LEARNING-BASED PROCEDURAL CONTENT GENERATION

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2014

By
Jonathan Roberts
School of Computer Science

Contents

Abstract **10**

Declaration **11**

Copyright **12**

Acknowledgements **13**

1 Introduction **14**

1.1 Context and Motivation 14

1.2 Contributions 18

1.3 Thesis structure 19

1.4 Glossary 19

1.5 Nomenclature 23

2 Machine Learning **25**

2.1 Fundamentals 26

2.2 Classification and regression 27

2.3 Supervised learning algorithms 28

2.3.1 Assessing accuracy 28

2.3.2 Artificial Neural Networks 32

2.3.3 Ensembles 33

2.3.4 Support Vector Machines 34

2.3.5 Decision Trees 35

2.3.6 Random Forests 37

2.3.7 Expectation-maximization 37

2.3.8 Active learning 38

2.4 Unsupervised learning algorithms 42

2.4.1 Dissimilarity measures . . . 42
2.4.2 k-means and k-medoids . . . 44
2.4.3 Principal Component Analysis . . . 44
2.5 Evolutionary Algorithms . . . 46
2.6 Crowd Sourcing and Machine Learning . . . 48
2.7 Conclusions . . . 49

3 Procedural Content Generation 50
3.1 Video Games . . . 50
3.1.1 Game Categorization . . . 50
3.1.2 Game Description . . . 53
3.2 Modelling players in games . . . 56
3.2.1 What is fun? . . . 57
3.2.2 Modelling using play-log features . . . 58
3.2.3 Modelling using content creation parameters . . . 59
3.2.4 Categorizing players . . . 60
3.2.5 Beyond play-logs . . . 61
3.2.6 Summary . . . 62
3.3 Procedural Content Generation . . . 62
3.3.1 Traditional Procedural Content Generation . . . 63
3.3.2 Non-adaptive SBPCG . . . 69
3.3.3 Rule-based player-adaptive SBPCG . . . 77
3.3.4 Learning-based player-adaptive Algorithms . . . 78
3.4 Togelius et al's Open Questions . . . 80
3.5 Summary . . . 83

4 Learning-Based Procedural Content Generation 85
4.1 Introduction . . . 85
4.2 Motivation . . . 86
4.3 Framework . . . 90
4.4 Enabling techniques . . . 94
4.4.1 ICQ . . . 94
4.4.2 CC . . . 96
4.4.3 GPE . . . 98
4.4.4 PDC . . . 100
4.4.5 IP . . . 101

5 LBPCG-Quake **106**
5.1 Introduction . . . 106
5.2 Quake . . . 107
5.3 OBLIGE . . . 109
5.4 Challenges . . . 113
5.5 Data types . . . 114
5.6 Data Collection . . . 115
5.7 LBPCG-Quake models . . . 116
5.7.1 Quake-ICQ . . . 116
5.7.2 Quake-CC . . . 118
5.7.3 Quake-GPE . . . 119
5.7.4 Quake-PDC . . . 119
5.7.5 Quake-IP . . . 120
5.8 Results . . . 120
5.8.1 Quake-ICQ results . . . 121
5.8.2 Quake-CC results . . . 124
5.8.3 Quake-GPE results . . . 132
5.8.4 Quake-PDC results . . . 137
5.8.5 Quake-IP results . . . 143
5.9 Summary . . . 156

6 Concluding Remarks **158**
6.1 Summary . . . 158
6.2 Limitations . . . 161
6.2.1 LBPCG limitations . . . 161
6.2.2 LBPCG-Quake limitations . . . 162

Bibliography **164**

A OBLIGE parameters **173**

B Quake Play-log format **177**

C LBPCG Software Project **180**

D Play-log feature rank table **187**

E IP Evaluation Data **192**
E.1 Player 8 192
E.1.1 IP State Transitions (Player 8) 192
E.2 Player 10 193
E.2.1 IP State Transitions (Player 10) 193
E.3 Player 11 194
E.3.1 IP State Transitions (Player 11) 194
E.4 Player 13 195
E.4.1 IP State Transitions (Player 13) 195
E.5 Player 15 196
E.5.1 IP State Transitions (Player 15) 196
E.6 Player 16 197
E.6.1 IP State Transitions (Player 16) 197
E.7 Player 18 198
E.7.1 IP State Transitions (Player 18) 198
E.8 Player 19 199
E.8.1 IP State Transitions (Player 19) 200
E.9 Player 20 200
E.9.1 IP State Transitions (Player 20) 201
E.10 Player 21 201
E.10.1 IP State Transitions (Player 21) 202

Word Count: 44653

List of Tables

1.2 Mathematical Notation. 24

5.1 OBLIGE/Quake parameters. 112
5.2 Quake-ICQ False Positive. 123
5.3 Quake-ICQ False Negative. 124
5.4 Rules for the Quake-MCC Model. 125
5.5 PDC Model Feature importance (Top 10) 142
5.6 Mean/Standard Deviation of play-log attributes for binary fun . . 142
5.7 IP Evaluation Survey questions and available responses 144
5.8 IP Evaluation self-assessment 144
5.9 Quake-IP PRODUCE state analysis 154

D.1 PDC Model Feature importance 191

E.1 IP State Transitions (Player 8) 193
E.2 IP State Transitions (Player 10) 194
E.3 IP State Transitions (Player 11) 195
E.4 IP State Transitions (Player 13) 196
E.5 IP State Transitions (Player 15) 197
E.6 IP State Transitions (Player 16) 198
E.7 IP State Transitions (Player 18) 199
E.8 IP State Transitions (Player 19) 200
E.9 IP State Transitions (Player 20) 201
E.10 IP State Transitions (Player 21) 202

List of Figures

1.1 Typical 1970s game: Pong (1972). 15
1.2 Typical 1980s game: Treasure Island Dizzy (1988). 15
1.3 Typical 1990s game: Mr Nutz (1993). 16
1.4 Typical 2000s game: Oblivion (2006). 16
1.5 A recent game: Skyrim (2011). 17

2.1 Pool-based Active learning (taken from [66]). 39

3.1 Gunn's taxonomy [28]. 53
3.2 A terrain generated with a fractal algorithm. 66
3.3 Automatically generated buildings. [27] 67
3.4 CGA recreation of Pompeii [23]. 68
3.5 Game map (background) generated from simple image (top left) [64]. 68
3.6 An automatically generated world in Charbitat. [55] 70
3.7 Terrain using GA [56]. 70
3.8 Evolved Tracks 1 [79]. 72
3.9 Evolved Tracks 2 [79]. 72
3.10 An evolved Pac-man like game [81]. 74
3.11 Galactic Arms Race [30]. 78

4.1 Content Generators. 87
4.2 Learning-based procedure content generation (LBPCG) framework. 91
4.3 The CATEGORIZE state in the IP model. 103
4.4 The PRODUCE state in the IP model. 104
4.5 The GENERALIZE state in the IP model. 105

5.1 Quake monsters. 108
5.2 The classic first-person shooter Quake. 109

5.3 GUI for OBLIGE. 110
5.4 LBPCG-Quake Client Server system. 116
5.5 Active Learning Error Plot 121
5.6 Active Learning F-Measure Plot 122
5.7 CC Test discovery 126
5.10 Quake-CC confusion matrix. 128
5.8 CC average error on Validation Set 128
5.9 CC per-class error on Validation Set 128
5.11 CC Learning discovery 129
5.12 CC error rates while sweeping reject threshold 130
5.13 CC average error 130
5.14 CC rejection rates 131
5.15 t-SNE plot for games selected using active learning. Key: Very Easy (Red), Easy (Green), Moderate (Blue), Hard (Yellow), Very Hard (Orange). 133
5.16 t-SNE plot for all MCC classified games. Key: Very Easy (Red), Easy (Green), Moderate (Blue), Hard (Yellow), Very Hard (Orange). 133
5.17 Binary Fun. 134
5.18 Ordinal Fun. 135
5.19 Ordinal Fun Distribution. 135
5.20 Binary Fun Distribution. 136
5.21 Binary Fun Deviation Distribution. 136
5.22 Remaining users as alpha/beta threshold swept. 138
5.23 Remaining surveys as alpha/beta threshold swept. 138
5.24 PDC Error on Test set vs Class Threshold. 139
5.25 PDC Error on Test set vs Reject Threshold. 140
5.26 PDC Rejection Rates on Test set vs Reject Threshold. 140
5.27 IP Evaluation: What kind of gamer would you describe yourself as? 145
5.28 IP Evaluation: How would you rate your skill level at action video games? 145
5.29 Binary Fun Metric for Quake-IP Evaluation 147
5.30 Simple Metric for Quake-IP Evaluation 147
5.31 Advanced Metric for Quake-IP Evaluation 148
5.32 Friedman test for binary metric (MedCalc). 149
5.33 Friedman test for simple metric (MedCalc). 150

5.34 Friedman test for advanced metric (MedCalc). 151
5.35 PDC Error on IP Evaluation Surveys vs rejection threshold 155
5.36 PDC Rejection Rates on IP Evaluation Surveys vs rejection threshold 155
5.37 PDC Error on IP Evaluation Surveys vs class threshold 156

C.1 LSP Libraries. 180
C.2 LSP Executables. 181
C.3 LSP Executables. 182
C.4 LSP: UtilLib 183
C.5 LSP: MachineLearningLib 184
C.6 LSP: GameSurveyLib 185
C.7 LSP: LbpcgLib 186
C.8 LSP: QuakePluginLib 186

Abstract

Learning-Based Procedural Content Generation
Jonathan Roberts
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2014

Procedural Content Generation (PCG) has become one of the hottest topics in Computational Intelligence and Artificial Intelligence (AI) game research in the past few years. PCG is the process of automatically creating content for video games, rather by hand, and can offer great benefits for video games companies by helping to bring costs down and quality up. By guiding the process with AI it can be enhanced further and even be made to personalize content for target players. Among the current research into PCG, search-based approaches overwhelmingly dominate. While search-based algorithms have been shown to have great promise and produce several success stories there are a number of open challenges remaining. In this thesis, we present the Learning-Based Procedural Content Generation (LBPCG) framework, which is an alternative, novel approach designed to address some of these challenges. The major difference between the LBPCG framework and contemporary approaches is that the LBPCG is designed to learn about the problem space, freeing itself from the necessity for hard-coded information by the game developers. In this thesis we apply the LBPCG to a concrete example, the classic first-person shooter Quake, and present results showing the potential of the framework in generating quality content.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor Ke Chen for his contributions to this thesis and his support during my PhD. I would also like to thank my mother Gillian, father Colin, sister Emma and partner Ieva for their support during my PhD, as well as my former house-mate Adam, who provided many interesting discussions of machine learning. Finally, I'd like to give thanks to all those who participated in the Quake experiments.

Chapter 1

Introduction

In this thesis, we introduce a new framework for Procedural Content Generation (PCG) in video games. In Section 1.1 we will introduce the context and motivation of this work, in Section 1.2 we will state the contributions, in Section 1.3 we will give the structure for this document and in Section 1.5 we will provide mathematical notation that is used in subsequent sections.

1.1 Context and Motivation

The video games industry originally catered to a relatively small hobbyist crowd and has drastically grown into a global phenomenon since. The estimated global revenue for the video games industry was $60.4 billion in 2009, which is expected to rise to $70.1 billion in 2015 [57]. The video games industry has already surpassed the size of the movie industry [85]. Increases in computational power and the wide-spread availability of high-end graphics hardware have resulted in a demand for video games with much deeper game-play and realistic graphics. This in turn has increased the complexity of game development. Many early games such as Space Invaders (1978), Pacman (1980) and Pong (1972) were constructed using relatively basic game worlds consisting of a finite state-space of possible movements. 2D sprites were used for graphical representation and AI was hard coded to control enemy behaviour. In contrast, modern video games such as Grand Theft Auto V (2013), Just Cause 2 (2010) and Skyrim (2011) consist of large continuous 3D worlds with cutting edge rendering techniques, advanced physics engines and increasingly life-like AI. Figure 1.1, Figure 1.2, Figure 1.3, Figure 1.4 and Figure 1.5 show the progress of games since the 1970s. Large sums

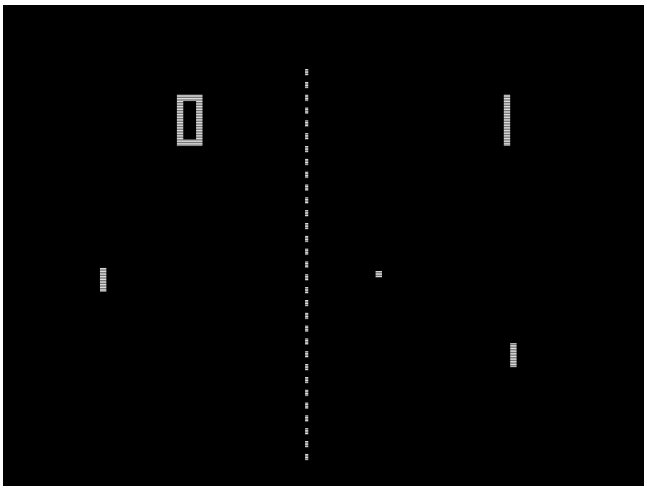

Figure 1.1: Typical 1970s game: Pong (1972).

Figure 1.2: Typical 1980s game: Treasure Island Dizzy (1988).

of money and personnel are required for the development of new games and it is estimated that only 20% of games that enter production turn a profit [63]. This means that while video games development can be very lucrative, it is also highly risky. As such, anything that can assist the games development process, be it a new high-level work-flow or a new software tool, is of high value.

Most cutting edge video games are driven by a *game engine* written by *game programmers*, usually in an object orientated language such as C++. Most games development companies don't create their own game engines, rather they pay a license fee for an existing commercial game engine and modify it to suit their own needs. Example game engines are *Source*, *Unreal*, *CryEngine* and *Unity*. The engine loads and interacts with *game content*, which is created by *artists* and *game designers*. Artists have the job of producing *art assets*, and examples

Figure 1.3: Typical 1990s game: Mr Nutz (1993).

Figure 1.4: Typical 2000s game: Oblivion (2006).

Figure 1.5: A recent game: Skyrim (2011).

include: (a) 3D meshes representing objects such as weapons, vehicles and creatures and structure of the world such as terrain and buildings, (b) animations for characters, (c) sound effects and music and (d) particle effects representing events such as explosions, dust clouds and magical spells. Most game engines come with high-level tools that allow for the creation of content without any programming knowledge.

Game designers specify and implement the *game-play features* of the game. They are responsible for describing the purpose of the game, how the player interacts with it and tuning its level of challenge. They construct the game world by placing art assets and design the storyline and challenges that need to be overcome by the player. It is also common for game designers to construct the AI for non-player characters (NPC) in the game, although this is sometimes a job performed alongside or solely by the game programmers. Games companies also employ a number of internal testers that continually test a game as it proceeds through the production process. It is becoming increasingly common, especially in multi-player online games, for companies to utilize *open beta tests*, which allow members of the public to test the game before release. These members of the public, known as *beta testers*, provide feedback and bug reports for the developers.

Throughout this thesis, *game content* will be terminology used to denote both art assets and game-play features. Every game has unique requirements on content, for example, a space exploration game requires ship models, particle effects and sound, but may not necessarily require character animations or buildings. The creation of game content by humans is both costly and time consuming as

it requires a diverse range of personnel such as software engineers, sound technicians, producers, concept artists and 3D modellers. Artists usually make use of model and animation software such as *3D Studio Max* and *Maya*.

Procedural Content Generation (PCG) is the process of constructing video game content using algorithms, as opposed to generating content by hand. PCG is not restricted to creating content before a game released, it can also be used after a game has been shipped to provide players with a unlimited supply of content. While it has been argued that PCG has a key role to play in the video game content creation pipeline in the future, PCG is not an easy task to perform [18]. Unconstrained PCG can result in undesirable results that are neither playable nor satisfying for the user. An article by Joe Ludwig of Flying Lab Software [47] discusses some of the pitfalls they encountered while attempting to integrate PCG into Pirates of the Burning Sea (2008).

Machine learning is a field that constructs algorithms that have the ability to learn on their own with limited human interaction. For example, rather than programming a robot to navigate an environment, one can instead use a *reenforcement learning* algorithm, such that the robot would learn behaviour by being rewarded and penalized by a human trainer. The most important feature of a machine learning algorithm is that it has the ability to solve problems beyond those that it has been presented with during training. As with most traditional algorithmic approaches, existing PCG methods can be viewed as constrained as they only operate within the bounds a human defines. As such, machine learning is something that, theoretically, can act as a controller for a PCG algorithm allowing it to produce unexpected but desirable game content

1.2 Contributions

In this thesis, our main contributions are summarized as follows:

1. We propose a novel framework for PCG in video games, termed the Learning-Based Procedural Content Generation (LBPCG) framework, which is data-driven in nature and aims to overcome weaknesses in existing PCG techniques.

2. We develop enabling techniques to support the proposed LBPCG framework.

3. For a proof of concept, we apply the LBPCG framework to the classic first-person shooter game, Quake, leading to a prototype that verifies high quality content is produced, based on our experiments.

1.3 Thesis structure

This thesis is organized as follows. In Chapter 2 we introduce basic Machine Learning techniques that will be critical to the rest of this thesis. In Chapter 3 we will describe basic video games concepts, such as categorization and description and then move on to review cutting edge techniques in player modelling and PCG, critically analysing the flaws in the current approaches. In Chapter 4 we introduce the LBPCG framework, covering the justification for its approach and the enabling techniques that are used. In Chapter 5 we detail how the LBPCG was applied to the video game Quake, using the level generator OBLIGE and provide results showing its success. Finally, in Chapter 6 we conclude by summarizing the contributions of this thesis and further work that could be done in the future.

1.4 Glossary

The following table gives a glossary of terminology used throughout this thesis.

Term	Description
Active Learning	A learning algorithm that interactively queries the user to gain knowledge of the problem domain.
Alpha (Sensitivity)	Rate at which positive examples are identified as such.
Artificial Intelligence (AI)	The field of creating human-like intelligence using computers.
Artificial Neural Network (ANN)	A form of machine learning based on creating models using brain-like neuron components.
Balanced Model	A Model used to evaluate the LBPCG which selects equal amounts of games from each category.
Beta (Specificity)	Rate at which negative examples are identified as such.
Beta Test	A phase of game development when the product is given to testers for evaluation.
	Continued on next page

Table 1.1 – continued from previous page

Term	Description
Category	A class that content can be mapped to.
CATEGORIZE State	A state the IP Model can enter such that it attempts to identify the player with a category of content.
Cellular Automata	A PCG method based on manipulating grids of cells.
Classifier Model	A model that, when presented with an input value, predicts its class.
Content	An entity that can be loaded into a video game and experienced by the player.
Content Categorization (CC) model	A component of the LBPCG that maps a Content Description Object to a Category.
Content Description Object	An abstract representation of content which can be passed into a Content Generator producing content.
Content Generator	Software that receives a Content Description Object and produces content.
Content Space	The space of all possible content for a Content Generator.
(Shannon) Entropy	The measure of information contained in a message.
Decision Tree	A procedural structure where each node represents a question and each sub-node represents the path that should be followed based on the response to the question.
Ensemble	A learning algorithm that makes use of multiple weak models.
Expectation Maximization (EM)	An iterative machine learning algorithm that optimizes it parameters based on estimated values for hidden variables.
Experience-Driven PCG (EDPCG)	A framework for PCG with emphasis placed on optimizing the experience of the player.
Evolutionary Algorithms (EA)	Optimization algorithm that uses the principles of evolution and manipulates the problem variables directly.

Continued on next page

Table 1.1 – continued from previous page

Term	Description
First-person Shooter (FPS)	A type of game controlled in first-person view where the player fights NPCs or other players, typically with guns.
Fractal Algorithm	A PCG algorithm based on the principle that recursively smaller parts of the content being generated are related in structure to the content as a whole.
Game	When used in terms of the Quake-LBPCG, a game is a level that be generated a played in Quake i.e. a type of content. In more general usage, "game" could also refer to "video game", i.e. software that be loaded onto a computer and played.
Game Description Language (GDL)	A language that be used to describe a video game.
General Game Playing (GGP)	The field of constructing agents that are designed to play multiple games, usually described using a GDL.
General Player Experience (GPE) model	A component of the LBPCG that captures public opinion on a fixed set of games.
GENERALIZE State	A state that the IP Model can enter wherein it generates non-personalized content.
Genetic Algorithm (GA)	Optimization algorithm that uses the principles of evolution by manipulates bit string representations of the problem.
Genetic Programming (GP)	Algorithm that uses the principles of evolution and manipulates parse trees representing programming languages.
Individual Preference (IP) model	A component of the LBPCG framework that attempts to personalize content for a target player by determining their categorical preference.
Initial Content Quality (ICQ) model	A component of the LBPCG framework that filters unacceptable content.
k-means/k-medoids	Two algorithms that look for clusters of points in data sets.

Continued on next page

Table 1.1 – continued from previous page

Term	Description
k-Nearest Neighbour (KNN)	A classifier that operates by looking for the closest known example to an input point.
Learning-Based PCG (LBPCG)	A data-driven PCG framework that learns how to categorize content and match target players to content interactively.
Machine Learning	The field of constructing algorithms that have the ability to learn on their own with limited human interaction.
Multi-Layer Perceptron (MLP)	An ANN with multiple hidden layers.
Non-player character (NPC)	A computer controlled character in a game.
OBLIGE	An open source Content Generator for the FPS game Quake.
Perlin Noise Function	A function that attempts to mimic the patterns experienced in nature.
Player Experience Modelling (PEM)	The act of modelling the player's experience as a function of game content and the player's playing style [90].
Principle Components Analysis (PCA)	A method that identifies patterns in high-dimensional data and facilitates the projection of data to a lower dimension.
PRODUCE state	A state the IP Model can enter wherein it produces content from a specific category for the target player.
Play-log	A vector summarizing the events that occurred when player experienced an item of content.
Play-log Driven Categorization (PDC) model	A component of the LBPCG that identifies whether a player had fun or not with an item of content based on a play-log.
Procedural Content Generation (PCG)	The automated generation of content for video games.
QRACK	A fork of Quake with extra graphics effects, which we use on Microsoft Windows.

Continued on next page

Table 1.1 – continued from previous page

Term	Description
Quake	An open source FPS game.
QuakeSpasm	A fork of Quake, which we use on Linux.
Random Forest	A classifier that uses an ensemble of decision trees.
Random Model	A Model used to evaluate the LBPCG which selects completely random games from the Content Space.
Regression Model	A prediction model that outputs a continuous or ordered whole value.
Search-Based PCG (SBPCG)	A family of PCG algorithms, with the general approach being to generate and then evaluate content.
Single-Layer Perceptron (SLP)	An ANN with a single hidden layer.
Support Vector Machine (SVM)	A classification/regression model that solves problems by forming a hyper-plane in the input space separating the training examples into their respective classes.
t-Distributed Stochastic Neighbour Embedding (t-SNE)	A state-of-the-art dimensionality reduction algorithm.

1.5 Nomenclature

Table 1.2 lists the mathematical notation used in Chapter 4 and Chapter 5 where we formally introduce the LBPCG and the test implementation for Quake.

D	The number of parameters in the content description vector
$\mathcal{G}$	The content space
Φ_{ICQ}	ICQ model
T_{ICQ}	Validation set for the ICQ Model
F_i	Content feature i
$\mathcal{F}$	The set of all content features
Φ_{CC}	CC model
$\mathcal{C}$	The set of all categories $(\cup_{i=1}^{\lvert\mathcal{F}\rvert} F_i)$.
T_{CC}	CC Model validation set
$\mathcal{G}_a$	Acceptable content in $\mathcal{G}$
Φ_{GPE}	The GPE model
$\mathcal{G}_{GPE}$	Content chosen to give to the beta-testers during public test phase
P	The total number of beta-testers in the public test stage
$y_n^{(p)}$	Feedback received from beta-tester p for content $\mathcal{G}_{GPE}^n$
$\alpha^{(p)}$,$\beta^{(p)}$	Sensitivity and specificity for beta-tester p
$\hat{y}_n$	The predicted quality of $\mathcal{G}_{GPE}^n$
$\mathcal{L}$	Space of all play-logs
Φ_{PDC}	The PDC model
O	Total number of play-logs available for training the PDC model
$\mathcal{D}_{PDC}$	Training set for PDC model
M	Number of weak learners used for PDC ensemble
$\mathcal{D}_{PDC}^m$	Training set for weaker learner m in PDC ensemble

Table 1.2: Mathematical Notation.

Chapter 2

Machine Learning

Machine learning is the development of intelligent algorithms that learn from data and then have the ability to generalize, that is, provide predictions on data that hasn't been seen before. In Chapter 3 we will review many papers which make use of machine learning for the purposes of procedural content generation, and indeed our own framework described in Chapter 4 makes extensive use of it. As such we will provide a review of the relevant literature in this chapter.

An example problem that machine learning can solve is hand-written character recognition. People write letters in a variety of ways, even though the core structure of each letter is the same. Machine learning algorithms could be deployed in such a situation by training them on a number of available examples. If sufficient training data has been provided, then the algorithm should be able to accurately predict what letter is represented in examples it has not seen before.

There are a variety of categories of machine learning algorithm that are used in the PCG literature and that we make use of in the LBPCG framework. In this section we will review them. In Section 2.3, we will describe *supervised algorithms* that learn from data that has been labelled. In Section 2.4 we will review *unsupervised algorithms*, such as clustering, which learn from data in the absence of labels. In Section 2.5 we will look at evolutionary algorithms and finally in Section 2.6 we will briefly review crowd sourcing-related machine learning algorithms.

Further information on the algorithms covered in this section can be found in most machine learning books [6] [53].

2.1 Fundamentals

From a Bayesian standpoint, Machine Learning algorithms are governed by the laws of probability and statistics. This is a very powerful abstraction as it allows for the analysis and comparison of algorithms in a theoretical manner. This section will introduce the main elements of the theoretical Bayesian view. The probabilistic features of particular algorithms will be covered as they are introduced in later sections.

A *random variable* X is the result of an observation. The value that a random variable takes is determined by a *probability distribution*. The notation $Pr(X = v)$ denotes the probability that the variable X will take the value v. A *discrete* random variable is one which can take a finite number of values i.e. labels. The probability distribution for discrete random variable X is characterized by a *probability mass function* of the form $P(X)$. In this case $Pr(X = v) = P(X = v)$. A *continuous* random variable Y is one that can take on an infinite number of values. It is characterized by a *probability density function* of the form $p(Y)$. The sum of the probability of all values in the distribution sums to one i.e. $\sum_{v \in Vals(Y)} p(v) = 1$ and the probability for any specific value is zero i.e. $\forall_{v \in Vals(Y)} p(Y = v) = 0$. The probability density function can be used to find the probability of a range of values by taking the integral between the two endpoints of the range as such:

$$Pr(Y \in [a, b]) = \int_a^b p(X)dx$$

Two random variables X and Y are said to be conditionally independent if $Pr(X|Y) = Pr(X)$ and $Pr(Y|X) = Pr(Y)$. X is said to be conditionally independent of Y given Z if $Pr(X|Y, Z) = Pr(X|Z)$. This can be generalized by saying that X is conditionally independent of a set of variables given another set of variables i.e. $Pr(X|Y_1, ...Y_n, Z_1, ...Z_n) = Pr(X|Z_1, ..., Z_n)$.

In general terms, one can view the goal of machine learning to be to find the highest probability hypothesis h from a set of hypotheses H given observed data D. The prior probability that a hypothesis is true, $Pr(h)$, reflects our certainty that h is the correct hypothesis before any data is observed. In formal terms, we seek $h \in H$ that maximises the posterior probability $Pr(h|D)$. We call this hypothesis the *maximum a posteriori hypothesis*. A *maximum likelihood hypothesis* is that which maximises $Pr(D|h)$. *Bayes Theorem* breaks down $Pr(h|D)$ in the

following manner:

$$Pr(h|D) = \frac{Pr(D|h)Pr(h)}{Pr(D)}$$

If the prior probability of all hypotheses is uniform, then Bayes theorem states that the maximum a posteriori hypothesis is the maximum likelihood hypothesis.

When attempting to find a maximum likelihood hypothesis, it is likely that many small floating point values will be multiplied with each other. This means that values may be produced that cannot be handled by a computer. As such, many algorithms instead work with the *log-likelihood*, which is defined as $ln(Pr(D|h))$. If we are optimising the log-likelihood of a hypothesis then we are also optimising the likelihood of the hypothesis, since there is a monotonic relationship between a number and its log. Rather than multiplying probabilities e.g.$Pr(a) = Pr(b) * Pr(c)$, we instead add them as such $ln(Pr(a)) = ln(Pr(b)) + ln(Pr(c))$.

2.2 Classification and regression

Classification is the act of identifying members of a set of examples with one of a discrete set of categories. For example, given a vector of parameters describing an applicants previous credit history, a loan company may want to map this into one of two classes, indicating whether the applicant is reliable or not. More formally, a classification problem involves determining a function $\phi : A \rightarrow B$, where A is the set of all possible examples and B is the set of classes.

Some types of classifier that we will later describe not only have the ability to map an example to a category, but can also indicate the *posterior probability* of it lying in the category. For example, a classifier ϕ may indicate that input vector v has label x, with probability p. If the problem is binary, then the probability according to ϕ that v belongs to the opposing class to x is $1-p$. The probability can be used to indicate a level of certainty in the classifiers decision. A binary classifier that indicates the label for an input vector v resides in class x with probability 0.55, clearly isn't as confident in its decision as if the probability was 0.95. In this thesis we formulate this into a *confidence score*. In the binary classification example, we can say that given a probability p, the confidence of the classifier is $|p-(1-p)|$. If $p = 1$, then the confidence is 1, whereas if $p = 0.5$, then the confidence is 0.0. In binary classification, a class boundary is a threshold that determines what class a particular value belongs to i.e. if the value is below

the threshold then it belongs to the first class and if it is above it then it belongs to the second class. In binary classification it is possible to use the posterior probability as a boundary. Suppose we are constructing a classifier $\phi : A \rightarrow B$, where A is the set of all possible examples and B is the binary class set $\{c_1, c_2\}$. Also suppose that $p(a \in A, b \in B)$ gives the conditional probability, according to ϕ that a belongs to class b. One can assume that if $p(a_1, b_1) > 0.5$ then ϕ is telling us that a belongs to class b. In this case it is possible to treat the value 0.5 as a type of class boundary. The value of the boundary can be changed to any particular value we choose, and this may be useful in biasing the classifier towards one of the two classes, which may potentially increase its accuracy.

In addition to classification problems, *regression problems* are also encountered quite frequently in various fields. A regression problem attempts to map an input vector to a real value. An example regression problem would be where there exists a function $f : A \rightarrow \mathbb{R}$, where A is a set of input vectors describing the various attributes of a day and the range is a temperature in degrees Celsius. Our objective in this case is to be able to predict what the temperature may be given previous data acquired in the form of $(a \in A, b \in \mathbb{R})$.

2.3 Supervised learning algorithms

In this section we will review supervised learning, which includes algorithms that are trained using a *training set* of data consisting of ordered pairs of the form (x, y), where x is an example input and y is the output of a target function f for it. Once trained, such algorithms are expected to be able to predict the output of the target function. An example of a supervised learning algorithm in a video game would be providing a computer chess player with a number of board configurations and the next move a human player made with the aim of the algorithm learning how to mimic a particular player.

2.3.1 Assessing accuracy

As discussed, supervised learning algorithms usually train a classification or regression model using a training set, which provides a set of ordered pairs of example inputs and the desired outputs. After such a model has been trained it is desirable to get an estimate of how accurate it is likely to be on examples it hasn't seen before. One could assess the accuracy on the training set, but this

isn't a good approach as it may be the case that the model has *over-fitted* to the training set. This means that while it might capable of classifying examples in the set well it may not be able to generalize beyond them. It is therefore a good idea to have a separate *validation set*, which is disjoint from the training set. It is important that validation sets contain adequate amounts of examples, either from each class in a classification problem, or from a good range of values in a regression problem. Failure to do so may result in an *unbalanced* validation set, which may negatively affect the accuracy assessment. For example, having 1 sample from class A and 99 samples from class B in a binary classification problem will not really allow us to assess the accuracy of the model for class A. One technique for constructing validation sets is *n-fold cross-validation*. In n-fold cross-validation we assume the existence of a set of labelled instances. This set is divided into n disjoint subsets. We then choose $n - 1$ of these subsets for training the model, and then use the remaining 1 for validation. The process can be repeated by selecting different subsets to be used for validation.

In this section we will constrain ourselves to discussing the error rates of classification models only, which is the main type of model we will use in this thesis. Information on calculating error rates for regression models can be found in most machine learning hand-books such as [53].

In a binary classification problem, we can determine the *error rate* of a model on the validation set in a straightforward manner. This is simply the total number of examples the model classified incorrectly divided by the total number of examples. This, however, is quite a basic measure and it better to know the accuracy of the model individually on each class. A classifier that is 100% successfully at identifying members of class A, and 0% successful at identifying examples from class B may still have quite a respectable error rate depending on the distribution of the validation set, even though it is clearly not useful.

If in a binary classification problem we have two classes (i.e. a positive and negative class). We define a *true positive* as an positive example that was correctly identified and a *true negative* as a negative example that was correctly identified. We define *false positive* and *false negative* as incorrectly identified negative and positive examples, respectively. *Sensitivity* is defined as the number of true positives divided by the total number of positive examples. The *specificity* is the number of true negatives divided by the total number of negative examples. In some problems the sensitivity may be more important than the specificity or

vice-versa. For example, it is more important to identify that patients have a disease, even if we are sometimes wrong, rather than assume they are not when they do in fact have the disease.

It is often useful to analyse validation results using two scores: (a) *precision*, which is equivalent to the sensitivity, and (b) *recall*, which is defined as $tp/(tp + fn)$, where tp is the number of true positives and fn is the number of false negatives. A high precision indicates that the classifier returns mostly correct results, i.e. the examples it classifies as positive that are actually positive. A high recall indicates that the classifier returned most of the results it needed to. The *f-measure* is defined as the harmonic mean of the *precision* and *recall*, more specifically: $2 * ((p * r)/(p + r))$, where p and r are the precision and recall scores, respectively. The harmonic mean is used as it gives less significance to high-value outliers.

