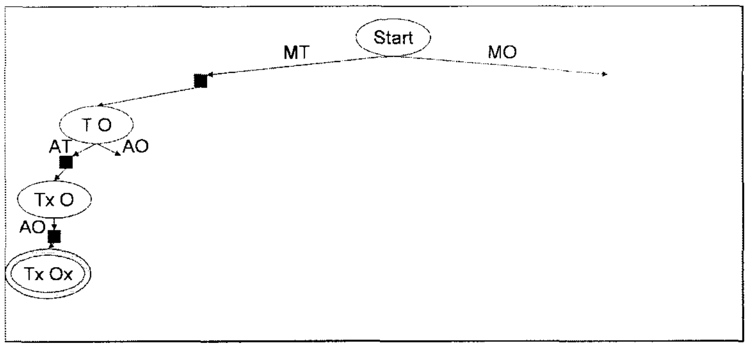


Figure 3.6: Completing the Branch

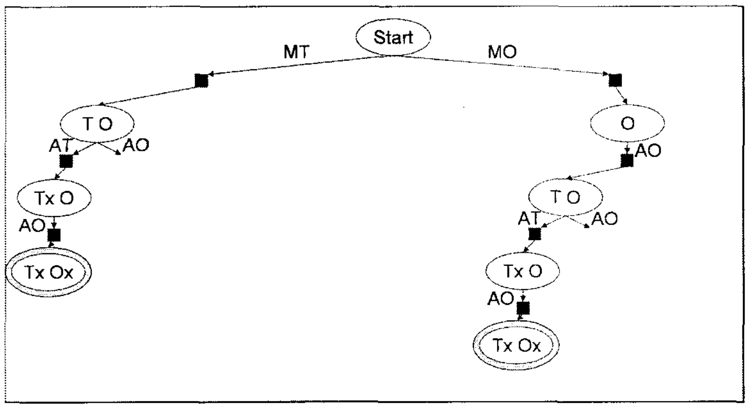


Figure 3.7: Completing a Second Branch

So far each chance node in the tree has only been visited once, but now to fill out the deeper levels of the tree some actions have to be repeated. Looking at the example tree, it can be seen that when the hero moves toward the troll, the left branch is followed and the left chance node is selected. Currently the only resulting state of this chance node brings the hero in contact with both monsters. Once the hero is in this situation he can do two actions; kill the troll or kill the ogre. The action "kill the troll" has already been explored at this point but the action "kill the ogre" has not. To explore this action the simulator must be reset and start from the root node. This is because the real state of the world was not saved at this node. To explore the action the same abstract state must be reached. From the root node the hero must move toward the troll to fill out the unexplored actions below. The simulator starts moving the hero toward the troll but it is possible for the resulting state to be different this time.

Assuming that instead of reaching both monsters the hero instead simply meets the troll. The Model Builder takes this new state and generates a choice node. The new choice node is then added to the chance node from which it resulted as seen in Figure 3.8.

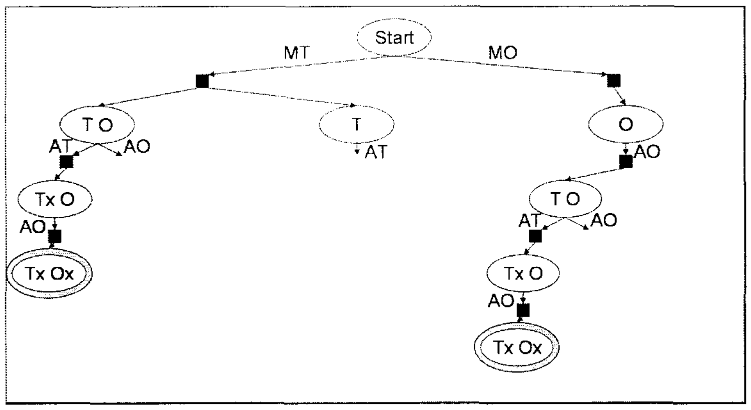


Figure 3.8: New Result from a Chance Node

The Model Builder continues to run simulations. For each new simulation the Model Builder systematically makes different choices at the choice nodes. The process is repeated until each leaf node is visited X times (the default is 3). To guide the choices made by the Model Builder each choice node records if all leaf nodes below it have been visited enough for each of the available actions. When all actions from a choice node have been visited enough to satisfy all leaf nodes below it, the choice node informs its parent that it is satisfied. When all choice nodes below a chance node are satisfied the chance node informs its parent choice node that it is satisfied. Once the root node is satisfied, each leaf node has been visited X times and the CCTree is deemed complete. In the example the completed CCTree is shown in Figure 3.9.

Unlikely events and cycles in the CCTree could cause the Model Builder to run countless trials to obtain a complete tree. An unlikely event is an event that occurs at a chance node less than $X\%$ of the time (the default is 1%) and is ignored when determining if the tree is complete. A cycle is a sequence of states that keep being repeated. The shortest cycle would be to take action 1 in state A which leads to state B and to take action 3 in state B which leads to state A. The Model Builder would be stuck in such a loop until an exterior force applies a change of state such as a monster arriving next to the hero. The Model Builder aborts cycles and considers them complete since they have an action for all states encountered. The Policy Evaluator will deal with cycles in Section 3.3. Even with these time-reducing techniques the Model Builder could take an excessive amount of time to build the model if the abstraction is too fine-grained. The abstraction must be modified to reduce the state space in this case.

Notice that there are duplicate abstract states in the CCTree. For example state TO occurs three times. These states are not merged because the underlying real states depend on the entire history of the scenario and not only on the current state (See Section 5.2).

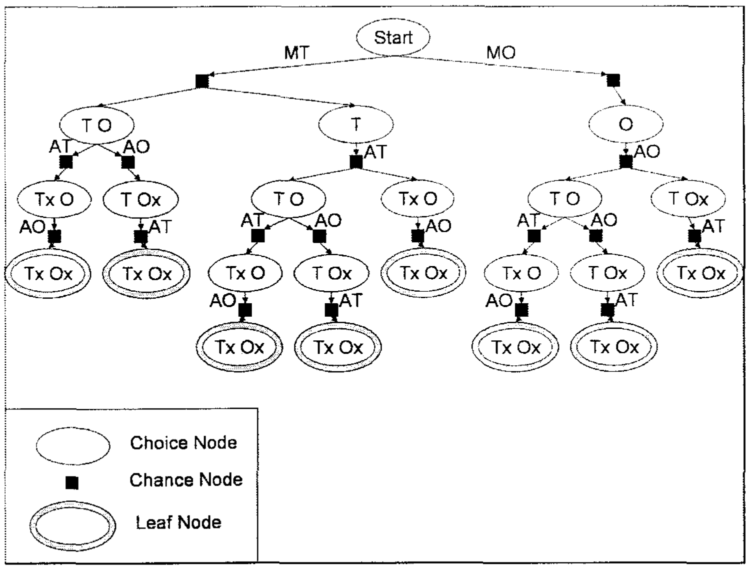


Figure 3.9: The Complete CCTree

3.3 The Policy Evaluator

Purpose: The Policy Evaluator evaluates all policies for the given scenario and obtains the outcome data for each policy.

Inputs: CCTree

Outputs: Policy List, Policy Results

Workings:

The Policy Evaluator has two tasks. The first task is to extract all, or a sample of, the policies represented by the CCTree. The first task is explained in detail in Section 3.3.1. Once the policies are extracted the second task of the Policy Evaluator is to evaluate each policy in turn. To evaluate a policy the Policy Evaluator executes the policy enough times to generate a significant number of trial samples. The results of each trial is stored in an array associated with the policy. The pseudo-code for executing a policy is in Table 3.2. Note that policies, including those with cycles, will be executed until a predetermined time limit is reached. This can result in a state that has never been seen before since the Model Builder aborted the cycle early. A cycle that results in a new state being encountered will invoke the Model Builder and this cycle will no longer be aborted by the Model Builder. Once both tasks are complete the Policy Evaluator returns the array of data accumulated for each policy to the Game Analyzer.

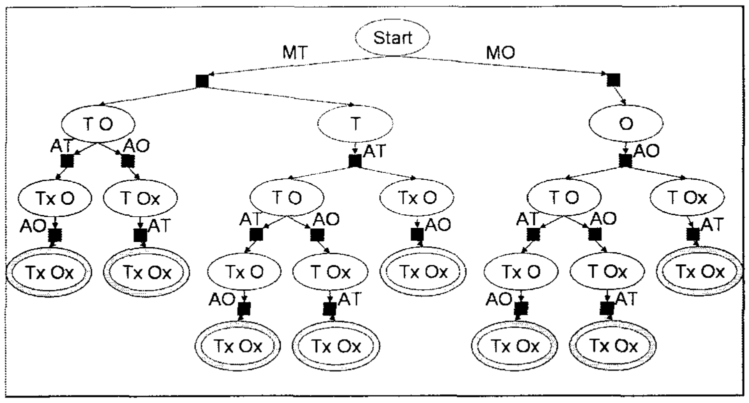
```
Get the Interface Layer to reset the scenario
While the scenario is not over and the time limit has not been reached
    Select the action for the current state according to the current policy
    If there is no action selected
        (this is a state never seen before for this policy)
        If the node is above threshold for unlikely events
            Abort the evaluation and invoke the Model Builder
        Else
            Pick the first available action
    End If (there is no action)
    Get the Interface Layer to execute the action
End While (the scenario is not over)
Store the results of the execution in the array
Return execution results
```

Table 3.2: Policy Execution Algorithm

3.3.1 Policy Extraction

To extract policies the Policy Evaluator uses leaf policies and partial policies. Policies, leaf policies and partial policies are defined here. The algorithm for policy extraction can be found in Section 3.3.2.

A policy defines a single action for every given state. Note that a policy which defines an action for all states is unnecessary. It may be the case that a certain state can never be reached with a given policy. Consider the example used previously in Figure 3.9 reprinted here for convenience.



An example of a complete policy is seen in Table 3.3.

Taking the example above, the policy at the start state is to move toward the ogre which leads down the right branch, the state "T" where the hero is adjacent only to the troll will

In State	Action Identifier	Abstract Action
Start	MO	Move to the ogre
O	AO	Attack the ogre
T O	AT	Attack the troll
T Ox	AT	Attack the troll
Tx O	AO	Attack the ogre

Table 3.3: Complete Policy

never be encountered. This means that all policies that take the action "move toward ogre" at the start state will never need to specify an action for the state "T".

A policy will therefore be considered complete if it defines an action for every state that can be reached when following the policy. In the example above the policy is complete even if no action is defined for the state "T" since the action defined for the start state is "move toward ogre". Since policies define actions for specific states they control which action will be taken at a choice node of the CCTree. All other branches of this choice node can be ignored when creating the policy. Chance nodes behave differently. All branches from a chance node must be considered when building the policy since the branch that will be followed during any given simulation cannot be predicted. The policy must have an action for any resulting state from the chance node.

Leaf Policies

When a path is traced from the start node to a leaf node many choice nodes are crossed. The combination of all actions taken to reach that particular leaf node is considered a leaf policy. In the previous example if the left action was always taken, the resulting leaf policy is shown in Table 3.4.

In State	Action Identifier	Abstract Action
Start	MT	Move to the troll
T O	AT	Attack the troll
Tx O	AO	Attack the ogre

Table 3.4: Leaf Policy

This leaf policy leads to the leftmost leaf node assuming that the chance nodes cooperate. Therefore this is considered the leaf policy of the leftmost leaf node.

Partial Policies

A partial policy is a policy that does not have actions for all reachable states. For a choice or chance node a partial policy is a policy which is complete for states between this node and all leaf nodes under this node. In addition the partial policy, like a leaf policy, contains all actions taken to reach this node. During policy extraction (Section 3.3.2) the partial policies get combined gradually until they become complete policies at the start node.

3.3.2 Policy Extraction Algorithm

The first task done by the Policy Evaluator is extracting policies. As seen previously, a policy does not contain an action for every state, instead it contains an action for each reachable state. Since this is the case, it is impossible to enumerate all policies or generate policies at random without exploring the CCtree.

To extract the policies from the CCTree, start by first assigning leaf policies to all the leaf nodes. Once each leaf node has a leaf policy recursively back them up through the levels of the tree. At a choice node the partial policies are obtained from each child and put together in a list. At a chance node the partial policies from each child are stored in a separate list. These lists are then combined to produce the list of partial policies for the chance node. Since the outcome of a chance node is unknown, each policy for the chance node must contain a partial policy for each branch of the chance node. This is accomplished by combining one partial policy of each branch together to form a single partial policy. This process is illustrated in Section 3.3.3. Repeat this process for all possible combinations of partial policies. Once this process is complete, a tentative list of partial policies is obtained. This list must be examined and any duplicates removed. With the duplicates removed the final list of partial policies is complete for this chance node. The partial policies will continue to be backed up recursively.

3.3.3 An Example of Policy Extraction

To illustrate policy extraction a small portion of a hypothetical CCTree will be used. Each partial policy will be represented as an array containing actions and indexed by state. For example the partial policy in Figure 3.10 means that action A will be taken in state 1, action C will be taken in state 3, action B will be taken in state 4 and state 2 and 5 have no action assigned in this partial policy.

Figure 3.11 shows partial policies obtained at a certain stage of policy extraction. This will be the starting point for this example. The choice node labeled with the number one must now combine the partial policies from its children. The partial policies the children passed up must be different because a different choice was made for each branch. In this

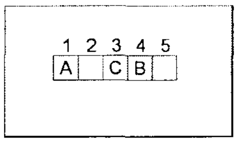


Figure 3.10: Sample Policy

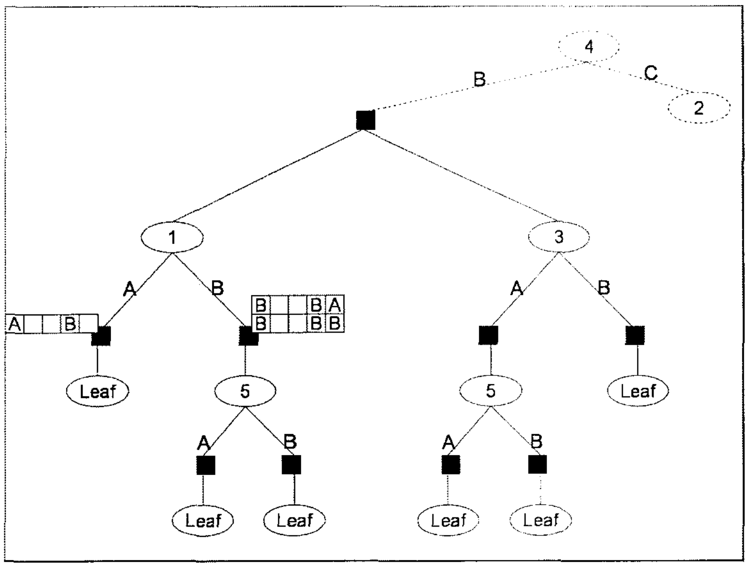


Figure 3.11: Partial Policies

case the choice node was selecting an action for state one since the choice node was labeled by its state, one. To combine the partial policies at a choice node they are simply added to a list as seen in Figure 3.12 essentially creating the union of the two lists.

At this stage the choice node would pass this list up to its chance node parent. The chance node would also obtain the lists from its other children. The resulting lists are shown in Figure 3.13.

At a chance node the sets of policies from the children are combined by taking their cross product. This is necessary because each policy of the chance node must specify actions for every reachable state under all its children. Start by taking the first partial policy of the first child (state 1), in this case A••B• and combine it with the first partial policy of the second child (state 3) ••ABA. To combine two partial policies compare each action taken in each state, if they match then continue, if one is a blank then take the action from the other, but if they don't match then discard the entire combined policy. In the example A••B• combined with ••ABA gives A•ABA. Once the two partial policies are combined the resulting partial policy would be combined with the first partial policy of the next child

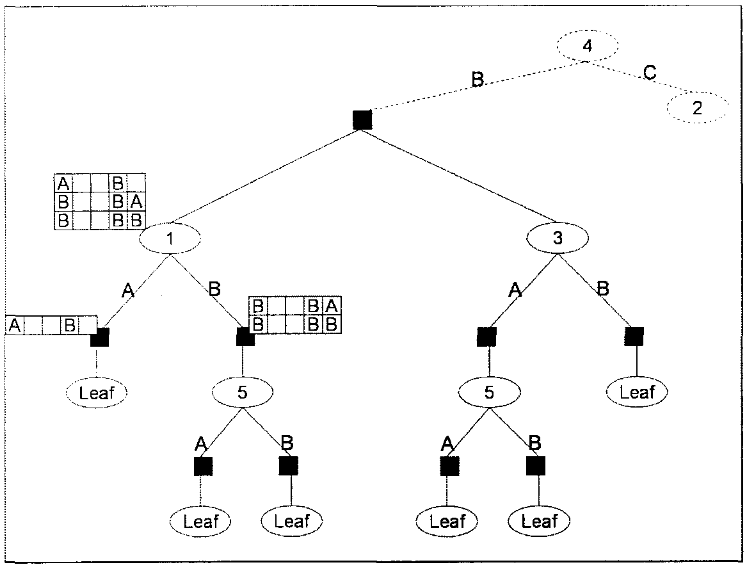


Figure 3.12: Choice Node Policy List

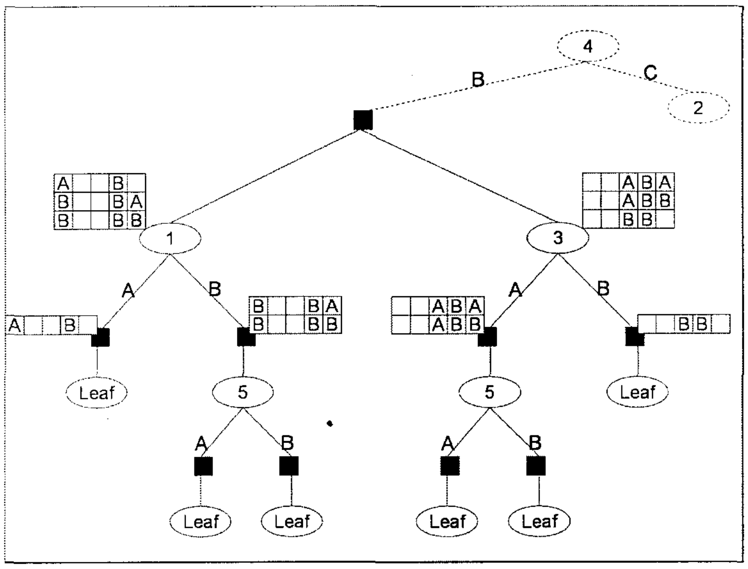


Figure 3.13: All Choice Node Lists

and this process would continue until the partial policy is combined with a partial policy from each child. In the example there are no more children and the resulting partial policy is A•ABA. This will be the first partial policy in the set of possible partial policies for this chance node. The next partial policy of the first child is then combined with the first partial policy of the second. In the example B••BA is combined with ••ABA. The resulting partial policy is B•ABA. The partial policies are combined in this fashion trying all possible combinations. In the example when the third partial policy of the first child is combined with the first partial policy of the second child a problem arises. The two partial policies are B••BB and ••ABA. These partial policies disagree on the last state. The first partial policy selects action B where the second partial policy selects action A. This conflict cannot be resolved and the policy is discarded. Once all the partial policies are combined and those that do not combine are discarded the list shown in Figure 3.14 is obtained.

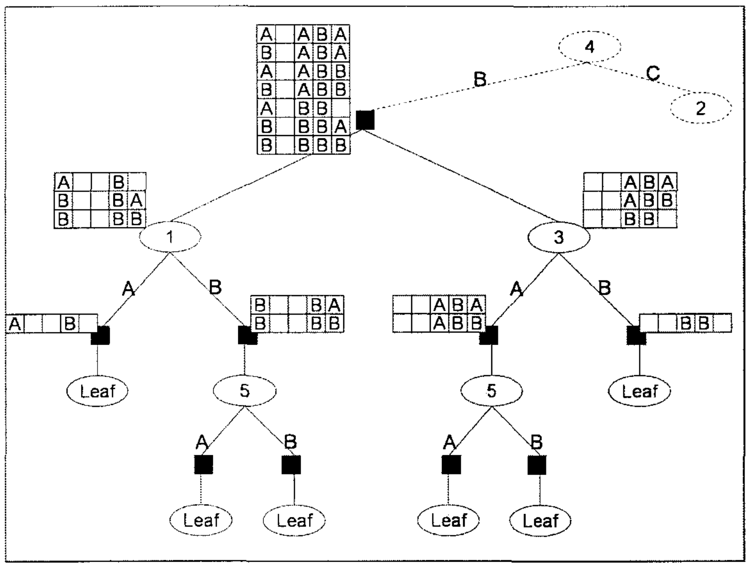


Figure 3.14: Chance Node Combined Policies

Once the partial policies are backed up all the way to the start node a complete list of policies for the tree is obtained. This list can then be used to evaluate the scenario.

3.3.4 Sampling

The policy space defined by the CCTree can quickly become very large, too large to extract in its entirety. Due to limited resources the computer may not be able to hold the set of policies explicitly in memory at one time. To evaluate policies from such large policy spaces a sample of the policies is taken. Since the Game Analyzer holds no prejudice on how to play a given game each policy in the space is equally likely. Therefore sampling from a uniform distribution over the policy space would be the appropriate thing to do.

Round Robin Sampling

This is the sampling method currently used in the Game Analyzer. It is a non-random sampling method that simulates sampling over a uniform distribution.

The sampler starts from the start state with a blank policy. At a choice node (like the start state) the sampler selects an action, fills in the current policy with the action selected at the current state and recursively follows down that branch of the CCTree. The sampler selects an action by looking through the list of possible actions for the current state in the order they are listed. When considering an action the sampler calculates the percentage of times it already sampled this action compared to the number of times it has sampled any action in this state. From the information stored in the node, the sampler retrieves the estimated percentage of policies that take this action at this state. The calculation of this estimate is discussed in Appendix A. Once the sampler has calculated these numbers it compares them. If the percentage of samples is smaller than the percentage of policies then this action is under-sampled and should be sampled again; otherwise the sampler will move on to the next action.

At a chance node, the sampler recursively follows down each branch in turn. Since each choice node fills in part of the policy the chance node sends the resulting policy from one branch into the next branch. This prevents choice nodes with duplicate states from choosing different actions. If the current policy has an action already selected for the choice node's state then that choice node will automatically select that action.

Using the Round Robin Sampling method reflects the true policy distribution reasonably accurately. The results of sampling will be demonstrated in Section 4.3.2.

3.3.5 Policy Evaluation

The set of policies extracted from the CCTree is evaluated to produce the output for the developers. Each policy in fact represents a strategy to be used to play through the scenario. For each state encountered the policy currently being used should have an action selected. Because of the randomness in the games evaluated, not all trials using the same policy will result in the same end state. In other words, when a single trial is executed only one branch of each chance node encountered is explored. The policy contains information for all branches of every chance node encountered. To test the policy thoroughly multiple trials must be executed for the same policy. The number of trials actually run is a fixed number for each policy and is set by the developer. At the end of each trial the resulting data requested by the developer is stored in an array associated with the policy. Once the Policy Evaluator evaluates all policies without discovering any unlikely events the data is stored to file and can be visualized.

3.3.6 Policy Expansion

Since the Policy Evaluator is running a much larger number of trials than the Model Builder it is possible for the Policy Evaluator to encounter a chance event that was not seen by the

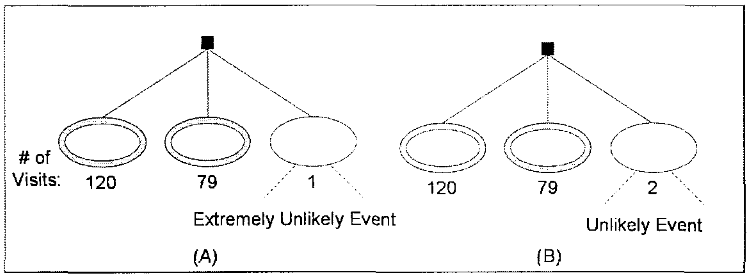


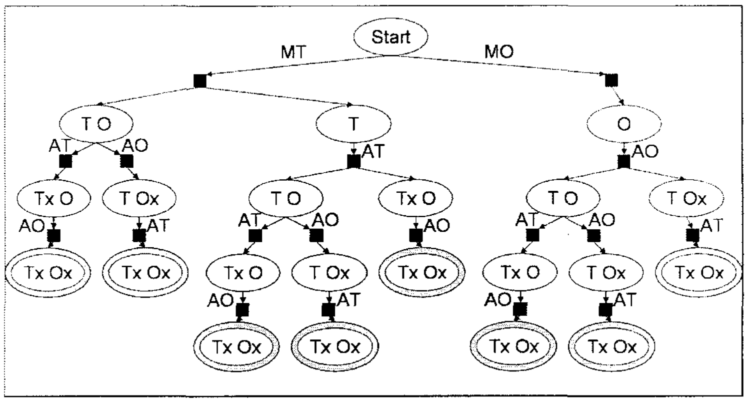
Figure 3.15: Unlikely Events

Model Builder. This event will be detected because the resulting state from the choice node will not match any of its likely children in the CCTree. These occurrences can be sorted into three groups:

- the state is an end state. Resolve by adding a leaf node to the CCTree and connecting it to the chance node.
- the state is not an end state and is below the threshold. These are called extremely unlikely events.
- the state is not an end state and was an extremely unlikely event but this visit pushes it above the threshold. This state is now called an unlikely event.

The probability of extremely unlikely events occurring is so small that there is no need to explore this portion of the CCTree in detail. Their combined outcome does not significantly affect the end result of the evaluation. The event will be explored whenever it occurs but the policy down that branch will take default actions. If an extremely unlikely event keeps reoccurring and passes the theshold for being considered extremely unlikely, it becomes an unlikely event that should be explored in depth. For example, in Figure 3.15 the rightmost leaf in (A) is an extremely unlikely event because it has occured fewer than 1% of the time. When it occurs again (B) it exceeds the 1% threshold and becomes labeled an unlikely event.

When an extremely unlikely event becomes an unlikely event, the Policy Evaluator will instruct the Model Builder that the model is incomplete, and the Model Builder will expand this section of the CCTree. Once the expansion is completed, the Policy Evaluator will extract the new policies from the expanded CCTree and continue with the evaluation. When instructing the Model Builder the Policy Evaluator will create an incomplete policy to direct the Model Builder to the chance node that generates the unlikely event. The incomplete policy will contain only actions for the states that lead directly to the offending chance node. The Model Builder will then use this policy to complete the CCTree.



The following example illustrates how the incomplete policy is used to expand the CC-Tree. In the previous tree building example the hero faced a troll and an ogre. The CCTree in Figure 3.9 (reprinted here for convenience) was considered complete and sent to be evaluated. Consider the following. From the start node if the hero chooses to move toward the troll, corresponding to the left branch of the CCTree, a chance node is encountered. Assuming the result of the chance node is to encounter the troll alone, state T is the resulting state. Now the only action is to attack the troll and the two resulting states from this chance node are TO, where the ogre reaches the hero before he kills the troll, and TxO, where the ogre reaches the hero as he kills the troll. From this chance node it is unlikely but possible that a third state is encountered where the hero kills the troll while the ogre still has not reached him, state Tx. Since state Tx is highly unlikely the Model Builder may have considered the tree complete before ever seeing this state. If this state is encountered often enough the Policy Evaluator will direct the Model Builder to expand that region of the CCTree.

The incomplete policy produced indicates how to get to the offending chance node. The Policy Evaluator traces its steps backwards from the offending chance node and creates an incomplete policy that can only lead to this node. In the example the policy would be, at the Start node "move toward troll" and from state T "attack the troll". The rest of the policy would be blank. The policy is depicted with bold arrows in Figure 3.16.

Even though the chance node cannot be forced down the path to state T, any trial which does not lead to state T can be aborted and therefore the expansion of the CCTree is sped up. After the unlikely event has been observed this section of the CCTree is built following the algorithm discussed earlier in Section 3.2.1. In the example the CCTree obtained is shown in Figure 3.17.

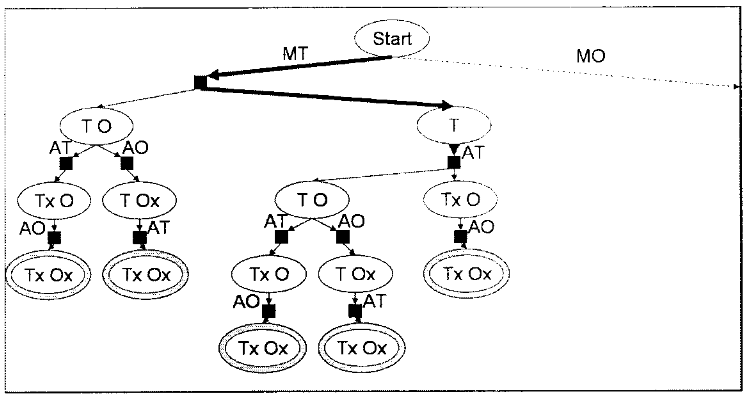


Figure 3.16: Incomplete Policy

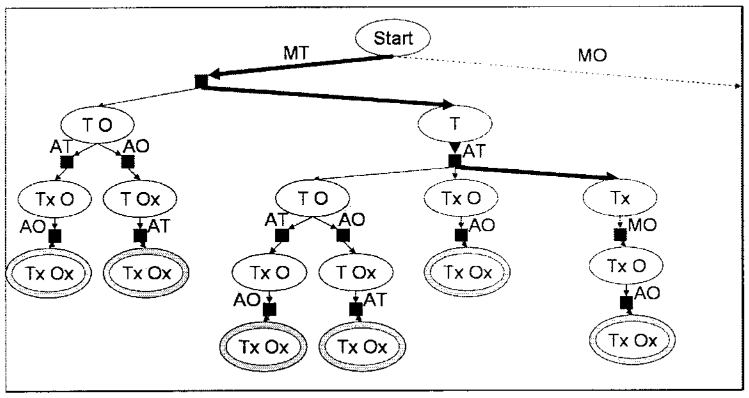


Figure 3.17: Updated CCTree

3.4 Output Files

The files created by the Game Analyzer contain the data gathered for each policy as well as a definition of the policy. For example, consider the policy in Table 3.3 for "The Troll and the Ogre" scenario, and suppose the developer is only interested in the average time it takes for a human to play the scenario. The policy definition file in Table 3.5 contains all the information that uniquely identifies the policy and the results file in Table 3.6 contains the associated results. This example only shows a single entry but in reality there is an entry for all the policies that have been evaluated.