Several of the concepts for binary classification that we have discussed can be extended to multi-class classification in a straightforward manner, such as the overall and per-class error rates. In addition, it is often useful to construct a so-called *confusion matrix*, which has dimension $n * n$, where n is the number of classes. The element at row i and column j tells us the number of examples of class i that were classified as j. If our classifier is 100% accurate, then only the diagonal values will be non-zero. If our classes are ordinal (i.e. arranged in an order), then visualizing the matrix allows us to see clearly whether incorrect classifications are close to the desired class or not.

It is very common for researchers in the field of machine learning to compare their algorithms against competing algorithms on public data sets. One approach to this would be to calculate the average error rate of each algorithm on each data set and declare the winner to be the one with the lowest error rate. Demšar is very critical of this approach in a paper in which he examines submissions to the Internal Conference on Machine Learning (ICML) [14]. He states that one reason why this isn't a good idea is that the error rates between data sets may simply not be comparable with each other. Secondly, he states that taking the average may cover up extremely poor performance on a few data sets i.e. outliers. The standard approach to comparing results in scientific experiments is to use a *statistical hypothesis test*. The aim of such a test is to establish that the results seen are *statistically significant* in that they weren't the result of random chance, or in other words, to reject the *null-hypothesis* that states there

is no relationship between two measured phenomena. If the phenomena under scrutiny are the results from classifiers on a number of data sets, then this means that the classifiers perform to the same accuracy, or that more information is needed before a conclusion can be drawn.

One of the most common statistical tests is the *paired student's t-test* (or "t-test" for short). Demšar [14] states three criticisms for using the student's t-test for machine learning. Firstly, the test assumes that results on the different data sets are commensurate, in that the results have similar meaning on each set. This is also a problem that straight comparison of the average error has. Secondly, if the number of data sets isn't large enough, then the test requires that the data conforms to a normal distribution. In the case of the comparison of machine learning algorithms, this would mean that the reported accuracy for each model is distributed normally, something which Demšar claims in general cannot be guaranteed. We note that tests such as *Kolmogorov-Smirnoff* [32] and *Shapiro-Wilk* [71] can be used for testing whether a data set conforms to the normal distribution. The final criticism of the t-test is that may not be able to handle outliers correctly. As a result of these weaknesses with the t-test, Demšar [14] recommends that the *Wilcoxon signed-ranks test* (or "Wilcoxon test" for short) is used instead. Demšar argues that the Wilcoxon test ignores the magnitude of values when comparing error rates, which helps to alleviate the problem of commensurability between data sets, and it does not assume that the data conforms to a normal distribution. In addition, due to the ranking mechanism that it is based on, outliers have less of an effect that on the t-test. He does, however, note that the t-test can under certain circumstances be more powerful than the Wilcoxon test.

The Wilcoxon and t-test are useful when comparing the results from two algorithms across multiple data sets, however, in the later experiments performed in this thesis we will be comparing results from three different algorithms. In his paper on statistical testing for machine learning, Demšar examines two tests, namely *ANOVA* and the *Friedman test*. Demšar criticises ANOVA in a similar manner to the t-test, in that it assumes the data is drawn from a normal distribution. He also criticises ANOVA because it requires that a *spherecity* condition is met, which places a limitations on the variances between the results of tests. For these reasons, Demšar recommends the Friedman test instead, which avoids these issues.

The following definition of Friedman Test is based on Demšar's paper [14] and the book "Handbook of parametric and non-parametric statistical procedures" [72]. Let us assume we have k classifiers and want to determine if there is a significant difference in performance between each of k on a data sets D. Let r_i^j denote the *rank* of classifier j on data set i, ordered by descending accuracy. Let $R_j = \frac{1}{N}\Sigma_i r_i^j$ denote the *average rank* of algorithm j. The *Friedman statistic* is calculated as follows [72]:

$$X_F^2 = \frac{12}{Nk(k+1)}[\Sigma_{j=1}^{k}(\Sigma R_j)^2] - 3n(k+1)$$

If the null-hypothesis holds then each R_j should be statistically similar [14]. This can be tested by checking whether the Friedman statistic is less than a *critical chi-square* value [72]. If the test rejects the null-hypothesis, then several post-hoc tests, such as the *Nemenyi test* and *Bonferroni-Dunn test*, can be used to detect pair-wise statistical differences between each classifier.

2.3.2 Artificial Neural Networks

Artificial Neural Networks (ANN) [6] [53] were originally created with the idea of mimicking the biological processes of the brain. The structure of an ANN consists of a number of layers of neurons connected via weighted synapses. ANN are suitable for learning solutions to problems where the training data may be quite noisy, such as input sensors for real-world problems. The main issues with ANN are that they usually take a long time to train and that they often need to be regarded as black boxes as it is difficult to gain any information as to how a problem was solved by them. However, ANN are widely used in the video game research literature.

Each neuron in an ANN applies weights to its inputs and passes the sum through an *activation function*. Single Layer Perceptrons (SLP) are ANN consisting of a single layer of neurons with an output that thresholds its inputs into a boolean value (0 or 1). An SLP can represent linearly separable classification problems only, such as AND, OR, NAND and NOR logic gates.

A more capable type of ANN is the *Multi-Layer Perceptron* (MLP), which has multiple *hidden layers* consisting of one or more neurons. Unlike the SLP, it has the ability to solve non-linearly separable problems. It is important that the functions used for activation are differentiable as this enables the influence of each

weight on the error produced by activation function to be calculated. The popular *back-propagation* algorithm can then be used, which in broad terms, calculates the error of the network at the activation function and moves backwards through the nodes calculating to what proportion each node and each node's weights were responsible for the error, and adjusting them as appropriate.

Another type of network that is used for visualization is Kohonen's *Self Organizing Map* (SOM). A SOM is a ANN consisting of a number of layers of neurons. Each neuron has a vector of weights equal to the dimensions of the input data. A SOM can be thought of as a method for visualizing a multi-dimensional data set in a lower dimension. It is trained by presenting a sequence of data vectors to it. The neuron in the network with the most similar weight vector to the data vector has its weight vector modified to resemble the data vector more closely. This adjustment is also applied to adjacent neurons. As such, the SOM learns the topological structure of the input data set.

2.3.3 Ensembles

An *ensemble* [39] is a supervised learning algorithm that makes use of multiple models. The ensemble makes an overall prediction based on the individual predictions of each model. Ensembles are constructed such that each model is accurate, in that they have better than random prediction, and the models are diverse, in that they have a different error rate from each other on new data. There are a number of arguments in favour of ensembles. Statistically the ensemble can be seen as averaging the prediction of the individual models, making it more robust to error, provided the models have an appropriate minimum error rate. Individual models in the ensemble may be incapable of representing the true target function i.e. they may be stuck at local optima. As such, by combining multiple models seeded from different points an ensemble can produce a more accurate representation of the target function.

Bagging [6] is an ensemble method that constructs each of the models in the ensemble by randomly training sets for each based on the original training data. The selection function allows multiple occurrences of training examples to appear in each set. This means that some models are trained with a greater emphasis on certain training examples, which occur more than once, and less emphasis on other examples, which don't occur often or at all. Implicitly this means that some models are better at predicting certain training examples. *AdaBoost* is a

classifier ensemble algorithm that is similar to bagging in that it manipulates the training sets. It is iterative in that it builds each classifier in the ensemble in sequence based on the result of the previous iteration. The process starts by constructing a classifier based on the training set. Each training example that is incorrectly classified by the classifier is weighted higher, such that more emphasis is placed upon it. The weighted training set is used to train the next classifier and the process repeats. After all required classifiers have been constructed, each of them is weighted based on its success at classifying the original training set. The ensemble operates by querying each classifier and weighting its prediction.

2.3.4 Support Vector Machines

A *Support Vector Machine* (SVM) [6] is a model that solves classification problems by forming a hyperplane in the input space separating the training examples into their respective classes. A *Support Vector* is a sample that when removed from the training set results in a different hyper-plane. A *margin* in this context refers to the distance the hyperplane has from any two selected points of differing class in the input space. The goal is to form a hyper-plane that maximises the minimum margin between points of differing class, as this increases its accuracy. The reasons for this are that a hyperplane with a high margin on the training data will tend to separate the training examples better and examples that it has not seen are likely to be closer to the space of training examples of the same class.

The problem of classification cannot usually be solved using linear classification. Often the two classes cannot be separated using a hyperplane in the input space. As such, *kernel* methods are commonly used to translate the input data into a higher dimension, while maintaining the distance relationship between input samples in the higher dimension. It is in these higher dimensions that hyperplanes can be found to separate the data. Many different variations of kernels exist, which are suitable for different types of data.

In the subsequent chapters of this thesis we will make use of SVMs, with the extra requirement that they output posterior probabilities along with classification labels. However, SVMs do not output such a probability by default. Fortunately, the LIBSVM [12] library, which is a popular SVM implementation for C++, adds support for this. In a binary classification problem consisting of two classes C_1 and C_2 the posterior probability for C_1 can be written as such, where D is the input vector [6]:

$$Pr(C_1|D) = \frac{Pr(D|C_1)Pr(C_1)}{Pr(D|C_1)Pr(C_1) + Pr(D|C_2)Pr(C_2)} = \frac{1}{1 + exp(ln(\frac{Pr(D|C1)Pr(C_1)}{Pr(D|C_2Pr(C_2)}}$$

The function present in the derived equation, $\sigma X = \frac{1}{1+exp(-a)}$, is called the a *logistic sigmoid function*. Chang and Lin provide an explanation of how they implemented support for LIBSVM [11]. Their work is based on an approach suggested by Platt [62], which proposes that the posterior probability of an SVM can be approximated by a sigmoid function, and that the best fitting parameters for this can be determined by solving a regularized maximum likelihood problem. Lin et al go on to suggest several improvements that they have made to avoid several numerical difficulties. For pseudo-code detailing how the LIBSVM extension was implemented and further details please see [44].

2.3.5 Decision Trees

A Decision Tree [53] is a procedural structure where each node represents a question and each sub-node represents the path that should be followed based on what the response to the question is. The leaves of the tree represent the final decision of the process. Decision Trees can be used for both regression and classification problems. Given an input, the evaluation begins at the root node of the tree and progresses downwards on a path determined by what conditions are met. Decision Trees aren't specific to machine learning and are common in many different fields.

Decision Tree Learning is the process of constructing a decision tree by learning about the problem. Let D be a classification data set, with set of classes C and attributes A. Let $V(A)$ give all the values for the attribute A. The goal of Decision Tree Learning is to construct a Decision Tree, which takes an input I composed of values for each attribute. At each node the Decision Tree asks a question about one of the attribute values in I. The leaf nodes provides an answer in the form of a class $c \in C$ to the query I. There are many different algorithms for constructing trees, each with their own advantages. As an example, we will demonstrate the basic operation of the ID3 algorithm for classification tasks.

Entropy is the measure of uncertainty of a random variable or set of data. The entropy for $D' \subset D$ is defined as: $Entropy(D') = -\sum_{c \in C}(p(c)log_2 p(c)$, where $p(c)$ represents the probability of class c in D' i.e. how often it occurs. For

$v \in Values(a \in A)$, let D'_v represent the number of examples in D' where the value for attribute a is v. For Decision Tree Learning, we define the information gain for an attribute $a \in A$ with respect to D' as:

$$InfoGain(D', a) = Entropy(S) - \sum_{v \in Values(a)} ((|S_v|/|S|) * Entropy(S_v))$$

This metric tells us the reduction in entropy (i.e. decrease in uncertainty) that would be achieved if we were to know the value of a particular attribute. The ID3 algorithm is designed to construct a tree where at each node a question is asked about the attribute with the highest information to be gained. The recursive ID3 algorithm is described in Algorithm 2.1 and proceeds with $D' = D$ in the first iteration.

Algorithm 2.1. ID3 algorithm.

Purpose: To train a decision tree capable mapping input vectors to a class.
Input: (a) A training set D' of input values along with classification labels, (b) a set of attributes A of which each input vector is made up of.
Output: A decision tree.
if All examples in D' are of class c. **then**
  Create a leaf node with classification c.
else
  if $|D'| = 0$ **then**
    Create a leaf node with the most popular class c.
  else
    For the current data set D', find the attribute $a \in A$ with the highest information gain.
    Set the current node to ask the question "What is the value of a?".
    for all $v \in Values(a)$ **do**
      Create a sub-tree for the response to the question as v
      Recursively go to (1) with $D' = D_v$ to build the sub-tree.
    end for
  end if
end if

2.3.6 Random Forests

Random Forests [8] are a machine learning algorithm combining the techniques of decision trees and ensembles and have several advantages that make them particularly appealing, including the ability to report a so-called *out-of-bag* error, which is similar to a cross-validation error, and the ability to report what the most influential features are in the decision making process i.e. a form of feature selection. Random Forests can be viewed as a framework of sorts, where the particular implementation details of the algorithm can be swapped out for alternatives. The central concept is that a number of decision trees are trained on subsets of the training data and when new data is presented, each decision tree makes a decision and the mode is taken as the output classification. One approach is to first choose a number of trees, e.g. 5. For each tree, a percentage of the training data is chosen as the training set and the rest as the validation set. The tree is then trained on the training set, and then tested on the validation set producing the out-of-bag error, which is an estimate of the accuracy of the tree. It is also possible to rank the features of a data set based on importance, which involves calculating the out-of-bag error after training the random forest and then comparing it against the out-of-bag error after randomly permuting the values of each feature in turn in the training set. Random forests can be extended to output a probability value for a classification by calculating the rate at which all trees in the ensemble agree with the classification. More formally, let there exist a random forest consisting of n trees and a function $f_t(x, y)$ which gives the value 1 if the prediction of tree t for input x it y, or 0 otherwise. We can define a probability function $p(x, y)$ that gives the probability that example x has label y according to r as follows: $p(x, y) = (\Sigma_t f_t(x, y))/n$. This is also the method that the software library *librf* [42] uses, which we will make use of later in this thesis.

2.3.7 Expectation-maximization

Expectation Maximisation (EM) [53] is an algorithm designed to find the maximum likelihood estimates of parameters for a model using a data set where there are random variables affecting the data that cannot be observed. We assume we have a complete data set $D = \{(x_0, y_0), ..., (x_n, y_n)\}$ and that there is a set of parameters θ that influence the data. The set $X = \{x_0, ..., x_n\}$ is the incomplete set. We assume that we have access to X, but not D, and would like to determine

what both θ and D are. EM is quite elegant in its design, and in high-level terms it can be described using two steps shown in Algorithm 2.2.

Algorithm 2.2. Expectation Maximization.

1: **Purpose:** To estimate what the labels for a set of data are along with hidden parameters influencing them.
2: **Input:** An incomplete set of data X.
3: **Output:** (a) An estimation for hidden parameters θ, (b) An estimation of the complete set of data D.
4: Let θ_i represent the current estimate of the parameters
5: Assign θ_0 randomly
6: Let $i = 0$
7: **E-Step**: Given θ_i, guess what the values for $Y = \{y_0, ..., y_n\}$ are as Y_i'.
8: **M-Step**: Choose θ_{i+1} that maximizes the log-likelihood $Pr(Y_i'|X, \theta_{i+1})$
9: Go to **E-step** with $i = i + 1$

2.3.8 Active learning

Active Learning [66] is a semi-supervised machine learning method which can be used to build both classification and regression models. The input space for the problem that active learning tries to solve consists of a large number of unlabelled samples, which can be selected and then labelled by an oracle. The oracle could be a human, robot or even an experiment that needs to be performed. It is expected that the oracle is a limited resource and therefore examples from the search-space should be chosen in an intelligent manner, as to minimize the burden on the oracle. The general process for active learning algorithm is to use a strategy to choose the best members of the input space to label, have the oracle label these members and then add them as training examples for the model it is attempting to build. The process then repeats again until a termination condition has been reached. The process for *pool-based* active learning is summarized in Figure 2.1, which is a specific type of active learning that will be explained shortly. Settles [66] provides an in-depth literature review for active learning. In this section will summarize the important concepts from this paper and several others.

We have mentioned that the input space consists of a large number of unlabelled samples, from which the active learning algorithm can choose examples to be labelled. The manner in which the algorithm is allowed to sample the input space is very important. It may be that we are restricted in what samples we can choose as some may only be available at certain times, or there may be some cost

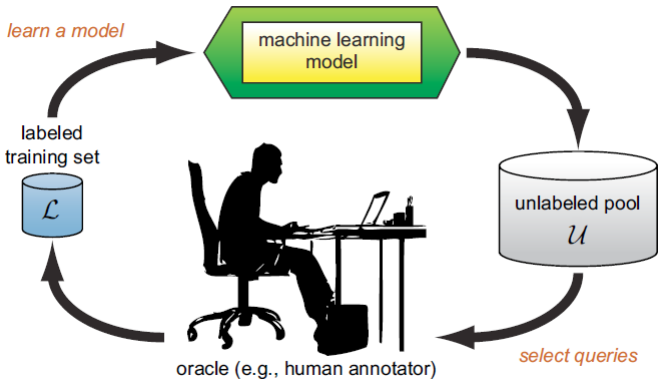

Figure 2.1: Pool-based Active learning (taken from [66]).

associated with retrieval. We will describe three sampling scenarios as identified by Settles [66]. In the *membership query* sampling scenario, the active learning process can request labels for any member of the input space. The active learner can synthesize the examples they wish to label, rather than be constrained to some distribution throughout the space. Settles [66] highlights one potential issue with this form of search if the oracle is a human, in that the learner could select an example which isn't particularly well-formed. He provides an example from Lang and Baum [40], in which the objective was to train a neural network to recognize hand-written characters. Since the learner was using membership query synthesis, it had a tendency to produce characters which were a mix of different letters. This was obviously potentially confusing for the oracles, and had the potential to train the neural network with invalid data. *Stream-based selective sampling* is another sampling scenario. Typically in this scenario the samples are provided to the learner in a sequence and the learner can choose whether or not to have the oracle label each sample it presented with. The important distinction between this and membership query synthesis is that the learner cannot arbitrarily choose samples from the input space, and its choice may be somewhat more constrained. The final form of sampling scenario is *pool-based sampling*, which is the most commonly used. It is also the scenario we will later encounter in the experiments in this thesis. In this scenario a large pool of samples are available and the learner may choose any to be labelled. Typically the members of the pool remain the same.

In the previous paragraph we have discussed the different types of sampling

ability an active learner may have. However, more important is perhaps the approach the learner takes in deciding which of the samples it has available to pass to the oracle for labelling. Settles [66] mentions several types of *query strategy*, of which we will describe four. The first strategy is termed *uncertainty sampling*, which is the approach we will take later on this thesis to solve several of the problems we encountered. In uncertainty sampling, the learner chooses the samples that it is the least sure about. In Section 2.2 we described how some types of classifier have the ability to output a probability along with any classification they make. We also described how this can be converted into a confidence value. One can immediately see that one straightforward approach to uncertainty sampling is to choose samples that the learner assigns the lowest confidence value to. Another approach is to make use of entropy, as described in Section 2.3.5. This approach is regarded as being better suited for problems that use multi-label classifiers. Entropy is a measure of how much information is needed to encoded a distribution and therefore it can considered to be a measure of uncertainty [66]. For binary classification problems, using the entropy for uncertainty sampling is equivalent to choosing examples with the lowest confidence [66]. *Query by committee* is another query strategy, in which multiple models are trained on the data. Those points which cause the most disagreement amongst the models are considered to be the best candidates to be given to the oracle for labelling. One could perceive of this approach as being useful in the case where the learner being used does not output probability, making it difficult to use an approach such as uncertainty sampling. *Expected model change* is a strategy that chooses to label points that would have the most effect on the model if the label was known and in a similar vein *expected error reduction* is a strategy that chooses points that are the most likely to reduce the error rate of the resulting model. The latter strategy can be implemented by estimating what the label is likely to be for the unlabelled instances in the input space, then determining how much the error is likely to be reduced on these estimated examples on labelling each member of the input space.

The query strategies we have mentioned so far have been rather general and applicable to any type of model. There are, however, specific approaches for several types of classification and regression model. For example, in their paper on active learning with SVMs, Tong and Koller [83] describe three approaches: *Simple Margin*, *MaxMin Margin* and *MaxRatio Margin*, which rely on the internal

structure of SVMs to determine either how uncertain examples are according to the model or how much of an effect they would have on the model if they were labelled.

In the subsequent chapters of this thesis we will make use of pool-based active learning. As of the submission of this thesis, no researchers have applied active learning to solve the exact same problems that we attempt to. However, several researchers have applied techniques in a somewhat similar manner to us. Settles highlights several papers on using active learning for text, image and video classification, information extraction, and speech recognition [66]. Of these, we examine a paper by Tong and Koller [83] to give an example of active learning using pool-based sampling, which is the approach we will later rely upon. In their paper, they aim to train SVMs to learn how to categorize documents from the Reuters-21578 data set and another large set constructed from articles in several newsgroups. They trained 10 different classifiers capable of recognizing 10 different categories on each set i.e. each classifier was binary. The inputs were word frequency vectors of dimension 10000. All documents had been hand labelled already, and as such the learning process was simulated by selecting subsets of data sets and removing the labels to form the pools. The authors then applied active learning using SVMs and three different types of query strategy: Simple Margin, Ratio and MaxMin. These were contrasted with a query strategy that selecting examples at random. The objective of their study was to show that the non-random query strategies performed better, which they did, and also determine what the best kind of query strategy was. This turned out to be a trade-off, as Simple Margin generally provided a rough approximation, but was quicker than the other two. Interestingly, the authors suggested that combining query strategies to form hybrid strategies may be useful. The pool sizes used in both experiments were fewer than 1000, which could be criticised as being rather small. It would be interesting to see how the models would have performed on larger sets and if the convergence rates would have stayed the same. In terms of their research, the authors have an advantage in that they have two large data sets available. This allows them to experiment with different types of algorithm and find out which is the best. One can question whether their results are generalizable to other problem domain, and this also brings about the question of how one can properly determine what the best active learning approach is when the input domain is very large, meaning that it cannot be exhaustively explored.

Active learning is not without its challenges. It is easy to assume that the oracle in an active learning approach is noiseless, that is, their feedback can be trusted entirely and their labels can be treated as a gold standard. However, this isn't necessarily the case. It may be that the oracle has limited ability to classify all member of the problem space or that the label being applied is actually subjective. Settles [66] highlights several papers that attempt to address these issues. Potential approaches include averaging out feedback amongst several experts and also calculating reliability scores for oracles. The solutions using the latter approach become potentially even more complex when one realizes that an oracles reliability may shift over time.

2.4 Unsupervised learning algorithms

In this section we will review unsupervised learning methods. These are algorithms that do not require the data to be labelled. Generally algorithms that fall into this category tell us the structure of the data presented and are useful for both clustering and visualization.

2.4.1 Dissimilarity measures

In the following sections we will go into detail about several types of clustering algorithm, which are algorithms based on grouping data using some form of distance metric. Before doing so, however, we need to specify exactly how a distance metric between two data items can be established. In this section we will consider the input to our problem to be a parameter vector consisting of attributes representing features of a problem domain. For example, the parameter vector may represent a patient and each attribute some feature of the patient such as age, previous illnesses, eye colour and so on. Our objective in this section is, given two such parameter vectors, to describe how *similar* they are. The immediate issue is that the components of the parameter vector could be of different types. For example, one attribute could be "gender", which is a *binary attribute* since it can take on two values "male" or "female". We may also have another value "miles exercised per year", which is a *numeric attribute* which may take on a real or integer value, perhaps from some restricted range of values. It is also conceivable there could be an *ordinal attribute* called "weight status", that can take on four values "underweight", "healthy", "overweight" and "obese". The values

have order to them, in each attribute value has a set place amongst the other values. Finally, we could conceive of another attribute called "favourite colour", which can take on one of several values with no particular order, such as "red", "blue", "green" and "orange".

In this section we restrict ourselves to the case where the parameter vector has attributes of different types, as this is the more generic case and the one which is most relevant to the techniques we require later on in this thesis. Our approach is taken from pages 75-76 of [29]. In our transcription of their approach we assume that the attributes of each parameter vector are always present, and therefore define our similarity metric as: $(\Sigma_{f=1}^{p} d_{ij}^{f})/f$, where p is the number of attributes and d_{ij}^{f} gives a distance value between parameter vectors i and j for attribute f. Put in a more verbose manner, the similarity metric tells us the average distance between each attribute pair in the two parameter vectors. It is immediately apparent that caution should be taken with the distance function d, as we have to ensure that no one or more attributes dominate the others due to the range of values they may take. This is why normalization needs to be included.

Let x_{if} give the value of attribute f in parameter vector i. For numeric attributes, we define d_{ij}^{f} as $|x_{if} - x_{jf}|/range_f$. We define $range_f$ as the range of values that attribute f can take and as such d in this case produces normalized distances in the range $[0, 1]$. For both nominal and binary attributes, we define $d_{ij}^{f} = 0$ if $x_{if} = x_{jf}$ or $d_{ij}^{f} = 1$ otherwise. In other words, if two binary/nominal attribute values are equal, then there is considered to be no distance between them, otherwise we consider there to be a distance of 1, regardless of what the two values actually are, which makes sense as there is no notion of order for such attributes. For ordinal attributes, however, there is a notion of order and therefore we can take this into account. We define t_{if} as the translation of nominal attribute f in parameter vector i. We define it as $t_{if} = (r_{if} - 1)/(m_f - 1)$, where r_{if} gives the rank of attribute f in i and m_f gives the total number of attribute values for attribute f. The rank of an attribute value is considered to be index, beginning at 1, at which the value occurs in the order of possible values. Once we have attained t_{if} and t_{jf} for vectors i and j we can treat the problem as we would a numeric attribute, noting that the denominator is simply 1. Note that for all definitions of d we have used, the value returned is always in the range $[0, 1]$, which ensures that no one attribute dominates the others when considering

distance.

2.4.2 k-means and k-medoids

k-means and *k-medoids* [6] are two closely related clustering algorithms. Clustering is useful as it allows us to group points in a data set based on their spatial distance from each other. We will first introduce k-means, then explain the difference between it and k-medoids. For any data set D of any dimension, the k-means algorithm will find a user specified c clusters, defined by a central point or *centroid*, and output what cluster each point in the data set is assigned to. The algorithm is defined Algorithm 2.3.

Algorithm 2.3. k-means.

1: **Purpose:** To identify clusters in an unlabelled data set.
2: **Input:** (a) Unlabelled data points, (b) The desired number of clusters to be identified c
3: **Output:** A list of centroids and, for each centroid, a list of points in the unlabelled data set associated with it.
4: Randomly initialize c centroids
5: Assign each data point to the nearest cluster, defined by a centroid
6: If no points were re-assigned, terminate the algorithm
7: Re-calculate the centroid for each cluster by averaging all the other points in the cluster
8: Goto (2)

The k-medoids algorithm is the same as k-means but an actual point from each cluster is selected as the centroid, rather than a mean value. The centroid must be that which minimizes the distance between all other points in the cluster. Caution must be made when applying such k-means and k-medoids to large data sets, as calculating distances can be computationally very expensive.

2.4.3 Principal Component Analysis

Principle Components Analysis (PCA) [6] is a method for identifying patterns in high-dimensional data. It can be used to reduce the dimensionality of data while minimizing data loss and as such is useful for both compression and visualization. Throughout the literature related to PCG in games it is directly useful and also forms the basis of several other machine learning algorithms.

For a thorough dissemination of PCA, it is necessary to first introduce two concepts from linear algebra, *eigenvectors* and *eigenvalues*. Eigenvectors can only be found for square matrices, (i.e. those matrices whose dimensions are the same length) and a *non-defective* n by n matrix has n eigenvectors of length 1 associated with it, which are all orthogonal. Multiplying a matrix with one of its eigenvectors result in a scaled instance of the original eigenvector i.e. $Me = \lambda e$, or in other words the matrix acts as a transformation on the eigenvector. Eigenvalues are the λ in the previous formula and tell us the scale of the transformation that the matrix has on the eigenvector. There are several methods to calculate eigenvectors/eigenvalues such as a Singular Value Decomposition (SVD).

For an n by m data set D with n examples of m variables, the variance of a particular variable is the square of the standard deviation of that variable i.e. for variable j: $var(j) = \sum_i^n ((D_{ij} - \overline{D_{*j}})^2)/(n-1)$, where $\overline{x}$ represents the mean. The variance is a measure of how much that variable changes throughout the set. The co-variance is an extension of this concept that tells us how much two variables vary in relation to each other and is defined as, for variables j and k: $cov(j,k) = \sum_i^n ((D_{ij} - \overline{D_{*j}})(D_{ik} - \overline{D_{*k}})/(n-1)$. The co-variance matrix is the square matrix giving the co-variance for each variable i.e. CM_{ij} gives the co-variance of variables i and j. Note that CM_{ii} gives the variance for variable i.

The basic idea of PCA is to calculate the eigenvectors and eigenvalues of the co-variance matrix of the data set. The eigenvectors in this case tell us the general directions of the data e.g. if the data generally forms a line in Cartesian space with some deviation from it then we should have one eigenvector in the direction of the line and then another eigenvector perpendicular to it to represent the deviation. We then project the data onto the principle eigenvectors. Algorithm 2.4 defines the steps for PCA reduction.

t-SNE

t-Distributed Stochastic Neighbour Embedding (t-SNE) [84] is a state-of-the-art non-linear dimension reduction approach, which can be used for the visualization of high-dimensionality data. t-SNE calculates the distances between the data points in the high-dimensional space and uses this to form a probability distribution representing similarities between them. For points that are close, the probability is high, whereas for points that are distant, the probability is infinitesimally small. The *Kullback-Leibler divergence* between two probability distributions is

Algorithm 2.4. Principal Component Analysis.

1: **Purpose:** To reduce the dimensionality of a data set while preserving trends within it.
2: **Input:** (a) A set of data points D of a high dimension, (b) the desired dimensionality, n, which the data should be reduced to
3: **Output:** (a) Eigenvectors E, (b) Eigenvalues V and (c) a data set D_r of dimensionality n
4: Let n be the desired reduced dimensionality.
5: Subtract the mean from each dimension of the data set D to form D'.
6: Calculate the covariance matrix C of D'.
7: Calculate the eigenvectors E and eigenvalues V of C.
8: Select n eigenvectors, ranked by V, and put them as columns in E.
9: Let the reduced data set be $D_r = E^T \times (D')^T$.

the measure of information lost when the first is used to approximate the second. t-SNE constructs a second probability distribution for the lower dimensional space, representing the distances between points, and attempts to minimize the Kullback-Leibler divergence between it and the higher dimensional probability distribution. This produces a lower dimensional representation, based on the second probability distribution, that preserves relationships from the higher dimensional space.

The main advantage of t-SNE over PCA is that t-SNE attempts to find a non-linear mapping between the high and low dimensional space, whereas PCA finds linear mappings which means that clusters may not get separated properly if the problem is non-linear in nature. The disadvantage of t-SNE is that it is quite costly to execute and is geared towards visualization, rather than being a general purpose dimensionality reduction technique.

2.5 Evolutionary Algorithms

Evolutionary Algorithms (EA) [53] are an approach to solving optimization problems whose design is influenced by biological processes. EA generate populations of solutions and use a fitness function to assess the quality of each member of the population. The best candidates are "bred" using the biological principles of recombination and mutation to produce a new population. The design of the fitness function is important, as a badly posed fitness function would result in the EA being misguided causing it to optimize solutions that aren't desirable.

The processes of recombination, which is the combining of two candidates, and mutation, which is the random modification of a candidate, also need to be well designed to make the search progress in a meaningful manner. EA have a long history and have been given different names by researchers, but in recent years, there has been a movement to unify them under one banner [36]. Algorithm 2.5 gives the general form of an evolutionary algorithm.

Algorithm 2.5. Evolutionary algorithm.

1: **Purpose:** To find optimized solutions to a problem using the process of evolution.
2: **Input:** (a) A fitness function f that can rate a candidate solution, (b) crossover and mutation operators for solutions.
3: **Output:** A set of solutions to the problem.
4: Create the first generation of candidate solutions G_0 randomly
5: Set $i = 0$
6: **while** G_i not optimal enough or time constraint not met **do**
7: Evaluate fitness of each member of G_i using fitness function f
8: Using crossover and mutation, reproduce using fittest members of G_i to produce a set of offspring O
9: Replace least fit members of G_i with offspring O to produce generation G_{i+1}
10: Set $i = i + 1$
11: **end while**

The two most prominent forms of EA used in research literature for PCG are Generic Algorithms (GA) and Genetic Programming (GP). GA traditionally operate on genotypes that consist of bit string representations of the actual solutions, also known as phenotypes. In principle, their theory is based on the Building Block Hypothesis (BBH), which states that solutions are made up of building blocks that can be recombined into an optimal solution. Both mutation and recombination are used to produce offspring. The main question when using a GA is the form that the genotypes take, as different forms of encoding can completely change the success of the algorithm. It is generally desirable that the extent of changes in the genotype would have a similar extent of change in the phenotype, but this can be difficult to achieve without good prior knowledge of the solution space. GP directly evolve programs and as such the solution space is represented using parse trees, with operation and value nodes located at the leaves of the tree.

Both types of evolution mentioned in this section have very critical configuration parameters that need to be chosen, such as the fitness function, genotype representation and the recombination and mutation operators. Identifying the best parameters is usually problem specific. In order to apply EA to PCG, one has to consider the challenge of parameters selection, and whether the approach taken is general enough to be applied to other problems or just the current problem at hand.

EA can be used to both train ANN and modify characteristics of an ANN to find one appropriate for a particular problem. Yao discusses many of the techniques that can be used to combine both EA and ANN [91], taking into account issues such as the *permutation problem*. The values of the weights in an ANN can be put under the control of an EA and evolved. This has the advantage that the activation function of the network need not be differentiable, since the hypothesis search is based upon the fitness comparison of members of a population, rather than modification of individual weights based on the output error of the network. Neuro-evolution also tends to require only sparse reinforcement from the environment, whereas back-propagation requires a lot of supervision.