3.5 Summary

This chapter has covered the entire function of the Game Analyzer. The parts shown in Figure 2.2 (reprinted as Figure 3.18) have been explained in detail. The Model Builder first

	States					
Policy #	Start	T	O	T O	T Ox	Tx O
. . .						
20	MO		AO	AT	AT	AO
. . .						

Table 3.5: Policy Definition File

Policy #	Average Run Time
. . .	
20	24.3
. . .	

Table 3.6: Results File

builds a model of the policy space. Using this model the Policy Evaluator extracts policies and executes them. The data resulting from executing the policies is stored in files for the game developer to visualize at his leisure. These results contain information specified by the game developer to help him fine tune his game.

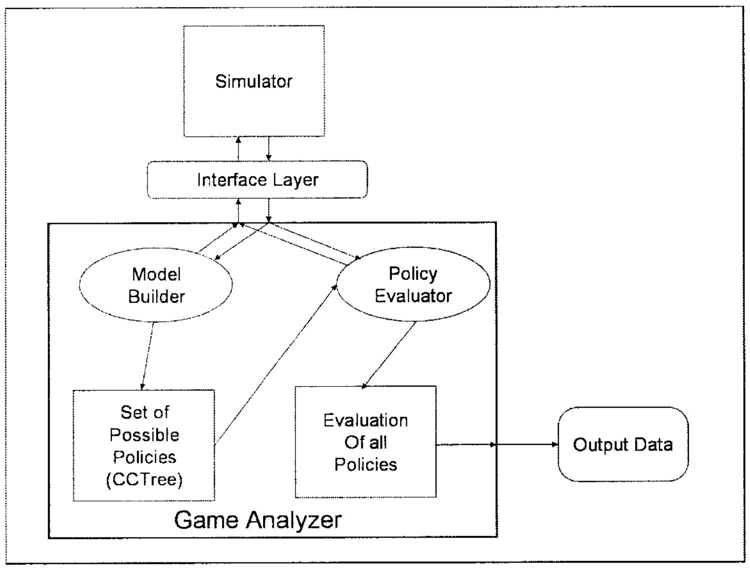


Figure 3.18: Overview of the Game Analyzer

Chapter 4

Experiments

This chapter presents the results of applying the Game Analyzer to various scenarios and evaluates the overall functionality of the Game Analyzer. Section 4.1 explains how the scenarios are created. Section 4.2 and Section 4.3 demonstrate the power of the Game Analyzer and discuss the results obtained from applying it to various scenarios. Finally Section 4.4 evaluates the performance of the Game Analyzer.

4.1 Experimental Setup

The experiments were done using a toy version of a Role-Playing Game (RPG). The game was based on Dungeons and Dragons to simulate games like Baldur's Gate and Neverwinter Nights.[1] Only the most basic of functions have been implemented in the simulator. Even though the simulator is limited, some of the scenarios used can be found in real games. In all the scenarios used here the player would control a single hero. Therefore the Game Analyzer will control this hero throughout the scenario. Since the Game Analyzer controls the hero as a player would, the Game Analyzer will use the set of allowable actions for the hero as defined by the game in question. The hero's opponents in these scenarios will have a deterministic scripted behavior. As mentioned previously, the Game Analyzer does not require that the opponents be deterministic but it is the case in these scenarios.

In the simulator each character is using a weapon and wears armor. Each character can strike at an opponent in the range of their weapon. To strike someone the striking character rolls a twenty-sided die and adds his skill with the weapon then compares the result with the armor of his intended victim. If the result is higher than the armor value the strike succeeds. If the strike succeeded then damage is dealt to the victim. Each weapon has a base damage value and a random damage value. A die is rolled to determine the random damage (the number of sides on this die depends on the weapon) and the base damage value is added to the result. To decide who strikes first each character rolls a ten-sided die and

[1]Baldur's Gate and Neverwinter Nights are video games developed by Bioware Inc.

adds their weapon speed (the lower the weapon speed, the faster the weapon) to it. Then each character does their action in turn starting from the lowest result.

4.2 Answers from the Game Analyzer

This scenario will be used as a working example throughout this section.

4.2.1 Example Scenario: Surrounded

Actors:

- The hero has a choice of two weapons and can change weapons every time he is given a choice of actions. The first weapon is a short sword which deals low variance damage. The second weapon is a heavy axe which deals high variance damage. The sword's average damage is slightly lower than the axe's. The hero can attack any monster as long as he is within his weapon's range of that monster. The hero can move to any monster as long as he is not within his weapon's range of that monster or any other monster.

- The armored ogre wielding a twig is hard to hit, has many hit points and deals insignificant amounts of damage. The ogre will always move towards and attack the hero.

- The naked goblin with the giant axe is easy to hit, has very few hit points and deals large amounts of damage. The goblin will always move towards and attack the hero.

- The troll is average in every respect. The troll will always move towards and attack the hero.

Setup:

The three monsters are placed at the same distance from the hero but in different directions. Therefore moving towards one monster moves the hero away from the others. When the hero chooses to move towards a monster, he will not have a choice of actions until a change of abstract state occurs. Once the hero is in range of a monster and chooses to attack it, the hero will not have a choice of actions until a change of abstract state occurs. The abstract state changes when a monster dies or when a monster comes within weapons range of the hero. This setup can be seen in Figure 4.1.

End States:

All three monsters are dead or the hero dies.

Abstract State Variables:

Hero is alive, Troll is alive, Ogre is alive, Goblin is alive, Troll is in range, Ogre is in range, Goblin is in range.

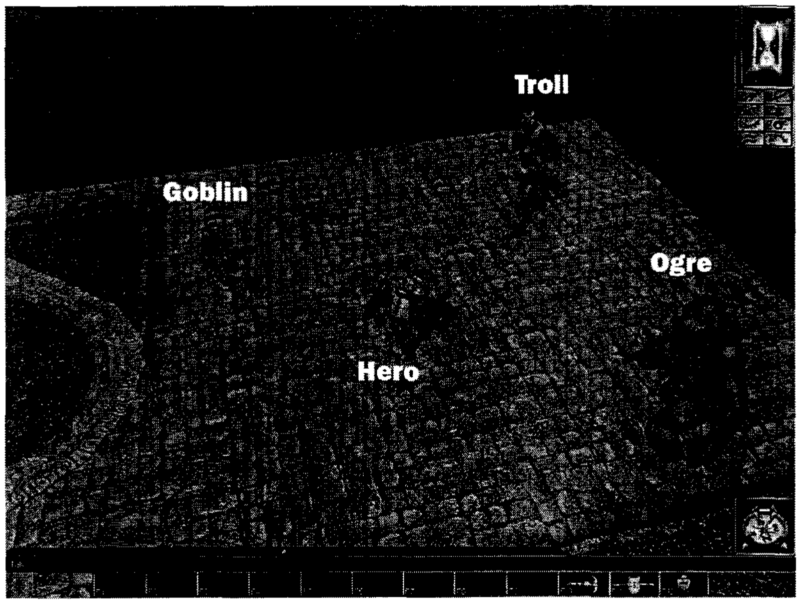


Figure 4.1: Example Scenario: Surrounded

In Chapter 1 it was stated that the Game Analyzer could help answer many questions. The Game Analyzer itself does not fully answer the questions. It provides the raw data that can be used to answer those questions. The graphical results seen below are produced with a prototype visualization tool. The visualization tool used throughout this document is primitive. The focus of this research is to gather the data that will help the game developers analyze their game and not the visualization of this data. The data gathered is so rich that even with this primitive visualization tool many useful observations can be made. A more powerful tool could be made to explore the data produced by the Game Analyzer in greater depth and answer many more questions. Hence the discussion will also include answers which cannot be provided by the visualization tool but can be provided by the data from the Game Analyzer.

The raw data collected in these scenarios will always be the final damage the hero has suffered and the time in seconds it would take a human player to complete the scenario. The data to collect is selected by the developer and does not have to be only two items. These particular two items were selected because they can be easily plotted and answer most of the questions.

The output for the example scenario from the visualization tool is shown in Figure 4.2. The Y axis represents the total time in seconds it would take a human player to complete the scenario and the X axis represents the total damage the hero took. Each dot represents the average result for a specific policy. The visualization tool has options to select policies

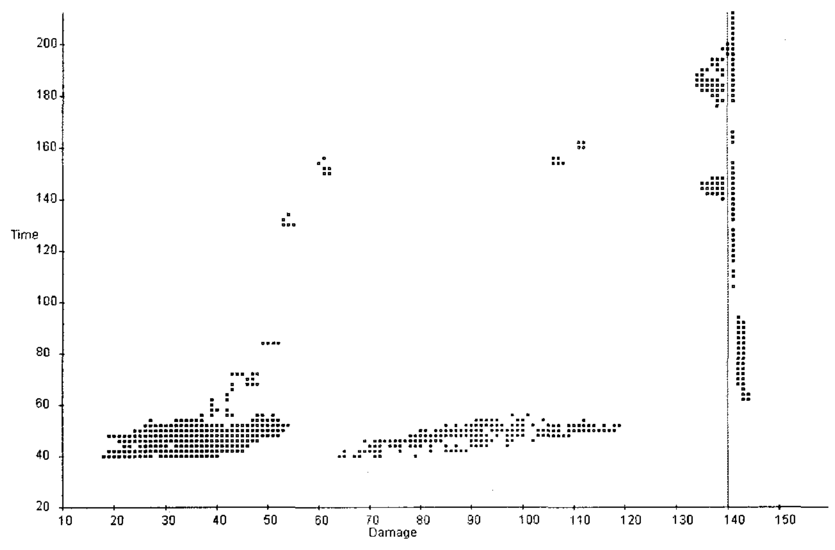


Figure 4.2: Scenario: Surrounded - Results

with a menu. Currently all policies are selected but the selection menu will be used in future examples to highlight certain policies. The vertical line along the right hand side is the amount of hit points the hero had when starting the scenario. For the test scenarios this will always be 140 to simplify graph interpretations. Any policy with an average result over the line ends with the hero dead almost all the time.[2] In Figure 4.2 the results are divided into two large groups. The first group is the two large clouds taking less than 60 seconds to complete the scenario. These are policies in which the hero succeeds in killing the monsters and takes anywhere from 20 to 120 damage. The second group of policies is the vertical cloud near the 140 damage line. Typically, these are policies in which the hero spends his time alternating between weapons while getting beat down by the monsters. The sequence of actions in which the hero continuously swaps weapons is allowed by the game. The Game Analyzer examines all legal sequences of actions, including sequences like swapping weapons continuously. Since most human players would not use a sequence of actions which always leads to the death of the hero these policies will be disregarded while answering most of the questions in this chapter.

Eliminating the policies where the hero continuously swaps weapons while getting beat down by monsters from the policy space is not entirely trivial. The action definition table could be modified to permit changing weapons only when not in combat. The graph result-

[2]It is possible for the hero to suffer more than 140 damage. For example, if he has already suffered 139 damage and he receives a blow for 9 damage the end result will be 148 damage.

ing from this change can be seen in Figure 4.3. This change is very limiting to the hero. The hero can no longer swap weapons between killing monsters if the second monster reaches him before he has finished killing the first. Since the hero can only swap weapons while out of combat if he is never out of combat he can never change weapons. It may be the case that it is preferable to change weapons while in combat since some weapons may be better suited against different monsters. In this particular scenario selecting a single weapon and fighting all the monsters with it is an effective way to play. The policies shown in Figure 4.3 are among the policies that minimize time and damage in Figure 4.2 but do not include the optimal policy.

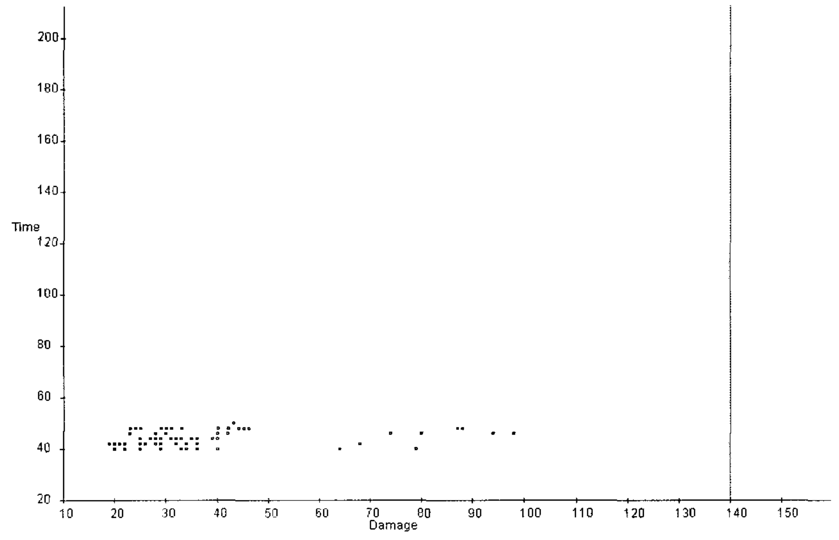


Figure 4.3: Scenario: Surrounded Modified - Results

Another modification could be to take into account previous actions and not let the hero change weapons twice in a row. The problem with this modification is that the state description would have to be expanded to include information about the past. The simplest change would be to add a statement "the hero changed weapons last round" to the state. This would prevent the hero from changing weapons twice in a row. The result of this change can be seen in Figure 4.4. Even though Figure 4.4 looks less populated, each dot in the graph represents many more policies than in Figure 4.2. To implement this change the abstract state has to be enlarged which doubles the total number of possible abstract states. With this increase in the number of abstract states the number of policies is also increased. But the removal of all the policies in which the hero swaps weapons continuously balances out the increase in most cases. Therefore this is a viable solution. The only problem that arises from this solution is that in some scenarios waiting is a good policy and swapping weapons simulated waiting.

In most of the scenarios the modified abstract state will be used to remove the weapon swapping problem. In scenarios that benefit from a policy that involves waiting the original abstract state will be used.

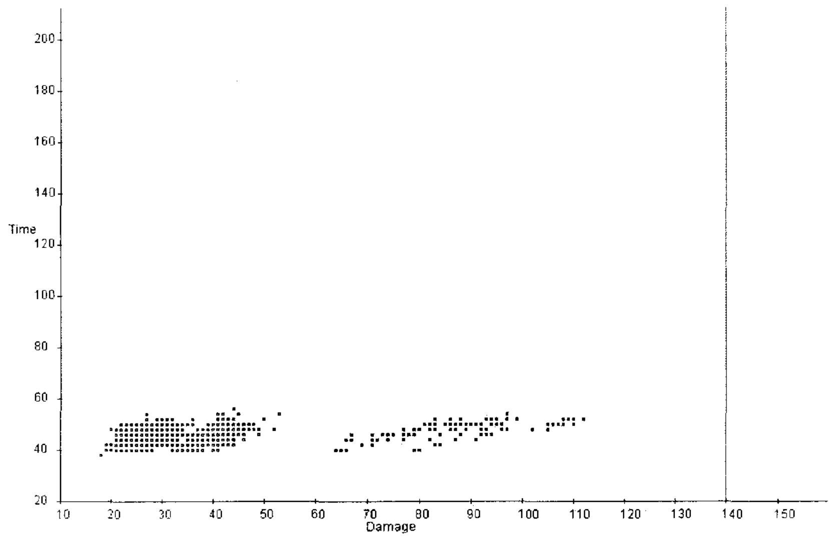


Figure 4.4: Scenario: Surrounded without Continuous Weapon Switching - Results

4.2.2 How Long Will this Scenario Last?

One consideration that is often important in designing a scenario is the time it will take a human to play the scenario. Scenarios that take too long become bothersome to some players and they might give up on the game. But in some circumstances, such as a big battle versus a dragon, the scenario should have a longer duration. Therefore the game designer would often want to know how long the scenario would last. The answer to this question can be easily extracted from the output. Looking at the output in Figure 4.4 it can be seen that on average the scenario will last 40 to 60 seconds.

4.2.3 How Much Damage Will the Hero Take?

Another consideration that is important in the scenario design is how much damage the hero will take. If the scenario is part of a sequence of scenarios with no rest in between then the hero should be able to survive the entire sequence if he is skillful. This question is similar to the previous question since both these questions are directed at the axes. Unfortunately the damage is not as clear cut as the time. Two large groups of policies cover a sizable portion of the damage range. The first group covers approximately 20 to 55 damage and the second group covers approximately 65 to 115 damage. All policies fall in one of these groups. This means that in this scenario a player can avoid taking damage by using skill or experience.

4.2.4 How Much Difference is there Between Different Policies?

The previous question illustrates the difference in the policies quite clearly. One group of policies will yield 20 to 55 damage while another group will yield 65 to 115 damage. These differences can be further illustrated by dealing with a subset of the policies. Figure 4.5 shows the same graph as in Figure 4.4 but this time only the policies where the hero starts by moving to and killing the naked goblin with the giant axe are in black and all other policies are in light grey. The black policies are in the first group and therefore reduce the damage the hero suffers. A skilled player would select one of these policies.

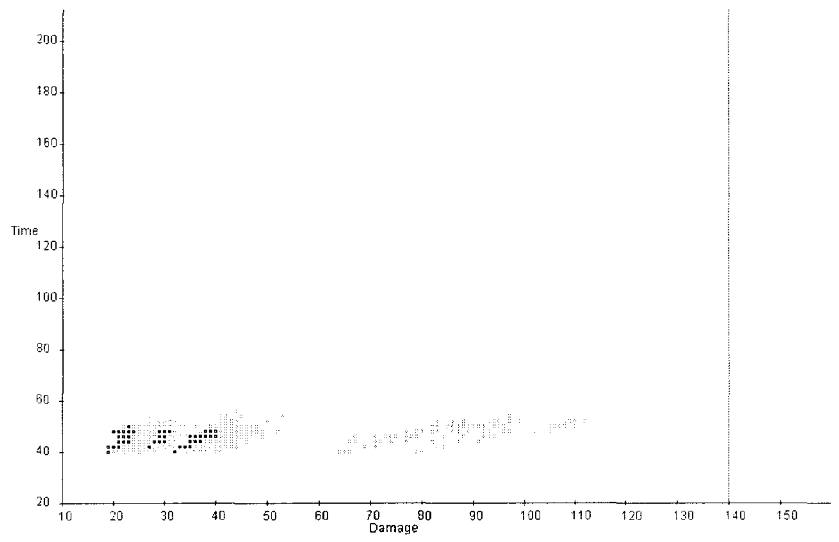


Figure 4.5: Scenario: Surrounded - Results of Goblin Policy

If on the other hand the hero were to start by moving to and killing the armored ogre wielding a twig the resulting policies would be those in black in Figure 4.6. The hero suffers more damage from these policies as they are in the second group. An unskilled player might select such a policy and face a grim outcome.

It is not always the case that different policies lead to significantly different outcomes. Figure 4.7 shows the resulting graph of a scenario taken from Neverwinter Nights, which

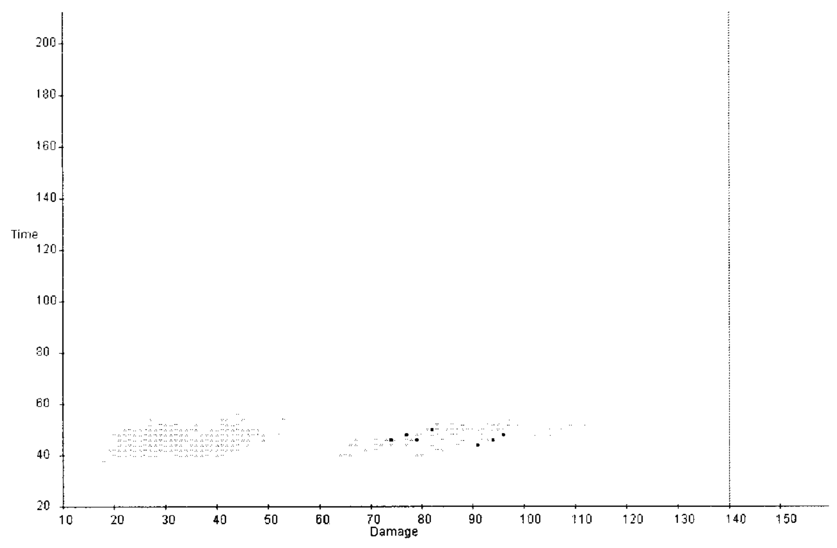


Figure 4.6: Scenario: Surrounded - Results of Ogre Policy

will be discussed further in Section 4.3.4. Figure 4.7 indicates that there is no real difference between the policies. Skill does not matter in such a scenario, all players would end up with a similar experience. This might not have been obvious to the designer without using the Game Analyzer.

4.2.5 What Happens if Certain Parameters of the Scenario are Changed?

This is easily answered by running the scenario a second time with the new parameters and seeing what the output yields. As an example, if the troll is given a giant club which doubles his damage output the resulting graph is that shown at the bottom of Figure 4.8. When compared with Figure 4.4, reprinted in the top of Figure 4.8, it is easy to spot the differences. The scenarios look similar but as expected the hero takes more damage in the new variation since the troll deals more damage. Another notable difference is that the two groups of policies are no longer as clearly separated as before. Since the troll now deals much more damage, the order in which to kill the monsters is no longer as easy to discern.

4.2.6 How Good is a Particular Subset of Policies?

A subset of policies can be selected from the data provided by the Game Analyzer and displayed with the visualization tool and the user can evaluate the resulting data. In Figure 4.6 the group of policies where the hero moves towards the ogre and kills it first is highlighted. The selected policies are in black and all other policies are displayed in light grey. The data

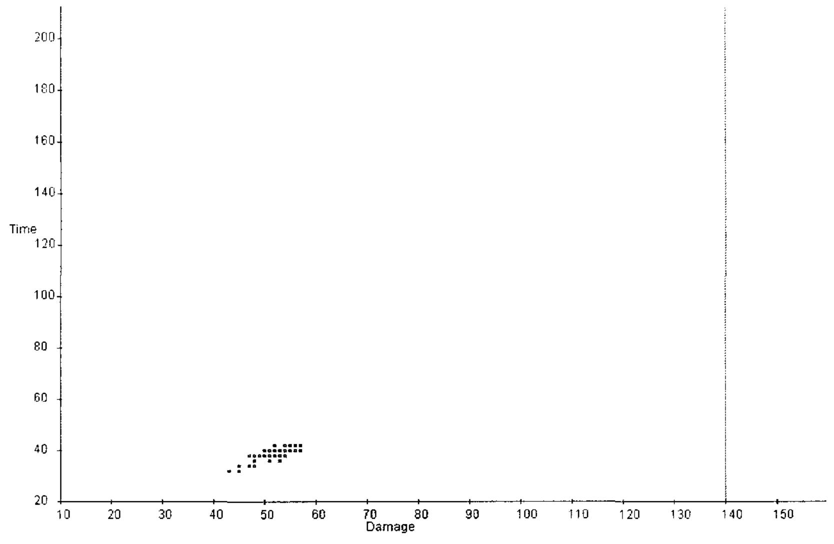


Figure 4.7: Scenario: Cornered by Trolls (Section 4.3.4)

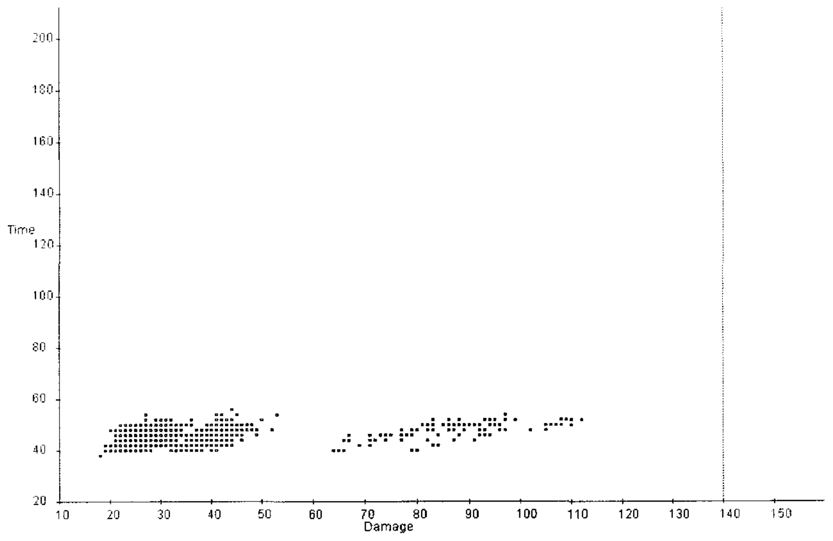
obtained by the Game Analyzer includes the results from each separate execution of each policy. With this data the mean, the variance and so on can be calculated.

4.2.7 Which Policies Yield the Best/Worst Results?

By looking at the graphs produced by the visualization tool the policies of interest can easily be isolated. The best policies are found on the left side of the graph where the hero takes the least amount of damage. In the case of best time the bottom policies would be selected since they are the fastest. If both time and damage are considered then only the bottom left policies would be selected. In any case, the goal is to select a region of interesting policies and find out which policies are there. Using the data provided by the Game Analyzer it is possible to select a region of interest and obtain the related policies. These policies can be compared to identify any similar traits. As an example, the best policy in Figure 4.4 involves killing the goblin, the troll and then the ogre. One thing to note is that the hero always uses the second weapon to kill the goblin in the best policies. Using these results the game designer could assess if changes need to be made to the scenario.

4.2.8 Which Policies Yield Similar Results?

With the visualization tool the user can click on a single dot and get all the policies which currently yield that result. Even though the visualization tool only displays averages, the data provided by the Game Analyzer is much richer. This data contains the time and damage



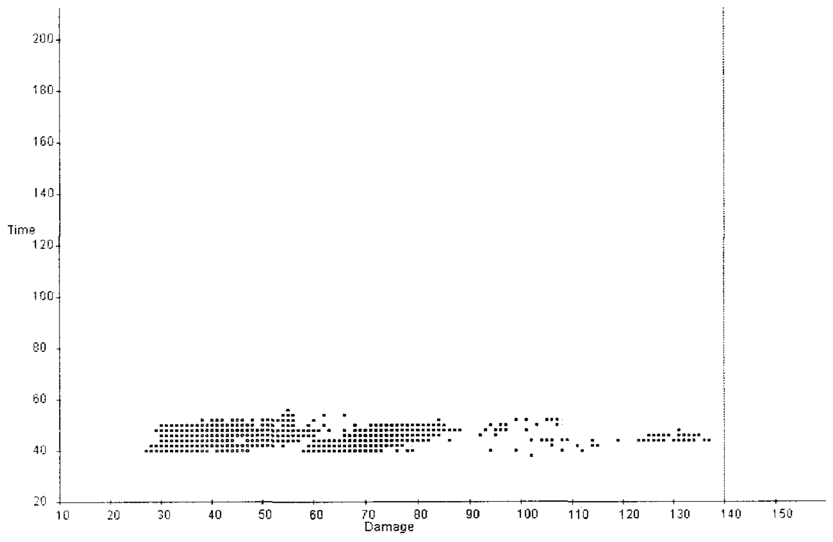


Figure 4.8: Scenario: Surrounded - Troll with Club Comparison

for every run of every policy. Therefore using the data provided by the Game Analyzer it is possible to use clustering algorithms or other software to obtain an accurate grouping of policies which yield similar results.

4.3 Test Scenarios

In this section many test scenarios will be described along with their results. In all test scenarios the following are always true unless otherwise specified. The hero has a choice of two weapons and can change weapons every time he is given a choice of actions. The hero can attack any monster as long as he is within weapons range of that monster. The hero can move to any monster as long as he is not within weapons range of that monster or any other monster. Also the scenario ends when all monsters are dead or the hero dies.

All the scenarios also share common abstract state observations. The abstract state keeps track of the deaths of the hero and any monsters described in the scenario. The abstract state also includes if any of the monsters are within range of the hero's weapons. The abstract state also notes which weapon the hero is currently using.

4.3.1 Scenario: The T Junction

Actors:

- The hero can move down the third branch of the hallway any time he is given a choice of actions.
- The two trolls are identical in every respect. They will move towards the hero and attack him.

Setup:

The two trolls are placed at either end of a long hallway. The hero is placed between the trolls in the hallway such that he is at a T junction in the hallway. The third branch of the hallway is only one person wide. If the hero chooses to go down the hallway he can create a bottleneck for the trolls and force them to fight him one at a time. When the hero chooses to move towards a monster, he will not have a choice of actions until a change of state occurs. Once the hero is in range of a monster and chooses to attack it, the hero will not have a choice of actions until a change of state occurs. The state changes when a monster dies or when a monster comes within weapons range of the hero. The scenario starting point can be seen in Figure 4.9.

Additional Abstract State Variables:

The hero is currently in the small branch of the hallway.