2.6 Crowd Sourcing and Machine Learning

Crowd sourcing is the act of delegating tasks to, often anonymous, members of the public. Crowd sourcing has been used successfully in commercial applications, such as "Amazon Mechanical Turk", which is a website allowing people to assign tasks, such as labelling, to the public. Of interest to us is the fact that it is possible to use "the crowd" as experts to train machine learning models. This has the advantage of providing a large number of people, perhaps with various levels of expertise, for labelling data and providing feedback. On the other hand, crowd sourcing brings with it the critical problem of handling unreliable feedback [75], which may cause harm to the training process. Researchers have attempted to make the process of crowd sourcing more robust, and learn attributes of the contributors such as competence, expertise and bias [86]. Recent research has also investigated how active learning and crowd sourcing can work alongside each other [1]. In Section 4.4.3 we go into more detail by describing one particular approach that attempts to do just this, an expectation-maximization model that

uses crowd sourcing [65].

2.7 Conclusions

In this section we have covered various machine learning concepts that will be of use in subsequent chapters of this thesis. In the next section we will see how several of these algorithms are used in the PCG literature.

Chapter 3

Procedural Content Generation

Chapter 1 provided a basic introduction to the field of PCG in video games and stated the main contribution of this thesis. The goal of this chapter is to provide an overview of video games (Section 3.1), player modelling (Section 3.2) and PCG (Section 3.3). Many of the techniques we will review will rely on the machine learning algorithms we have described in Chapter 2. Near the end of this chapter we will provide a critical analysis and taxonomy of existing PCG techniques, which gives motivation for the design decisions made with respect to the LBPCG framework, the primary contribution of this thesis.

3.1 Video Games

Before embarking on a detailed discussion of PCG in video games, it is necessary to take a step back and consider precisely what a video game is. In this section we will cover the topics of video game *categorization* and *description*. Categorization tells us the general nature of a video game, such as how the player interacts with it and what the objectives are. This is useful information, as it allows us to limit the scope of PCG algorithms to particular genres. Description on the other hand provides us with a layer of communication that facilitates the manipulation of game content by a PCG algorithm.

3.1.1 Game Categorization

The traditional method used for game categorisation is *genres*. It is very common to see genres used on game review websites such as *IGN* [35] and *Gamespot* [25].

Some examples of genres are:

1. **First Person Shooter (FPS)**: The player controls a character who is wielding a ranged or melee weapon. The objective is to navigate through various levels killing enemies as they are encountered. Examples: Doom (1993), Duke Nukem 3D (1996) and Halo (2001).

2. **Adventure**: The player controls one or more characters and moves through the world using a point-and-click interface solving various puzzles and uncovering a storyline. Examples: The Secret of Monkey Island (1990), The Longest Journey (1999) and Zak McKracken and the Alien Mindbenders (1988). Moving further back in time, classic adventure games were entirely text-based, such as Zork (1977).

3. **Role Playing Game (RPG)**: The player controls one or more characters, usually in a fantasy-based setting. The objective is to enhance their character(s) abilities to make them more powerful while progressing through a storyline. Examples: Daggerfall (1996), Final Fantasy (1987) and The Legend of Zelda (1986).

4. **Real-Time Strategy (RTS)**: The player controls a faction. The objective is to collect resources, construct buildings and units and engage in warfare against other factions. Examples: Command and Conquer Red Alert (1996), Warcraft (1994) and Starcraft (1998).

5. **Racing**: The player races a vehicle around racing tracks against opponents. Examples: Forza Motorsport (2005), Virtua Racing (1994) and Test Drive (1987).

It is not always clear what genre a game belongs to, for example, the popular game series Grand Theft Auto (1997-) includes elements of the racing and first person shooter genres. This is a common issue as games have become more complex by incorporating many game play features and has seemingly become more difficult to classify the exact nature of a video game using genres.

Frasca [24] states that one can view the genre approach as *narratological*, in that it views games as extensions of drama and narrative. He argues, however, that the narratological approach to defining video games is inaccurate and limits its understanding of the video game medium. He also introduces the formalist

discipline known as *ludology*, which is dedicated to the study of video games and focuses on the understanding of structure and rules with a view to create typologies and models for explaining the mechanics of games. Apperley et al [3] build on this argument and agree that many existing classifications of games are *representational*. By this it is meant that they are tied to existing descriptions of other media (e.g. films) and focus too much on the aesthetic qualities of a game. The authors argue that the way in which the user interacts with the game is not taken into enough consideration with this type of classification. This is true, since two games may look visually very similar, but their control schemes may be significantly different resulting in two very distinct experiences.

Lindley [45] defines four classification models for games. These models provide a number of dimensions in which a game can exist. Three recurring axes in each model are Simulation, Ludology and Narratology. The ludological dimension specifies how much a game is rule based. An example of purely ludological game would be Battlechess (1988). The simulation dimension specifies how much a game attempts to represent something real, for example an economy. A game such as Risk rests between the simulation extreme and the ludological extremes, as it attempts to simulate a war in a rule-based fashion The narratological dimension specifies how much a game is based upon a storyline that advances over time. At the extreme of this dimension are movies, with which the user has no interaction. Moving slightly away from this extreme are multi-path movies, in which the user can select the direction of the movie. Role-playing games, which tend to feature a storyline, simulated world and combat rules tend to sit in the centre of the ludological, narratological and simulation dimensions. Such a classification is useful for categorising the high-level nature of a game but does not say much about a game's mechanics.

Gunn et al [28] provide a ludological taxonomy of video games with the aim of providing a framework for matching AI techniques to games. They state that a ludological classification is the one that interests developers and academics. This standpoint is understandable from a machine learning perspective, as researchers in this field are interested in determining the mechanics of the game that they intend to have their algorithm interact with. The model in their paper provides a number of boolean values that in sequence can be used to describe the features of a game. Figure 3.1 shows a complete visualization of the model, along with mappings to appropriate machine learning algorithms. As an example, a game

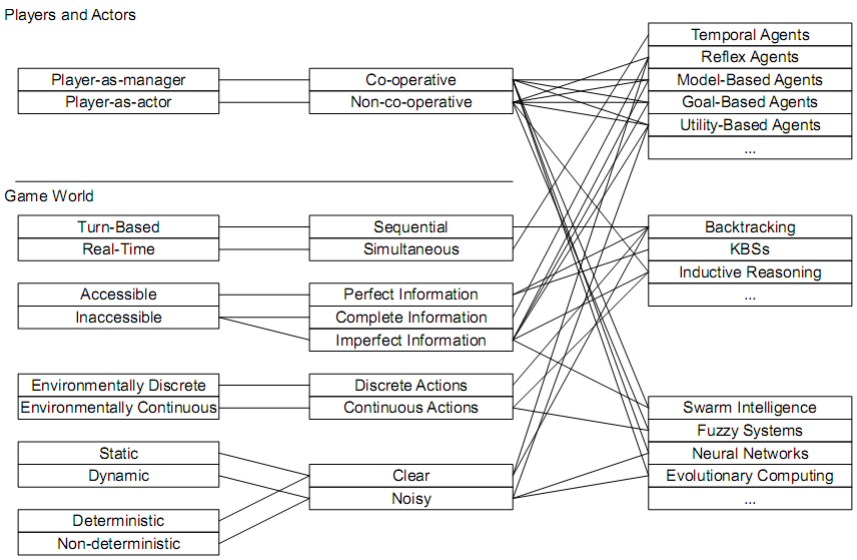

Figure 3.1: Gunn's taxonomy [28].

is player-versus-environment if it consists of rather limited AI and involves the player manipulating the game world directly, such as a puzzle game. A player-as-actor game involve the player directly controlling an actor in the environment, such as first-person-shooter. On the other hand player-as-manager games involve the player controlling more than one actor in the game world, such as a real-time strategy. Games can be co-operative, in that players within the game can work together or non-co-operative in which they work against each other. The model is useful as it provides a general mechanism for describing the traits of a particular game, beyond its aesthetic qualities.

In summary, there is no single generally agreed approach for categorizing video games and most professionals often resort to the use of genres. In the next section, we will review video game description, which is perhaps a more useful perspective on video games for the purposes of PCG.

3.1.2 Game Description

We begin this section on game description literature by first reviewing approaches that are used for traditional board games like chess. The Stanford Logic group have developed a first-order logic Game Description Language (GDL), which is intended to be used by General Game Playing (GGP) agents [46]. A GGP agent is an agent that has the capability to play different types of game having been

provided with a description of the game rules only. GGP agents can compete in various open competitions [7]. The Stanford Logic group state that the GDL can be used for "finite, discrete, deterministic, multi-player games of complete information". In terms of Gunn's taxonomy, this limits the supported games to those that are player-as-actor, accessible and environmentally discrete. As such, the scope is somewhat limited, especially with respect to the current generation of video games. The Stanford GDL is similar in syntax to Prolog and as such is unlikely to be of practical use by game designers, who tend to have minimal programming experience.

Zillions of Games [92] is a commercial product that provides a large number of games written in the Zillions Rule Language. It allows a great deal of interesting rule-based games with sprite graphics to be defined. The language is Lisp-like in that it is based on *S-Expressions*. The expressions in the language can specify players, valid moves and completion objectives of the game. Similar to the GDL, while the language is simple to use and can cover a wide-range of computerized board games, it is not sufficient to describe modern day video games which can involve large amounts of continuous elements that re-act in real time.

Browne et al [9] define a method for automatically generating combinatorial games. A combinatorial game is a two-player game which does not involve chance and is fully accessible, as in all information is visible to players. Their paper describes a general game system which includes a GDL and a GGP agent. Unlike existing general game systems that are used to evolve AI, the aim of their system is to evolve interesting games automatically. They introduce the Ludi GDL, which is more high-level than the Zillions of Games rule language. It is specifically tailored for creating combinatorial games and consists of *ludemes*, which are structured rules. The language is very easy to understand, but is limited to combinatorial games and cannot control the graphics displayed on the screen, unlike the Zillions of Games rule language.

Pizzi et al define a novel technique for debugging game storyboards [60]. In their paper they work with Eidos Interactive and make use of the game Hitman: Codename 47 (2000). The technique involves writing a complete logical representation of the high-level game play elements of the world in a propositional language. They use Heuristic Search Planning (HSP) to derive possible solutions to a level. Using this method they were able to show solutions to a level that the game designers themselves weren't aware of. Such an approach is appropriate for

games that progress through certain storyboard stages, but would not be useful for games such as first person shooters that tend to be purely action based.

Nelson and Mateas propose the use of a first-order event calculus to represent games [54]. They include a construct known as *fluents*, which is a predicate type whose truth value changes over time. In their paper, they write a description for simple grid-based game executing using Mueller's Discrete Event Calculus Reasoner [34]. In the game the player must acquire a key to open a door which they need to pass through to complete the game. The reasoner successfully identifies that the game is winnable. After adding diagonal movement to the representation, the reasoner is able to identify an error in the game logic that allows players to move diagonally through walls. As such, their technique can successfully determine whether a game is suitable to be played or not. Although this work is successful at describing tile-based games, it is also limited for the same reasons as the Zillions of Games language in that it would not be appropriate for most modern video games which have large non-discrete 3D environments.

A paper by Dormans defines a methodology for describing and generating action-adventure games [16]. In an action-adventure game, the player moves through an environment and solves puzzles while a storyline develops. A very prominent example of such a game is Tomb Raider (1995), where the player must navigate the central character through various 3D environments while fighting enemies and solving puzzles. Dormans observes that such games are made up of two separate components. The first of which are the "missions", including activities such as collecting items and solving puzzles, and the second of which is "space", physical entities such as rooms and corridors. The paper argues that *generative grammars* can be used to first generate the mission structure and then the appropriate space structure for it. In addition to this, he states that rules can be constructed that specify how new missions can be automatically generated. The author demonstrates that a level from the game Zelda: The Twilight Princess (2006) can be described in this manner. It is our opinion that this promising work could pave the way for a high-level description language for video games, although it does at this moment fail to incorporate a great deal of game elements such as physics, control and artificial intelligence. The use of grammars is becoming increasingly popular in the field of PCG. Recently researchers have used them to represent platform games [68] and have even developed a specific grammar for the representation of strategy games [49].

The purpose of this section has been to analyse the topics of game categorization and description. It is apparent that the ludological representation is more useful for the purposes of PCG as it describes the components of a game, rather than simply attempting to relate to existing media. A taxonomy by Gunn et al [28] provides a very good basis for ludological classification, which can express the suitability of a PCG algorithm for particular types of game. In summary there are many approaches to representing video games, with no particular agreement on which is the best for the purposes of PCG. In their paper on Search-Based Procedural Content Generation (SBPCG) [82], Togelius et al mention some very common approaches, of increasing abstraction, in which a video game can be represented for manipulation by a PCG algorithm. These are: (a) a grid where each cell represents an entity in the environment, (b) a list of positions, orientations and lengths of walls (c) a repository of pre-fabrications and a list of how they are distributed (d) a list of abstract values specifying the properties of the content (e) a random seed. Although making no reference to grammars, which are used in some PCG techniques, based on the literature review in this thesis, we consider Togelius et al's list of approaches to be a good summary.

3.2 Modelling players in games

In this section we will present the concepts of entertainment and review algorithms that attempt to model player satisfaction and other states in video games. Player satisfaction is a measure that tells us whether game content is appealing and due to its subjective nature it is very difficult to quantify exactly what constitutes it. For example, a player may be frustrated by the difficulty of a particular game, but will continue to return to it due to the challenge. In such a case, the player may not be having fun in the traditional sense, but the game content is surely appealing enough to be continually played. For a PCG algorithm, models that can predict satisfaction can be extremely useful tools as they allow the implicit detection of whether game content is desirable. If one was to apply an evolutionary algorithm to PCG, a player satisfaction model could be used as a fitness function that needs to be maximized to find desirable content.

3.2.1 What is fun?

Sweetser et al state that player enjoyment is the single most important goal for a computer game in their interpretation of the *Theory of Flow* applied to video gaming [78]. The Theory of Flow is a model of enjoyment proposed by Csikszentmihalyi, which defines eight elements required to provide an enjoyable experience [13]. Amongst their interpretation, they state that a game should provide sufficient challenge, players should feel in control of their actions and games should support and create opportunities for social interaction. Andrade et al put forth that argument that balancing challenge is a key component to player entertainment [2]. In their experiment they ask human players to fight against several AI agents. Some of the agents are adaptive and learn to fight against the player more affectively and analysis of game data and the use questionnaires suggested that the adaptive agents provided the highest level of player satisfaction. Interestingly, their results also showed that unpredictability also improved player satisfaction.

Using Malone's intrinsic qualitative factors [50] and the Theory of Flow [78], Yannakakis and Hallam attempt to correlate the relationship between difficulty, curiosity and fun [88] using Pacman as a test bed. Each player played games against two different agents selected from a collection of 5, each with different challenge, behaviour diversity and spatial diversity, reporting their preference between the two opponents. Two features from each match were also recorded: (a) curiosity, which was the deviation of time played and (b) challenge, which was the average time played. Neural networks were then used to learn the correlation between difficulty, curiosity and fun based upon the collected data. The inputs to the networks were the recorded features and the output gave an entertainment value. The idea was that the networks should eventually be able to determine what features of a game are preferable. Their results showed that entertainment is low when challenge is too high or curiosity is low. The results also suggested that keeping challenge at an appropriate level mitigated a lack of curiosity. These results seemingly matched Malone's hypotheses. The approach can be criticised in that it assumes that the two features recorded, (a) and (b), are in fact curiosity and challenge, and that the study relates only to predator/prey games.

Yannakakis and Hallam perform another experiment to model entertainment in a grid sensor game [89]. In this experiment they ask several school children to use the Playware physical interactive platform, which involves stamping on tiles as lights appear on the tiles. Neural networks are trained in similar manner

to their previous experiment. They then take the partial derivative of certain neural network outputs and inputs with respect to each other. For example, the rate of change of game speed with respect to entertainment. These partial derivatives are then used as criteria to adapt additional games as they are played. It was found that the school children prefer the adaptive games more than the non-adaptive game, suggesting that diverse game play is something that could increase entertainment.

A central argument of this thesis is that it isn't a good idea to rely entirely on the prior knowledge about a problem, such as the question of "What is Fun", and then encode this information to form some kind of assessment function. The Theory of Flow and Malone's intrinsic qualitative factors may hold for specific examples of traditional games, but when it comes to open world games or even educational/serious games they may not. A far more robust approach is to learn in a data-driven way the function that can assess content by monitoring people playing games. In the next few sections we will review techniques that do just this to create models of entertainment and other emotive states.

3.2.2 Modelling using play-log features

If one is to model player behaviour, then the type of data used as input requires in-depth consideration. In this section we will look at approaches that make use of what this thesis terms *play-logs*. Play-logs are records of events that occurred when a player played a game. For example, in the case of a first-person shooter, a play-log might list how many times the player killed a monster, what kind of weapons they preferred and how much of the level they managed to complete. The necessary requirement for the existence of such data is, of course, that a game has to be played by a player.

Pederson et al model human emotion while playing procedurally generated levels in Infinite Mario [58]. Infinite Mario is an open source version of the classic platform game Super Mario Brothers (1985), which has the ability to generate an endless amount of levels. The authors take the engine and produce sixteen different level variants, based on four level generation parameters. Members of the public were then asked to play two levels and report their emotional preferences, including fun, with respect to the two games. For example, the player could report that they thought game A caused more anxiety than game B. While playing, a number of play-log features were collected such as the number of jumps

the player made and how many enemies they killed. Empowered with a large amount of feedback from players, the authors performed analysis to detect relationships between level generation parameters, game play statistics and emotion preference. They also use two types of neural network (SLP and MLP) along with feature selection to build a model that can accurately predict a players emotional preference between two levels. They conclude that their model is successful, but suffers from the fact that it requires play-log features as inputs rather than just level generation parameters i.e. the levels must be played before they can be assessed. They argue that neural networks are the best candidate for modelling this problem based upon work on preference modelling in a paper by Yannakakis [89]. In this paper the authors compare techniques including neuro-evolution, a variant of SVMs called large-margin algorithm (LMA) and Gaussian Processes. According to their results, neuro-evolution achieved the best results. The main criticism of the work is that there were only a small number of level variants (sixteen) and it is not clear whether the same modelling approach would work for a larger set of games, or even if it would be possible to collect enough data for such a purpose. Another problem is the question of what would happen in the presence of unreliable players, who may provide noisy feedback.

Almost all of the studies in this literature review use data sets specifically built for academic purposes. However, in 2010 Mahlmann et al are given access to play-log data from 1000 people who played commercial game Tomb Raider Underworld. In their second paper on the subject [48], the authors goal is to predict the performance of a player, i.e. how much of the next level will be completed, based on the players activities in previous levels. This time, more features are extracted to be used as inputs and they use a combination of different machine learning algorithms including neural networks and Bayesian Networks. Unfortunately, they report only moderate performance, citing problems with noise in the data. This paper highlights the difficulties in working with data collected from the public, which could be infested with either unreliable feedback or simply be incomplete. Such issues can cause very detrimental effects on model performance.

3.2.3 Modelling using content creation parameters

As stated at the start of the previous section, using play-logs to predict what affective states game content produces is not always a desirable approach. It is more useful in some circumstances to be able to predict how good content

is based only upon the features of the content e.g. how many monsters does the level have? In further work on Infinite Mario, Shaker et al model player preference with respect to engagement, challenge and frustration across a larger number of controllable parameters resulting in 40 variants this time [70]. After playing two games the player reported their preference with respect to each of the three qualities. The objective this time was to detect preference based on the level parameters only, without consideration of the play-log. The authors also attempt to model preference based on the properties of sub-sections of each level as well as the entire the level, with an end goal of being able to generate content in real-time for a player. The best result achieved was an accuracy of 65.72% using neural networks, suggesting there is room for improvement, which could perhaps be achieved by using a different machine learning method. The work also suggests that including play-log features could improve the results.

3.2.4 Categorizing players

It is certainly conceivable that rather than every player being completely unique, there actually exist different groups of player who play in a similar style, with small deviations. Such information is highly useful, as it could be used as a filtering step in an algorithm attempting to optimize content e.g. "We have detected this player is a beginner gamer, therefore filter all difficult content".

Drachen et al attempt to categorize players using a smaller subset of the Tomb Raider Underworld data set which they acquired [48], consisting of play-logs from 1365 people [19]. They extract six play-log features related to character death and usage of help functionality. They apply Emergent Self Organizing Maps (ESOM) to cluster the data and find that there are four main types of player. Based on their findings, the concept of using clustering to identify different player-styles may be a good approach, although the data set used is somewhat limited meaning the number of player-styles that can be identified is also limited.

Martínez et al take an interesting approach to predicting affective state in their paper on the predator/prey "Mazeball" video game [51]. They first used ESOMs to identify types of player, which detected five different play-styles. They then used play-log data consisting of game events, key strokes and high-level statistics along with the detected category to predict challenge and fun. Their results suggest that combining the players category with play-log information provides a statistically significant performance increase. In general, this result suggests that

categories of player exist for the target game in question, but it remains to be seen if there are generic categories of player. If such generic categories exist, it is debatable whether the detection methods presented in this paper are sufficient to detect them.

Gow et al apply Linear Discriminant Analysis (LDA), a technique closely related to PCA, to recognize player style in two games: Snakeotron, a basic 2D game, and Rogue Trooper, a commercial 3D shooter game [26]. In the case of Snakeotron, 16 features were extracted from each play-log including statistics such as damage done and key-press information. There was a total of 215 unique participants in the study, resulting in a data set of 1450 surveys. The results showed that frequency of key presses was the most important feature for dividing players. For Rogue Trooper, 32 participants were asked to play 20 minutes of the first level of the game producing 35 level attempts and therefore 35 play-logs. The data was then fed into the LDA algorithm, which allowed the authors to identify that both health and movement were determining factors on player grouping. The method of applying LDA in this manner is interesting, as it implicitly detects groups of players. It remains to be seen whether using the categories found by LDA would allow an algorithm to detect player affective states more efficiently.

3.2.5 Beyond play-logs

All of the techniques in this section so far, perhaps with the exception of the work done by Yannakakis et al on the play-ware platform [89], have used cues from software to model the player, for example, by looking at events that occur during game-play. There are many examples of other approaches that use physiological data to predict player emotive state, which we consider beyond the scope of this thesis. However, an approach gaining traction is to use the physical expressions of the player during game-play. This is something that is actually a viable option on modern computers, as most laptops have web cams built into them. Papers have recently been produced by Asteriadis [4] and Shaker [67] that record players expressions while Super Mario is being played. In Shaker's work, a very high level of accuracy was reported when combining visual cues with play-log events to predict challenge and frustration. It does remain to be seen if the technique can be extended to other genres, and whether the system would be successfully on a wider selection of people. It is also questionable as to whether such methods could be used for online adaptation, for example, gamers may consider web-cam

logging as being rather invasive and may not wish to use it.

3.2.6 Summary

In this section we have reviewed existing methods for modelling players in video games. A direct approach to adapting content for a target player would be to present content to them and ask them if they enjoyed it with a dialogue box or similar. The search could then be updated base on this information, however, this method is very intrusive to the users experience. Research in this section has showed that it is possible, based on play-logs collected during game-play, to identify whether the player's experience was positive or not without asking the player any questions. This is a strong indicator that such methods could be employed to enable non-intrusive adaptive PCG. There are several pitfalls, however, such as dealing with outliers and ensuring that enough data is collected to train the player models [48]. Several papers indicate that players may fall into different categories, based on their play-style, which could be used to focus the search for player-specific content.

3.3 Procedural Content Generation

In this section we will introduce PCG in a taxonomy based on sophistication in terms of adaptivity to players. In Section 3.3.1 we will review traditional approaches to PCG, which are those that are the result of applying an algorithm with some level of randomness involved, perhaps where the measure of fitness is built into the algorithm itself. In Section 3.3.2 we will review search-based PCG algorithms, which are algorithms attempting to optimize a pre-defined fitness function. We distinguish algorithms in this section from the later algorithms by noting that they do not directly attempt to adapt content to a target player, rather they adapt the content to some general notion such as fun, challenge etc. In Section 3.3.3 we will review algorithms that use search-based methods to adapt content to a target player. The fitness function used to adapt to the player in this case has been constructed by hand i.e. pre-defined. In Section 3.3.4 we will review algorithms that learn models that adapt to the player and then make use of the model to adapt content to a target player.

Many of the algorithms that we will introduce in this section fall under the umbrella of Search-Based Procedural Content Generation (SBPCG) [82]. SBPCG

are a family of *generate-and-test* algorithms, meaning that content is generated then evaluated. At the heart of an SBPCG algorithm is a fitness function, which is capable of grading the content. The most common example of a SBPCG is an evolutionary algorithm, which generates populations of content, assesses them, then performs cross-over and mutation on the most fit examples to produce a new population.

Recently, another view of the PCG literature has emerged termed Experience-Driven Procedural Content Generation (EDPCG) [90]. In this framework, the emphasis is placed upon optimizing the experience of the player. Specifically, an EDPCG algorithm has four components: (a) "Player Experience Modelling" (PEM), which means the player's experience is considered to be a function of the game's content and the player, (b) "Content Quality", which means the quality of the content is linked to the player experience, (c) "Content Representation", which means the representative form of the content used by generator is designed well and (d) "Content Generator", which means the algorithm searches through the available content and attempts to optimize it with respect to the player's experience. The paper lists three types of PEM: (a) *Subjective*, which asks the player about their experience, (b) *Objective*, which monitors physiological conditions of the player while experiencing the game such as their facial expression, posture and speech, and (c) *Game-play based*, which observes the play-logs generated by the player. The authors define three general approaches to assessing the quality of content: (a) *Direct Evaluation Functions*, which extracts features from the content for evaluation, (b) *Simulation-Based Evaluation Functions*, which evaluate content by having artificial agents play it, and (c) *Interactive Evaluation Functions*, which evaluate content based on how the player interacts with it. These definitions are useful to bear in mind while considering the various PCG techniques mentioned throughout this section.

3.3.1 Traditional Procedural Content Generation

In this section we will examine traditional methods of PCG and highlight their significance. Togelius et al put forward strong argument as to why PCG in games is important [82]. Their first argument is that it allows the reduction in size of the game content. If the content is generated by an algorithm at run-time then there is no need to distribute it. Although this is not such an important consideration in modern times due to large storage space, it was certainly a benefit in the

early days of video games. Their second argument states that it reduces the cost of producing content. They use the example of *SpeedTree* which is software capable of placing foliage in the game world. This frees game artists from the tedious role of performing such a task and allows them to focus on more important development. Their third argument is that PCG can produce an endless stream of game content, which would extend the re-playability a game by a potentially infinite amount. Their final argument is that PCG can augment the human creative process, to provide a basis on which the human can develop their game.

Procedural Content Generation (PCG) is an approach that has been used in commercial video games for a long time. Since video games are software, which can be abstracted into high-level APIs for content generation, PCG has the potential to generate *any* type of content. This includes the physical game world such as terrain, the non-player characters (NPCs) that the player interacts with, the shaders used for graphical enhancement, the storyline, quests and so on. There are many examples of PCG in commercial games, although it is unfortunately predominantly the case that the source code for such games is unavailable, and as such it is difficult to determine the exact methods that have been used. The following list describes some of the major examples of PCG use in games, providing detail where it is available.

- **Borderlands (2009)**: A multi-player First Person Shooter where weapons in the game are procedurally generated to provide an endless and interesting supply of items to collect. This contributes greatly to the re-playability of the game because players don't know what they will see next.

- **Elite (1984)**: A very early 3D space game in which the player could explore the universe. Star systems in the game were randomly generated as they were encountered, which provided an endless game world to explore.

- **Rogue (1980s)**: A classic ASCII game that randomly generated dungeons. This has spawned a large number of similar games called *Rogue-likes*. Diablo (1997) and Torchlight (2009) are isometric and 3D commercial games, respectively, which can be considered to be rogue-likes as they generate dungeon content and items procedurally.

- **Spore (2008)**: An ambitious god game in which the player controls all aspects of the development of a species from its emergence as a cell, to

Darwinian evolution, war with other species and eventually colonization of space. The breeding and evolution of the creatures, which can pass characteristics onto their children can be viewed as a form of PCG.

- **Elder Scrolls II: Daggerfall (1996)**: A 3D action RPG that had a vast world of 161,600 squared kilometres and over 15,000 towns, cities, villages and dungeons. Most of this was procedurally generated, as it would be impossible to develop so much content by hand.

- **Far Cry 2 (2008)**: A sandbox First Person Shooter set in war-torn Africa. The development team used a form of PCG to generate the terrain before release, which was likely to have been hand edited afterwards. The developers also spoke publicly that enemies in the game were procedurally generated. They suggest that enemies had random faces selected during run-time to provide dynamic appearance.

- **Left4Dead (2008)**: A single and multi-player First Person Shooter in which a team of four people have to fight through way through a zombie infested city. The game flow is controlled by an "AI Director", which dynamically places spawn points for zombies based upon the skill of the players.

- **X-Com: Enemy Unknown (1994)**: A strategy game in which the player must control an organization that researches UFO technology and co-ordinates a defence against attack. The isometric levels where the combat took place were procedurally pieced together from tile-sets, providing an endless supply of locations in the world.

Many of the PCG techniques used in games are quite simple in design but can produce spectacular results. A *fractal* is a shape where smaller parts of the shape are similar to the shape as a whole. There are many examples of the use of fractals to generate visually impressive imagery, however, a fractal is of particular interest to PCG as the shapes found in nature tend to be less based on Euclidean geometry and more on the self-recursive and irregular attributes of a fractal. As such, there has been a large amount of research into and usage of fractal algorithms to generate 3D geometry such as terrain. Such algorithms tend to begin by constructing the general features of the terrain and then recursively

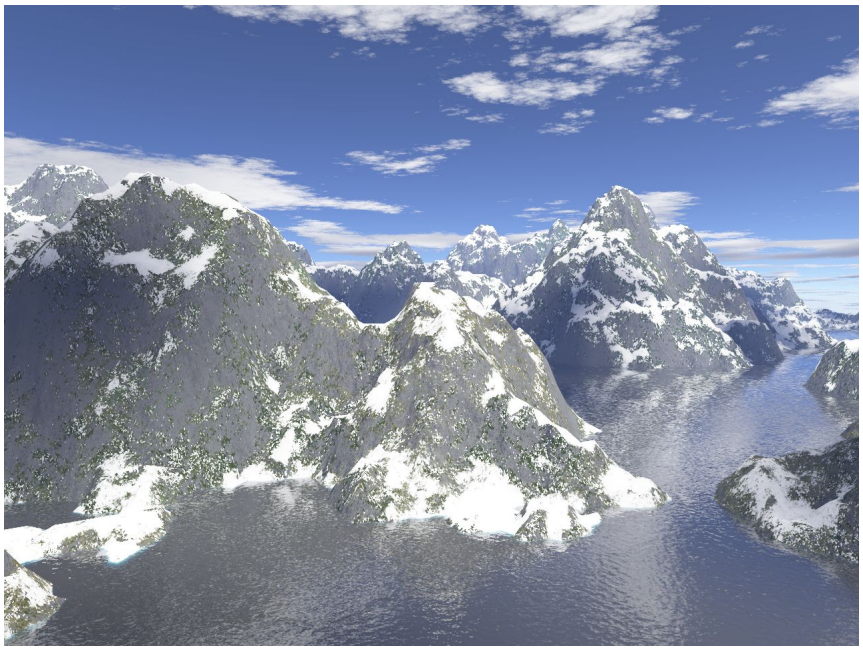

Figure 3.2: A terrain generated with a fractal algorithm.

refining parts of it until a level of realism is reached. An example fractal terrain is displayed in Figure 3.2.

Another common method used for PCG is *Perlin Noise* [59]. It is observable that the natural world is not made up entirely of completely random variations. It tends to be made up of groups of variation, such as large mountains with huge variation, hills with less variation and small objects with very little variation. A Perlin Noise function can be used to simulate this natural phenomena and produce realistic environments. It is constructed by creating several noise functions with varying frequency and amplitude, which are smoothed using one of several interpolation techniques. When queried for a random value, the Perlin function samples each of the noise functions and produces a sum, which represents the result. Perlin noise was used in the open-source game Infiniminer (2009) to produce multi-player blocked-based environments with interesting structure.

Cellular Automata are a type of machine that use a mesh of adjacent cells, where the mesh may have any dimension e.g. a cube, line or 2D plane. The value of a particular cell is affected by those cells around it based upon a rule, which is shared by all other cells in the mesh. The value of every cell is determined at the same time as every other cell, although some variations of the technique pick random cells to be updated at separate times. Babcock describes a method

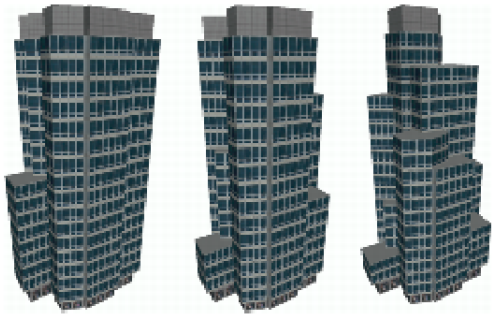

Figure 3.3: Automatically generated buildings. [27]

that uses cellular automata to create cave-like levels for use in a Rogue-like game [5]. The map consists of a grid of empty spaces, which may have a wall placed in them. In the initialization step, walls are placed randomly using given probability value. The algorithm then updates every space during each iteration. The rule is that a wall should be placed in a space s if: (1) there are 5 walls near to s, or (2) if s is surrounded by no nearby walls by a given distance. This simple algorithm is successful in constructing maps that resemble cave-like labyrinths. Babcock's work goes on to define modifications to the rule to produce more enclosed variants.

PCG can also be used to generate artificial structures. Greuter et al describe a resource sensitive algorithm to generate pseudo-infinite cities [27]. Their technique randomly generates polygons representing each floor of a building, which are then extruded upwards to induce third dimensionality. Texture co-ordinates are generated during the algorithm which enables building textures to be drawn onto the surface of the geometry. A city is created by forming a grid of such buildings. The technique produces relatively simple structure (see Figure 3.3), but is particularly notable as it generates content on-the-fly as the user moves through the world.

In a similar vein, Wonka et al describe an interpreted grammar called CGA Shape, which can be used for the modelling of 3D cities [87]. Iterative rules written in the grammar generate content by adding more and more detail. Rules may specify operations upon basic shapes such as scaling, rotation and surface subdivision. Additional procedures such as occlusion testing also exist, which can be used to test for clipping between shapes in a model. A designer can also make use of random variables, such that a single script can generate many different buildings. An impressive result in the paper shows the reconstruction of ancient

Figure 3.4: CGA recreation of Pompeii [23].

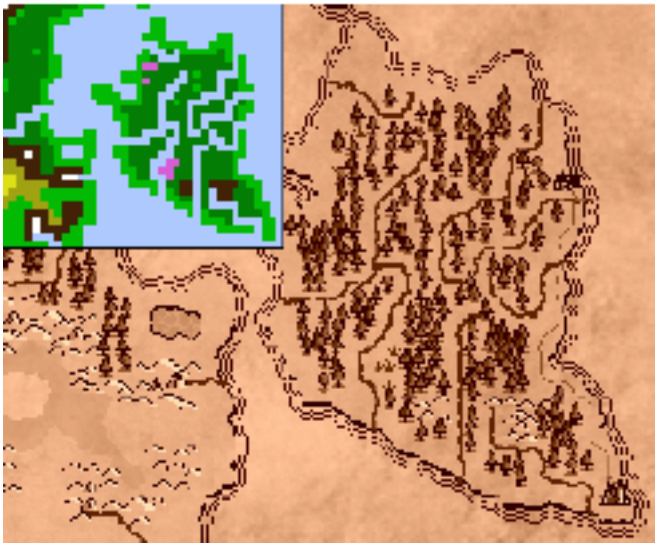

Figure 3.5: Game map (background) generated from simple image (top left) [64].

Pompeii using the grammar. The main criticism of this work is that to be fully exploited the method does require a good deal of human input in the form of script authoring.

The user interface for games can also be enhanced using PCG. Prachyabrued et al present an algorithm [64] that generates visually pleasing maps from basic pixelations, which identify key features of the terrain. The artist need only draw a simple map of the game world, and the algorithm will draw a map that can be presented to the player in the game. An example is shown in Figure 3.5. The algorithm is interesting as it can be said that it is extrapolating features based on user input.

PCG can also be used to enhance the rendering properties of video games. El-Nasfr et al [22] [21] describe a technique to dynamically create lighting in a game. Their system attempts to achieve several goals such as directing attention to important objects, evoking moods to support events in the game and maintaining

visual continuity between scenes. Several of these goals conflict with each other, so the system carefully weights and balances their priority to produce an ideal set of lights with distinct placement, colour and orientation.

A staple of Role-Playing Games (RPG) is the ability to converse with Non-Player Characters (NPC). Strong and Mateas [77] observe that game dialogue is still largely created by teams of writers and designers who hand-author every line. In their paper they propose a method to automatically generate discourse in a soap opera-like drama game. In the game two NPCs have a heated argument in which they call upon knowledge of past events until the player is asked to side with one NPC or the other. Both knowledge and the basic execution structure for an argument are written in a Hierarchical Task Network (HTN) planner application known as SHOP2. The results show that their system can automatically generate well-formed dialogue that is based on an expanding knowledge base. They express a desire to extend their research to natural language processing in the future.

Perhaps the more ambitious applications of PCG are those which generate the world on the fly as the game is played. In a paper, Nitsche et al present a project known as *Charbitat* [55], which is a game written for the Unreal engine. The most interesting feature of Charbitat is that it procedurally generates segments of the game world, which is divided into blocks. Terrain height maps and object placement are all under the control of the PCG algorithm. Based on how the player interacts with the world, the theme of tiles in the players path is changed. For example if the player interacts a lot with the "Fire Element" in game, then the game will generate fire-based world blocks. This work is of particular significance as it procedurally generates a 3D game world including enemies and objects, which is not something many PCG algorithms attempt.

This section has described some of the traditional methods for PCG. The majority of these algorithms tend to be constrained by hand-written rules and driven by a random seed. In general, this is somewhat restrictive on the content that is created as it doesn't allow new content of a high quality outside of the original intentions of the game developers to be found. In the next section we will move onto looking at algorithms that attempt to achieve just this.

3.3.2 Non-adaptive SBPCG

In this section we will review more advanced PCG algorithms, that employ an intelligent search algorithm to explore the solution space. We restrict this section

Figure 3.6: An automatically generated world in Charbitat. [55]

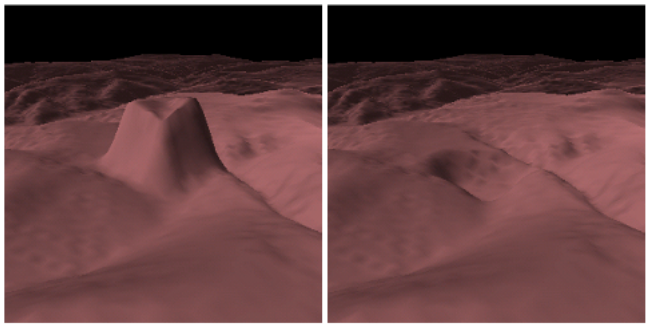

Figure 3.7: Terrain using GA [56].

to include algorithms that are not player adaptive i.e. don't try to generate content to satisfy a particular target player.

Ong et al make use of Genetic Algorithms (GA) to generate realistic height map terrain [56]. Their algorithm is divided into two stages. The user first provides a basic sketch of the terrain, dividing it into areas of different terrain type. A GA then evolves these dividing lines to create realistic transitional boundaries. Height map vertices are then imported from satellite scans to be used as a seed for evolution. Areas of influence are overlaid onto the height maps, which have domain over all vertices within their radius. A set of operations are defined which can be performed upon them such as rotation, translation and height scaling. When an operation is applied to an area it affects all of the vertices within it. It is these operations that the GA encodes and evolves. A terrain is considered fit if its variance from satellite data of the same terrain type is not too large. The result is a varied and realistic looking terrain (see Figure 3.7).

Doran et al have recently used a technique using multiple agents to generate terrain [15]. Their technique is rather unique in that it makes use of multiple agents, each with their own agenda, to generate the world. Their algorithm switches between three modes: "coastline", "land form" and "erosion". In each mode agents are deployed, of which there are five different types, which can view and change the height of any part of the world depending on what the mode's objectives are. The agents can in fact interfere with each others work causing quite interesting effects. The authors argue that their agents satisfy the criteria of Ong et al [56] in building an entertaining world environment. The technique is very interesting, but it is unclear what the advantage of using this method is without the presence of a fitness metric. The technique seems to breaking the global task of generating an environment up into smaller tasks only.

Togelius et al [79] experimented with several types of ML algorithm to generate racing tracks for a 2D racing game. Tracks in the game consist of b-splines which can be parametrically modified to produce varied curvature. The first phase in their methodology was to create an agent that had the ability to drive around a track. The agent was trained upon examples of a human playing the game using K-Nearest Neighbour (KNN), a Multi-Layer Perceptron (MLP) and a GA algorithm known as Cascading Elitism (CE). The trained agent is used as a tool to measure fitness during evolution of the race track content. They incorporate the work of Malone [50], Koster [38] and their own informal ideas into the fitness measure. Essentially, a good balance of challenge, variation in challenge and the ability to go at fast speeds are all positively rewarded. The experiment did not produce entirely desirable results likely due to the choice of mutation operators used in the second phase. Two examples of evolved tracks are shown in Figure 3.8 and Figure 3.9. In similar area of application, Loiacono et al apply genetic algorithms to the more advanced 3D Open Racing Car Simulator (TORCS). In this case they do not use bots, and instead evolve tracks using two fitness functions based on the entropy of track curvature and potential speed. Their results are promising in that they show more highly evolved tracks are preferred to less evolved tracks by 71.43% of players, validating that the evolutionary algorithm is producing increasingly desirable content. It remains to be seen if tracks can be evolved which are as, or more, appreciated than hand-built tracks.

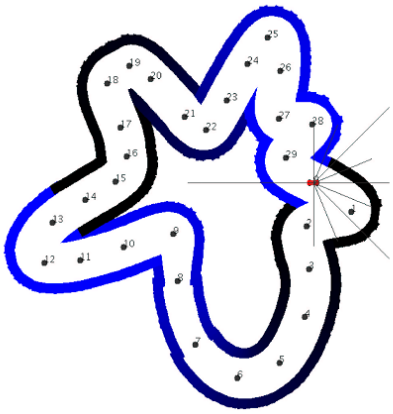

Figure 3.8: Evolved Tracks 1 [79].

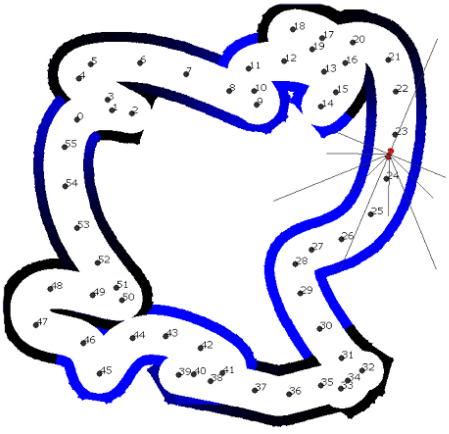

Figure 3.9: Evolved Tracks 2 [79].

In his MSc thesis, Laskov discusses his usage of the Temporal Difference Learning (TDL) algorithm SARSA to generate levels in a side-scrolling 3D platform game [41]. The player can move left, move right, jump and collect treasure. Levels that provide multiple branches, or paths through the level, are assigned a higher fitness along with levels that have treasure and dangers in them. He addresses the issue of accessibility by using a ballistic agent, which is a simulation of the player than moves through the level and performs a jump at every possible location. A lack of accessibility in a level can severely penalise the fitness of a level dependant on how far into the level the problem occurs. His results show that playable levels can be generated within reasonable time constraints and he states that it could be used to reduce the amount of manual work required by game designers.

Togelius et al present an attempt at evolving the rules of a game in an environment similar to Pac-Man [81]. The game world consists of a discrete grid, walls and coloured objects (see Figure 3.10). The player controls one of the objects and attempts to increase their score before they are killed. Variables under control by evolution include the number of items of each colour, their movement and the effect of a collision between two types of coloured object. The rules of the game are evolved using a Hill Climber algorithm that starts with a random set of parameters. The fitness of a game is analysed in two stages. The first stage uses agents with random behaviour to play the game. If they score well the game is too easy and is discarded. The second stage trains an MLP to play the game. The fitness function assigns a very low fitness to easy or very hard games based upon the MLP's behaviour. Games that can be learnt quickly are given a high score. The results of the experiment produced playable games but not games that were particularly well designed or fun.

Sorenson et al use the concept of *rhythm groups* as the basis of a fitness function to evolve levels for Super Mario Brothers and Zelda [76]. Rhythm groups alternating periods of game play with low and high challenge, which are separated by periods of rest, where the player is not being particularly taxed. The authors argue that the calculation of how fun a level is is based upon the way in which these groups occur within the level. Their method requires a function $c(t)$ which can identify the challenge at any particular time in a level. For both application games this is defined statically. For example, in the case of Super Mario they define it as a formula based upon the distance between platforms and the length

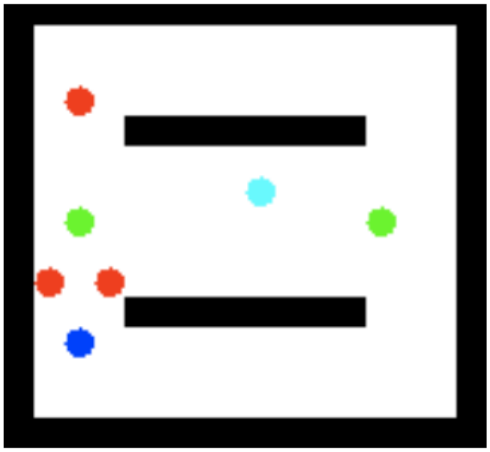

Figure 3.10: An evolved Pac-man like game [81].

of platforms that the main character has to jump on to. An algorithm is then employed which looks for rhythm groups in any level, which sets up boundaries between rhythm groups by looking for periods of rest. A fitness function defined which rewards levels where the total challenge distributed in different rhythm groups is of the appropriate scale. The fitness function is then used as the basis for a genetic algorithm, which is constrained by several hand tailored rules to ensure a minimum level of quality. They do not verify their system using human participants, rather they validate the levels produced based on industry experience of what constitutes good levels. We would argue this is not a good approach to validation, since level designers often regard the games they produce as being good, but public opinion can vary drastically. The method also makes a very large assumption that challenge is critical to game enjoyment, whereas in some games there is no true notion of challenge at all. The approach could perhaps be enhanced by incorporating different measures to be optimized beyond challenge, but it might be difficult to map the concept of rhythm groups on to these.

Basing the idea of fitness on a single quality, such as challenge, may not be a good approach. The idea that fitness is multi-faceted and may require multiple functions to be optimized is a better approach. Togelius et al use Multi-objective Evolutionary Algorithms (MOEA) for this purpose in their paper on evolving maps for multi-player RTS games [80]. MOEA are a special type of EA that attempts to satisfy multiple fitness functions by means of a *Pareto front*. The authors target applications are an imaginary RTS, which has many of the features of commercial games, and the game Starcraft (1998) by Blizzard Entertainment.

Their representation scheme allows bases and resources to be placed and allows the terrain features to be changed. They defined a number of fitness functions based on the concepts of interestingness, fairness, playability and skill differentiation based loosely on the theories of Malone [50], Czikszentmihalyi [13] and Koster [38] and then run MOEAs on single and double and triplet combinations of the functions. The evaluation allowed players of evolved maps to tag the map with various values including "Interesting", "Fun" and "Good Game-play" and the results show that the fitness functions defined, in conjunction with the MOEA do indeed evoke the desired properties. The main criticism of this work is that the fitness functions are statically defined and therefore very specific to RTS games, and perhaps even just the games defined in this study. The ideas of tagging content i.e. categorizing it and being able to optimizing multiple properties of content is useful.

In terms of the representations Togelius et al describe in their paper on SBPCG [82], most approaches in this literature review are high-level. However, in a paper by Cardamone et al, four relatively low-level and representations are used [10]. Their target platform is the open source multi-player first person shooter Cube 2, which has some functionality similar to Minecraft (2009). Cube 2 allows the player to actually edit the game as it is being played, raising terrain, adding models and adding various power-ups and spawn points to the level. The genotype representations used by the author allow the structure of a single-floor level to be defined, placing walls and open spaces. The four representations are: (a) grid-based, a very direct representation where the genotype controls the activation state of $9.9 = 81$ blocks, (b) "all-white", where the entire level is assumed to be empty and the genotype controls the placement of walls, (c) "all-black", where the entire level is full and the genotype controls the extraction of cavities and (d) where the genotype represents the behaviour of an agent that "digs out" the level. The fitness of a level is a static metric based on how long players fight each other, assuming that this results in more action and therefore is more enjoyable. The authors use bots, instead of human players, citing that it would be impractical for humans to play so many levels. The authors evaluate the results of the experiments with all four representations by hand. This paper raises some interesting points, including whether bots can be used affectively in the evaluation of content in a SBPCG algorithm and that low-level representations can be viable. The method can be criticized since it can be very difficult to construct

bots that accurately represent human behaviour and their choice of fitness metric was very limited. One can imagine that a player, for example, spend much of their time fighting, but also having a bad time.

In another paper for PCG in Super Mario Brothers, Shaker et al attempt to generate levels using grammars [68]. The grammar consists of rules that govern the location of environmental features like hills, cannons and platforms as well as the location of monsters. Their generator evolves content using an algorithm based on Genetic Programming (GP). The fitness function rates content based on the number chunks placed in the level and conflicts between chunks. They evaluate their generator by comparing its output with that of the original level generator included with Infinite Mario Brothers and the authors level generator from a previous paper [69], which generated content based on six features. The generators are evaluated by assessing their *expressiveness*, based on a framework designed by Smith et al [74]. The metric incorporates: (a) linearity, which considers the variation in hills throughout the level, (b) density, which measures the stacking of hills on top of each other (c) leniency, which is based on difficulty and (d) diversity, a score that looks at gaps, platforms and monsters. Their results do not show that any of the generators has a clear advantage over the others, with each of them being better in one score while being worse in others. However, the paper does show an interesting approach to evaluating expressiveness in the case of a 2D game. The use of grammars is interesting, in that it potentially increases the diversity of the content being generated, but in this case, the fitness function for the GP was not entirely geared towards optimizing content quality for players. Shaker et al have recently applied grammar evolution to puzzle games, and other researchers have also recently been experimenting with grammars for PCG, including Dormans et al [17] and Smith et al [73], with promising results.

The precise definition of content is rather wide. Kerssemakers et al ask the question, "Can PCG generate PCG generators?", or in other words, can the content generators themselves be generated [37]? Their target is Infinite Mario Brothers. Each level generator consists of agents that move through the level painting it with features. Each agent has attributes including its spawn time, starting position, movement style and the actions it performs. In many ways, one can consider each generator as being in similar in operation to the method described in Section 3.3.1 by Doran et al for terrain [15]. The generator generator, or "outer generator" as they term it uses a *interactive* genetic algorithm, where

the fitness function is a human who can evaluate members of the population. The authors do not provide concrete results regarding how well their system works, but the approach itself is quite unique. The approach, if it were to be made more feasible, would need a more automated method for evaluating inner generators, as the developer of the system would eventually become very fatigued from reviewing the progress of each generator that the GA creates.

3.3.3 Rule-based player-adaptive SBPCG

In this section we will review two SBPCG approaches that attempt to adapt content for a target player. This approach to SBPCG is quite challenging, as such algorithms have to deal with player types that may not have been encountered before-hand. This section will only address algorithms that adapt to the player, but do so using some form of static defined metric, for example a rule-based solution.

Hastings et al discuss their use of NeuroEvolution of Augmenting Topologies (NEAT) in the game Galactic Arms Race (GAR) [30]. GAR requires the player to man a space craft and destroy enemies using upgradable weapons. The unique feature is that the weapons in GAR are evolved using an extension to NEAT known as content generation NEAT (cgNEAT). The basic idea of NEAT is to evolve not only the weights in a neural network but the network topology itself using GA. The primary fitness measure of a weapon in the game is how much a player decides to use it. Powerful and fun weapons are more likely to be used by a player and as such their fitness and probability to have their properties appear in future weapons is increased. Their results showed that the game did indeed evolve entertaining and surprising weapons. They claim that automatic content generation is a viable technology and could be used to provide a user with a continual stream of new content. Hastings et al propose an extension to GAR that allows the player to influence the creation of new weapons more directly [31]. In this addition to the engine players can view the neural network structure of a particular weapon and modify it. This concept is interesting as it suggests that the machine learning mechanisms in the game can be exposed to players to make the game more interesting. The adaptation algorithm makes a big assumption, however, that weapons that are used more frequently are the most fun. It may be simply that such weapons make the game easier, thus reducing the overall quality of the game.

Figure 3.11: Galactic Arms Race [30].

Every algorithm in this literature so far has concerned itself with adapting game content such as enemies, weapons and the game world structure. In their paper, Plans and Morelli, instead adapt music for Super Mario in real-time using genetic algorithms [61]. They frame their work under the principles of EDPCG and SBPCG. Their approaches first evaluates the frustration, challenge and fun in real-time by using metrics that evaluate various game-play events such as the number of shells kicked, coins collected and amount of time alive. These three values are fed into simple weighted formula that estimates what level "excitement" is being experienced. Their system then adapts the music to match the level of excitement in real-time. Their evaluation proved inconclusive due to the number of participants in their study. The main criticisms of this work are that the low-level metrics used are quite specific to Super Mario, and require developer expert knowledge to determine. Such knowledge may not be available and may actually vary for different types of player.

3.3.4 Learning-based player-adaptive Algorithms

In this final sub-section, we will review algorithms that learn a metric for assessing content, before using it to adapt content to a specific player. There are currently only a few examples of algorithms that meet this criteria.

The first example we review in this section is a recent paper by Liapis et al which aims to evolve the aesthetics of space-ships in a simple 2D game [43]. The ships are represented using a form of Neural Network known as Compositional Pattern-Producing Networks (CPPNs). The CPPN receives a sequence 2D co-ordinates as input representing 15 points in a circle and returns a sequence of 2D co-ordinates giving the pattern of the ships base shape. The activation functions

of the CPPN modify the shape of the space ship its represents. The authors define four functions which calculate aesthetic properties of a ship, such as its symmetry and simplicity, based on the polygons representing it. The four functions are combined with weights to form an overall aesthetic score. The authors evaluate a combination of off-line and online evolution. For off-line evolution, NEAT and a technique known as FI-2Pop which evolves two separate populations with migration are used. For the online adaptation, a form of interactive evolution is used, which asks the player what kind of content they like and then updates the aesthetic weights. It is interesting to note that the initial population given to the user in the online stage are actually ships that were evolved off-line, demonstrating the combination of of-fline and online models. The main criticism of this work is that it requires the player to give direct feedback. If such a technique was to be applied in a real game the end user may quickly get fatigued and no longer wish to play, especially if the content being involved was not restricted to one thing. A better approach would be to implicitly detect if the player enjoyed the content, such as in Galactic Arms Race [30]. In addition, the aesthetic functions making up the weighted aesthetic score are pre-defined functions which may be open to human error in their definition.

Shaker et al build on their previous work described in Section 3.2 by taking the player models for fun, frustration and challenge and using them as functions to adapt content to a target player [69]. Their prediction model for fun is an MLP trained on the previous data set with feature selection, taking a play-log as input and four level generation parameters. Their approach to evaluation is interesting, in that they adapt content not only for players, but also for two bots. Their argument is that it is much easier to train bots to evaluate content than to get real players to do so. The two bots are taken from a recent competition on Super Mario AI and attempt to mimic human behaviour. The content space consists of levels generated using four parameters, with enough variation to produce 12000 levels. Their system works by presenting a level to the player, extracting the play-log features, then feeding these into the MLP along with potential candidate levels for the next iteration. The level parameters that optimize fun are used to generate the next level. In their first experiment they apply the algorithm for each bot, and in the second experiment they apply it to a hybrid bot, consisting of behaviour from both bots to test the ability of the approach to deal with varying game-play. They report success in both experiments in optimizing fun

reported by the MLP, but only 60% the players report preferring the adaptive model over a random model. The main criticism of this approach is that the adaptive model is being evaluated based on what the trained MLP says is fun, whereas the actual level fun the real players is having may be very different. In the case of the bot, there is of course no actual fun, so the reported results are somewhat arbitrary. It also isn't clear what the dependence between the play-log features and the level itself are. It could be that these two sets of parameters affect each other, making it unclear if the MLP can accurately predict fun for the next level, or only for the level that they were generated upon. In another paper, Shaker at al have combined the idea of adaptation with the principles of grammar representation in the paper described in Section 3.3.2 [68]. They use a similar approach involving two agents for evaluation, but this time adapt engagement, frustration and challenge. It is difficult to draw conclusions from their results, as they do not compare the performance of their adaptive model based on feedback from real people.

3.4 Togelius et al's Open Questions

In this section, we quote and discuss several questions posed by the seminal paper on the subject of SBPCG by Togelius et al [82].

"Which types of content are suitable to generate?". Togelius et al state that some types of content are easier to generate than others using optimisation algorithms and pose the question of just what can be generated quickly and reliably enough with sufficient quality. They state the answer depends on whether the content is being generated off-line or online and whether the content is optional or necessary. For example, the generation of online content may sacrifice quality to meet the demands for speed. We would add that much of the content currently generated using SBPCG tends to be relatively basic in nature, as algorithms tend to have only high-level control over the content produced. We believe it remains to be seen if SBPCG can be successful when it has more fine control over content, such as the vertices in 3D models or detailed control over levels such as the position of enemies and items.

"How can we avoid catastrophic failure?". Togelius et al state that one of the issues preventing the adoption of PCG by the commercial games industry is the belief that the content generated is unreliable. They state that presenting

the player with any content that is unplayable is unacceptable. They suggest that limiting the content representation space or using simulation-based evaluation functions to automatically play through the content (e.g. bots) may be approaches towards tackling this issue, but also warn that this can reduce the diversity of content produced and also may be prohibitively computationally expensive. We agree with this analysis but would add that there are different types of catastrophic content. For example, in the case of level generation there is content that is simply unplayable, perhaps because of incorrectly placed walls or gaps which can't be crossed, and also content that is unplayable because the difficulty so high that no player of any level of skill would enjoy e.g. a room full of monsters that requires a huge amount of luck to beat. We would say that in academic research, presenting catastrophic content isn't too bad, but presenting such content to people who have paid to buy a game certainly is not acceptable. In the existing research constraints have been applied to evolutionary algorithms [76], which is one approach to solving the problem, but this brings the additional challenge of requiring the developers of the game to have in-depth knowledge of the problem space at hand, which cannot be guaranteed.

"How can we speed up content generation?". Togelius et al state that the computational expense of SBPCG can be prohibitive for both online and off-line generation. They state that depending on representation and the shape of the search space different types of algorithm can be applied which make increase efficiency. We would add that this challenge is closely related to the question of what type of content can be generated. As the content required becomes more complex in nature, the computational power required to generate it is also likely to increase. We suggest that one approach that could potentially speed up content creation is to divide the PCG process up into two parts. The first part would be off-line, generated models and perhaps pre-generating content, whereas the second part would be online and be responsible for using the models to create content for the target audience. The combination off-line and online adaptation has been attempted already [69] and it would be interesting to find out how far this concept could be pushed forward.

"How is game content best represented?". Togelius et al state that there are a number of different approaches to representing content and say that the most appropriate representation for a problem is likely to be dependant on the

required amount of novelty and reliability. They say that generating a representation scheme may in itself be a considerable task and impact on the development of the game. We would add that finding a good content representation scheme is a very difficult task, which we have touched upon in Section 3.1.2. The sheer amount of content that could potentially be generated by a PCG algorithm, including geometry, music and behaviours makes it very difficult to specify a single representation scheme. Togelius et al suggest there are five common representation schemes in their paper on SBPCG [82], and we also observe that recently there has been a surge of interest in using game description languages. We agree with Togelius et al and state that the question of representation isn't just about increasing the scope/novelty, but also ensuring that the representation does not create a very large, or awkward, search space to negotiate. We note that something not really discussed in the research literature is the interchangeability of the representation formats. For example, can a description language be converted into a higher level vector representation for easier manipulation?

"How can player models be incorporated into evaluation function?". Togelius et al state that most SBPCG approaches so far use theory-driven evaluation functions, which try to optimize some feature of content that has been assumed to be desirable. They state that in may be more advantageous to move towards evaluation functions that are based on based on information gained from experiments with real players i.e. player models. They state that existing approaches have managed to use data-driven evaluation functions for direct evaluation, that being, the assessment of feature values extracted from content. They add that it is an open question as to whether player models for simulation-based evaluation, that being, evaluation of content by playing it using artificial agents, and also whether player models can be used to evaluate content that isn't represented in a straightforward manner. We agree that basing evaluation functions on player models is probably the best approach. Pre-defining an evaluation function using prior knowledge about the problem is potentially unsafe because, quite simply, the prior knowledge may be incomplete. We believe that the first part of adapting content for target players is to find a good evaluation function, and that searching for the content itself is actually the second part of the problem. In our taxonomy of PCG in Section 3.3, we order algorithms based on their adaptation capabilities. At the high-end of the scale we review algorithms that learn their evaluation functions, which are then applied to the task of adaptation. Most of

the SBPCG approaches in this literature review focus entirely on the search side of the problem.

"Can we combine interactive and theory-driven evaluation functions?". Theory-driven evaluation functions are those which the developer uses their intuition to design. This includes almost all of the examples of PCG in Section 3.3. Togelius et al ask whether such functions can be combined with interactive methods, which evaluate content as it is being played based on explicit or implicit feedback from the player. They state that this might be one approach to speeding up SBPCG when only sparse human feedback is available. We have already mentioned the weakness in theory-driven evaluation functions, which is that the developer must have a very good understanding of the content space, whereas in actuality they may very well not. We would therefore like to extend the question posed by Togelius et al, and ask whether data-driven evaluation functions, those based on knowledge gained from player experiments, can be combined with iterative evaluation functions. Stated a different way, we ask whether it is possible to detect if the player is having fun based on knowledge gained from player experiments and therefore improve our search of the content space. The examples in Section 3.3.4 do approach this problem somewhat, but it remains to be seen if there is an abstract method for approaching the problem.

3.5 Summary

In this chapter we have reviewed video game description and categorization, player modelling and PCG. Our goal has to been to highlight the foundations on which cutting edge PCG is based, and highlight the major challenges that need to be addressed. The most common approach to PCG in the research literature at the moment is SBPCG, which has shown a lot of promise in various applications. However, the most common approach to SBPCG are evolutionary algorithms such as GAs and Genetic Programming (GP), which by their very nature bring two major problems with them. The first of which is content representation, or in other words, the genotype-to-phenotype mapping. It is very uncommon to use a direct encoding, although some papers have done this [43]. Most approaches uses an indirect encoding, which is not guaranteed to preserve the locality principle, meaning that items close in the genotype space may not be close in meaning in the phenotype space. For example, if a bad representation

was being used for a model whose responsibility was to filter out unacceptable content, then a lack of locality could result in poor content being very close to high quality content in the genotype making the model very difficult to train. Evolutionary approaches also usually require a large amount of content to be assessed as populations are evolved, which make it a good approach for off-line PCG, but not online PCG. The final major issue with current SBPCG approaches is that many of them use hand tailored fitness functions for the assessment of content. Constructing such functions is very difficult, as they often require a very good knowledge of the genotype space. In this thesis we argue that searching for content is only one part of the problem, with the other equally large problem being that of knowing what makes content good or bad.

We have summarized the current state of art in PCG and analysed the issues with the most popular approach SBPCG. In the next chapter we will introduce the Learning-Based Procedural Content Generation (LBPCG) framework, which aims to address several of said issues.

Chapter 4

Learning-Based Procedural Content Generation

4.1 Introduction

In this chapter [1] we will introduce the main contribution of the thesis, the novel *Learning-Based Procedural Content Generation* (LBPCG) framework, which aims to address some of the main issues in SBPCG highlighted in Section 3.4. The major difference between LBPCG and many existing SBPCG approaches is that the LBPCG insists that the models used to evaluate content are built using data-driven machine learning. The LBPCG builds up robust models from different contributors in the typical video games development life-cycle, requiring much less prior knowledge about the problem domain than contemporary methods do. The LBPCG does not attempt to solve the entire problem of PCG via a single model, rather, it uses a five models working in tandem to generate content for a target player. Since the LBPCG is less dependent on hard-coded content evaluation and splits the problem of PCG across different models we believe it offers a more robust approach to PCG than existing methods. It also provides several other advantages such as avoiding interruption to the player's experience, which will be explained in this section.

[1] This chapter is adapted from J. Roberts and K. Chen, "Learning-Based Procedural Content Generation", arXiv:1308.6415 [cs.AI], 2013.

4.2 Motivation

The commercial video games development process, like most software engineering processes, is well-structured after decades of experience. Unlike many software engineer processes, video games development needs to incorporate work-flows for content including everything from music and sound effects to skeletal animation and level design. Each type of content itself has its own development process, usually requiring several tools such as 3D Model editors, 2D art software and level editors. The games development process can require a large cohort of skilled personnel including programmers, artists, managers, testers and sound artists. Throughout this thesis, we refer to such people as *developers*. These are experts who are "in" on the development process and have the ability to change content. It is expected that they are very familiar with video games and can provide great insight from past experience. The games development process includes a lot of iterations involving internal testing. When the game being developed reaches the required standard, it is sometimes released to the public in a so-called *beta-test*. Beta-tests are very common in Massively Multi-player Online Games (MMOG) such as World of Warcraft (2004), which regularly does such tests whenever a new patch or expansion set is released. Throughout this thesis we refer to all people involved in the beta-test as *beta testers*, and they are noted for their separation from the internal development process. Beta-testers usually participate because it means they get to see the content early or just enjoy contributing to a games product. It is also a possibility that beta-testers could be rewarded in some way for participating. Beta-testers are an invaluable resource, as they not only find bugs, but can also provide quality feedback on whether the game is any good or not. Depending on feedback, it is common for the developers to remove, modify or even add new features to a game. While beta-testing is not standard in all genres or games companies, the contribution of beta-testers forms an integral part of the LBPCG framework. After the games development process is complete, a game "goes gold" and is released to the buying public. The end-users of the game are referred to as *target players* throughout this thesis.

In this thesis, we define *content generators* to be software that when provided with a high-level *content description object*, produce *video game content* as shown in Figure 4.1. We call this type of configuration a *content generator system*. As reviewed in Section 3.3, the video game content to be generated can be almost anything perceivable, from music to entire game levels and the content description

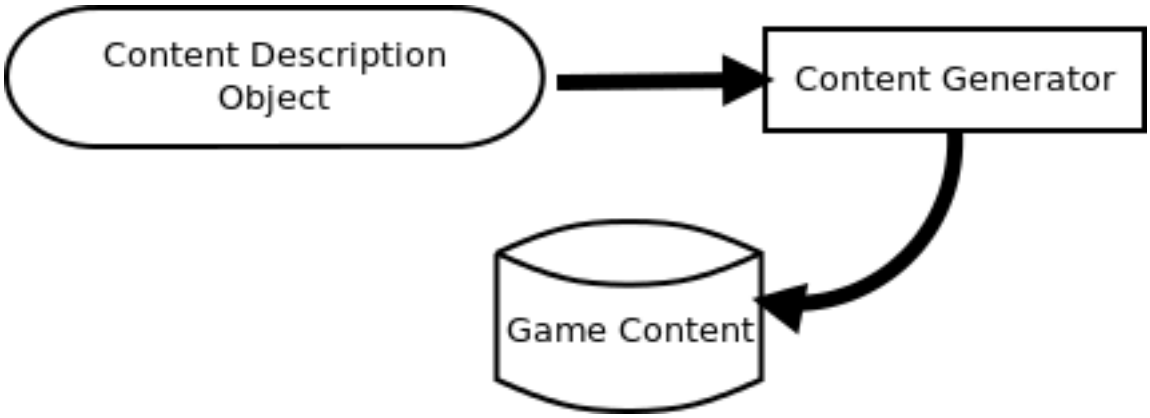

Figure 4.1: Content Generators.

object includes all formats reviewed in Section 3.1.2 such as grammars, high-level vectors and grid-like representations. One example of such an environment would be a level generator, which takes a list of corridors and object co-ordinates as input, then converts all of this to 3D geometry loading the 3D objects into place and producing a level as output. In the LBPCG, we confine the *content description object* to be a parameterized *content description vector*, which is a vector consisting of real or discrete values. Each value in the vector can describe the magnitude, quantity or ratio of a particular attribute of the video game content, for example, in the case of a level generator the content description vector could include values specifying the number of monsters, types of weapons available and aesthetic qualities of the level. While at first this format seems restrictive, many of the other representation formats can be converted to such vectors, for example almost any of the genotype representations encountered in Section 3.3. By varying all values of the content description vector we can define a *content space* which is the space of all possible content that can be generated by the content generator. Usually the content space is too large to be exhaustively searched. We define the goal of the LBPCG to be to manipulate the parameters of the content description vector to build the best content for a target player's individual preferences. We have already discussed the challenges that current approaches, such as a SBPCG, encounter while trying to solve this problem in Section 3.4.