This scenario was created in an attempt to predict the outcome as a game designer normally does. In this discussion "I" refers to the author playing the role of the game designer. By creating the bottleneck for the trolls I expected a big difference between a good policy and a bad policy. I was expecting all policies that started by heading down the bottleneck to be in the upper left area of the graph thus taking longer to execute since

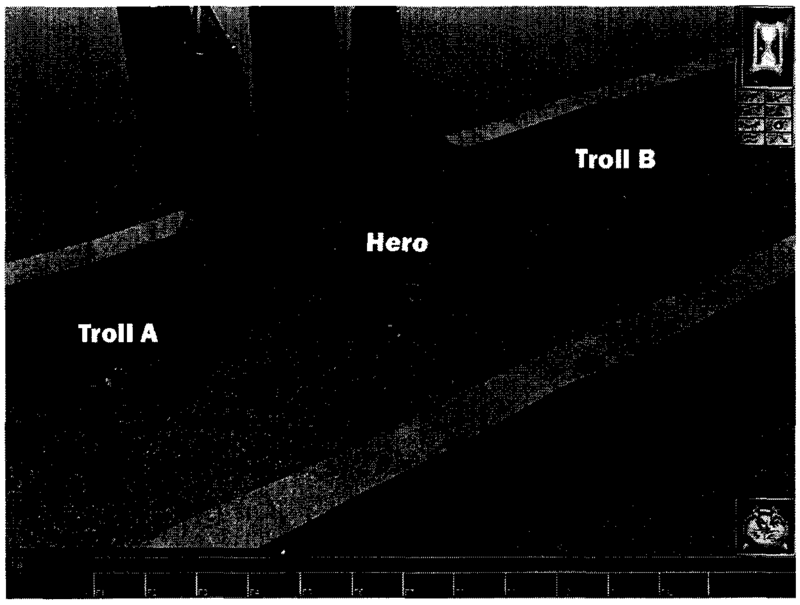


Figure 4.9: Scenario: T Junction

the hero runs down a hallway and waits for the trolls but taking less damage since he faces them one at a time. I was also expecting those policies that did not use the bottleneck to be in the lower right area, taking less time and more damage. I was surprised by the results in Figure 4.10. Waiting for the trolls to come down the small hallway is an important part of the scenario therefore the here is allowed to swap weapons continuously in this scenario but all the policies that do not waste an inordinate amount of time are in the circlular group in the bottom left. These include both the policies that use the bottleneck and those that do not. The policies using the bottleneck are in the upper left portion of the circle and those that do not are in the lower right portion. But this was not the graph I wanted. The Game Analyzer indicated that there was not much difference between the bottleneck and just running to one troll and then the other.

I applied modifications to the scenario and repeated the test using the Game Analyzer until I achieved the results I expected from the scenario. I modified the shape of the hallway and the distance between the hero and the trolls. After a few iterations the scenario behaved as I had intended. The whole process took approximately one hour and I obtained the results in Figure 4.11. There is a clear separation between the upper left potion of the circle and the lower right portion. In the upper left portion the hero is using the bottleneck and in the lower right he is not.

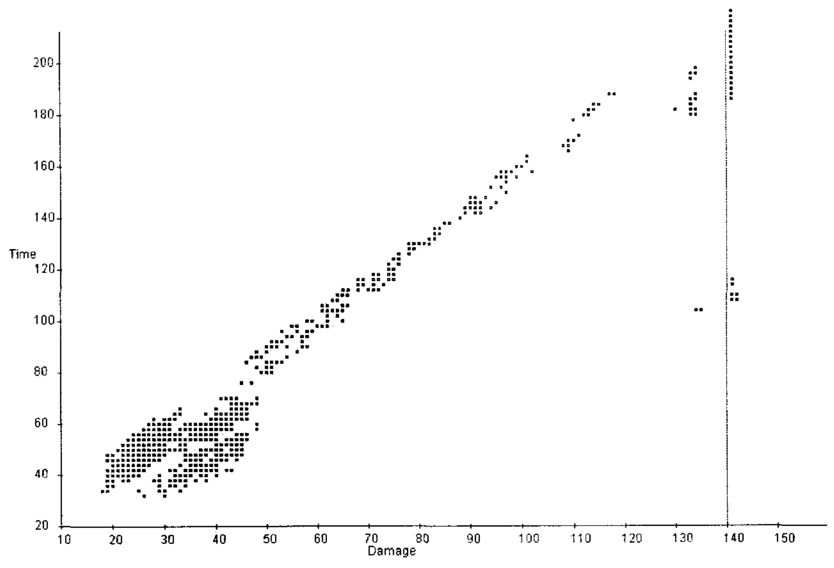


Figure 4.10: Scenario: The T Junction - Results

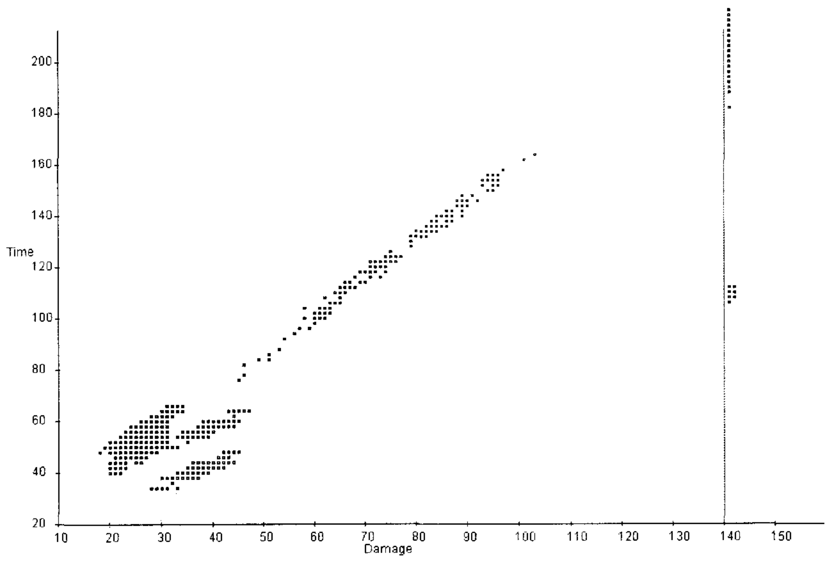


Figure 4.11: Scenario: The T Junction Modified - Results

4.3.2 Scenario: The Maze

Actors:

- The hero is exactly as in the standard description.
- The three trolls are all identical. They will move towards the hero and attack him.

Setup:

The trolls and the hero are placed about a small maze. The trolls as well as the hero magically know the location of everyone in the maze. When the hero chooses to move towards a troll, he will not have a choice of actions until a change of state occurs. Once the hero is in range of a troll and chooses to attack it, the hero will not have a choice of actions until a change of state occurs. The state changes when a troll dies or when a troll comes within weapons range of the hero. This setup can be seen in Figure 4.12.

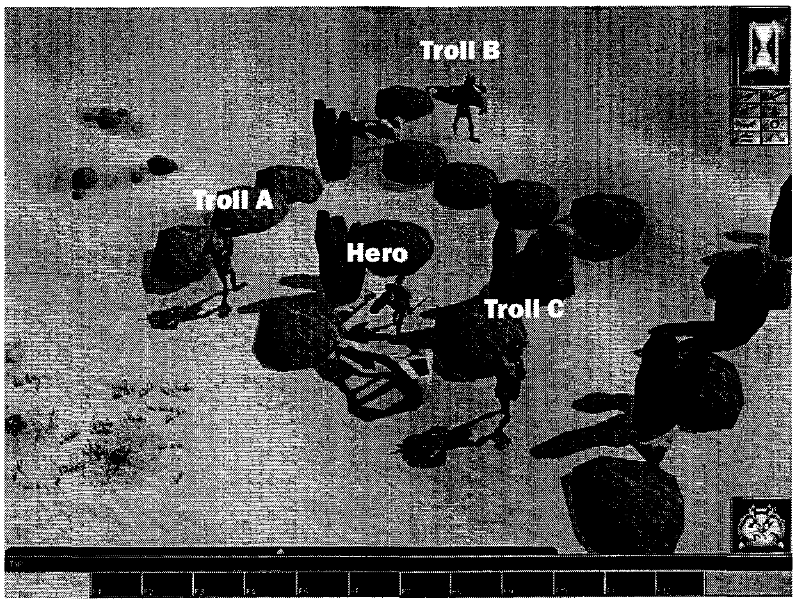


Figure 4.12: Scenario: The Maze

This scenario was created to test the sampling method used. Having a maze with monsters strewn about it results in many policies. The Game Evaluator was executed with different sampling thresholds to see if a small sample was representative of the larger picture. In Figure 4.13 there are 58,216 policies giving the results a thick slanted "V" shaped look. In Figure 4.14 only 581 policies were sampled, a 99% reduction. Even with this reduction the resulting graph closely resembles the one shown in Figure 4.13. The only real difference being that it is not a completely fleshed out "V" but the skeleton is there.

These results show that with a 99% reduction in the number of policies sampled the Game Analyzer still produces useful data. The reduced set of policies demonstrates the range of different policies and their tendencies. The information about the scenario obtained from the small set of policies is similar to that obtained from the large set.

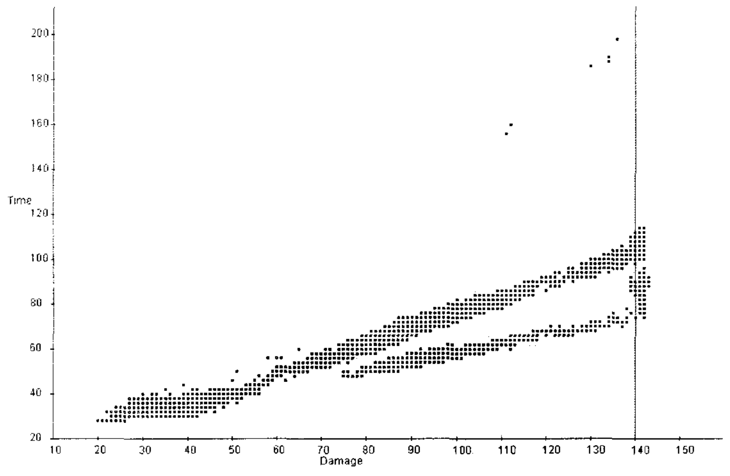


Figure 4.13: Scenario: The Maze - Results

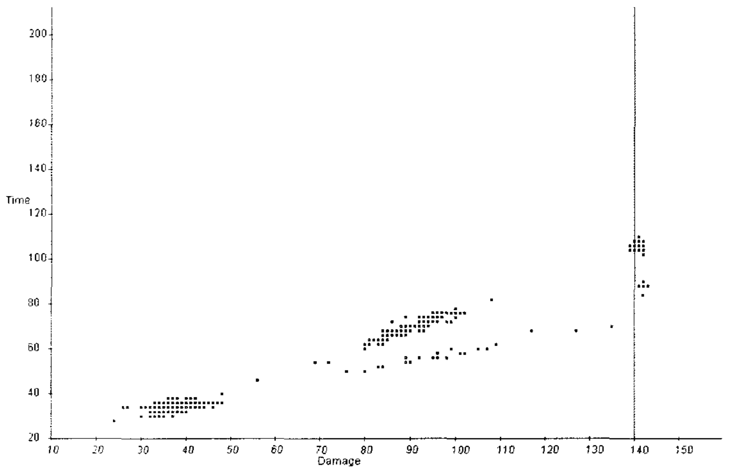


Figure 4.14: Scenario: The Maze - Sampled

Unfortunately the small set of policies does not always capture the extremities of the policy space. In Figure 4.14 the optimal policies are not present. There are eight policies that result in less than 23 damage in Figure 4.13, and none of these are captured after reducing the sampling one hundred-fold. This is not entirely surprising since the odds of selecting one of the above mentioned eight policies are quite slim even with a perfectly uniform distribution of samples over the 58,216 policies.

4.3.3 Scenario: Cornered

Actors:

- The hero can attack any monster but cannot move.
- The armored ogre wielding a twig is hard to hit, has many hit points and deals insignificant amounts of damage. The ogre will always attack the hero.
- The naked goblin with the giant axe is easy to hit, has very few hit points and deals large amounts of damage. The goblin will always attack the hero
- The troll is average in every respect. The troll will always attack the hero.

Setup:

The scenario starts with the hero being cornered by the three monsters described above. When the hero chooses to attack a particular monster, this monster must be killed before the hero gets another choice of actions.

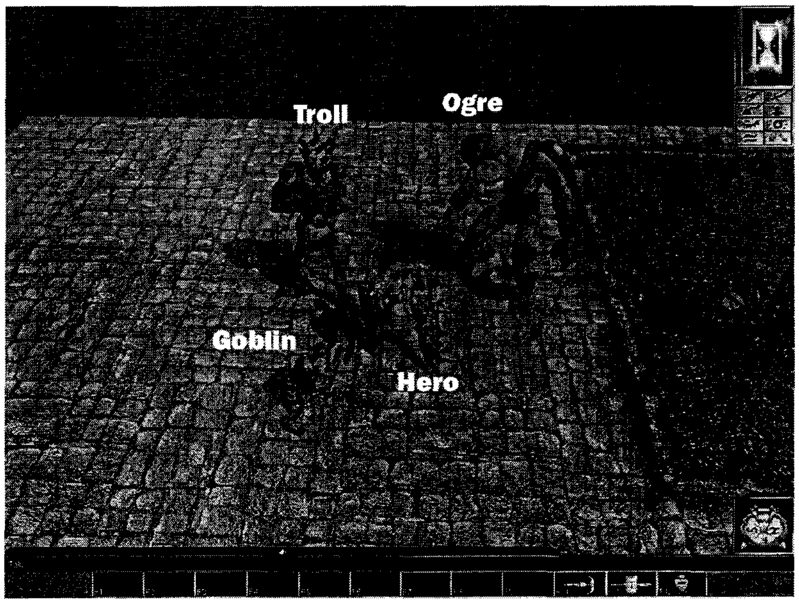


Figure 4.15: Scenario: Cornered

This scenario is almost identical to the sample scenario "Surrounded". The difference here is that the hero starts his journey adjacent to all the monsters. In the Surrounded scenario the hero starts a few paces away from all the monsters. Starting the hero adjacent to all the monsters limits the possible actions taken and reduces the number of policies produced.

In this scenario three monsters of different strengths and weaknesses attack the hero. The hero can defend himself with one of two weapons and has at most four options; changing weapons or attacking one of the three monsters with the weapon in his hand. With such a small branching factor the policies can be enumerated exhaustively. The resulting data is displayed in Figure 4.16.