One of the major challenges for content generation system is that the content space may contain *unacceptable content*. This is content that is not appropriate for any player. This definition does not only include technically unbeatable

content, e.g. levels that have badly placed walls making then impossible to complete, but it also includes games that couldn't really be enjoyed by any player no matter their tastes e.g. a map that consists of rooms full of monsters that in most circumstances give the player little to no chance to retaliate. In this chapter we argue that optimizing content for a target player is a multi-faceted problem, in that it is a requirement to first filter out the unacceptable content, and then search the acceptable content to optimize entertainment for a target player and that the best approach is to divide responsibility for this between separate models. As reviewed in Section 3.3 other approaches have attempted to combine models before with some success. As of the current research literature, it remains unclear as to the extent that models of different functionality can be combined to form robust composite models.

The personalization of content is not only driven by how acceptable it is or what level of challenge it poses, but also by a player's affective and cognitive experience [90]. This means that modelling target players goes beyond just matching them to a skill level. Traditionally, content was divided using difficulty. For example, in many games it is possible to select a difficulty level ranging from "Easy" up to "Hard", and the effect of changing the parameter is usually that the ability of the enemy NPCs in the game increases. Difficulty can go beyond this, for example in the adventure game Monkey Island 2 (1992) there are two difficulty levels. Selecting the easier difficulty results in simpler puzzles which require less ability to solve. We argue that difficulty is simply a specialization of a more abstract concept, that being, matching a player to a particular *category* of content. For example, a player may like "Easy opponents", but also like levels that have "Lots of plasma guns", combining these two features results in the content category "Levels with easy opponents and lots of plasma guns", which we can match the player to. Learning how to categorize the content space is a big problem that has not yet been tackled, but in the LBPCG framework it forms a major component of the approach for personalizing content to a target player.

The act of modelling a player is itself another problem, which we reviewed in Section 3.2. If the goal of a content generation system is to match players to a particular category of content, then there are two basic approaches. The first of which is to present the player with content and ask them whether they enjoyed it, which is quite intrusive and could actually be to the detriment of the process. The second is to observe the player's behaviour, and then update the model of player

preference. In the LBPCG, we prefer the second approach and a substantial part of the framework is designed around providing this functionality. Many existing approaches to player modelling are based upon pre-defined player types and styles from psychological studies. We argue that taking such an approach may not encapsulate all player types and can lead to problems in identifying the player's preferences and that a more robust approach is to build up models for player preference by learning from the players themselves.

So far we have argued in favour of the learning-based approach to PCG, which builds up its models using data from players and developers of games. In Section 3.3 and Section 3.2 we have reviewed papers where researchers have already tried similar approaches. A major problem with training models is that not only is the feedback subjective, but there is the risk that players can also provide unreliable feedback. It can take many thousands of surveys to build up an effective player model and even then the process can be hampered by missing or invalid entries. This is especially the case if one is to train models using the beta-testers, who could just be random samples of people from the internet. In a recent paper on Tomb Raider Underworld, researchers attempted to build up a predictive model to determine if a player would complete the next level of the game based on their behaviour on the previous two levels [48]. Even with a pool of 10,000 players, the researchers had difficulty due to the existence of many outliers. In this context, outliers refer to people who play the game in an irregular manner. The detection and handling of outliers and removal of unreliable experts is one problem that the LBPCG framework aims to address.

The final major problem in PCG that we will address here is that of player's preference drifting over time. Shaker et al touch on this subject in their paper on personalizing content for Super Mario [69]. Their approach was to record a play-log from the previous level that the player played and then update their evaluative function to find appropriate content using this information. Their approach was seemingly successful in that it managed to adjust the content accordingly for artificial agents, but when tested on human players it was reported that only 60% of them preferred the adaptive system to the non-adaptive system. The notion of a player changing their preference or even play-style is not far fetched, as for one example, it can be imagined that a players skill level would increase over time and thus so would their behaviour and preference for more challenging content. A PCG system is not very useful if it cannot continue to generate appropriate

content over a long period of time.

In this section we have introduced the main challenges, identified from the research literature, that the LBPCG aims to solve. In the next section we will introduce the LBPCG framework detailing each of its component models in abstract detail.

4.3 Framework

It is useful for the LBPCG framework to split the games development process into three distinct stages: (a) the development stage, which covers the process of creating the game including both its code and the content and also includes internal testing, (b) the public test stage, which is the period when the game is released to the beta testers and (c) the adaptive stage, which is the period when the game is considered complete and released to the target players.

The LBPCG framework aims to solve five challenges highlighted in the previous section:

1. Avoiding unacceptable content in the content vector space.
2. How to categorize the content in the content vector space.
3. How to exploit potentially unreliable information acquired from the public.
4. How to categorize players in terms of player type/style.
5. How to tackle the issue of concept-drift, where a player's preference changes over time, without asking invasive questions such as "Are you having fun?"

The LBPCG is designed in such a way as to learn from the developers in the development stage and from the beta-testers in the public test stage to gain as much information about the content space and player behaviour as possible, so that when the game is released to the target players it can adapt content to them as quickly as possible and minimize any interruptions to their experience. To attain this goal the LBPCG uses five separate models to learn about the content space, learn about player behaviour and then adapt to the target model. Figure 4.2 shows the three stages of development and the corresponding LBPCG sub-models that are trained during each stage. The *Initial Content Quality* (ICQ)

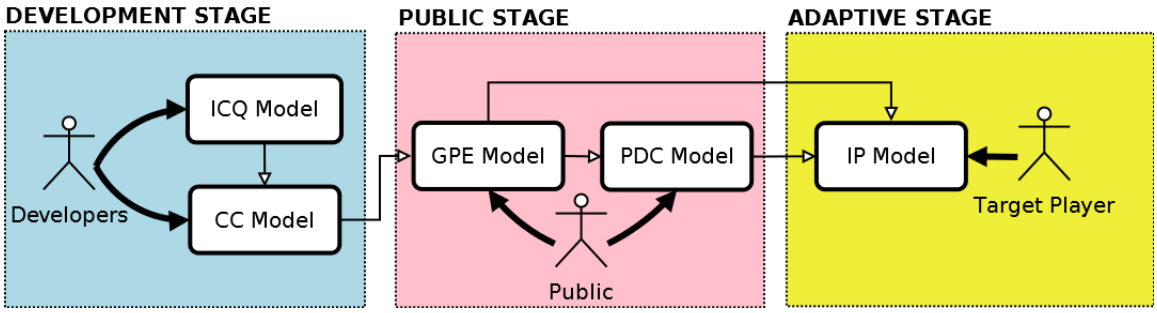

Figure 4.2: Learning-based procedure content generation (LBPCG) framework.

Model and *Content Categorization* (CC) Models are trained during the development stage and aim to encode the developer's knowledge of about what content is legitimate and how content categories are defined in order to tackle issues (1) and (2). The *Generic Player Experience* (GPE) and *Play-log Driven Categorization* (PDC) models are trained during the public test stage and encode beta testers knowledge regarding the public consensus on content and how play-logs map to player's enjoyment, addressing issues (3) and (4). The *Individual Preference* is deployed during the adaptive stage and is designed to adapt content for the target player by monitoring their play-logs, addressing issue (5).

The goal of the ICQ model is to recognize whether content is acceptable or not. If the ICQ model is capable of achieving this, then it can be used as a filter on the content vector space to carve out a manifold of vectors representing acceptable content. Any content given to the target player should be selected from within this manifold and, if the amount of unacceptable content is significantly large, can help reduce size of the search space. The problem the ICQ model aims to solve is a binary classification problem in that content is either acceptable, or it is not. If the developers had perfect knowledge of the content vector space, then they could theoretically hand craft a rule-based function that could solve the problem. In reality, it is highly unlikely that the developers would have such knowledge, and as such the LBPCG prefers a learning-based methodology. The general principle for training the ICQ model is that a few well selected examples of content are presented to the developer who provides a label for each. The content description vector for the content, along with the labels, is then used to train the ICQ model. This training method itself brings about several issues that need to be surpassed. Firstly, how to choose the well selected examples and keep them to a minimal amount to avoid asking the developer to label too

much content, and secondly, how to train up a learner with good generalization, especially if only a small number of examples are available due to constraints on the developer's time.

The CC model is designed to categorize content. The motivation for categorizing content is the observation that players often prefer content similar to content they have enjoyed before. For example, a player who enjoys a level categorized as "Hard", is likely to prefer other levels categorized as "Hard". This allows the LBPCG to assign players to categories based on their preference. The definition of what the categories are is down to the developer themselves. However, the LBPCG treats the problem of determining exactly what makes a piece of content belong to a particular category as a problem to be solved in a data-driven way, and as such the CC Model is trained in a similar fashion to the ICQ Model, by presenting the developer with well-chosen examples until enough data is available to train a classifier with good generalization ability. One can view each category as an index, and thus the problem that the CC Model aims to solve as being a mapping from the content vector space to either an ordinal or regression value. Since it is only desirable to present acceptable content to the target player, the CC Model is actually a mapping from the manifold of acceptable content in the original content vector space. The CC Model training process has the same critical issues as the ICQ Model does.

The GPE model is designed to model the public's consensus as to what content is good and bad. The input to the GPE model is a content description vector and the output is a binary value specifying whether the content was enjoyable or not. One could theoretically train a model for determining what content is good and bad by just using the developers themselves, but the developers are only a small group of people with probably a very large amount of gaming experience, making them not very representative of all possible players. This is the justification for training the GPE model on the beta-testers, which is a much large group of people more likely to consist of different types of player. In the LBPCG framework, an adequate amount of representative acceptable content is given to the beta-testers to play, who provide labels indicating whether they enjoyed the content, and this data forms the basis of the training set for the GPE Model. As we have previously mentioned, it is unlikely that feedback from the beta-testers is going to be completely reliable and there may be a large amount of noise in the training data. It could even be the case that participants have deliberately mislabelled

data. As such, the two critical issues that need to be solved in the training process are the question of how to assign a reliability score to each beta-tester and then the question of how to use this information to train the GPE Model.

The goal of the PDC Model is to implicitly detect whether a play-log represents a positive or negative experience. Recall that play-logs are generated whenever a player experiences a particular piece of content. The PDC model enables the LBPCG framework to maintain a model of a players preference without ever interrupting them with potentially intrusive questions such as "Was that fun?". Again, the PDC Model could be trained on only the developers, but this is not a good approach for the same reasons as with the GPE Model. It is likely that the developers are a small number of people with very specific play-styles and preferences, and training the PDC Model on them would probably reduce the generality of the model. The PDC model is therefore trained on the play-logs produced during the training process for the GPE Model, making use of the reliability information for each beta-tester to filter out misleading information. The inputs to the PDC Model is a play-log, along with the corresponding category for the content the play-log was generated for, and the output is binary corresponding to either a positive or negative experience. The content category is passed in as it allows the classification model to discriminate based on category, potentially making the problem easier to solve.

The IP model is designed to personalize content for a target player. It achieves this goal by strategically making use of the other four models in the LBPCG framework. By its nature, the IP model is well-suited to be a state machine that controls the process of delivering content to the player. The ideal scenario for the IP model is that it first presents a number of well-selected examples of content, which are deemed acceptable by the ICQ model, to the target player. By using the resulting play-logs and the output of the CC Model the IP model can then determine which category of content the player prefers, and then proceed to produce content from this category until the players preference drifts into another category. If the player's preference cannot be detected over a long period of time, the GPE model can be used to find content that is generally regarded as fun, and present such content to the player instead. As such, the IP model has to address three major issues. Firstly, the question of how many games need to be presented to the target player to detect their categorical preference. Secondly, how to ensure that games presented to the player from a particular category have

good variation and are not too similar. Thirdly, how to quickly detect concept drift and respond to it.

To summarize, the LBPCG framework aims to provide a systematic solution to many of the problems in cutting-edge research into PCG. In terms of the EDPCG [90] framework, one could view the LBPCG as using game-play based Player Experience Modelling and as using an interactive evaluation function since it uses play-logs in conjunction with the PDC model to assess player experience and the quality of content with respect to the player. However, this isn't such a straightforward mapping, since prior knowledge is gained before the adaptive stage from the developers and beta-testers causing the IP model to also form somewhat of a direct evaluation function. In the next section we will go into detail about the enabling techniques that take the LBPCG framework from a conceptual level to a partial implementation level.

4.4 Enabling techniques

In this section we will present the enabling techniques we use for each model in the LBPCG framework. In Section 4.4.1 and Section 4.4.2 we detail how active learning can be used to train the ICQ and CC models, respectively. In Section 4.4.3 we introduce the expectation-maximization based crowd sourcing algorithm used to train the GPE model. Finally, in Section 4.4.5 we show our state-based machine solution for the IP model.

4.4.1 ICQ

The goal of the ICQ model is to be able to recognize whether a content description vector represents acceptable content or not. We denote a content space of D parameters as $\mathcal{G} \subset \mathbb{R}^D$ where $\boldsymbol{g} \in \mathcal{G}$ is the parameter vector of some content denoted by $\boldsymbol{g} = (g_1, g_2, \cdots, g_D)$. The ICQ model can be formalized as a mapping $\Phi_{ICQ} : \mathcal{G} \rightarrow \{+1, -1\}$, where $+1/-1$ indicates acceptable or unacceptable content, respectively.

To tackle the problem, the LBPCG employs a binary classifier for Φ_{ICQ} that can output a confidence, or something that can be converted to a confidence such as posterior probability, for each classification it performs [20]. Please see Section 2.2 for discussion of precisely what is meant by confidence and its relation to probability. If $\Phi_{ICQ}(g^*) = +1$, but the probability is 0.6, then the ICQ model

believes that g^* is acceptable, but is not very confident of it. The content space in most non-trivial PCG scenarios will be too large to be exhaustively searched, which means it is not possible to label each piece of content individually. It is also assumed that prior knowledge about the domain is minimal or non-existent meaning that Φ_{ICQ} needs to use learning to achieve its functionality. Since the ICQ model is trained during the development stage, the only people available to train it are the developers of the game, who are a limited resource. This means that it is not really feasible to present them with thousands of examples of content to classify and the ICQ training process must intelligently select examples that when labelled provide the greatest insight into the problem.

Algorithm 4.1. Active ICQ Learning.

1: **Purpose:** To train a classifier capable of recognizing acceptable content using active learning.
2: **Input:** The content space $\mathcal{G}$ consisting of vector representations of all content we can generate.
3: **Output:** (a) A set of labelled training data, and (b) the trained ICQ Model ϕ_{ICQ}.
4: **Initialization**: Annotate a few randomly selected games from $\mathcal{G}$, ensuring they do not already exist in T_{ICQ} and add them to the training set for the classifier. Train the classifier using the training set.
5: **while** classifier's error on T_{ICQ} is not acceptable **do**
6: Determine the best $\boldsymbol{g}^* \in \mathcal{G}$ based on a chosen query strategy
7: Have developer annotate $\boldsymbol{g}^*$, if it was not annotated already and add it along with its true label to the training set.
8: Re-train the classifier on the training set.
9: **end while**
10: Train ϕ_{ICQ} on the resulting training set.

While there are many different methods that can be used to train a classifier, the LBPCG framework advises that active learning is used (Section 2.3.8), as it is a query-based approach that aims to reduce the burden on the oracle, who is the developer in the case of the ICQ model. In our conceptions of how the LBPCG is likely to be used, we believe that the content space is best represented as a pool of examples, and as such pool-based active learning is the likely sampling scenario. We do not consider other types of sampling scenario in this thesis. Our active learning algorithm trains Φ_{ICQ} by iteratively selecting examples from $\mathcal{G}$ based on a chosen query strategy, which will provide the most value to the learning process. Each example is provided the developer for labelling, and the result is

added to the training set on which the classifier is re-trained. In Section 2.3.8 we highlighted several types of query strategy, which can be dependent on the model that needs to be trained. In this general description of the ICQ, we leave the decision of what strategy to use to the implementers of the LBPCG framework. One could, for example, use a uncertainty sampling approach if the model has probabilistic outputs. We defined how posterior probabilities can be calculated for two such models, random forests and SVMs, in Section 2.3.6 and Section 2.4.1, respectively. The entire learning process for the ICQ model is summarized in Algorithm 4.1.

The termination condition for training the ICQ model is reached when a satisfactory error is achieved on a test set T_{ICQ}, although, other metrics such as the f-measure (described in Section 2.3.1) could potentially be used. What constitutes a satisfactory error is also something we leave to the implementer of the system. This is likely to be based on examining the examples that the ICQ model correctly identifies and those that it generally gets wrong, to provide some context to the error rates. The generation of T_{ICQ} is itself another problem that needs to be solved. It is desirable that T_{ICQ} is as representative of the space as possible, but we are constrained by the fact that the developer needs to generate T_{ICQ} and thus we need to keep the size of the set to an appropriate level. To generate T_{ICQ}, we recommend that first a clustering algorithm is applied to $\mathcal{G}$, such as those mentioned in Section 2.4. T_{ICQ} should be generated based on the centroids of the clusters, which are then labelled and added to T_{ICQ}. A clustering algorithm ensures that the centroids are relatively spread out, and while not an being a perfect or sophisticated solution, the approach does guarantee some degree of spread in $\mathcal{G}$ of the test examples. While labelling the members of T_{ICQ}, we also recommend that category labels and any other useful annotation is collected, as this can be potentially re-used in the subsequent stages of the LBPCG.

4.4.2 CC

The purpose of the CC Model is to categorize acceptable content. A requirement for the CC Model is that the developers define content features $\mathcal{F} = (F_1, \cdots, F_{|\mathcal{F}|})$. Example content features include challenge, frustration and anxiety. Each feature takes a discrete or continuous value, for example, if F_i represents difficulty then it could take a value from the set $\{\text{“}Easy\text{”}, \text{“}Medium\text{”}, \text{“}Hard\text{”}\}$. The Cartesian product of all possible content feature values $\mathcal{C} = \cup_{i=1}^{|\mathcal{F}|} F_i$ is the set

of all categories. The CC Model problem can therefore be formalized as learning a mapping $\Phi_{CC}: \mathcal{G}_a \rightarrow \mathcal{C}$. Note that the CC model is expected to only be able to classify acceptable content.

Algorithm 4.2. Active Learning for CC.

1: **Purpose:** To train a classifier capable of recognizing the category of any content in $\mathcal{G}_a$ using active learning.
2: **Input:** The content space $\mathcal{G}_a$, consisting of all acceptable content according to ϕ_{ICQ}.
3: **Output:** A labelled training set for ϕ_{CC}, and the trained classifier ϕ_{CC}.
4: **Initialization**: Annotate a few randomly selected games that do not already exist in T_{CC} and add them to the training set. Train each of the $|\mathcal{F}|$ learners with the training set.
5: **for all** $F_f \in \mathcal{F}$ **do**
6: **while** test error on T_{CC} is unacceptable in terms of F_f **do**
7: Determine the best $\boldsymbol{g}^* \in \mathcal{G}$ based on a chosen query strategy
8: Have developer annotate $\boldsymbol{g}^*$ if it was not already annotated and add it along with its ground-truth label to the training set.
9: Re-train the fth learner with new training set.
10: **end while**
11: **end for**
12: Train ϕ_{CC} using the training set.

The CC model has the same potential problems as the ICQ model, in that the original content space $\mathcal{G}$ is too large to be exhaustively searched and that it is desirable to avoid asking the developers to label too many games. While it is desirable to keep the number of queries to a minimum, the CC model must also have good generalization. To ensure the CC Model meets these requirements, the search-space for the CC model is the original search space $\mathcal{G}$ filtered by the the ICQ Model Φ_{ICQ}, which is a sub-space of acceptable content $\mathcal{G}_a$. To speed the process up, the LBPCG framework advises that labelled data from the ICQ model training process can be re-used in the CC model training process, as long as the respective content is labelled as acceptable. To evaluate the CC model a validation set T_{CC} is formed, which consists of representative acceptable games chosen from the space $\mathcal{G}_a$. The CC Model Φ_{CC} is trained using active learning as described in Algorithm 4.2. As with the ICQ model training process, we leave the details of the implementation, such as the query strategy, to the implementers of the framework.

T_{CC} can be generated in a similar manner to T_{ICQ}, by applying clustering to

the space $\mathcal{G}_a$ and labelling the centroids. However, care should be taken to ensure that there are enough representative examples from each category. As such it is advisable, if after labelling each centroid there is insufficient representation, to continue to label randomly selected games from G_a until sufficient amounts of representatives from each category are found. If available, using prior knowledge to increase the speed of test set generation is a good idea.

After an acceptable error rate is achieved in the validation set T_{CC}, the active learning process is terminated resulting in a trained model Φ_{CC}, capable of recognizing the category of any acceptable content in the content space. It is imperative that the error rate is low on the CC model, otherwise the LBPCG framework may produce content from the incorrect category for a target player, even if the target player's preferences are correctly identified. As such, it is important that the CC Model is able to output a confidence along with its output, which says how confident it is in its decision. The confidence value allows the LBPCG framework to reject content that the CC Model is unsure about, thus potentially reducing the error rate. A possible side-effect of this is to reduce the diversity of the content, and this is a trade-off that the implementers of the LBPCG framework need to consider.

4.4.3 GPE

The GPE model has two objectives. The first objective is to enable the LBPCG framework to identify content that is likely to be appreciated by the majority of people. This is useful, as it can be used as a fall-back solution if the LBPCG framework fails to adapt to a target player. This problem is solved by learning a mapping $\Phi_{GPE} : \mathcal{G}_a \rightarrow \{+1, -1\}$, where $+1$ and -1 correspond to desirable content and undesirable content, respectively. The second objective of the GPE model is to identify the reliability of beta-testers. This information is useful both in the training of the GPE model and the PDC model, described in the next section. To train the GPE model, the implementers of the LBPCG framework need to choose a set of highly representative examples of content, $\mathcal{G}_{GPE} \subseteq \mathcal{G}_a$ to give to P beta-testers during the public testing phase of development. Each beta-tester will play as much content from $\mathcal{G}_{GPE}$ as they choose. After each example they play, they provide binary feedback representing either a desirable or undesirable experience and a play-log corresponding to their behaviour during the experience is also recorded. The feedback for player p with content example

$\mathcal{G}^n_{GPE}$ is denoted as $y^{(p)}_n$. It may also be useful to collect more feedback from the beta-testers for evaluation purposes, such as information pertaining to emotive states or desirability expressed as an ordinal value. This allows the implementers of the LBPCG framework to perform statistical analysis of their data.

The GPE model problem can be further formalized by stating that for each sample of content $\mathcal{G}^n_{GPE}$ played by the beta testers, the GPE model must find the desirability of it according to public consensus, $\hat{y}_n$ with confidence level γ_n as well as a reliability factor for each player p. The reliability factor for a player is described using sensitivity and specificity, which are the true positive rate and true negative rate of the player respectively. We denote the sensitivity and specificity of a player p as $\alpha^{(p)}$ and $\beta^{(p)}$. See Section 2.3.1 for further discussion of this terminology.

Our requirements for the GPE Model are that it is capable of assessing the true label of examples given potentially noisy and incomplete feedback, and that it can also estimate reliability information for the labellers. A crowd-sourcing algorithm created by Raykar et al [65] meets these requirements. We refer to this algorithm as *Crowd-EM* throughout this thesis. Crowd-EM is an expectation-maximization algorithm (see Section 2.3.7) and its details are given in Algorithm 4.3. Since each beta-tester might not play all content in $\mathcal{G}_{GPE}$, one simply needs to substitute P, the number of all beta players, with P_n, the number of beta players who actually played content $\mathcal{G}^n_{GPE}$, as well as N in Algorithm 4.3, the number of all games in $\mathcal{G}_{GPE}$, with N_p, the number of games that player p actually played.

In addition to a functional classification model Φ_{GPE}, on completion of the GPE model learning process, the LBPCG framework has access to two important pieces of information which are used by both the PDC model and IP model. The first of which is the reliability factor $(\alpha^{(p)}, \beta^{(p)})$ of each of the P beta-testers, and the second of which is a desirability score for each sample of content in $\mathcal{G}_{GPE}$. This desirability score can be used to rank each sample of content in $\mathcal{G}_{GPE}$ and therefore produce a list of top samples of content from each category. The reliability scores are used by the PDC model to filter out unreliable training data, so it can build up a more robust model of player behaviour, and the ranked list of games are used by the IP model to help find a players categorical preference.

Algorithm 4.3. Crowd-EM for Learning GPE.

Purpose: To train a model ϕ_{GPE} that captures public opinion on the content space and additionally assess the reliability of participants in the GPE Model training process.
Input: A labelled set of data consisting of content from $\mathcal{G}_{GPE}$, along with labels provided by the public participants.
Output: (a) The model ϕ_{GPE}, and (b) sensitivity and specificity scores for each participant, α and β.
Initialization
Let $N = |\mathcal{G}_{GPE}|$.
For $p=1, \cdots, P$, set $\alpha^{(p)}(0) = \beta^{(p)}(0) = 0.5$.
For $n=1, \cdots, N$, set $\gamma_n(0) = \frac{1}{P}\sum_{p=1}^{P} y_n^{(p)}$.
Train a regression model, $f(\boldsymbol{g}, \Theta)$, by finding optimal parameters, $\Theta^*(0)$, with the training set of N examples, $\left\{\left(\boldsymbol{g}_n, \mu_n(0)\right)\right\}_{n=1}^{N}$, and set $t=1$.
E-Step
For $n=1, \cdots, N$, calculate

$$h_n(t) = f(\boldsymbol{g}_n, \Theta^*(t-1)),$$

$$a_n(t) = \prod_{p=1}^{P} \left[\alpha^{(p)}(t-1)\right]^{y_n^{(p)}} \left[1-\alpha^{(p)}(t-1)\right]^{1-y_n^{(p)}},$$

$$b_n(t) = \prod_{p=1}^{P} \left[\beta^{(p)}(t-1)\right]^{1-y_n^{(p)}} \left[1-\beta^{(p)}(t-1)\right]^{y_n^{(p)}},$$

$$\gamma_n(t) = \frac{a_n(t)h_n(t)}{a_n(t)h_n(t) + b_n(t)[1-h_n(t)]}.$$

M-Step
For $p=1, \cdots, P$, update

$$\alpha^{(p)}(t) = \frac{\sum_{n=1}^{N} \gamma_n(t) y_n^{(p)}}{\sum_{n=1}^{N} \gamma_n(t)},$$

$$\beta^{(p)}(t) = \frac{\sum_{n=1}^{N} [1-\gamma_n(t)](1-y_n^{(p)})}{\sum_{n=1}^{N} [1-\gamma_n(t)]}.$$

Re-train the chosen regressor, $f(\boldsymbol{g}, \Theta)$, by finding optimal parameters, $\Theta^*(t)$, with the training set of N examples, $\left\{\left(\boldsymbol{g}_n, \mu_n(t)\right)\right\}_{n=1}^{N}$, and set $t=t+1$.
Repeat both *E-Step* and *M-Step* **until** convergence.
Let ϕ_{GPE} be the model f.

4.4.4 PDC

The goal of the PDC model is to accurately detect whether an experience with a sample of content was positive or negative based on a play-log of the experience and the category of the content. The space of all possible play-logs of

L play-log features is denoted as $\mathcal{L} \subset \mathbb{R}^L$, and $l \in \mathcal{L}$ represents the play-log for a single experience where l_i represents the value for a particular play-log attribute. Formally, the goal of training the PDC model is to create a mapping $\Phi_{PDC} : \mathcal{L} \times \mathcal{C} \rightarrow \{-1, +1\}$, where $+1$ and -1 represent a positive experience and negative experience, respectively. The problem is therefore binary classification, but could also be formulated as a multi-classification problem if ordinal rather than binary output is desired.

During the public testing phase, a set of O play-logs along with corresponding feedback from beta-testers are recorded using a data collection system. In addition, each play-log is associated with a content sample in $\mathcal{G}_{GPE}$, for which there is a category label provided by the developers. As such, the PDC model has a training data set $\mathcal{D}_{PDC} = \left\{[(l_i \in \mathcal{L}, c_i \in \mathcal{C}), y_i]\right\}_{i=1}^{O}$. Due to the likelihood of unreliable feedback from the beta-testers, the GPE model needs to take into account the $(\alpha^{(p)}, \beta^{(p)})$ reliability information for each player p and only use data from people whose sensitivity and specificity are above a certain threshold, and thus considered to be reliable. However, selecting an appropriate threshold is a difficult problem. It is conceivable that people who are labelled with a lower sensitivity or specificity may in fact simply be outliers, in that they are reporting their true preference but that preference is different to the vast majority of other people. Such people are not liars and their feedback is valid. To tackle this problem, the LBPCG framework uses an ensemble learning algorithm to train multiple classifiers based on subsets of $\mathcal{D}_{PDC}$ created by varying thresholds on the sensitivity and specificity. This results in M classifiers, whose weight can be adjusted by calculating their potential accuracy on cross-validation sets. Algorithm 4.4 describes the algorithm used for training the GPE Model.

4.4.5 IP

The IP model has three objectives: (a) to discover a target player's categorical preference, (b) to generate content for the target player from their preferred category, and (c) to avoid complete failure in the case where the target player's categorical preference cannot be identified. The general methodology of the IP model is to present content to the target player and observe how they interact with it. By examining the play-log of the interaction, the IP model should determine what the target player's categorical preference is. These functional requirements suggest that the IP model is well suited to be implemented as a finite state

Algorithm 4.4. Learning for PDC.

1: **Purpose**: To train a model ϕ_{PDC} that can recognize whether a player enjoyed some content from $\mathcal{G}_a$ by looking at the play-log and content category.
2: **Input**: (a) Training data $\mathcal{D}_{PDC}$ consisting of play-logs and category labels, and (b) sensitivity and specificity scores for each participant in the public survey, α and β.
3: **Output**: The trained model ϕ_{PDC}.
4: **Initialization**: Divide the training data set, $\mathcal{D}_{PDC}$, collected in beta tests into M training subsets, $\mathcal{D}^1_{PDC}, \cdots, \mathcal{D}^M_{PDC}$ by setting a number of thresholds for $\alpha^{(p)}$ and $\beta^{(p)}$, respectively. Choose a classification model for $f[(\boldsymbol{l}, \boldsymbol{c}), \Theta]$, which will be a new member of the ensemble.
5: **for** $m = 1$ **to** $m = M$ **do**
6: Train $f[(\boldsymbol{l}, \boldsymbol{c}, \Theta_m)]$ on $\mathcal{D}^m_{PDC}$ via cross-validation by finding optimal parameter, Θ^*_m.
7: Record its accuracy, u_m, on cross-validation.
8: **end for**
9: Calculate weights:

$$w_m = \frac{\exp(u_m)}{\sum_{k=1}^{M} \exp(u_k)} \quad \text{for } m = 1, \cdots, M.$$

10: Construct the ensemble classifier:

$$F[(\boldsymbol{l}, \boldsymbol{c}), \Theta^*] = \sum_{k=1}^{M} w_k f[(\boldsymbol{l}, \boldsymbol{c}), \Theta^*_k].$$

machine that runs in an infinite loop during the adaptive stage. The LBPCG framework's enabling technique is therefore a finite state machine consisting of three modes with multiple states each, to deal with the IP models three objectives in turn.

Figure 4.3 shows the CATEGORIZE state, which is the first state entered by the IP model. It is designed to detect a new target player's categorical preference by presenting content from each category and analysing the resulting play-log to detect if the experience was desirable. Since the CATEGORIZE state is the first state entered, it is important to present content that is of good quality as soon as possible. If the player is immediately presented with poor quality content while the IP model tries to find out more about them, then it is possible they will simply give up, defeating the purpose of the entire framework. The best approach to the problem, without prior knowledge about the target player's individual preference, is to present content $\mathcal{G}_{GPE}$ in the order of rank determined during the GPE model

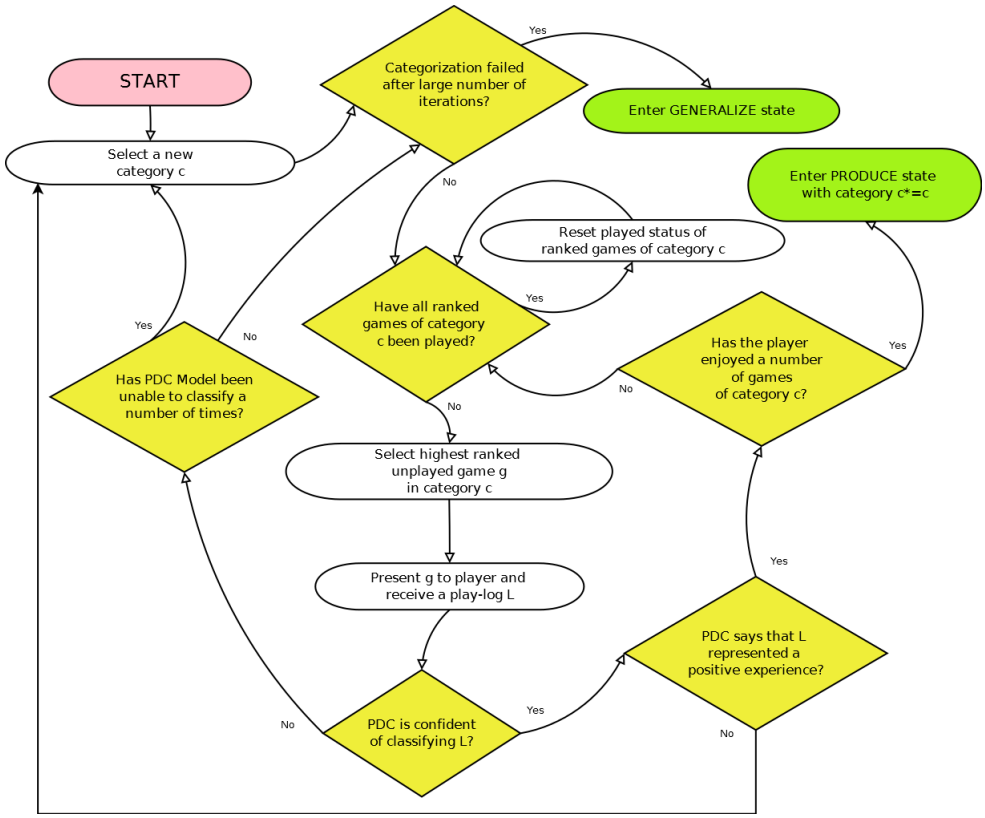

Figure 4.3: The CATEGORIZE state in the IP model.

training process. Even if the top content does not match the player's categorical preference, it has at least been judged to be of a certain level of quality by the public. It is also helpful that the exact category of each member of $\mathcal{G}_{GPE}$ is already known, since it was labelled by the developers. If it is determined that the target player player enjoys enough content of a certain category, by observing their play-logs and passing them through the PDC model, then the IP Model switches to the PRODUCE state. If the IP Model cannot determine the player's categorical preference, then it switches to the GENERALIZE state.

Figure 4.4 shows the PRODUCE state, which is entered when it is detected that the target player enjoys content of a particular category. The PRODUCE state continues to present content from the determined category, until a play-log produced by the player indicates they are not longer having fun. Whenever this concept drift is detected, the IP model switches back to the CATEGORIZE state.

Figure 4.5 shows the GENERALIZE state, which is designed to mitigate the situation where the target player's categorical preference is unknown. This is not the ideal state to enter, as the whole point of the LBPCG is to match content

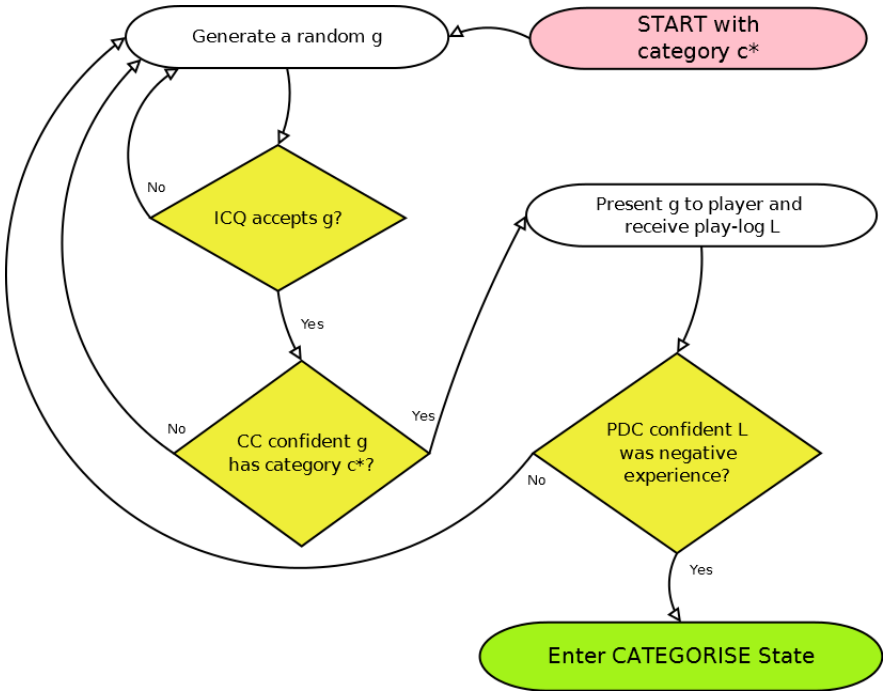

Figure 4.4: The PRODUCE state in the IP model.

to a player's preferences, however, it is a good fall back solution that guarantees some degree of content quality. The GENERALIZE state uses the ICQ and GPE models to find content from $\mathcal{G}_a$ that is acceptable and likely to appreciated by members of the general public. It uses the CC model to find the category of any games that are presented, with the hope that the player's categorical preference will eventually become evident so the IP model can switch into the PRODUCE state.

It should be noted that throughout all states that the output of the PDC model and CC model is only trusted if its confidence of classification is above a certain threshold. The reason for this is that it is likely that these models will occasionally be unable to classify the data they are presented, and it is important that the IP Model responds accordingly rather than taking a risk and presenting the target player with undesirable content.

To summarize, the IP model is a state machine that makes use of all other models in the LBPCG to generate personalized content for an arbitrary target player whose initial preferences are unknown. It should be emphasized that the IP model requires only play-logs detailing the target player's experiences with content, avoiding interruption to their experience.

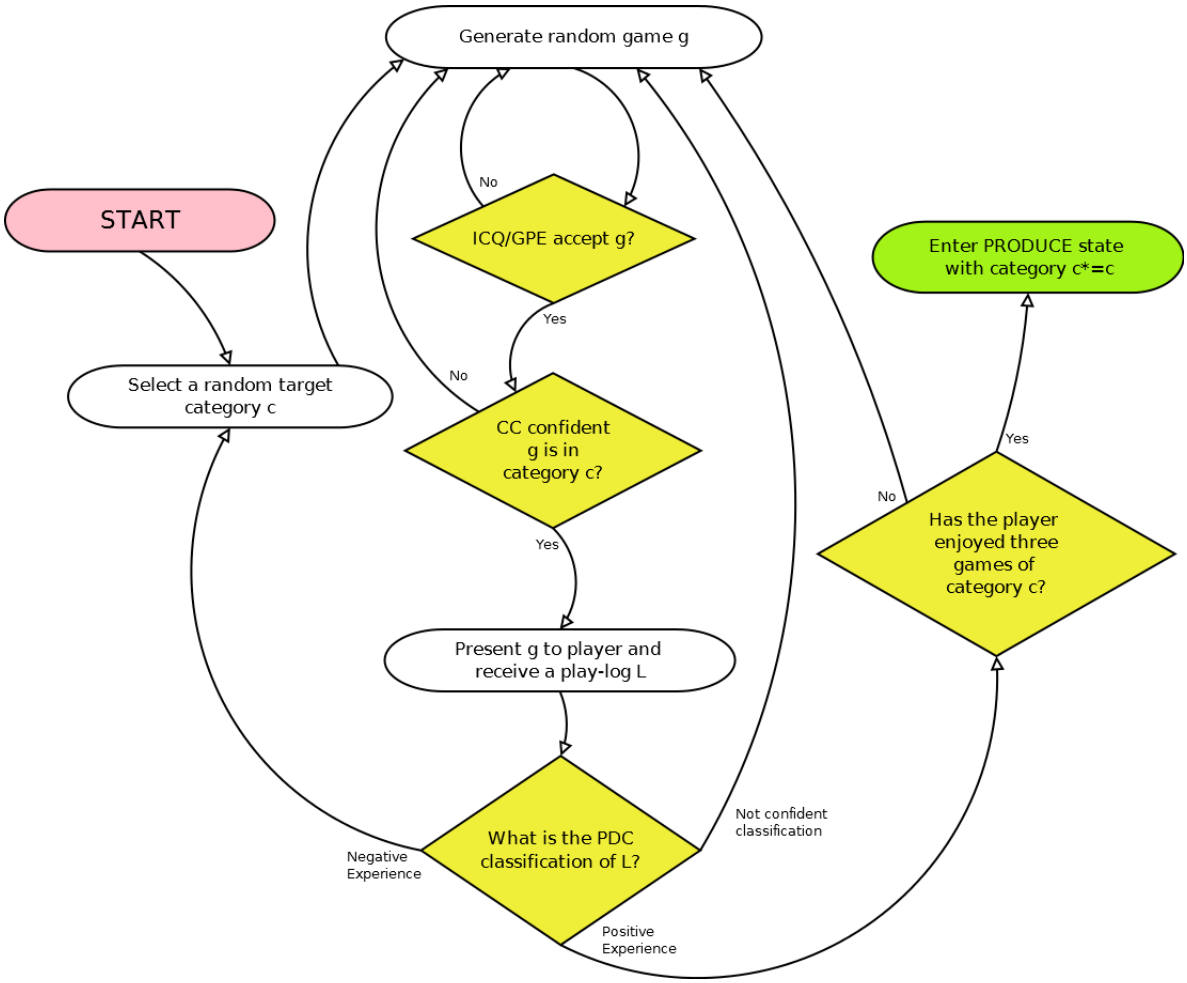

Figure 4.5: The GENERALIZE state in the IP model.

Chapter 5

LBPCG-Quake

5.1 Introduction

In Chapter 4 the core concepts behind the LBPCG framework were introduced along with enabling techniques for each sub-model in the framework. The framework was justified by analysing problems in the existing literature. In this chapter [1] concrete evidence for the framework is provided by applying it to the classic first-person shooter, Quake. Throughout the remainder of this thesis, the application of LBPCG to Quake will be referred to as *LBPCG-Quake*, and the specific enabling technique for each LBPCG model when applied to Quake will be preceded by *Quake-* i.e. Quake-ICQ, Quake-CC, Quake-GPE, Quake-PDC and Quake-IP.

In Section 5.2 Quake is introduced and in Section 5.3 the level generator, OBLIGE, that was used to create content for it. Section 5.4 will detail the problems that the LBPCG framework faces in particular when applied to Quake. Section 5.5 will list all of the concrete data types involved in the LBPCG-Quake framework, including play-logs, content vectors and the like. In Section 5.6 we will describe the data collection process and briefly touch on the software used. Section 5.7 will show how the enabling techniques in Section 4.4 can be applied to Quake. In Section 5.8 the experimental protocol is provided along with all corresponding results both on an individual model basis and global basis. Finally, in Section 5.9 we summarize the results with LBPCG-Quake and draw conclusions.

[1] Some parts of this chapter are adapted from J. Roberts and K. Chen, "Learning-Based Procedural Content Generation", arXiv:1308.6415 [cs.AI], 2013.

5.2 Quake

Quake is one of the most significant games of note in gaming history, and one of those most fondly remembered by the author of this thesis. Devised as a follow-up to the popular 2.5D game Doom by iD Software, Quake was completely 3D and had strong multi-player elements. The graphics were incredibly advanced for the 90s and the game pushed forward the boundaries of the expectations of players. It was one of the most popular packages distributed between people using the Shareware license at the time of its release.

In the single-player mode of the game the player takes control of a marine stranded in multiple demonic worlds, fighting off hordes of creatures with various ranged weapons or, if necessary, an axe. The general flow of the game is that the player starts the level, battles their way through various rooms and outdoor areas, then eventually reaches an exit pedestal which completes the level. Monsters come in many varieties, including humanoid undead marines, flying monstrosities that shoot plasma at the player and also large tank-like ogres which soak up a great deal of damage before dying. A diagram of monsters is shown in Figure 5.1. The player can pick-up health to restore damage and various power-ups that increase their damage ability, add invisibility or add protection against toxic elements in the environment. Levels contain interior, exterior and underwater areas. Maps in Quake also usually contain many hidden areas containing extra power-ups and more powerful weapons, requiring the player to be very observant and curious to get the full potential from their game-play. In summary, Quake has all the basic elements of a modern first-person shooter, albeit it in a more classic form and as such is a good representation of the first-person shooter category. In Gunn's taxonomy [28], one would identify Quake as being non-co-operative player-as-actor and real-time.

Quake was released under the GNU Public License (GPL) in 1999, which is the primary advantage of using it as an example game for the LBPCG. The open source nature of the game allowed it to be modified in various ways for the purposes of experimentation. It was desirable to have as large a group as possible to train the LBPCG-Quake framework and as such we decided to open up the project to both Linux and Windows users. This meant that we had to use two different source ports of Quake named qrack and QuakeSpasm, for Windows and Linux respectively. Both of these ports add stability and several additional improvements to the game such as enhanced particle effects and bloom lighting

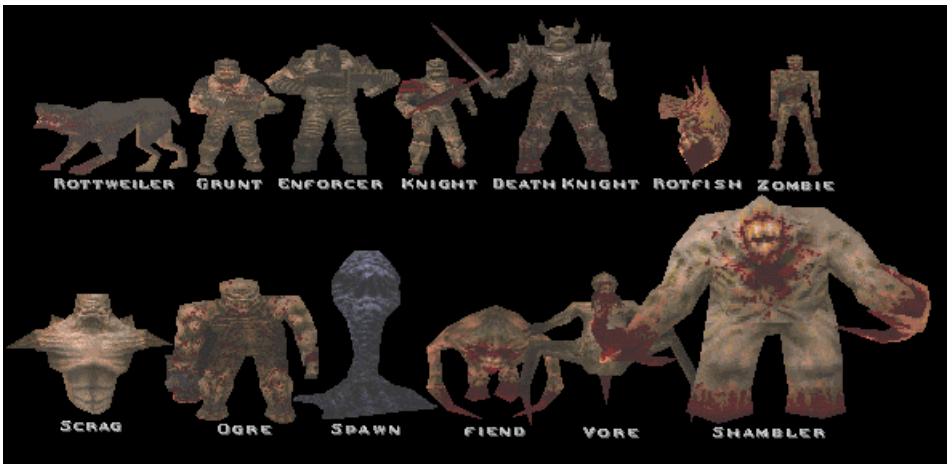

Figure 5.1: Quake monsters.

effects. This makes both games much more palatable to the public who are used to more graphically appealing games. The list of modifications we made to both QuakeSpasm and qrack is as follows:

1. After a game is played, both qrack and QuakeSpasm now produce an event-log, which is a raw list of time-stamped events that occurred during game play. Examples of events are the player firing their weapon, a monster being killed and the player moving the mouse. This modification was the most important as it allows the construction of a high-level play-logs from the low-level events, which are used as input to the Quake-PDC model.

2. The user interfaces for both qrack and QuakeSpasm have been modified to "lock" the user in. Once a level has been launched, the player cannot switch to another level and the only way of exiting is to quit the game via the menu or by completing it. This prevents the player from playing multiple games, which would cause the event-logs to be mixed together.

On its own, Quake is just a game engine and provides no way to generate new content. The goal of the LBPCG is to optimize content and as such a requirement is for the existence of a content generator. In the following section we will introduce one such content generator for Quake, named OBLIGE. Throughout this section we refer to Quake levels as *games*, and the content vector as a *game vector*.

Figure 5.2: The classic first-person shooter Quake.

5.3 OBLIGE

OBLIGE is a level generator written by Andrew Apted for various iD Software games including DOOM, DOOM 2, Hexen, Heretic and Quake. OBLIGE has been in development since 2005 and as such is quite mature. Figure 5.3 shows the GUI for the program, which allows the user to select the target game, set various generic and game-specific options and specify a random seed. Appendix A gives all the options that can be set for Quake. Although OBLIGE does have the ability to produce good quality playable levels, it is missing some functionality such as generating liquids, traps and levels featuring bosses. The levels it produces are also less aesthetically pleasing and interesting than those that can be produced by hand. Producing a level is quite a fast process and tends to take approximately 5-10 seconds for a small level. We note that OBLIGE does not produce levels that are broken, in the sense that the geometry of the level makes it "physically" unplayable e.g. the player starting the game in a cuboid with no exits.

The driver for the software, including the GUI, are written in C++ but most of the functionality is written in the scripting language LUA. We have modified OBLIGE in the following ways:

1. A few bugs in the LUA code were fixed which prevented several of the parameters from having any effect.

2. The number of monsters spawned has been toned down across the board to produce more sensible difficulty scaling.

3. The first room in a map has been coded to have less monsters, which still

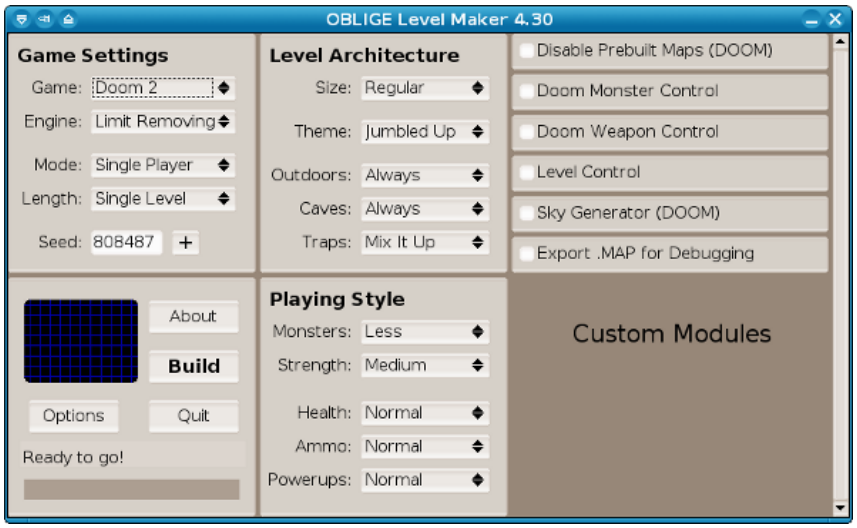

Figure 5.3: GUI for OBLIGE.

gives players a chance to find a weapon, as opposed to the original behaviour which gradually scaled up the number of monsters from the first room onwards. This means that the monster parameters have a more readily apparent effect on levels, which allows the player to assess them more quickly.

4. The parameters which specified how many monsters there were of each type were modified to be pairwise ratios, stating what proportion of the total monster count each monster type should occupy. This is opposed to the original code which meant each monster parameter affected the total number of monsters, rendering the overall monster count parameter less meaningful.

5. OBLIGE has been modified to generate levels which have one weapon available throughout, making weapon selection much more game-changing.

As is apparent from Appendix A, OBLIGE has many parameters. We have combined and excluded many of these to reduce the search-space, while maintaining meaningful game variation. Table 5.1 lists the parameters that we designed for the project. The majority of removed parameters affect aesthetic qualities of the level, making them somewhat less interesting. For parameters that aren't varied any more, we fixed their value to something sensible e.g. the median value. The important parameters, which affect monsters and weapons have been included in the game vector.

There are several artificial game vector parameters, such as the *weapon* and the *monsterset* parameters. These artificial parameters wrap several OBLIGE parameters into one to reduce the search-space. *monsterset1* specifies the frequency of Rottweilers, Grunts and Enforcers. *monsterset2* specifies the frequency of Knights, Scrags and Ogres. *monsterset3* specifies the frequency of Fiends, Zombies and Deathknights. *monsterset4* specifies the frequency of Shamblers and Vores. The frequency of a monster determines how probable it is to appear instead of the other monsters. If, for example, *monsterset1* is set to scarce and *monsterset2* is set to heaps then the player will see a lot more Rottweilers, Grunts and Enforcers than the player will see Knights, Scrags and Ogres. Monster frequency is also influenced by *strength*, which gives a preference based on how powerful the monster is. The *mon* parameter has the overall say in the total number of monsters in a level. It should be noted that in our initial analysis we found that the *skill* parameter did not have much of an effect on the level. In some circumstances it produced more enemies, but not by a substantial amount. As such, this parameter can be regarded as being not very influential, if at all. It was not expected this parameter would not be very important when making any decisions about content such as acceptability or category.

Table 5.1 lists the type of each attribute used. All variable attributes, excluding the weapon parameter, were ordinal, in that some feature of the level was increased as the value of the attribute increased (note that the random seed was not controllable and therefore not used for measuring distance). The weapon parameter was considered to be nominal, as there was no clear order to the values. For example, the lightning gun may cause quite a lot of damage to enemies, but it runs out of ammunition fast, making it less useful than the nail gun in many situations. In other words, there was no clear advantage to any of the weapons over the others, they all being rather situational. We used the mixed attribute measure mentioned in Section 2.4.1 as a dissimilarity metric between content vectors, primarily for the purposes of clustering.

There are a total of 116,640 possible games, excluding variation by the random seed. If one was to consider the random seed a controllable variable, which isn't the case, then there is an effectively infinite supply of games.

Each game vector also contains a random seed. This means that two game vectors which are identical except for the random seed will result in two games that look different. The game-play should still be similar for both games. The

seeds for the games presented to the public during training the Quake-GPE Model are fixed, which means each player will experience exactly the same content as others playing the same game vector.

Name	**Notes**	**Type**	**Values**
skill	Increases difficulty	Ordinal, Passed to game, not OBLIGE	0, 1, 2
mons	Increases monster occurrence	Ordinal	scarce, less, normal, more, heaps
health	Amount of health packs	Ordinal	none, scarce, normal, heaps
ammo	Amount of ammo in game	Ordinal	none, scarce, normal, heaps
weapon	Weapon availability	Nominal, Artificial	{ nailgun, supershotgun, grenadelauncher, supernailgun, rocketlauncher, lightninggun }
monsters1	Frequency of monsters in set 1	Ordinal, Artificial	none, less, more
monsters2	Frequency of monsters in set 2	Ordinal, Artificial	none, less, more
monsters3	Frequency of monsters in set 3	Ordinal, Artificial	none, less, more
monsters4	Frequency of monsters in set 4	Ordinal, Artificial	none, less, more
seed	Random seed adding variation to level	Discrete	Integer

Table 5.1: OBLIGE/Quake parameters.

In the previous two sections we have introduced the target game for the LBPCG and the content generator used to generate levels for it. In the following section we will describe precisely what the expected outcomes from applying the LBPCG to Quake are and the challenges faced.

5.4 Challenges

The objective, simply stated, is to generate Quake levels for a target player that are optimal in terms of enjoyment. The target player could be anyone, from someone who has never played a game before, all the way up to people who are very experienced gamers. This means the LBPCG-Quake framework must be versatile enough to recognize behaviour from, and adapt to, many different types of player who it is likely have widely different preferences and play-styles. The objective is to generate maps that have just the right amount and type of monsters, health packs, power-ups and the correct types of weapon for the target player. We will later measure the success of the LBPCG-Quake framework by comparing it against two other models (see Section 5.8.5).

OBLIGE, with the specialized parameter set, has a content vector space consisting of a total of 116,640 games. If one was to play each of these games for 3 minutes all day long, it would take approximately 243 days, which is certainly infeasible even for the most dedicated gamer. This means that the content space must be intelligently explored and any learned models must be able to generalize very well. On randomly sampling the content space it becomes clear very quickly that a large amount of the content is nightmarishly difficult. This is difficult in the sense that no player, no matter how skilled, would want to play such levels. An example unsuitable game would be one where the player enters the first room and is overwhelmed by hordes of monsters before the player is able to acquire any kind of weapon. Another challenge that is faced therefore is to filter such illegitimate content, as presenting even one such game could ruin the experience for a player to the point where they don't want to play any more.

These general problems are not the only ones faced. There are many subtleties with the parameters we have chosen to vary. For example, the overall number of monsters is controlled by one parameter, but there are four other parameters that control the proportion of different types of monsters that appear.

The final problem faced is that the event-logs generated from each experience with a game are very large, consisting entirely of timestamped events. There is a question of how to break these event-logs down into meaningful play-log representations.

In the following section we will move onto detail the precise data types that the LBPCG-Quake framework will need to handle including such objects as play-logs, feedback values and content vectors.

5.5 Data types

All of the OBLIGE parameters in Table 5.1, except the random seed, form the content vector for the LBPCG-Quake framework. Whenever a new content vector is generated, a random seed is also generated which is fixed and then associated with the game vector. In other words, the LBPCG-Quake framework has control over all the aforementioned parameters except the random seed, but the random seed is still implicitly used in the content generation. The content vector is passed to the content generator (OBLIGE) which creates a map that can be played in Quake. We use the terminology *game* to mean a map that can be played in Quake.

Categorization requires that one or more content features is selected. To keep the experiment as simple yet realistic as possible we chose a single feature, which was the most intuitive, namely *difficulty*. Difficulty has, throughout the history of games, been one of the most ubiquitous options that can be set before or during game-play. It is a standard method for grading players and altering the content that is produced. It therefore makes sense that this is a feature that most players would have a preference for and something that the LBPCG-Quake framework could adapt to a target player. We set the difficulty content feature to have five possible ordinal values, namely { Very Easy, Easy, Moderate, Hard, Very Hard }.

In addition, we define acceptability as a binary value in $\{Illegitimate, Legitimate\}$, ordinal fun is a single ordinal value in $\{VeryBad, Bad, Okay, Good, VeryGood\}$ and binary fun is binary value in $\{NotFun, Fun\}$.

As previously discussed, the event-log is a log of timestamped events that occurred when a game of Quake was played. Note that many of these events provide additional parameters. For example, *QS_Quake_DmgFiend* additionally states how much damage was done. Other events, such as *QS_GotWeapon_Nailgun* do not have additional parameters. The play-log is a real vector consisting of statistics gained from an event-log. Its structure is described in Appendix B. Throughout this table *tick* refers to a single pulse of the game engine.

In this section we have defined all the basic types that will be used as inputs and outputs to the LBPCG-Quake framework. In the following section we will detail the exact implementation of each member of the framework, translating the enabling techniques in Section 4.4 into concrete procedures.

5.6 Data Collection

In this section we will go into detail about the data collection process for each model in the Quake-LBPCG framework. This section will focus primarily on software and the techniques used in collection. Statistical breakdowns and analysis of the resulting data collected will be given in Section 5.8. The software used to store and receive data was written in C++. More details of the software are given in Appendix C.

Both the Quake-ICQ and Quake-CC were trained using a single expert, who was responsible for generating both the training sets and test sets for the models. This can be viewed as a special case of developer training, where there is a single developer.

As per definition, the GPE training set required a large number of members of the public to be trained. It was decided that the best way to acquire such a data set was to collect data from anonymous people via the internet. This of course had some drawbacks, in that the data collected was not guaranteed to have an equal spread across different types of player, which would be advantageous to the later adaptive stages of the Quake-IP model. In a realistic games development environment, the developers would be able to selectively choose the people to participate in the beta trials of the game and thus construct a more optimal training set.

To collect data from the internet, a client/server system was developed and placed on a website. The website was then advertised to members of the public on websites such as reddit, the OBLIGE forums and Facebook. We will give a breakdown of the data received in Section 5.8.3. Figure 5.4 shows the basic design of the data collection system. The server controls the client, which is responsible for launching Quake and OBLIGE on the player's computer. The server transmits games across a network connection, and receives feedback and play-logs from the client after a game has been played. The player runs the game via a launcher which is responsible for executing the client and also checking the internet for updates. This allows any bugs to be fixed remotely. The server builds up a database of surveys which are then used in the Quake-GPE training process. The questionnaire consisted of two questions: (a) "Did you enjoy the level?" (yes/no) and (b) "How do you rate it?" (Very Bad/Bad/Average/Good/Very Good). The justification for (a) is that there is not much room for misinterpretation as to what the question means. The player either had fun, or they didn't. This also allows

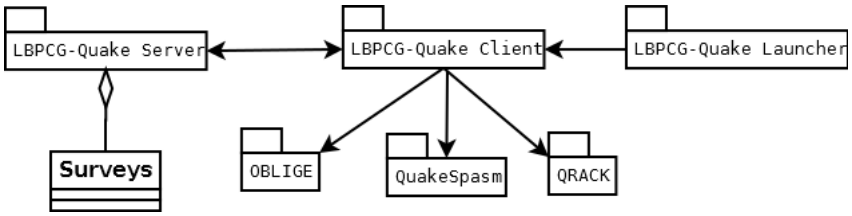

Figure 5.4: LBPCG-Quake Client Server system.

us to form a binary classification problem, which is likely to be easier to model than a multi-class problem in this initial study. The justification for (b) is just that we simply wanted more information for the purpose of analysis e.g. to see how people's interpretation of ordinal feedback is related to binary feedback. We do not use the information from the response to (b) as training for the LBPCG-Quake framework. We note that other authors have used systems such as *4-AFC*, which allow the player to choose their preference between multiple examples [70]. The main advantage of this is that it is usually quite easy to express preference between two games. It isn't immediately clear how such feedback would be used to train the models in the LBPCG-Quake framework as we have currently defined them, but integration with feedback using protocols such as 4-AFC would be an interesting topic to study in the future.

5.7 LBPCG-Quake models

5.7.1 Quake-ICQ

In the application of the LBPCG to Quake, the first model that needs to be trained is the ICQ Model. The role of the Quake-ICQ is to classify games as being either acceptable or unacceptable. An example unacceptable game would be one which is highly unlikely to be enjoyed, where each room is full to the brim with ogres, but the player is not provided with enough ammunition or health packs to be able to progress, leading to a frustrating experience. An acceptable game would be any for which there exists a reasonable number of players who would enjoy the content. In general, these are games where the player is provided with enough resources able to complete the level. We have previously stated that OBLIGE does not produce "broken" levels, in the sense that the geometry of the level makes the game unplayable. While this is not a problem for Quake/OBLIGE, it could very well be an issue that the ICQ Model

would need to deal with in other content spaces.

The content vector space for the target, Quake/OBLIGE, is too large to be exhaustively explored and as such the ICQ model's approach of using active learning to present those games that will add the most value to the training set and therefore potentially gain a good understanding of the search-space more quickly is a good idea. Active learning does, however, require the trainer to play games and it would be a bad idea to present the user with an unreasonable amount of examples, say over 500.

The first step in training the ICQ Model is to generate a test set T_{ICQ}, which can be used to validate the success of the model after, or during, its training process. We want T_{ICQ} to be as varied as possible, so we first apply k-medoids to $\mathcal{G}$, which was reviewed in Section 2.4.2. K-medoids is not a particularly sophisticated algorithm, but provides all of the functionality required. The content vector space is also sufficiently small, meaning the algorithm is unlikely to run for an extended period of time. We chose k-medoids as the centroids it uses are actually points in the content vector space, rather than averaged values that would be used in an algorithm such as k-means. This means the centroids are playable games. The k-medoids algorithm is terminated after no games have moved to different clusters within a single iteration. After convergence, the developer played every single centroid and provided acceptability feedback on each, forming the set T_{ICQ}. We chose k, and thus $|T_{ICQ}|$ to be 200, because we believe this is a reasonable number of games to expect the developer to play.

The ICQ active learning process requires that the implementers of the LBPCG choose a query strategy that will choose which games to give to the oracle at each iteration. Although many approaches were available, we decided to make use of the SVM implementation provided in the popular library libsvm [12], which has probabilistic output. With such probabilistic output, it is possible to use query using uncertainty sampling by converting the probability to a confidence score as detailed in Section 2.2. As our problem was likely to be non-linear, we chose an RBF kernel for all SVMs.

The active learning process was continued until the positive and negative error rates on T_{ICQ} converged, indicating that the ICQ Model had similar levels of ability at recognizing both acceptable and unacceptable content. This is important, as too much bias in one direction (e.g. biased towards classifying games as acceptable) may lead to good overall performance on the test set, but may lead

to poor performance on the actual data encountered when being used on new data, which may be balanced in the other direction (e.g. lots of negative games).

5.7.2 Quake-CC

The next model to be trained in the LBPCG-Quake framework is the Quake-CC Model, responsible for categorizing acceptable content, which is an active learning process. This means that the training and test sets for the Quake-CC Model should be composed of games that the Quake-ICQ deems as acceptable only. The Quake-ICQ model is therefore applied to $\mathcal{G}$ to filter out any unacceptable content and produce $\mathcal{G}_a$.

Training the Quake-CC Model poses an immediate problem. The developer of the game might have some insight, but doesn't really know much about what makes a game difficult. Also there are five levels of difficulty, and specific assignment to each of these categories based on a content vector is even less clear. Fortunately, the method proposed in Section 4.4.2 tackles this problem by building up an evaluative categorization model that is built using learning. By presenting strategically selected games, the developer's knowledge is tapped indirectly.

We decided to require a minimum of 20 games from each category as a basis for T_{CC}. As such, we applied k-medoids with $k = 100$ to $\mathcal{G}_a$, with the ideal situation being that the centroids consists of 20 games from each category. In reality, this is of course extremely unlikely, so after labelling each of the 100 centroids, games were uniformly sampled from $\mathcal{G}_a$ and labelled until at least 20 games from each category were found.

Finally, the Quake-CC learning algorithm was implemented as specified in Algorithm 4.2. As with the ICQ learning process, our query strategy was uncertainty sampling and the underlying binary classifier used were SVMs, with added probabilistic output provided by libsvm [12]. SVMs were used as the code was already in place from the ICQ training process. We did not experiment with other classifiers so cannot report if they would give better performance. For the resulting model, after learning had finished, we used an ensemble of random forests. We chose random forests for the ensemble as these allowed us to easily print out feature rank information for each class. Again, we make no claim as to whether random forests would perform better than another type of classifier and this might form interesting further research.

5.7.3 Quake-GPE

The goal of the Quake-GPE is to capture public opinion about the games OBLIGE produces for Quake with respect to fun. In addition, the GPE Model should assess the reliability of each individual used in the public survey so that play-logs can be ranked by reliability for use in the training process of the Quake-PDC model.

The first step in the training process for the Quake-GPE was to select candidate games to give to the public. It was important that there were equal numbers of games from each difficulty class, so that a good spectrum of feedback, and play-logs, were received for the training process of this and subsequent models. It was decided that 100 total games would be a sufficient number to give to the public, as results from training the Quake-ICQ and Quake-CC models showed that this was sufficient for training purposes. In retrospect it is apparent that the Quake-CC test set, T_{CC}, is the best candidate game set for this purpose. However, a separate active learning search process, was used to discover 100 games for the public to play. Details of the data collection process used to conduct the public surveys will be discussed in Section 5.6.