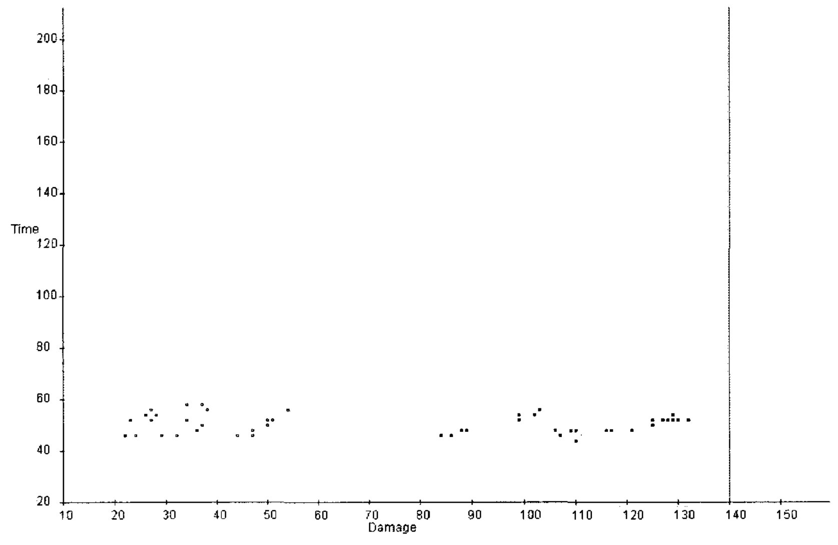


Figure 4.16: Scenario: Cornered - Results

With only 48 policies each policy can be examined in detail. The large gap between the two groups of policies is caused by the ogre. The ogre takes a long time to kill because he has many hit points. If the hero does not start by killing the other two monsters the scale of damage is completely shifted to the right. The left group consists of policies where the ogre is left for last and the right group of policies are those that incorporate killing the ogre with other monsters still alive. There are six different sequences in which the monsters can be killed. Each sequence in which the monsters are killed actually form eight policies. These eight policies involve swapping weapons and killing the monsters. Even though the hero is not allowed to swap weapons twice in a row, the following eight policies can be formed. S will indicate swapping weapons and K will indicate killing a monster. The policies are:

KKK, $SKKK$, $KSKK$, $KKSK$, $SKSKK$, $SKKSK$, $KSKSK$ and $SKSKSK$. The eight policies that involve starting by killing the ogre are the eight policies found at 125 damage and beyond.

4.3.4 Scenario: Cornered by Trolls

Actors:

- The hero can attack any troll but cannot move.
- The healthy troll has many hit points. He will always attack the hero.
- The wounded troll has fewer hit points than the healthy troll but identical in every other respect. He will also always attack the hero.
- The sickly troll is at death's door but otherwise identical to the other two trolls. The sickly troll always attacks the hero.

Setup:

The scenario starts with the hero being cornered by the three trolls described above. When the hero chooses to attack a particular troll, this troll will be killed before the hero gets another choice of actions.

Figure 4.17: Scenario: Cornered by Trolls

This scenario was created as a simple way to illustrate that this method can be applied to real commercial games. This particular scenario takes place in Neverwinter Nights.

Starting the game as a warrior and playing the module published with the game, this small scenario will be encountered repeatedly over the first few levels. Since the hero has no special powers or spells as an early level warrior, this scenario is almost identical to those found in Neverwinter Nights. Since all the monsters in this scenario are identical except for their hit points there is much less variance in the results of different policies.

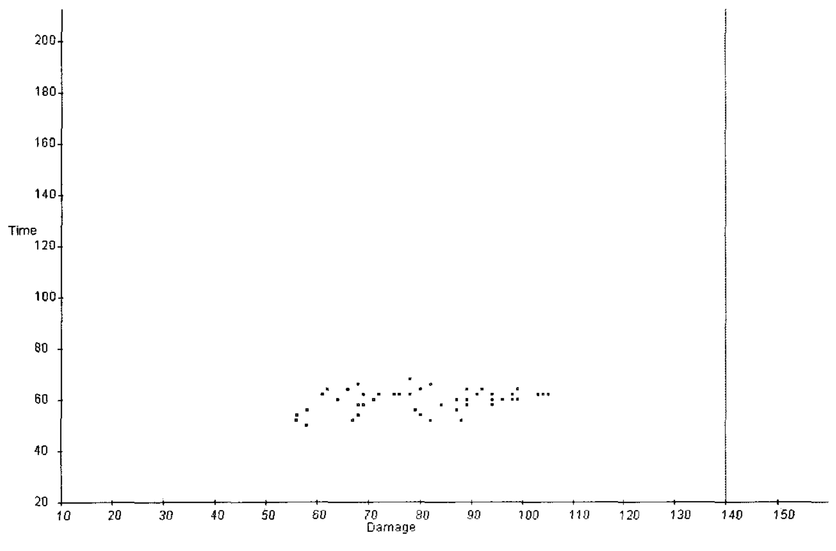


Figure 4.18: Scenario: Cornered by Trolls - Results

Figure 4.18 shows that there was an impact on the damage taken depending on which monster the hero chose to kill first in this scenario. Killing the monster with the least number of hit points first yielded the best results.

This scenario was first attempted with all the trolls having the same number of hit points as this is often the case in Neverwinter Nights. The results shown in Figure 4.19 were obtained from this scenario. It can be easily noted that the policy chosen does not matter much in this case. But the results are still interesting since they show that this encounter is limited to a maximum of approximately 42 seconds and 56 damage as well as a minimum of 32 seconds and 42 damage.

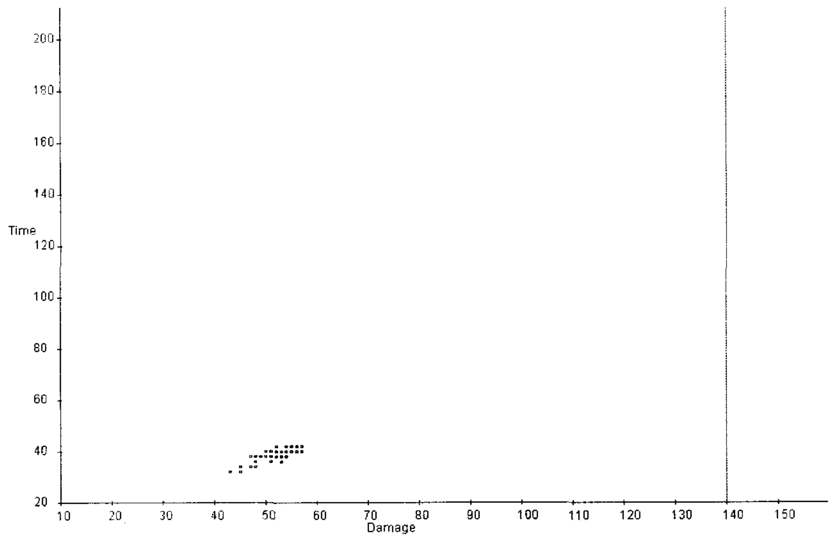


Figure 4.19: Scenario: Cornered by Clones - Results

4.3.5 Scenario: The Hallway

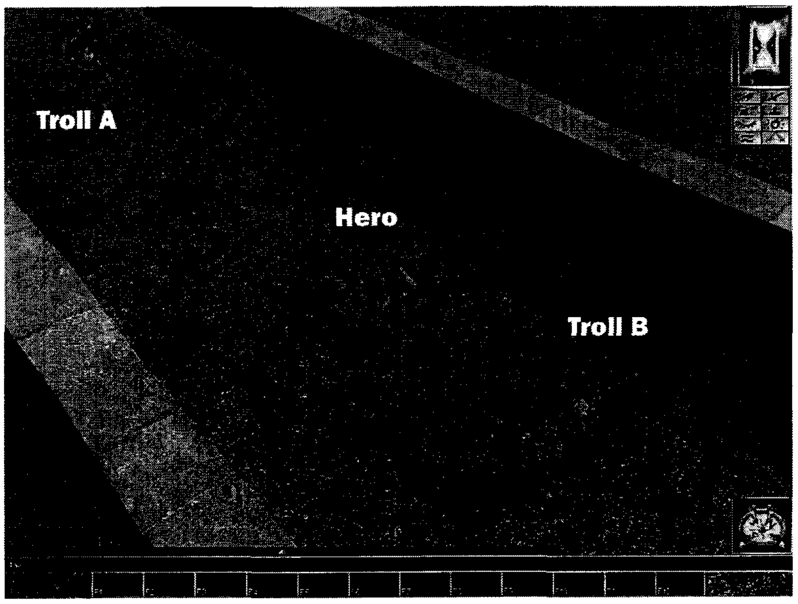


Figure 4.20: Scenario: The Hallway

Actors:

- The hero is exactly as in the standard description.

- The two trolls are identical in every respect. They will move towards the hero and attack him.

Setup:

The two trolls are placed at either end of a long hallway. The hero is placed between the trolls in the hallway such that he is much closer to one of the trolls as seen in Figure 4.20. When the hero chooses to move towards a troll, he will not have a choice of actions until a change of state occurs. Once the hero is in range of a troll and chooses to attack it, the hero will not have a choice of actions until a change of state occurs. The state changes when a troll dies or when a troll comes within weapons range of the hero.

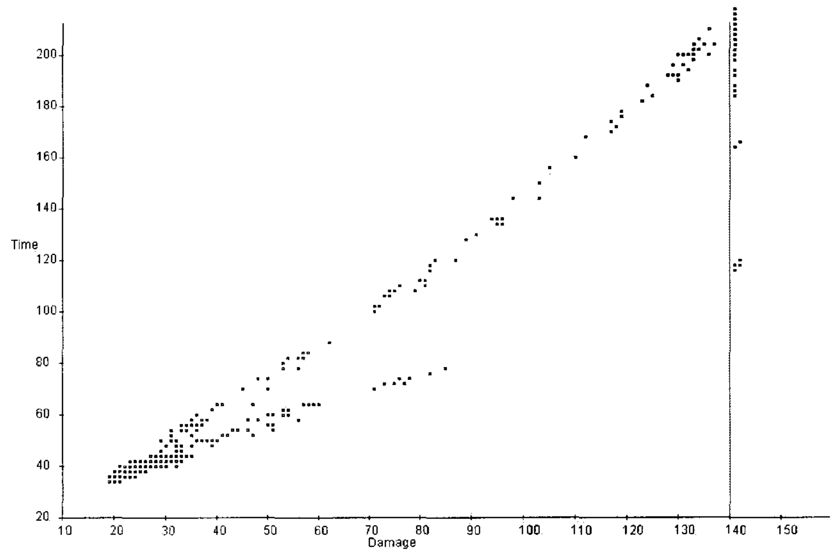


Figure 4.21: Scenario: The Hallway - Results

This scenario was another attempt at predicting the outcome. The scenario is simple, one troll at either end of a tunnel and the hero much closer to one than the other. As the designer, I anticipated that killing the troll closer to the hero followed by the other troll would yield the best results. I also anticipated that running to the far troll while the other chases the hero and then killing one or the other would end in bad results. The exercise was successful and the Game Analyzer helped me verify that the scenario was working as I had predicted. In Figure 4.21 the results resemble a slanted "V". The vertex where the two branches of the "V" meet is composed of policies where the hero kills the closer troll first. The lower of the two branches corresponds to moving towards the far troll first. The hero takes more damage in this branch than in the vertex since he ends up fighting both trolls simultaneously. The upper branch are those policies that tend to waste time such as waiting for the trolls to reach the hero instead of actively seeking them out.

4.3.6 Scenario: James Bond

Actors:

- The hero is exactly as in the standard description. From the starting position the hero can sneak around and reach one trolls without alerting the other.
- The two trolls are identical in every respect. They will move towards the hero and attack him.

Setup:

The two trolls are placed in a room with vision impairing obstacles. If the hero sneaks around to reach one of the trolls, the other will only be alerted once the combat begins but will normally arrive too late to help his friend. If the hero does not sneak from the start he will alert both trolls and will no longer be able to sneak. The path used for sneaking is shown in Figure 4.22. When the hero chooses to move towards a troll, he will not have a choice of actions until a change of state occurs.

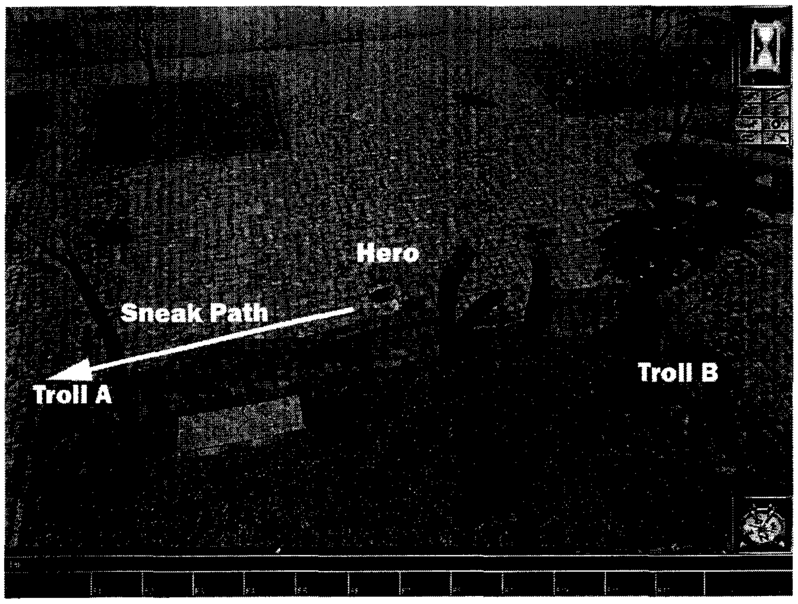


Figure 4.22: Scenario: James Bond

This scenario is used to demonstrate that the Game Analyzer is not only useful for standard "hack and slash" RPGs. In this scenario the hero takes the role of James Bond and tries to stealthily dispatch the two guards in the control room. The bottom left of Figure 4.23 where the hero takes less than 45 damage corresponds to the policies in which the hero uses stealth and dispatches the opponents quickly. Towards the right of the graph the hero is taking more damage and this is mostly due to the lack of stealth. A bottleneck effect where the hero stands in the door frame and fights the trolls one at a time was also observed when the hero was allowed to wait by swapping weapons continuously. This bottleneck does not provide the optimal policies but it does provide some of the policies in the 40 to 50 damage range. Thus the hero is allowed to swap weapons continuously in this scenario and this swapping is what is causing many of the deaths seen in the rightmost portion of the graph.