The Quake-GPE was implemented exactly as in Algorithm 4.3 making use of Raykar's crowd sourcing algorithm. Raykar's algorithm requires an underlying regression model, which is used to output a probabilistic classification. By default, the underlying classifier is a logistic regression model. Several ideas were explored to improve the flexibility of the model, including using an ensemble, but the simplest solution was to replace the logistic regressor with an SVM regressor with RBF kernel. As per Raykar's paper [65], the algorithm was terminated when the log-likelihood function reached a local maximum.

5.7.4 Quake-PDC

The Quake-PDC is designed to detect whether a player had an enjoyable experience playing a game generated by OBLIGE for Quake. It requires only the difficulty of the game and a play-log of the interaction of the player with the game. Algorithm 4.4 shows the learning process and operation of the resulting model, which is a weighted ensemble. For the underlying classifier, we chose to use random forests as these provide a fast method to extract cross-validation error i.e. the out-of-bag error. In addition, they provide statistics regarding the weight of each feature, which is useful information for debugging the model. We

applied four different thresholds, $0.0, 0.3, 0.6$ and 0.9, on $\alpha^{(p)}$ and $\beta^{(p)}$, which resulted in a combination of $4^4 = 16$ different training sets. 16 random forests were then trained on the sets to produce the basis of the Quake-PDC ensemble. We calculated the confidence, as specified in Section 2.2, by defining the Quake-PDC's posterior probability to be the weighted sum of the posterior probabilities reported by the random forests in the ensemble, using the weights w_k in Algorithm 4.4. Please see Section 2.3.6 for information on how to calculate posterior probabilities for random forests.

5.7.5 Quake-IP

The state machine of the Quake-IP model was implemented as specified in Section 4.4.5. The rank we chose to use when selecting games in the CATEGORIZE state for a particular category of content was based on controversy i.e. games were ranked more highly if there was more disagreement on them between the members of the public. This means that players will be presented with games that divide opinion first, as opposed to games which are regarded by most people as being good or bad. This reduces the uncertainty of what category the player falls into.

5.8 Results

In this section we will describe both the experiments conducted and results gained in the process of verifying the LBPCG-Quake framework. A number of the models in the LBPCG-Quake framework were verified using validation sets. Other models, for example the IP Model, require an evaluation process involving target players. Each sub-section in this section will describe the results for each sub-model in the LBPCG-Quake framework.

Throughout this section and beyond we define *positive error (rate)* as the rate at which positive examples were misclassified as negative and *negative error (rate)* as the rate at which negative examples were misclassified as positive. We define *average error* as the average of the positive and negative error. Positive error is represented using the notation *+error* and negative error using the notation *-error* in all graphs throughout this thesis.

5.8.1 Quake-ICQ results

The Quake-ICQ learning algorithm was trained for a total of 108 iterations, as it was around this point that the f-measure (Section 2.3.1) began to fall. The positive error, negative error and average error at each iteration is given in Figure 5.5, as reported by training an SVM at each step. Figure 5.6 gives the f-measure. It is immediately observable that all error rates gradually decrease over time, indicating that the active learning process worked. The positive and negative errors converge at around 80 iterations, indicating that the Quake-ICQ Model is equally able to classify both positive and negative points. The resulting Quake-ICQ Model, consisting of an SVM with RBF kernel, had an average error of 0.1, positive error rate of 0.05, negative error rate of 0.15 and f-measure of 0.94. These measures give us a degree of confidence in the generalization capabilities of the model, but don't really say how effective the model is. We did not compare the active learning process to any other approaches such as random sampling, so cannot make any claims as to whether active learning is the most effective algorithm to use.

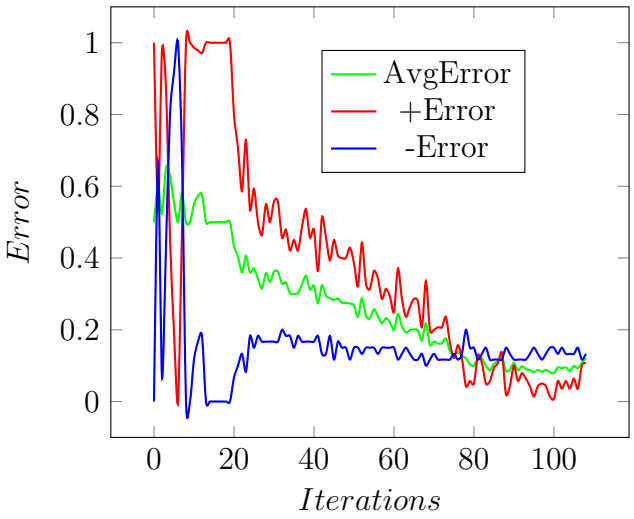

Figure 5.5: Active Learning Error Plot

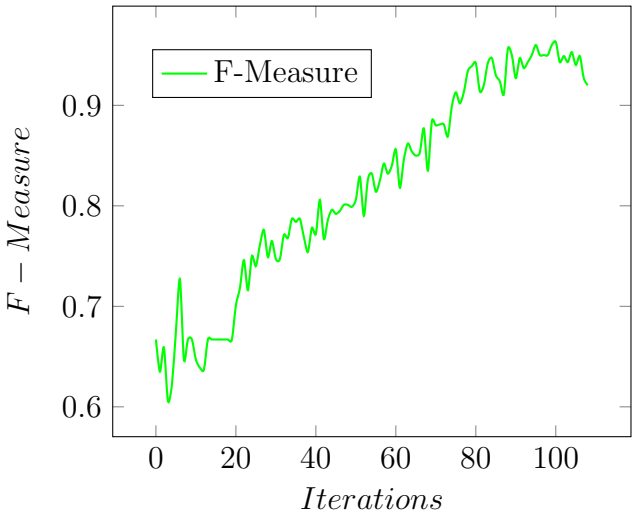

Figure 5.6: Active Learning F-Measure Plot

Based on the literature review of PCG approaches in Section 3.3, there is no equivalent research that is quite similar enough to compare these error rates against. The Quake-ICQ model is somewhat unique in that it is a data-driven model trained by experts that is used to identify acceptable content in a specific game. Most existing approaches attempt to satisfy some metric such as fun or tune a feature such as frustration. Even our definition of "acceptable content" is less lenient than other approaches that attempt to exclude only "broken content", that being content that is unplayable due to badly placed geometry and other issues. Our definition goes further in that any content that is unlikely to be enjoyed by anyone is also excluded, even if it is technically playable. This specificness makes it difficult to find other comparable results.

The most meaningful analysis we can provide is to look at the what types of content the ICQ Model classified incorrectly, and state what kind of consequences this could have. It was generally found by the developer that the types of game that were unacceptable were those which had: (a) overwhelming amounts of monsters, with not enough resources or (b) games that had no monsters at all i.e. with the monster1, monster2, monster3 and monster4 parameters set to none. Almost all other types of game were playable to some extent and could be perceived as being enjoyable to some particular user. On filtering $\mathcal{G}$ with the Quake-ICQ Model, it was apparent that the model filtered out every game where there was no ammunition present. This makes sense, since such games are very rarely enjoyable in any way. However, the Quake-ICQ Model failed to filter out

all games with no monsters in them, which is probably due to the active learning process not exploring this exact combination of parameters during its iterations thoroughly enough. This means that if we were relying on the opinion of the Quake-ICQ Model only to decide what games to play, the user could potentially be presented with unacceptable games consisting of no monsters, even if this is a rare occurrence.

We also provide two examples of games the Quake-ICQ Model incorrectly classified to add more context. First, we give an example game that the Quake-ICQ thought was acceptable, when in fact the developer thought it was not. The parameters for the game are described in Table 5.2. On beginning this game, it is understandable as to why the Quake-ICQ might think it is acceptable. There aren't that many monsters, and you are provided with a decent weapon. However, it soon becomes clear that there are quite a lot of powerful monsters, and the player quickly runs out of all ammunition. With no health at all in the level, the player is almost guaranteed to die unless they sprint through the entire level ignoring all monsters. It was for these reasons the game was labelled as unacceptable i.e. due to a lack of resources. In this case, the Quake-ICQ's hasn't learnt the subtleties just mentioned. The second game is one that the Quake-ICQ thought was unacceptable, while the developer found it to be acceptable. The parameters for the game are given in Table 5.3. In this game, there are a large number of monsters, and it is extremely difficult to move more than 50 metres before getting shredded by fiends or zapped by shamblers. The weapon is a grenade launcher, which is not suitable for close quarter combat. However, the game could still be enjoyable by a very dedicated player. As such, the decision as to whether the game is acceptable or not was very close.

Parameter	**Value**
skill	1
mons	scarce
health	none
ammo	scarce
weapon	supernailgun
monsters1	less
monsters2	less
monsters3	less
monsters4	less

Table 5.2: Quake-ICQ False Positive.

Parameter	Value
skill	1
mons	heaps
health	none
ammo	scarce
weapon	grenade
monsters1	none
monsters2	none
monsters3	less
monsters4	less

Table 5.3: Quake-ICQ False Negative.

The results from the Quake-ICQ Model suggested that the model was mostly capable of recognizing acceptable content. The active learning process converged, in the sense that the positive and negative errors were the similar, within 80 iterations and it is certainly reasonable to assume that a developer in a commercial setting would be willing to play this amount of games. Perhaps the main criticism of the approach is of the active learning process. In general, it remains to be seen if the active learning process would converge as quickly, or have as high a success rate, if the problem was substantially more complex. In addition, our experiments were performed with one type of active learning only, namely uncertainty sampling using probabilistic SVMs. It would be interesting to see if other configurations of SVM, using different kernels and degrees of complexity, and SVM-specific active learning strategies, such as MaxMin [83], perform better or worse. It would also be interesting in future research to find if other query strategies such as query by committee, and models such as random forests and neural networks provide any advantage. More importantly, a general approach to assessing, perhaps using prior knowledge, what the best active learning configuration is likely to be would be very useful.

5.8.2 Quake-CC results

To help verify the Quake-CC model and the game selection process for the Quake-GPE, a *Manual Content Categorization* (MCC) model was constructed for Quake, which we term *Quake-MCC*. This is a rule-based version of the Quake-CC model which relies entirely on developer knowledge. The rules for the model are given in Table 5.4. Any game that did not match a rule in the MCC is marked as unknown

by the MCC. In general, the MCC acts as an effective contrast for the Quake-CC model, which is learning-based. It also acts as an alternative method for selecting the games in $\mathcal{G}_{GPE}$ and a comparative study will be given in Section 5.8.3.

Class	**Condition**
"Very Easy"	Lowest skill, very small number of monsters, no ogres/-fiends/shamblers, only one type of enemy, no grenade launchers, no rocket launchers and enough ammo and health
"Easy"	Like "Very Easy", but more monsters, two types of monster and rocket launchers allowed
"Moderate"	Like "Easy" but harder skill allowed, more monsters, some fiends allowed and no super shotguns
"Hard"	Like "Moderate" but more monsters, enforced normal skill level, shamblers allowed but not lots, no mixing of shamblers with ogres/fiends, no lightning gun, only just enough ammo and health
"Very Hard"	Like "Very Hard" but hard skill, lots of monsters, constrained health/ammo and all monsters allowed

Table 5.4: Rules for the Quake-MCC Model.

Figure 5.7 shows the construction process of T_{CC} using the clustering/search process described in Section 5.7.2. After labelling 165 games the algorithm has found at least 20 examples of each class. This means that 65 games had to be played beyond the centroids found by clustering, which while acceptable in terms of developer time usage, does indicate that the uniform sampling process could go on for quite a while. This could potentially be improved by using a more intelligent search, such as active learning. The final iteration leaves T_{CC} with 23 "Very Easy" games, 47 "Easy" games, 43 "Moderate" games, 33 "Hard" games and 20 "Very Hard" games. The graph shows that there were much less "Very Easy" and "Very Hard" games spread throughout the centroid set, and that the uniform search had to spend most of its time looking for examples from these two classes. This might be because games of this difficulty are less prevalent, or simply that they are not as spread out through $\mathcal{G}_a$ as the other classes.

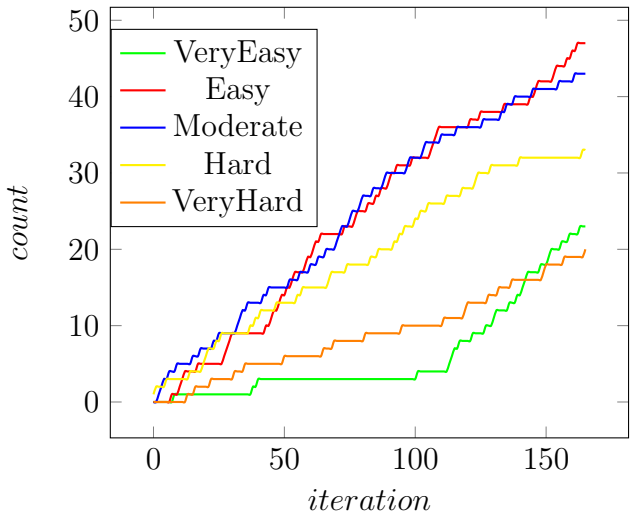

Figure 5.7: CC Test discovery

Figure 5.9 shows the progression of the training process. Each curve shows the error rate of the Quake-CC on a particular difficulty class. We define error rate for a particular class as the rate at which members of that class were classified incorrectly. The graph shows that the process seems to stagnate after approximately the 45th iteration. After this point the error rates for each class seem to vary around a average values. This indicates that this iteration may be a good place to end the training. Figure 5.8 shows the average error of the Quake-CC learning process, that is, the average of all the individual class error rates. This graph indicates that the average error continues to decrease until approximately the 60th iteration where it seems to stagnate. The best iteration to stop at, based on the average error, was found to the the 65th with an error of 0.0205. The error rates for the "Very Easy", "Easy", "Moderate", "Hard" and "Very Hard" classes were 0.043, 0.149, 0.349, 0.33 and 0.15, respectively.

It isn't really possible to draw a conclusion as to the success of the Quake-CC model using these error rates alone. For the Quake-CC model, we are in a similar situation as to that of the Quake-ICQ model, in that there is no equivalent research to compare our model against. The Quake-CC is a data-driven approach attempting to classify content into specific categories. In the case of the LBPCG-Quake framework, the categories happen to be difficulty levels. However, it is still difficult to contrast these results with existing research into approaches such as *dynamic difficulty adjustment* [33]. One approach to determine how good our

results are is to compare them against the Quake-MCC, by applying the Quake-MCC to T_{CC}. Unfortunately it was found that the Quake-MCC was incapable of classifying almost all of the examples provided, due to how strict the rules added were. This gives a small amount of evidence towards the idea that encoding developer intuitions as sets of rules may be very difficult to do successfully. The developer involved in this experiment was a very experienced gamer, but their concept of difficulty, as encoded in the Quake-MCC may have been far too specific. Whether this argument has any weight beyond the developer and the game Quake is uncertain without further research.

Another meaningful manner in which to analyse the results, in the absence of contemporary work, is to determine what the effect of an incorrect classification is. To this end, Figure 5.10 gives the confusion matrix for the Quake-CC Model's run test on T_{CC}. The matrix shows that most of the incorrect classifications land in adjacent classes, meaning that when the Quake-CC Model incorrectly labels a game, the label is "almost" correct. For example, we observe from the confusion matrix that even though the "Moderate" difficulty class is the one the Quake-CC struggles with the most, most of the incorrect classifications land in the "Easy" class. In terms of the LBPCG-Quake framework, this would mean that if the Quake-IP entered in a PRODUCE state for a player identified with a particular category C, then the player might get given games that are either somewhat harder, or somewhat easier. The net effect of this is uncertain.

Figure 5.11 shows the counts for each class as the active learning process proceeds. This shows us how differently the training process for the Quake-CC model proceeds in its exploration of the space in contrast to the test set generation process. For example, the training process discovered much fewer "Moderate" and "Hard" games than the test set generation process.

$$\begin{pmatrix} 22 & 0 & 1 & 0 & 0 \\ 4 & 40 & 3 & 0 & 0 \\ 3 & 12 & 28 & 0 & 0 \\ 1 & 1 & 6 & 22 & 3 \\ 0 & 0 & 0 & 3 & 17 \end{pmatrix}$$

Figure 5.10: Quake-CC confusion matrix.

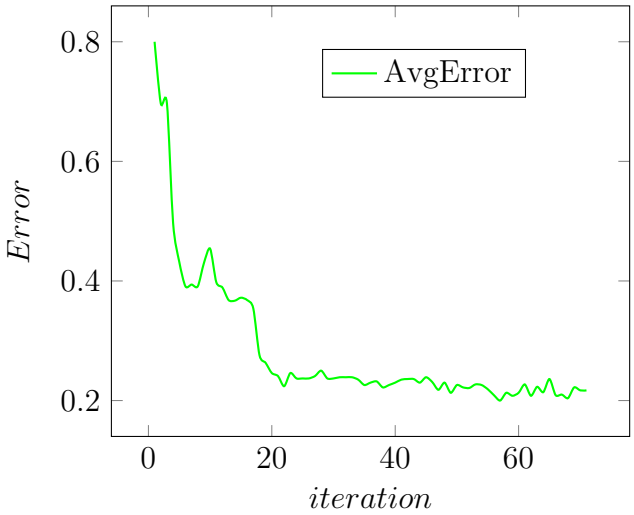

Figure 5.8: CC average error on Validation Set

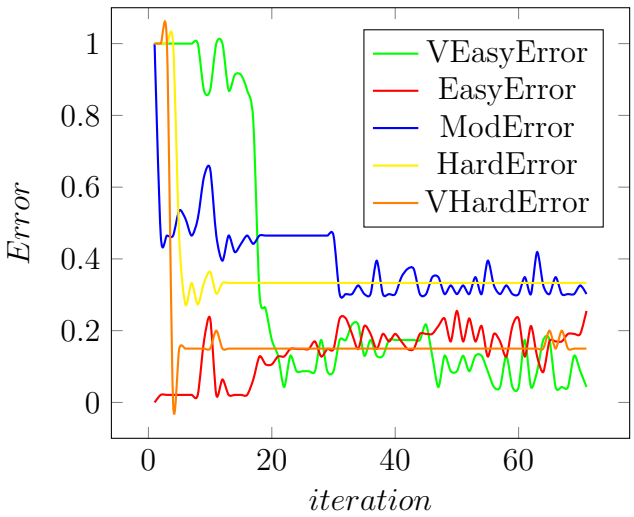

Figure 5.9: CC per-class error on Validation Set

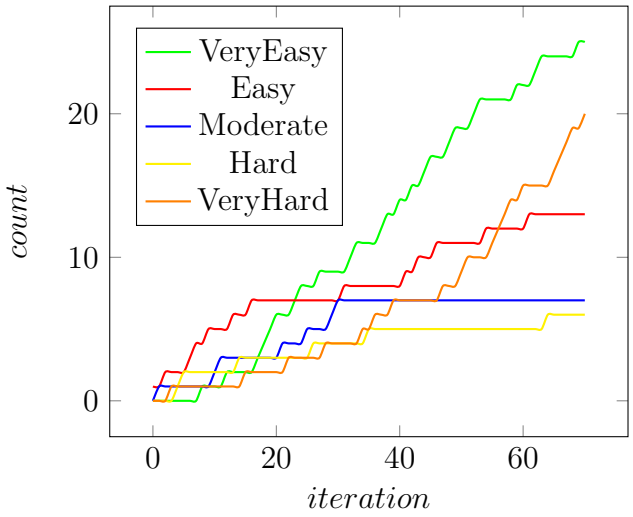

Figure 5.11: CC Learning discovery

One approach to improve the accuracy of the Quake-CC may be to reject games that the model is not confident of classifying correctly. For the purposes of testing, this means that any game that the model gives a confidence of being below a certain confidence threshold does not contribute to the reported error rate. Figure 5.12 and Figure 5.13 show the effect of applying a confidence threshold to Quake-CC model, showing the per-class error rate and average error rates, respectively. Figure 5.14 shows the rate of rejection, that being, the fraction of games from each class that the model rejects as the threshold is varied. From all three figures, it is apparent that there is no advantage to using a confidence threshold. Increasing the rejection threshold reduces the error on some classes, but raises it on others, and in general raises the rejection rate too high. Since the Quake-CC model is used as a filter in the Quake-IP model, this means that fewer games will be allowed to pass through it. The net effect of this is less diversity in the games presented to the target player. As such, applying a confidence threshold would make the Quake-CC somewhat less useful, and it was decided that the Quake-CC should not do so.

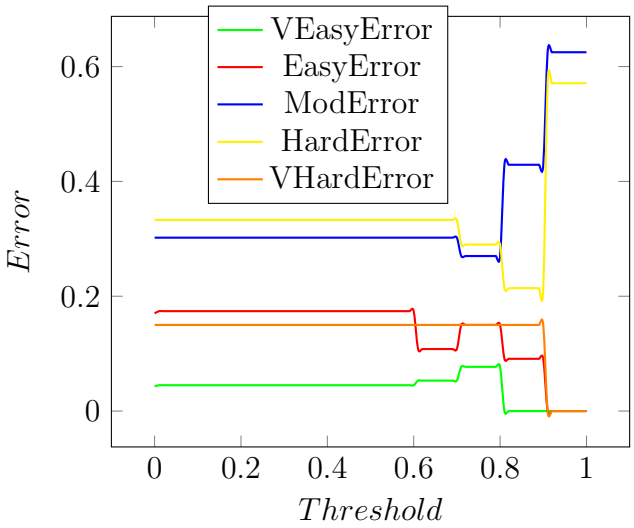

Figure 5.12: CC error rates while sweeping reject threshold

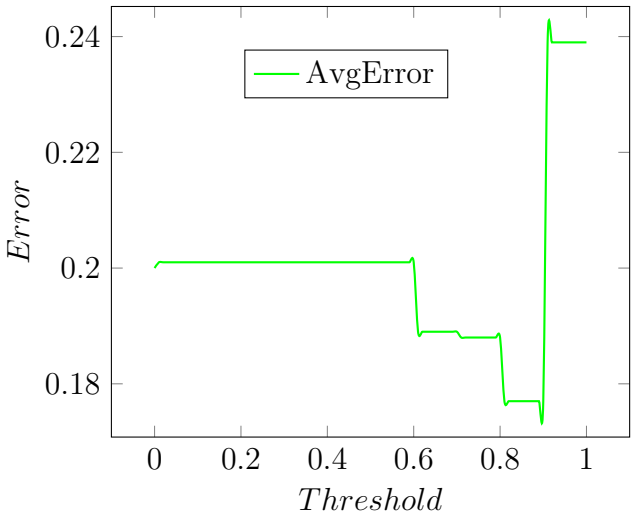

Figure 5.13: CC average error

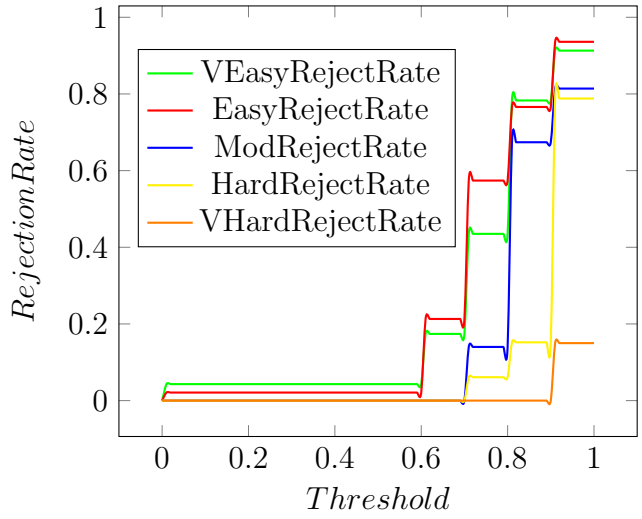

Figure 5.14: CC rejection rates

In summary, the results show an overall error rate of Quake-CC of 0.205. The "Moderate" and "Hard" classes have higher error rates than the other classes, with the Quake-CC getting approximately one-third of test set examples incorrect for each of these classes. The confusion matrix showed that incorrect classifications tended to fall in adjacent classes. Therefore, the net effect would be that if a game of a specific category was requested by the Quake-IP model by querying the Quake-CC model, then the resulting game is more than likely of the correct category, or will be of a similar category. The training process for the Quake-CC model concluded in an acceptable number of iterations, 65. The advice of the LBPCG framework in Section 4.4.2 is that the data from the Quake-ICQ model training process is re-used for training and testing purposes with the Quake-CC Model. However, it was found that this data was particularly skewed towards different categories, as the data had been derived using active learning with different objectives. It was found that a far better approach was to start from afresh with T_{CC} and the training set for the Quake-CC Model. Unfortunately, enhancing the model with a rejection threshold proved unsuccessful and it might be useful to see if different approaches could be used to improve the model further.

Our main criticism of the approach we have taken is, as with the Quake-ICQ, the active learning process used. We restricted ourselves to one type of active learning algorithm, namely uncertainty sampling using SVMs. We also restricted ourselves to one type of final model, namely an ensemble of random forests, due to the conveniences it offered. Both of these approaches haven't been

contrasted with other active learning algorithms or classification models and as such, in future work it would be interesting to know if there are, in general, recommendations for the concrete techniques used in training the Quake-CC.

5.8.3 Quake-GPE results

The first step in training the GPE model is to select the games that are to be provided to the public. The process by which this was done is specified in Section 5.7.3. A desirable feature of the GPE games is to ensure that they are as spread out as possible geometrically in the content vector space, therefore increasing the likelihood of the information gain. To measure the spread of the games we applied t-SNE as follows. Firstly, k-medoids was run on $\mathcal{G}_a$ to form 500 clusters and then all the centroids were extracted to form a set B. B is a condensed representation of the general spread of all acceptable games in $\mathcal{G}_a$, which means that plotting it along with the games from $\mathcal{G}_{GPE}$ would give a good indication of how spread out the public games are. The GPE games, along with C were fed into the t-SNE algorithm to produce the plot in Figure 5.15. To contrast the result, the MCC Model (see Table 5.4) was applied to $\mathcal{G}$, and the results fed into t-SNE along with B to form the plot in Figure 5.16, showing how spread out content would be if the the MCC model was used to select games. Comparing the two t-SNE plots shows that the active learning approach does provide more diverse selection, as the MCC model tends to select its games from very distinct clusters, compared to the dispersion of the active learning selected games.

In total 895 surveys were recorded from a total of 140 people. Each game was played roughly nine times each and players played on average approximately three to five games. However, one particularly enthusiastic participant produced 154 surveys on their own. The user's surveys were removed from the data set to prevent their feedback having too large an influence on the framework.

Figure 5.17 shows the distribution of the binary fun feedback question amongst the surveys received from the public. 40% of the surveys represented a positive experience. Figure 5.18 shows the equivalent distribution of the ordinal fun feedback question. It is interesting to note that 33% of games were labelled "Very Bad" or "Bad", a discrepancy of 7% between the two feedback questions, indicating the likely scenario that a number of users used the "Average" feedback value to indicate a negative experience. This shows that caution should always be taken when asking for feedback from members of the public, who have their

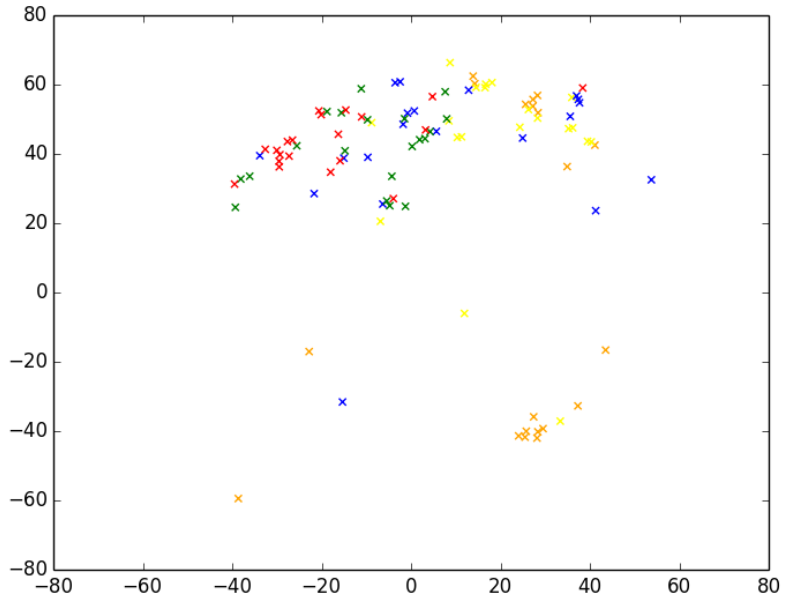

Figure 5.15: t-SNE plot for games selected using active learning. Key: Very Easy (Red), Easy (Green), Moderate (Blue), Hard (Yellow), Very Hard (Orange).

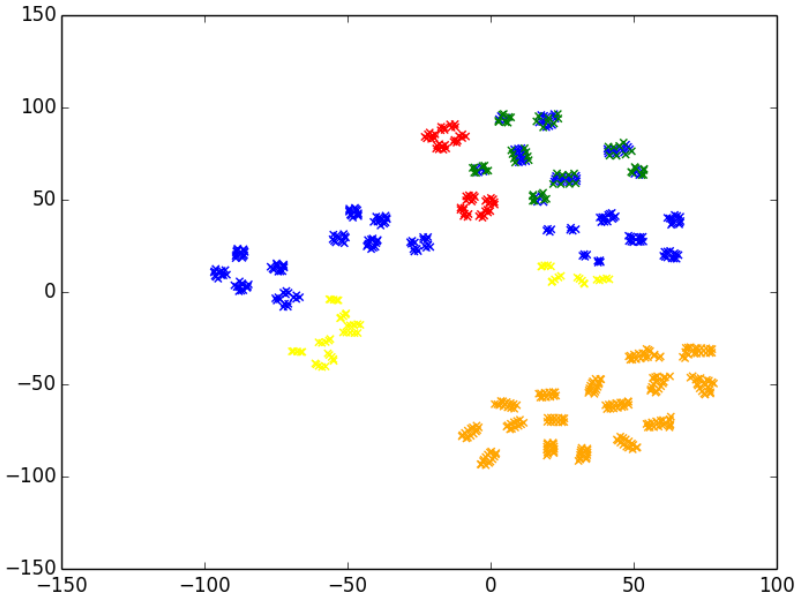

Figure 5.16: t-SNE plot for all MCC classified games. Key: Very Easy (Red), Easy (Green), Moderate (Blue), Hard (Yellow), Very Hard (Orange).

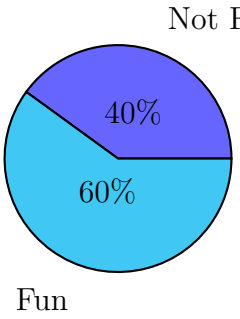

Figure 5.17: Binary Fun.

own interpretation of subjective questions. In general, the split of positive and negative experiences was a good result, since it meant that there would be an acceptable split of training data for the Quake-GPE and Quake-PDC models.

Figure 5.19 and Figure 5.20 show the distribution of feedback amongst games of each difficulty class. Interestingly, the graphs show that as game difficulty increases, the number of "Very Good" labels also increases, but the "Very Hard" class also has the most labels of "Very Bad". The middle difficulties "Easy" to "Hard" have the least number of people labelling them as "Very Bad". Additionally, as difficulty increases, fewer surveys are labelled as "Average", indicating more polarized view points.

It is actually quite desirable to present controversial games to the public, as games that divide opinion are likely to segregate players into different categories. Figure 5.21 provides further analysis of the controversy. For each game, we determined the average binary fun feedback value and assigned the game to a bucket based on the value. A game with average fun feedback value of 0.5 was clearly very controversial, and labelled as such. A game with a low or high average fun feedback back e.g. 0.1 or 0.9, was clearly not as controversial. The graph indicates that the "Very Hard" and "Very Easy" games cause the most disagreement amongst participants, whereas the middle difficulties caused less disagreement. This indicates that the GPE games generated using the active learning process were well selected since they did indeed cause a fair amount of controversy.

In summary, we analyzed the public feedback on the set of games, G_{GPE}, used to train the Quake-GPE model. The method of using active learning to discover games of different categories for public consumption was successful, based on the feedback received, which indicated a lot of controversy and also showed patterns in feedback for games across categories. In retrospect, one change we would have

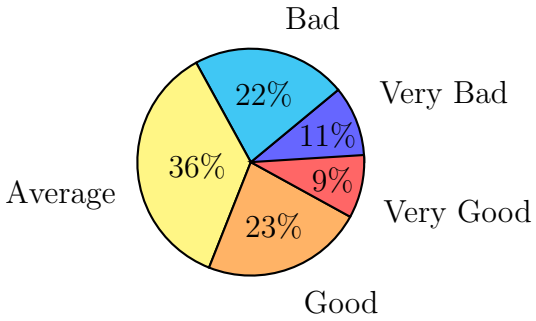

Figure 5.18: Ordinal Fun.

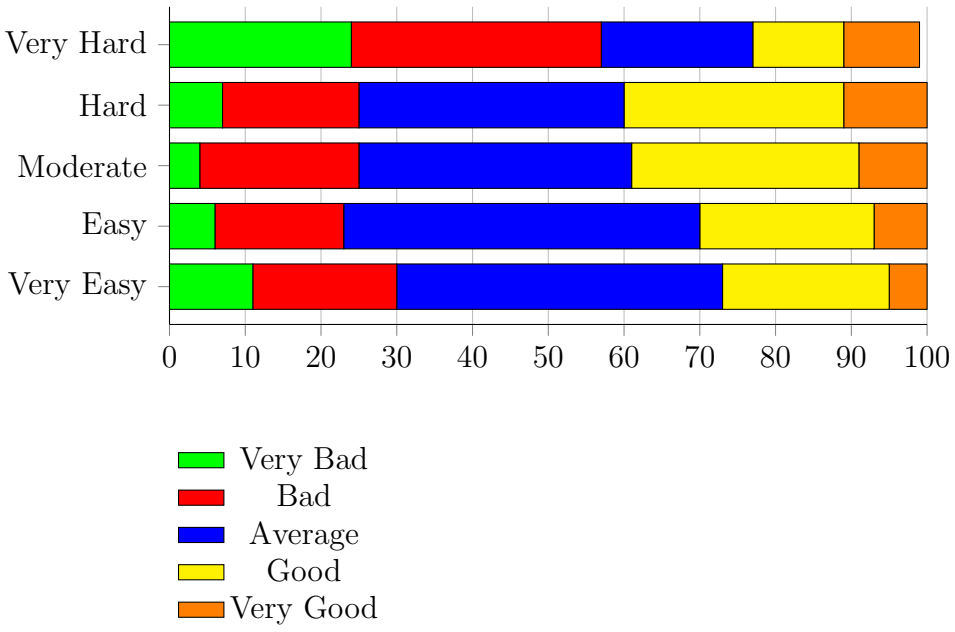

Figure 5.19: Ordinal Fun Distribution.