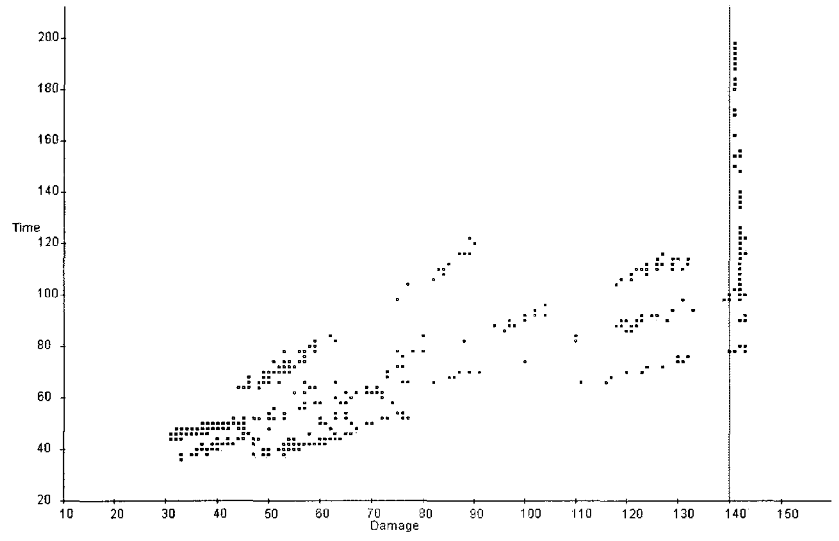


Figure 4.23: Scenario: James Bond - Results

4.3.7 Scenario: RTS

Unlike the previous scenarios this one takes place in a toy Real-Time Strategy (RTS) engine. In this engine damage is deterministic and the characters never miss a shot but there is an added effect called cool-down. Cool-down time is the amount of time a unit has to wait after firing before firing again. Armor in this scenarios is represented by hit points, it takes longer to destroy a tank than it does to kill a marine. Since damage is deterministic, all the scenarios will run for the same length of time and the unit controlled by the Game Analyzer cannot lose.

Actors:

- The hero in this case is an allied Tank firing a 50 caliber machine gun. This gun has no cool-down period and does little damage. The Tank can shoot at any opponent.
- The Enemy Marine wielding an M-16 has average hit points, causes very little damage and has no cool-down.
- The Enemy Heavy Weapons Expert wielding a Rocket Launcher has almost no hit points, causes massive damage but has a long cool-down.
- The Enemy Armored Personnel Carrier (APC) has high hit points, causes average damage and has an average cool-down.

Setup:

The scenario starts with all the units placed in range of each other. The hero gets to choose an opponent and will shoot at it until it is destroyed. The opponents will continuously shoot at the hero.

End States:

All three opponents are destroyed. The hero cannot be destroyed as this is guaranteed by the deterministic setting.

This scenario was created to help solve a small question. When the hero tank meets up with three opponents the goal is to find the optimal sequence in which to kill the opponents to reduce the damage the hero takes. This is a one vs. N real-time strategy scenario and the question is "Is there an easy formula to assign target selection?" In the simplest case the following equation can be used:

$(EnemyDamagePotential)/(TimetoKillEnemy)$

By introducing a cool-down period in which the opponent cannot deal any damage this formula breaks down. The example in this scenario is one for which this formula does not work. Using the damage over time formula the heavy weapons expert should be killed first then the marine and then the APC. But because of cool-down it is possible to kill the marine and the heavy weapons expert before the heavy weapons expert fires a second time. Therefore the optimal solution is to kill the marine first then the heavy weapons expert followed by the APC. Since this scenario is deterministic and very small the Game Analyzer returns the results in Figure 4.24 instantly. The left most dot is the optimal solution just described.

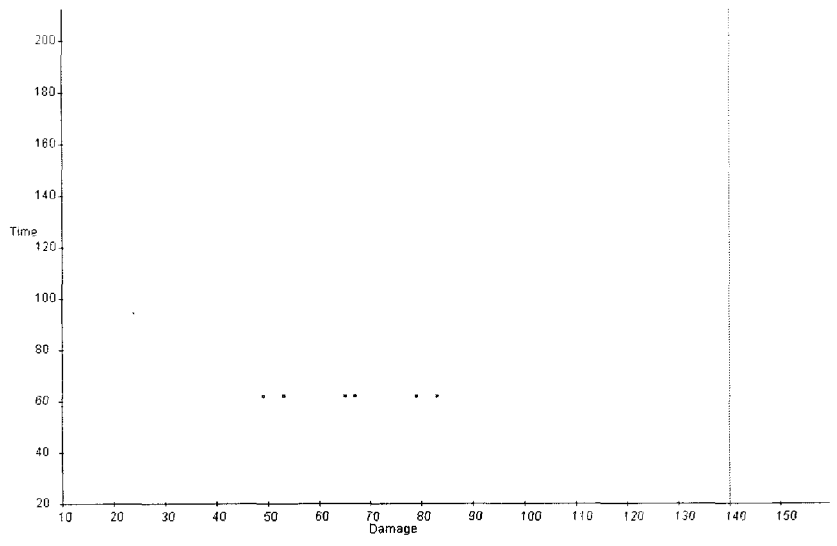


Figure 4.24: Scenario: RTS - Results

4.4 Evaluating the Game Analyzer

Data was gathered during the executions of all the test scenarios to evaluate the functioning of the Game Analyzer. Each test scenario was executed multiple times to insure accuracy. The data was gathered to establish the Game Analyzer's usefulness. The Game Analyzer is meant to help game developers produce better games. The test scenarios have demonstrated that the results are quite useful but nothing has been said about the costs associated with these results.

In the scenario "The T Junction" (Section 4.3.1) it was mentioned that fine tuning the scenario took approximately an hour. Creating the scenario and defining the abstract states and actions in the first place took about the same time. But creating subsequent scenarios in the same environment and using the same abstract actions and abstract state takes much less time. Subsequent scenarios took approximately 15 minutes to create. This time will vary based on the skill of the person using the Game Analyzer.

The time the Game Analyzer takes to execute a scenario can be measured. Of all the test scenarios "Cornered by Trolls" (Section 4.3.4) was the fastest to execute. It took less than one second for the Game Analyzer to build the model and evaluate it completely. The CCTree contained on average 225 nodes and always generated 48 policies. Each policy was executed one hundred times. The Model Builder played the game on average 500 times to build the CCtree and all policies were discovered on the first build.

"The Maze" (Section 4.3.2) was the longest scenario to execute. It took fifty three minutes to build the entire model and and evaluate all the resulting policies. The CCTree contained more than eighteen thousand nodes. The CCTree contained over fifty-eight thousand policies and each policy was executed one hundred times. The Model Builder played the game on average 110,000 times to build the CCTree. The Model Builder had to expand the tree two to four times to discover all the policies. For each expansion run the Model Builder played the game an additional two to five thousand times.

The sample scenario "Surrounded" used in Section 4.2.1 took on average two minutes to execute. The CCTree for this scenario contained slightly more than three thousand nodes. The CCTree contained 3808 policies which were executed one hundred times each. The Model Builder needed to play the game approximately twenty-one thousand times to generate the tree. The Model builder needed to expand the tree at most once during the evaluation and needed to play the game less than five hundred times to complete the expansion.

The data shows that the Game Analyzer can play a scenario approximately three thousand times per second. This is an average from all test scenarios described in this thesis. Depending on the scenario the Game Analyzer varied between one and six thousand scenario executions per second the average being approximately three thousand. The test scenarios took anywhere from 20 seconds to over 3 minutes to finish depending on the policy used. If the game engine used in these tests were to be played by a human it would be run in real time and a game tester playing the test scenarios using a single policy would take from twenty seconds to three minutes to play the scenario once. On average a good policy would take the human forty seconds to a minute to play. In forty seconds the Game Analyzer can play the scenario one hundred and twenty thousand times.

The Game Analyzer requires a few hours of work to create the state abstraction, the interface layer and to hook up to the game engine. Then the scenarios might take an hour or two to set up for the Game Analyzer. Assume that the total time is eight hours to hook up the Game Analyzer to the game engine and preparing a scenario. In these eight hours a single human tester could play the scenario about 500 times. After these eight hours the Game Analyzer would need a fraction of a second to play the scenario 500 times. Any subsequent scenarios would not require the initial setup and could be hooked up to the Game Analyzer in an hour or so.

Thus the Game Analyzer produces informative output at a reasonable cost. The time to set up the game engine to be used with the Game Analyzer is time well spent. The test scenarios show that even with a simple visualization tool many important questions can be answered.

Chapter 5

Related Work

5.1 Semi-Automated Gameplay Analysis

Research similar to the Game Analyzer has been conducted independently at the University of Alberta. SAGA-ML [11][14] is a machine learning program that attempts to discover unintended functionalities in gameplay. The goal of SAGA-ML is to augment the abilities of the game designer by providing a largely automated analysis of the game behavior. SAGA-ML has been applied to FIFA '99, a video soccer game developed by Electronic Arts. The results obtained from SAGA-ML identified sweetspots (exploitable weaknesses) and hardspots (unintentional difficulties) in the game of FIFA '99. Other research on commercial sports games involves genetic algorithms [5]. This research is similar to that of SAGA-ML but differs even more from the Game Analyzer.

SAGA-ML and the Game Analyzer share almost identical goals but the functionality of the two programs differ. SAGA-ML consists of an active learning methodology. SAGA-ML starts with a uniform random sampling of the parameter space. The initial setup of the scenario consists of parameters and the space of all possible parameter settings is what is being sampled. For a soccer one-on-one kick scenario these parameters could be the location of the goalie, the location of the kicker and the angle and power of the kick. The results from the samples are fed into a machine learning algorithm. These results are binary, did the kick score or not. The machine learner takes this binary output and attempts to build a decision tree. The learner guides future waves of sampling by putting emphasis on the ambiguous areas in order to find boundaries. This cycle continues until the boundaries between areas are clear and the decision tree is accurate. The results are then shown through a game specific visualization tool.

SAGA-ML starts the sampling process without ever building a model. The Game Analyzer builds a model of the entire scenario before attempting to sample. This major difference separates the two approaches. SAGA-ML is better suited for sports games where the Game Analyzer is better suited for Role-Playing Games.

5.2 Markov Decision Process

As described by Russell & Norvig [10] an MDP has three defining components: the initial state S_0, a transition model $T(s, a, s')$ and a reward function $R(s)$. The transition model is a table of probabilities $T(s, a, s')$ that denotes the probability of reaching state s' starting from state s and executing action a. The reward function $R(s)$ is used to guide the MDP solver in its search for an optimal solution.

The reason this simple transition model $T(s, a, s')$ works is because the transitions are assumed to be Markovian. Russell & Norvig [10] define a transition model as Markovian if "the current state depends on only a finite history of previous states".[1] Therefore the transition probabilities depend only on a finite history of previous states. This can be further simplified by using a first order Markov Decision Process which assumes that the current transition probabilities depend only on the current state.

A solution to an MDP takes the form of a policy π. Each policy contains entries $\pi(s)$ which represent a probability distribution over the actions to be taken in state s. The policies contain an entry $\pi(s)$ for every possible state s.

When comparing an MDP to the CCTree some similarities can be noted. They are both used for sequential decision problems. Both use Markovian policies. Even though the policies used with the CCTree are deterministic they can be considered a distribution over the action space where most actions have a 0 probability. Both the MDP and the CCTree have transition models. The transition model of an MDP is explicit and takes the form of a table of probabilities. This table is given to the MDP but can be learned prior to its use in the MDP. The transition model of the CCTree can be found in the chance nodes. They keep track of all resulting states from a state-action pair and they can be queried for statistics like probabilities of reaching a certain state.

Even though MDPs and CCTrees are similar they are fundamentally different. There are superficial differences between the MDP and CCTree such as MDPs are typically used to solve for the optimal solution and the CCTree is used to accumulate data on all solutions. But the real differences are much more profound. The CCTree does not use a reward function and it does not rank its policies in any fashion. But the biggest difference is that the CCTree is non-Markovian. Since the transition from a state in the CCtree to another depends on all previous states it is possible for a triplet $T(s, a, s')$ to appear more than once in the tree and with different probabilities. This is not possible in an MDP since the transition probabilities arising from state s depend only on the state s and the action a. This difference may arise because of the differences in how the state is observed. In an MDP it is assumed that the state is fully observable where the CCTree only sees a state abstraction

[1]Page 539

which can hide many of the details of the state.

5.3 Partially Observable Markov Decision Process

A Partially Observable Markov Decision Process (POMDP) is an MDP in which the state is not fully observable. To illustrate the difference let us consider the RPG scenarios we have been working with all along. In a normal MDP the entire state is observable, all the characters positions, exact health points and which weapon they are using are available and the MDP can base its choice on complete information. In a POMDP only a partial state is observable, it could be the case that the POMDP, like our game evaluator, only sees if characters can hit each other or are simply alive or dead. This makes decision making for a POMDP much more challenging and very similar to the Game Analyzer's situation.

The POMDP contains all the elements of the MDP. It has a transition model and a reward function but also contains an observation model $O(s, o)$. The observation model defines the probability of being in state s and observing the observation vector o. From this observation model the POMDP constructs belief states. A belief state is a probability distribution over all states detailing how likely we are to be in each state. Transitions from the transition model are converted using the belief states into belief transitions from one belief state to another. Once the belief state transitions $T(b, a, b')$ are obtained the POMDP over real states can be solved as an MDP over the belief states.

Even though belief states used in POMDPs seem similar to abstract states used in the CCTree, in reality they are quite different. Both the belief state and abstract state obscure the real state observations. A belief state is a set of real states along with the probabilities of being in each state. An abstract state corresponds to a distinct set of real states associated with the abstract state. The difference here is that a belief state contains a probability distribution over all possible states and could be any real state with some probability. The abstract state is a mapping from a group of similar real states to a single abstract state. Abstract states divide the underlying state space into disjoint sections where the belief states do not divide the state space at all. The belief state is used to solve the POMDP for real states and the abstract state is meant to group information from similar real states. The abstract state is used as a tool to abstract a complex space to facilitate search over this space. The belief space on the other hand is forced upon the user due to his innability to fully observe the real state.

Equipped with belief states a POMDP is reduced to an MDP. This MDP is now solved over belief states much like the CCTree and its abstract states. Yet the two remain different since again the transition model for the MDP $T(b, a, b')$ is a first-order Markov decision process. The transition to belief state b' is based only on the action taken at belief state b.

The CCTree depends ultimately on all previous states encountered in the path. The MDP groups all transitions $T(b, a, b')$ together no matter where they occur. The CCTree on the other hand keeps them separate unless the entire path they followed throughout the scenario was identical.

5.4 Reinforcement Learning

Sutton and Barto [12] describe Reinforcement Learning as "a computational approach to learning whereby an agent tries to maximize the total amount of reward it receives when interacting with a complex, uncertain environment."[2]

There are two ways to approach reinforcement learning. The first is the model-free approach. In this approach the algorithm is specifically looking for the optimal policy. Hill climbing is used and any non-optimal policies are discarded. This approach is entirely unlike the Game Analyzer.

The second approach is the model-based approach. In order to use a model-based approach a model of the complex and uncertain environment is created. This is typically done by learning an MDP. Once the MDP is learned there are multiple ways to solve it. Temporal-difference learning, dynamic programming and Monte Carlo methods are the three solutions discussed by Sutton and Barto [12].

In solving the MDP, policies are created which are similar to the policies used in the Game Analyzer. Each policy has an action or probabilistic distribution over actions available for each state. Unlike the Game Analyzer, the Reinforcement Learning policies are based on an MDP and because the MDP model contains no information about reachability each policy must account for all states in the environment.

The policies created are evaluated in a similar fashion to those of the Game Analyzer. The major difference here is that the Reinforcement Learner is trying to find an optimal solution to the problem presented. In order to find the optimal solution an accurate probabilistic model is needed. The probabilities of the state transitions must be accurate in order to find the optimal solution. The Reinforcement Learner strives to score each policy according to the probabilistic model in order to obtain the optimal policy quickly. The Game Analyzer does not need an accurate probabilistic model. The Game Analyzer needs a rough reachability model, as long as most reachable states for a policy have in fact been reached the results for that policy will be accurate.

Even though both the Game Analyzer and Reinforcement Learners guide an agent through complex and uncertain environments their goal is ultimately different.

[2]taken from the book jacket

5.5 Statistical Software Testing

Statistical software testing is used to test the inputs given to a program to see if the program conforms with the expected output. Whittaker & Thomason [13] describe this black box approach as "Given a program P with intended function f and input domain d, the objective is to select a sequence of entries from d, apply them to P, and compare the response with the expected outcome indicated by f. Any deviation from the intended function is designated as a failure."[3]

This testing method repeatedly runs the program trying different input sequences. The stopping criteria for the tests is based on the reliability of the program. The goal of this process is to test if the program is reliable and to discover any inconsistencies between the functionality of the program and its expected functionality.

The method used by Whittaker & Thomason [13] consists of creating Markov chains of input to the program and testing those chains on the program. A Markov chain can be considered a single policy in an MDP. Using these chains Whittaker & Thomason [13] test the program repeatedly to see if the program functions as expected. With these results they can calculate certain information such as the failure rate of the program and the mean time between failures.

With statistical software testing an MDP is used in a fashion similar to the way a CCTree is used. The tests are used in order to obtain information from as many policies as possible. The tests done in statistical software testing are completely Markovian, the belief is that once you reach a state it does not matter how you got there. These tests are normally used on software like menus, forms and databases in which case it is mostly true that the current state does not depend on previous states.

There are other statistical software testing methods such as the Cleanroom Approach [8] or SFAST [4]. These approaches use statistical methods to generate test data. The data is then processed to obtain results about the software's behavior. The results are completely skewed towards the program's performance in the way of bugs and crashes. These statistical software testing methods are made to analyze the failure rate of a specific piece of software and not its actual function.

Even though the similarities are substantial there are differences. The state space is not abstracted for statistical testing as the goal here is to find unique cases which cause fault. But the major difference is that the CCTree is non-Markovian, it contains duplicate states along different branches and the goal is not to find bugs or estimate the mean time between failures.

[3]pg. 812

5.6 Other Software Testing Methods

There are other methods that have been applied to software testing. Some of these methods are based on machine learning [3] [9]. The classification trees used here differ greatly from the CCTree. For example the trees used by Porter & Shelby [9] represent models of components based on their measurable attributes. The evaluation of the classification trees relates again to the traditional results expected from software testing. These methods are trying identify the troublesome parts of the program. Genetic algorithms [2] have also been used to test software with these goals in mind.

5.7 Heuristic Search

Kovarsky [6] used heuristic search to generate move sequences for Real-Time Strategy games during run-time. This research is meant to augment the scripted behaviors of the computer controlled units. This provides RTS game designers with an improved game AI that could enhance current RTS games. Research is currently being conducted at the University of Alberta to include the heuristic search algorithms developed by Kovarsky into the ORTS [1] system.

Kovarsky's research looks for optimal policies and does not build a model of any kind which is quite different from the Game Analyzer. Even though his particular goals are different, his overall goals are the same. To provide better tools to video game developers. This should in turn provide better video games to the players.

Chapter 6

Conclusion

6.1 Summary

This research started with the goal of helping game developers improve their products. The result is a tool that tests video games rapidly and provides useful feedback to the developer. This tool can be used to fine tune games to reflect the designer's original expectations. With the tool a developer can analyze the behavior of his video game quickly and make adjustments as necessary.

The user connects the Game Analyzer to his game engine through an interface layer. Once this is accomplished the Game Analyzer can be given scenarios to test out. The Game Analyzer will explore each scenario using the set of allowed actions defined by the user. As the Game Analyzer explores a scenario it creates a CCTree which contains information about game states and actions taken. The CCTree is a compact representation of the space of possible policies. Each policy contains the actions that the Game Analyzer will execute at any reachable state during this scenario.

Once the CCTree is completed the Game Analyzer can retrieve all or a sample of all the policies from the tree. Each policy is then executed repeatedly in order to accumulate data. The data accumulated can be anything that can be measured in the game engine. The user of the Game Analyzer can specify what kind of data should be accumulated. The data can then be used in visualizations or further computations. The visualization tool used in this document is only an example of what could be done using the data. The data could be used in clustering algorithms, statistical calculations, prediction models and more advanced visualization tools. These visualizations and computations can then be used by the game developer to fine tune his game. Once the scenario or game engine is modified the Game Analyzer can be used to produce a new set of data. By obtaining subsequent test results the game developer can track his progress.

6.2 Limitations

Like everything else in this world, the approach described in this document has limitations. At first glance the Game Analyzer seems to do away with game testers. A human tester playing the game takes hours to complete something that the Game Analyzer can do in minutes. The human tester can test out only a few strategies where the Game Analyzer can test out thousands.

Taking a closer look at the Game Analyzer some limitations can be found. The Game Analyzer is not as autonomous as a human tester. A human tester can be told to play the game and report any bugs found as well as any perceived flaws with the game. The human tester can then learn the game and play it. The Game Analyzer must be informed through files which actions it can take in the game and does not report bugs directly. The Game Analyzer does provide different feedback than a human tester since it is limited to taking only actions that have been specified and could miss an important flaw in the game. Unlike a human tester the Game Analyzer requires a lot of initial information about the game such as how to abstract states, which information to collect and which actions are possible.

The Game Analyzer is also limited in the types of games it can analyze. The Game Analyzer works very well with role-playing games. There have been many examples of RPG scenarios in Chapter 4. A real-time strategy game scenario was also present. This scenario was very simple due to the large branching factor in RTS games. The Game Analyzer cannot handle RTS games as well as RPG games in its current form. Some aspects of first person shooting (FPS) games can be approximated by the Game Analyzer as seen in Section 4.3.6. The Game Analyzer cannot approximate the use of GUI functions such as aiming and jumping. The current version of the Game Analyzer is only moderately useful in analyzing FPS games.

A Turn Based Game is a game in which players (including computer players) take turns to accomplish their goals. An example of such a game is chess. Each player has a turn and during this turn the player can do a specific number of actions. Once the player's turn is ended the other player gets a turn. New types of turn based games like "Syd Meyer's Civilisations" may require tuning of game parameters. The Game Analyzer has not been tested on this type of game for lack of time and game engine source code. The Game Analyzer is well suited for these types of games and would require little or no modifications to adapt to turn based games.

Sports Games are games that simulate a sport. Soccer, hockey, football and many more sports have been converted into video games. These games run in real time and attempt to follow all the rules and physics of the sport they simulate. The Game Analyzer could be used in sports games. The state and action abstractions would have to be well thought out in order to reduce the resulting policy space to a manageable size. The Game Analyzer may need some modifications to work with sports games but these would be minimal. The Game Analyzer is not as well suited as SAGA-ML[11] to analyze sports games but would work in a similar fashion.

In Racing or Simulation games the players takes the role of the driver or pilot for any number of vehicles. The game approximates the physics related to the vehicle and the player pilots the vehicle through a series of missions or races. The Game Analyzer is poorly suited to analyze these games. These games rely on near-optimal play from the user therefore the widespread policy evaluation of the Game Analyzer would not convey pertinent information.

The greatest limitation of the Game Analyzer is technology. Most games have a large branching factor. The branching factor causes an exponential explosion in the number of nodes in the CCTree. This increasingly large number of nodes takes an increasing amount of time to process. The experiments done in this document took between one second and five minutes to run. As the experiments get more and more complicated the time it takes to run them increases drastically. Therefore scenarios must be limited in size.

Another limitation is the game engine itself. The game engine has to be modified to allow for certain information to be extracted from the game state as well as to allow resetting the game engine to a start-of-scenario game state. Resetting the game engine is usually an easy task since many games allow you to save your current game state and load previous game states. Nonetheless some additional work has to be done in order to connect the game engine to the Game Analyzer. But some modifications must be made to the game engine even with a human tester. To evaluate the human tester's performance the game engine requires additional information gathering methods in order to accumulate data.

6.3 On the Road to Neverwinter Nights

This research first attempted to use Neverwinter Nights, an award winning Role-Playing Game created by Bioware, as a test bed. The problems encountered were minor but crippling. Lacking access to the source code the graphics could not be removed. The Game Analyzer would have to play each game exactly as a human would therefore losing its ability to compute thousands of times faster. With access to the source code, the game engine could have been stripped of its graphics and hooks could have been placed in order to allow the Game Analyzer to function with this game.

If the source code was available then the natural place to put the Game Analyzer would be in the Aurora Toolset. With the versatility of the Aurora Toolset supplied with Neverwinter Nights it could be possible to incorporate the Game Analyzer directly in the game.

The state abstraction would simply be a menu of check boxes in which the user can select what to track. Since the game mechanics are the same for all scenarios created with the toolset the state abstraction can also be standard. Start with state variables for the hero and add state variables for each monster. Then the user could point out with the click of a mouse useful locations or objects that the hero could go to or manipulate and add those to the abstraction. Then the user invokes the Game Analyzer. The results come back in an easy to use visualization tool and the user can see the analysis of his labor.

This tool could then be available to the public and the multitude of people already designing scenarios for Neverwinter Nights could benefit from it. Of course a more detailed tool could be made available only to the game designers. This tool could enable the tracking of all sorts of variables like gold accumulation or total time played.

6.4 Concluding Statement

Human game testers are a necessity for creating new and exciting video games. The goal of the Game Analyzer is not to replace these game testers but to enhance their ability to test the games. The speed and power of the Game Analyzer could reduce the development time of future video games drastically. The information provided by the Game Analyzer can be used by the game developers to enhance the games they create. In the end, with the Game Analyzer lies the promise of better video games.

Appendix A

Policy Estimation

The algorithms described in this appendix are used to estimate how many policies, $\#(N)$, can be found below a single node N. To obtain an estimate of the number policies represented by the CCTree compute $\#(Root)$.

The Model Builder normally requests an estimate of the number of policies, $\#(N, A)$, taking a single action, A, from a choice node N. This is found by estimating all policies below the chance node that results from this action. The pseudocode for the initial estimation can be found in Table A.1.

```
Start at the root of the CCTree
Recursively move down the tree
  Stopping condition: leaf node $L$
    Set $\#(L) = 1$ and return $\#(L)$
  Recursive condition 1: chance node $N$
    Recurse down all children, $C_i$, to compute $\#(C_i)$
    $\#(N) = \prod_i \#(C_i)$
    Return $\#(N)$
  Recursive condition 2: choice node $N$
    Recurse down all children, $C_i$, to compute $\#(C_i)$
    $\#(N) = \sum_i \#(C_i)$
    Return $\#(N)$
End Recursion
```

Table A.1: Number of Policies Estimation Algorithm

The initial estimate is stored in each node to speed up future calculations. When a policy containing duplicate states is found the $\#(N)$ and $\#(N, A)$ are revised for all the common ancestors of the duplicate states. In this way the estimates are reduced to closer reflect reality. The pseudocode for doing this revision is found in Table A.2. Note that if a common parent is found, the common parent will always be a chance node.

```
For each abstract state $X$, encountered during the current sample do
  Let $S_X = (N_i, E_i = 1, \forall A : \#(N_i, A))$
  where $N_i$ are all nodes whose state is $X$,
      $E_i$ is the "multiplying factor" associated with $N_i$,
      and $\#(N_i, A)$ is the number of policies following action $A$ at node $N_i$
  While $|S_X| > 1$ repeat
    Find the set of nodes $S$ in $S_X$ at the greatest depth
    Check if any of the nodes in $S$ have a common parent
    If there is no common parent then for each $N_i$ in $S$
      Remove $(N_i, E_i, \forall A : \#(N_i, A))$ from $S_X$
      Add $(P_i, E_i * \#(P_i)/\#(N_i), \forall A : \#(N_i, A))$ to $S_X$ where $P_i$ is the parent of $N_i$
      If $P_i$ is a choice node update $\#(P_i) = \sum_j \#(C_j)$ where $C_j$ is a child of $P_i$
      If $P_i$ is a chance node update $\#(P_i) = \prod_j \#(C_j)$ where $C_j$ is a child of $P_i$
    End If
    If nodes $N_a$ and $N_b$ have a common parent $N_P$ then
      Remove $(N_a, E_a, \forall A : \#(N_a, A))$ from $S_X$
      Remove $(N_b, E_b, \forall A : \#(N_b, A))$ from $S_X$
      Add $(N_P, 1, \forall A : \#(N_P, A) = Ea * \#(N_a, A) * E_b * \#(N_b, A))$ to $S_X$,
      Update $\#(N_P) = \sum_A \#(N_P, A)$
    End If
  End While
End For
```

Table A.2: Estimate Reduction Algorithm

Bibliography

[1] M. Buro. ORTS: A hack-free RTS game environment. In *International Computers and Games Conference*, pages 85–99, 2002.

[2] B. Chan, J. Denzinger, D. Gates, K. Loose, and J. Buchanan. Evolutionary behavior testing of commercial computer games. In *Proceedings of the 2004 Congress on Evolutionary Computation*, pages 125–132, 2004.

[3] T.J. Cheatham, J.P. Yoo, and N.J. Wahl. Software testing: a machine learning experiment. In *Proceedings of the 23rd Annual Conference on Computer Science*, pages 135–141, 1995.

[4] H. Chu and J. Dobson. A statistics-based framework for automated software testing, 1996.

[5] J. Denzinger, K. Loose, D. Gates, and J. Buchanan. Dealing with parameterized actions in behavior testing of commercial computer games. In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games*, pages 51–58, 2005.

[6] A. Kovarsky. Heuristic search applied to abstract combat scenarios. Master's thesis, University of Alberta, 2004.

[7] M. Molineaux and D.W. Aha. TIELT: A testbed for gaming environments. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*. PA: AAAI Press, 2005.

[8] J. Poore and C. Trammell. *Cleanroom Software Engineering: A Reader*. Blackwell Publishers Inc., 1996.

[9] A.A. Porter and R.W. Shelby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, 1990.

[10] S. Russell and P. Norvig. *Artificial Intelligence A Modern Approach*. Pearson Education, Inc., 2003.

[11] F. Southey, G. Xiao, R. Holte, M. Trommelen, and J. Buchanan. Semi-automated gameplay analysis by machine learning. In *Proceedings of the 2005 Conference on Artificial Intelligence in Interactive Digital Entertainment*, pages 123–128, 2005.

[12] R. Sutton and A. Barto. *Reinforcement Learning*. The MIT Press, 1998.

[13] J.A. Whittaker and M.G.Thomason. A Markov chain model for statistical software testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, 1994.

[14] G. Xiao, F. Southey, R. Holte, and D. Wilkinson. Software testing by active learning for commercial games. In *Proceedings of the 20th National Conference on Artificial Intelligence*, pages 783–788. PA: AAAI Press, 2005.