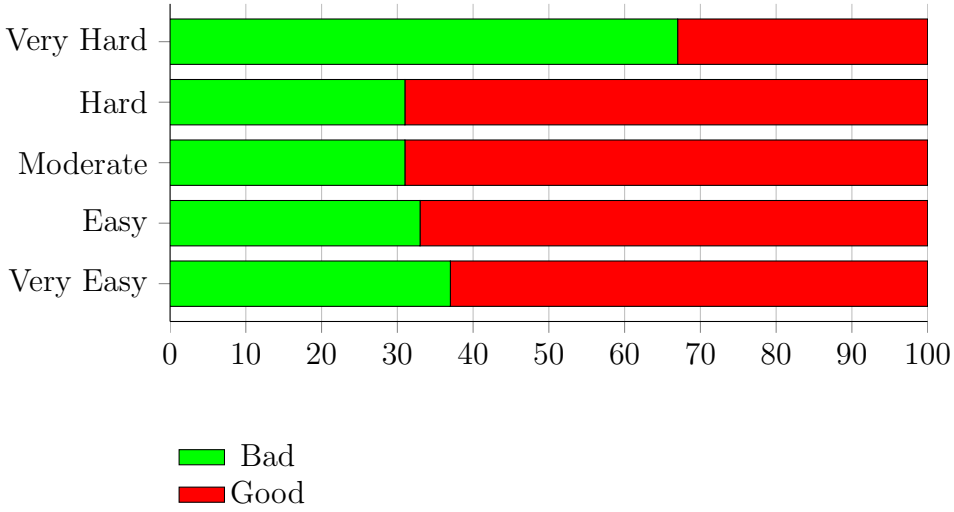

Figure 5.20: Binary Fun Distribution.

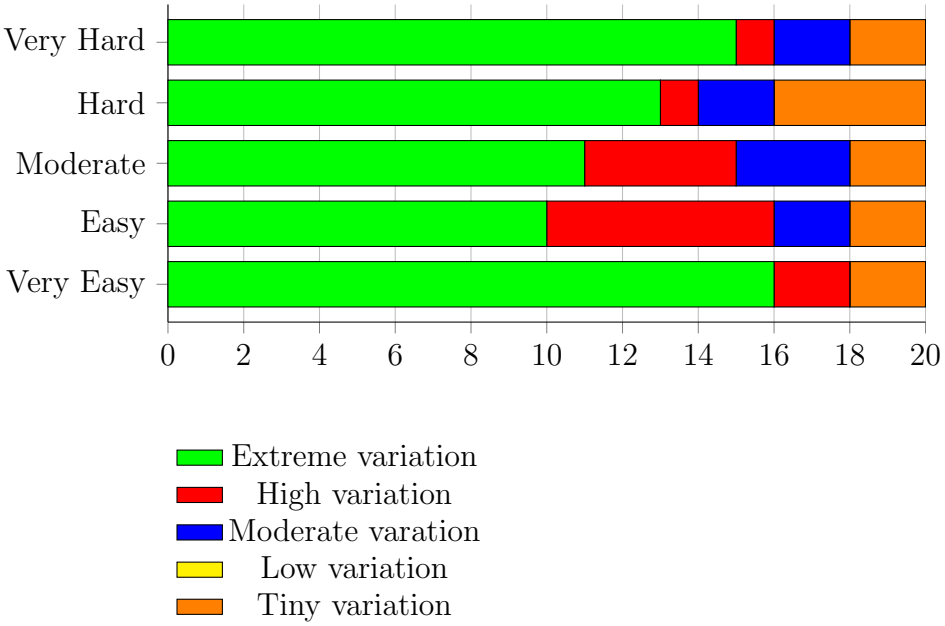

Figure 5.21: Binary Fun Deviation Distribution.

made to the selection of G_{GPE} would have been to simply re-use the Quake-CC testing set, T_{CC}, removing excess games from the categories to produce 100 games. The Quake-GPE model itself is rather difficult to verify, since it is not actually known what the true α and β (sensitivity and specificity) values for each player are. It was found that people who didn't play many games tended to have low α and β scores. We would hypothesize that people who didn't play many games are likely to be somewhat unreliable, having not gained enough experience to judge the games accordingly, although we have no direct evidence for this. This suggests that the Quake-GPE is performing as required, although this result cannot be truly verified.

5.8.4 Quake-PDC results

The first step in analysing the data received from the public was to examine the effect of thresholding the data using the α and β values learnt for each player by the Quake-GPE Model. Recall that each member of the public p has an $\alpha^{(p)}$ and $\beta^{(p)}$ associated with them, which provides an estimate of their sensitivity and specificity. By ignoring data from people with values below a threshold, it is possible to filter out unreliable data from the training set. Figure 5.22 shows the remaining players as the α and β threshold is swept. It is interesting to note that the initial increment of the threshold from 0.0 to 0.01 results in the sensitivity of 25 players being declared unreliable and specificity of 59 players being declared unreliable, which accounts for a fairly large chunk of the player base. This is an interesting phenomena, but Figure 5.23 sheds some more light upon it. This second graph shows the total number of remaining surveys as the data from people with unreliable sensitivity and specificity is removed. Interestingly, it shows that the people removed as the threshold is initially incremented only account for approximately 50 surveys. It is likely that the Quake-GPE model is detecting people who only briefly played the game and probably didn't provide very reliable feedback. This is exactly the result that was hoped for.

When providing a classification for a play-log, the Quake-PDC provides a confidence, based on the posterior probabilities outputted by its constituent random forests (see Section 2.3.6 and Section 5.7.4). If the confidence is above a specific value, by default 0.5, then the output of the Quake-PDC is considered to be a positive classification, otherwise it is considered a negative classification. Better results can be achieved by varying this class threshold, which adds bias towards

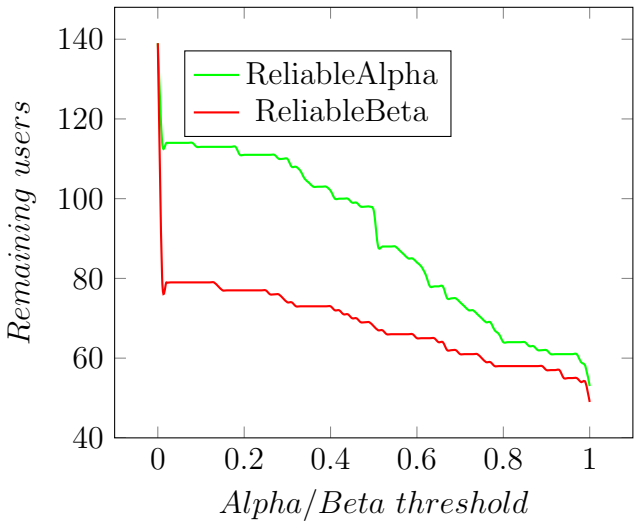

Figure 5.22: Remaining users as alpha/beta threshold swept.

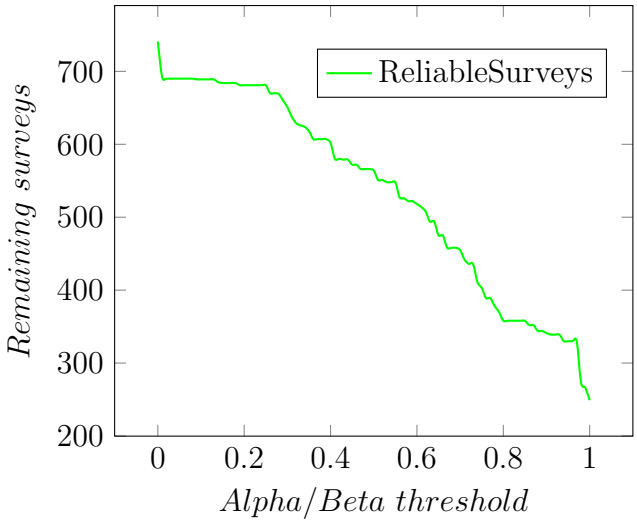

Figure 5.23: Remaining surveys as alpha/beta threshold swept.

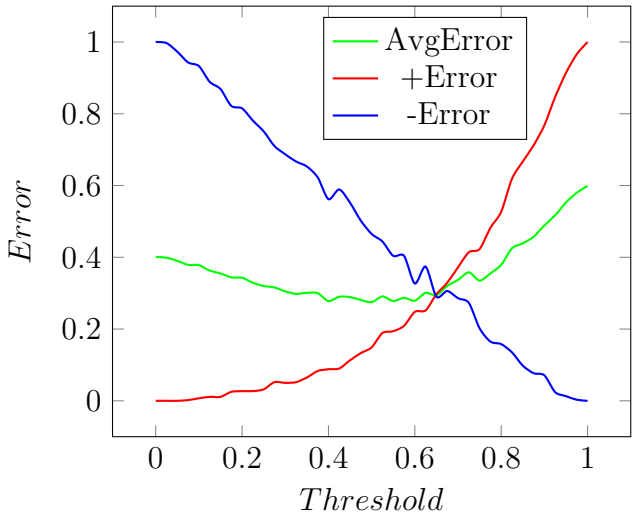

Figure 5.24: PDC Error on Test set vs Class Threshold.

one particular class over the other. The approach for testing the Quake-PDC is to use cross-validation on the training set and in Figure 5.24 we show the cross-validation error as the class threshold is varied between 0 and 1. It is apparent from the figure that the positive/negative error rates converge at a class threshold of 0.61, producing an average error of 0.21, which is an acceptable error rate given the issues faced with noisy, unreliable data. As such, we decided that 0.61 would be the class threshold used throughout the remainder of the experiments.

Another way in which the Quake-PDC cross-validation error can be improved is to apply a rejection threshold, in a similar manner to that which was applied to the Quake-CC in Section 5.7.2. Figure 5.25 shows the error rates as the rejection threshold is varied between 0 and 1, and Figure 5.26 shows the rate of rejected samples. Using a rejection threshold of 0.25 an average error rate of 0.24 is achieved, with a rate of 0.25 and 0.27 on the positive and negative classes, respectively. This is considered to be a reasonable trade-off and as such was chosen as the rejection threshold.

To see just how the PDC model was determining whether a player was having fun or not, we trained a random forest on the entirety of the PDC training set, minus the surveys from the player who played an excessive number of games, and then extracted the features sorted by importance score the random forest gave them. The full results of this are listed in Appendix D. Table 5.5 gives the top 10 features as extracted from the longer table. For each of the top ten attributes,

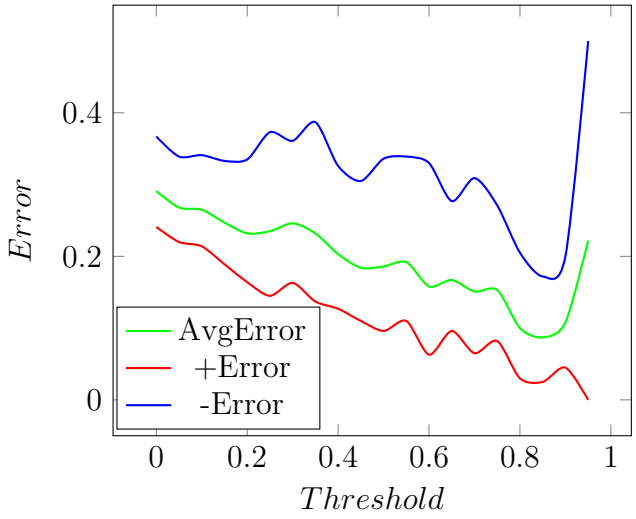

Figure 5.25: PDC Error on Test set vs Reject Threshold.

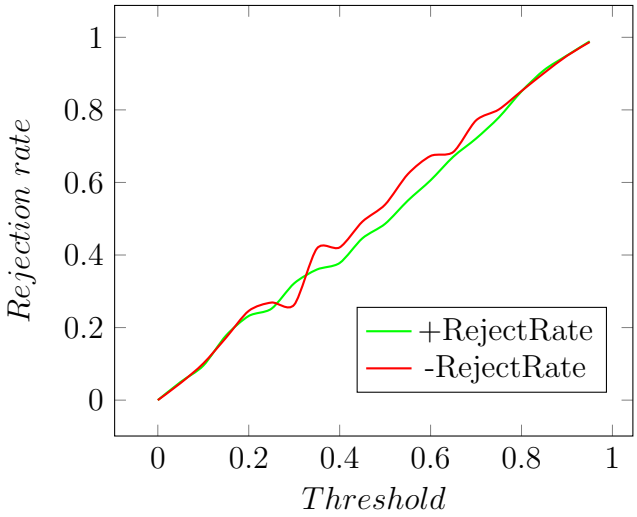

Figure 5.26: PDC Rejection Rates on Test set vs Reject Threshold.

Table 5.6 shows the mean and standard deviation of the attribute when the play-logs were marked as fun and not fun. This table is interesting because it shows us the effect on the players interpretation of fun as the attribute changes, although it cannot be entirely trusted as many play-log attributes are likely to act in groups.

The top two features are the player death count and completion, which by their score indicate they were much more important than the other features. The means suggest that a higher death count means it is less likely the player enjoyed the game. This makes sense, since more deaths are likely to lead to frustration, or indicate that the content is just not suitable for the players level of ability. A higher completion rate in a play-log is more likely to indicate a positive experience than a negative one according to the results. This makes sense logically, since a player is more likely to continue playing something they enjoy to completion, rather than something they do not enjoy. Interestingly, the number of grunts killed is ranked quite highly, with more grunts being present having a positive effect. Grunts are a humanoid monster that are easy to kill, and when killed with explosive weapons will often explode causing a rather satisfying experience. It could be that player enjoy levels with monsters they can mow down easily. This theory seems to stand, as killing more rottweilers, another easy monster to kill, also seems to increase the enjoyment of the level. Games that result in more fiends being spawned, which are quite a difficult monster to kill, seem to be less enjoyable than those games where fewer fiends are spawned. Fiends are difficult in that they launch themselves at the player, often resulting in the player damaging themselves if they are using an explosive weapon, and quickly killing the player in melee combat if not. Other attributes to rank highly include distance travelled, monsters killed and the number of input events, all of which are attributes linked with how much of the level was completed. The amount of health received was also influential. Strangely, the means for the health related attributes do not different much between enjoyed and not enjoyed games. This may indicate they only have an effect in conjunction with other attributes, or simply that small variations in this parameter are meaningful.

Rank	**Feature**	**Score**
0	Completion (Monsters Killed/Total Monsters)	6.36389303
1	Player Death count/ticks	5.66099739
2	Grunt killed	2.22189641
3	Total Input Events	2.14130044

4	Fiend spawns	1.08697462
5	Monsters killed	0.979292214
6	Total distance	0.794655621
7	Rottweiler killed	0.646714807
8	In Health/ticks	0.570297718
9	In Health	0.501151502
10	Total ticks	0.498960942

Table 5.5: PDC Model Feature importance (Top 10)

Attribute	**Mean (fun=1)**	**S. dev (fun=1)**	**Mean (fun=0)**	**S. dev (fun=0)**
Completion (Monsters Killed/Total Monsters)	0.739786	0.363648	0.409536	0.403648
Player Death count/ticks	0.00472	0.002354	0.002951	0.010833
Grunt killed	15.783784	23.823454	11.764310	31.556665
Total Input Events	1105.416626	1009.648560	799.323242	795.587463
Fiend spawns	17.349098	96.157455	41.956230	122.201279
Monsters killed	47.029278	58.440601	34.828281	61.966492
Total distance	216755.609	971947.125	114658.992	106461.969
Rottweiler killed	14.083333	19.817169	9.723906	21.440807
In Health/ticks	0.278697	0.549001	0.272439	0.648923
In Health	108.378380	214.095032	98.131310	189.794342

Table 5.6: Mean/Standard Deviation of play-log attributes for binary fun

In summary, after examining the cross-validation error, we identified a suitable rejection threshold and class bias for the Quake-PDC model. The ensemble approach of combining different data sets generated by filtering using different α and β threshold values was found to alleviate the potentially difficult problem of selecting only one threshold. Improvements to this model could look into the method for selecting different thresholds for the ensemble, rather than the basic approach we have taken. We defer analysis of the performance of the PDC Model

to the next section, the reason being that the best test of the Quake-PDC is to see how it performs on a group of target players in conjunction with the Quake-IP model, rather than those used specifically for training the system.

5.8.5 Quake-IP results

To determine the performance of the Quake-IP Model, and by implication the LBPCG-Quake framework as a whole, we decided to contrast it against two other models. We termed the first of these the *Random Model*. This model uniformly selects games from the original content vector space $\mathcal{G}$. It uses no prior knowledge, filtering or adaptation in its selection, apart from that built into the OBLIGE level generator itself. We named the second model the *Balanced Model*. This model presents an equal number of games from each difficulty category, as dictated by the developer. The games are chosen from the set of games given to the public during training of the GPE Model. Each of these has been labelled by the developer, meaning the label is precise. Since the Quake-IP Model uses these exact same games itself while in the CATEGORIZE state, one can view the Balanced Model as somewhat similar. However, for each category the Balanced Model uniformly samples the games, whereas the IP Model selects games based on popularity and controversy. In addition, the Balanced Model does not attempt to adapt to a target player, whereas this is the primary goal of the IP Model. Our expectations before performing the experiment was that the Random Model would perform the worst, given the uniform sampling of $\mathcal{G}$ and the prevalence of unacceptable content. We predicted, however, that the Balanced Model would present a challenge to the IP Model, in particular if the IP Model failed to identify the category of players and ended up presenting equal numbers of games from each category itself.

To contrast the three models we developed a network-based system, which was somewhat similar in functionality to the system used for training the Quake-GPE Model. The system presented 20 games from the Quake-IP Model, 10 games from the Balanced Model and 10 games from the Random Model to each participant. At each iteration, one of the three models was chosen randomly to present a game. Although we collected 20 surveys from each participant for the Quake-IP Model, only 10 of these were used in our subsequent statistical tests. We collected 20 to give the Quake-IP Model more to time to move into the PRODUCE and GENERALIZE states and provide further analysis. After playing each game, the

player was asked a number of questions, which are given in Table 5.7. Please note that such feedback was not used for the purposes of adaptation, it was used to assess the performance of the models only.

ID	**Question**	**Response**
A	How difficult was that?	Too easy, Just right, Too hard
B	Do you want to see more levels like that?	No, Yes
C	Did you find the level fun?	No, Yes
D	Do you think other people would enjoy it?	No, Yes
E	How do you rate the level?	Very Bad, Bad, Average, Good, Very Good

Table 5.7: IP Evaluation Survey questions and available responses

During the Quake-GPE Model training we observed that players, on average, played under 5 games each. In the Quake-IP evaluation we required that each player played 40 games, which made it somewhat more challenging to find candidates. Each participant was asked to give honest feedback and play each game for as long as they wished. Before the first game was presented, the players were asked two questions each, which are given in Table 5.8. This gave us an indication of the ability of each participant, of course depending on whether the participant was accurate in their own self-assessment.

Question	**Response**
What kind of gamer would you describe yourself as?	(a) I don't play games much - if at all, (b) I play games sometimes, (c) I play games regularly, (d) I'm a hardcore gamer
How would you rate your skill level at action video games?	(a) Inexperienced, (b) Average, (c) Good, (d) Very good

Table 5.8: IP Evaluation self-assessment

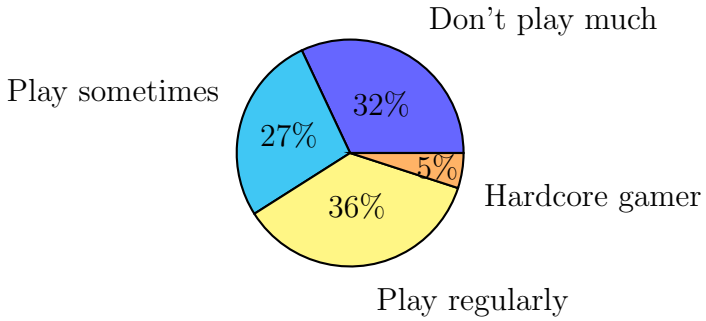

Figure 5.27: IP Evaluation: What kind of gamer would you describe yourself as?

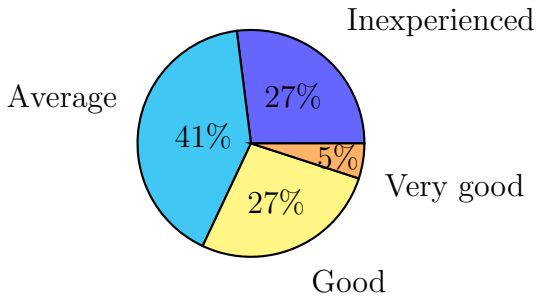

Figure 5.28: IP Evaluation: How would you rate your skill level at action video games?

The feedback from one person was removed as they answered positively for every response to the question C (binary fun). This meant that the experiment was successfully completed by 22 people. The self-assessment feedback is displayed in Figure 5.27 and Figure 5.28. Each participant answered 10 surveys from the Random and Balanced models and 20 from the Quake-IP Model, and each survey consisted of a play-log, a game, feedback for the game (see Table 5.7) and state information for the model used e.g. IP Model target category. We viewed each participant as a separate "experiment". As such, it was necessary to devise a metric for assessing the performance of each model in each experiment. We came up with three such metrics. The first metric, *binary fun*, was a simple sum of how many games received positive feedback for the question "Did you find the level fun?" (question C). For example, if the Random Model produced 6 games in an experiment which the player labelled as "Yes" for question C, then it received the score 6, or in other words 60% of the games it produced (6 of 10) were fun for this player. The second metric, which we termed the *simple metric*, was a scoring system which rewarded positive feedback on each question asked.

For some player p, model m and question q (from Table 5.7), let q_p^m denote the number of games produced by model m for p which received positive feedback for question q. For example, if q is Question A, and player p responded to 3 of the 10 games m produced with "Just Right", then $q_p^m = 3$. For Question E, we counted any feedback above and including a response of "Average" as positive. We define the simple metric, for player p and model m as $\Sigma_q q_p^m$ i.e. a sum over all positive responses.

The third metric, which we termed the *advanced metric* was a scoring system designed to assess the success of each of the model at providing for the player's categorical preference. The metric was designed to reward models which produced content from the player's favourite category, which was determined by summing up the positive feedback for each category of game presented to the player. For the 30 games played by player p, let $N_{p,c}$ and $N_{p,c}^E$ denote the number of the games of difficulty category c played and the number of games of difficulty category c enjoyed by them, respectively. Hence, the preference rate of player p for games of difficulty category c is $\rho_{p,c} = N_{p,c}^E / N_{p,c}$. For player p, let $N_{m,p,c}$ be the number of games of difficulty category c generated by model m. For model m and player p, the scoring metric $S_{m,p}$ is defined as

$$S_{m,p} = \sum_{c \in \mathcal{C}} \rho_{p,c} N_{m,p,c},$$

where $\mathcal{C}$ is the set of five content categories in the LBPCG-Quake framework. Intuitively, a higher score awarded to a model indicates that the model produces more games of the difficulty category enjoyed by the player. Thus, it is possible to use such a metric to measure the success of a model in terms of personalization. For games where no category label was available, we used the Quake-CC Model to provide a classification.

Figure 5.29, Figure 5.30 and Figure 5.31 shows the result of applying the three metrics to the IP Evaluation surveys.

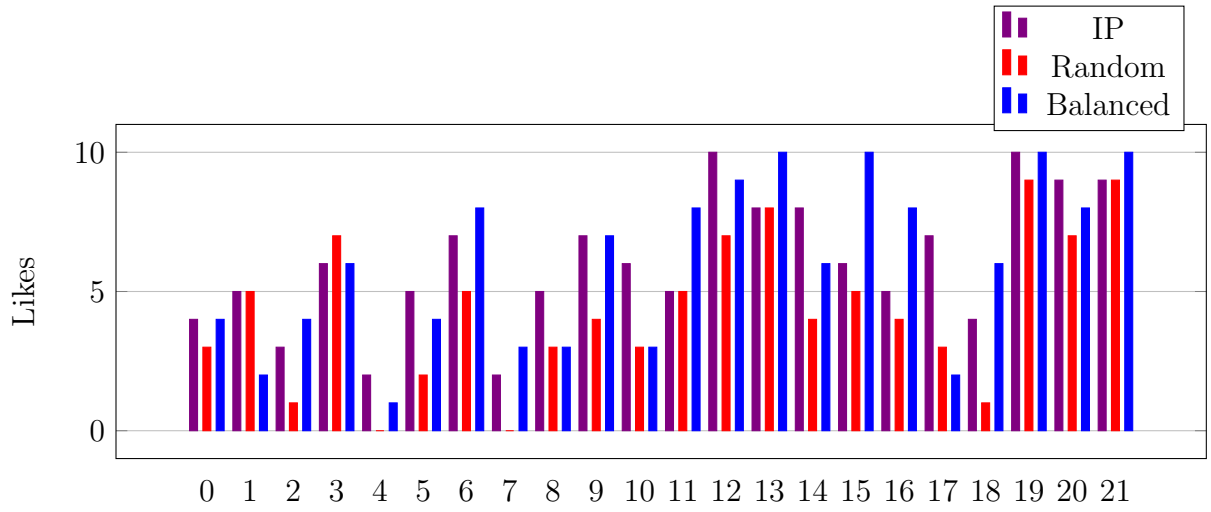

Figure 5.29: Binary Fun Metric for Quake-IP Evaluation

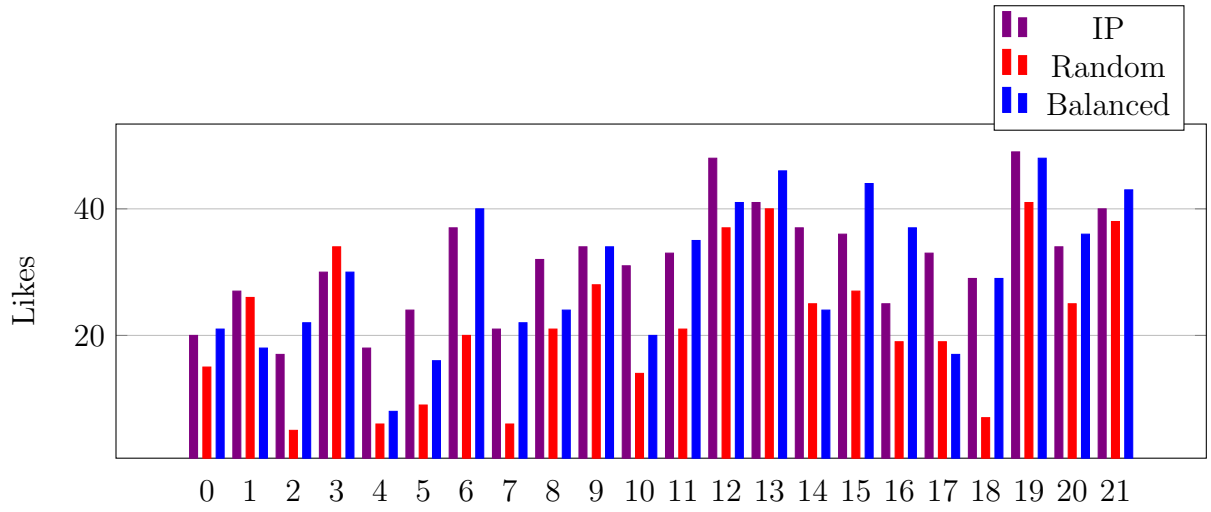

Figure 5.30: Simple Metric for Quake-IP Evaluation

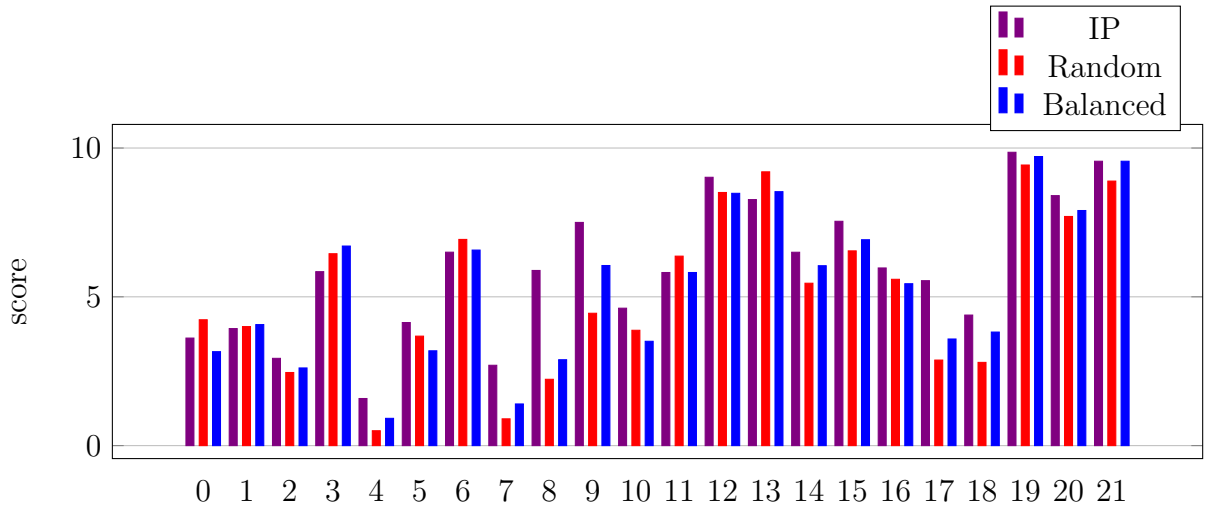

Figure 5.31: Advanced Metric for Quake-IP Evaluation

In Section 2.3.1 we discussed the fallacy of using average error rates to compare different machine learning algorithms, a stance taken by Demšar [14]. The three metrics we have presented could be considered to be a type of accuracy measurement, i.e. how accurate each model is in predicting whether a player will enjoy some content it proposes. Since we have three models, either Friedman or ANOVA are good candidates for use as statistical tests. Analysis using Kolmogorov-Smirnov in MedCalc [52] showed that the distribution of the results did conform to a normal distribution, which is a requirement for ANOVA. However, since we did not want to take the sphericity assumption required by ANOVA, we chose the Friedman test instead. The Friedman test was executed three times, one for each metric, using a statistical significance of 0.05 i.e. the null-hypothesis is rejected if its probability is less than 5%. The output of the statistics package MedCalc [52] for the Binary Fun metric, Simple metric and Advanced metric is given in Figure 5.32, Figure 5.33 and Figure 5.34, respectively. The important sections of these figures are the third and fourth tables. The third table gives the *p-value*, which if under our threshold of 0.05 allows us to safely reject the null-hypothesis, meaning that some of the models produce significantly different results. If the null-hypothesis is rejected, the post-hoc testing in the fourth table tells us which of the models were different and also their mean rank, indicating which performed better.

Cases in spreadsheet	22
Cases with missing values	0
Cases included in the analysis	22

Descriptive statistics

	n	Minimum	25th Percentile	Median	75th Percentile	Maximum
Binary_lp	22	2.0000	5.000	6.000	8.000	10.000
Binary_Random	22	0.0000	3.000	4.000	7.000	9.000
Binary_Balanced	22	1.0000	3.000	6.000	8.000	10.000

Friedman test

F	13.6500
DF 1	2
DF 2	42
P	0.00003

Multiple comparisons

Variable	Mean rank	Different (P<0.05) from variable nr
(1) Binary_lp	2.3636	(2)
(2) Binary_Random	1.3182	(1) (3)
(3) Binary_Balanced	2.3182	(2)

Minimum required difference of mean rank: 0.4565

Figure 5.32: Friedman test for binary metric (MedCalc).

Cases in spreadsheet	22
Cases with missing values	0
Cases included in the analysis	22

Descriptive statistics

	n	Minimum	25th Percentile	Median	75th Percentile	Maximum
Simple_lp	22	17.0000	25.000	32.500	37.000	49.000
Simple_Random	22	5.0000	14.000	21.000	28.000	41.000
Simple_Balanced	22	8.0000	21.000	29.500	40.000	48.000

Friedman test

F	18.3093
DF 1	2
DF 2	42
P	<0.00001

Multiple comparisons

Variable	Mean rank	Different (P<0.05) from variable nr
(1) Simple_lp	2.4318	(2)
(2) Simple_Random	1.2273	(1) (3)
(3) Simple_Balanced	2.3409	(2)

Minimum required difference of mean rank: 0.4474

Figure 5.33: Friedman test for simple metric (MedCalc).

Cases in spreadsheet	22
Cases with missing values	0
Cases included in the analysis	22

Descriptive statistics

	n	Minimum	25th Percentile	Median	75th Percentile	Maximum
Adv_Ip	22	1.5800	4.140	5.870	7.540	9.860
Adv_Random	22	0.5000	2.880	4.955	6.930	9.430
Adv_Balanced	22	0.9200	3.190	5.630	6.920	9.710

Friedman test

F	5.4176
DF 1	2
DF 2	42
P	0.00807

Multiple comparisons

Variable	Mean rank	Different (P<0.05) from variable nr
(1) Adv_Ip	2.5000	(2) (3)
(2) Adv_Random	1.6364	(1)
(3) Adv_Balanced	1.8636	(1)

Minimum required difference of mean rank: 0.5489

Figure 5.34: Friedman test for advanced metric (MedCalc).

The tests on the binary fun and simple metric both indicate that the IP Model and Balanced Model outperform the Random Model, but there is no statistically significant difference between the performance of the IP Model and Balanced models. However, for the advanced metric, the IP Model outperforms both the Random and Balanced Models, and there is no statistically significant difference between the performance of the other two models. The results for the advanced metric suggest that the Quake-IP Model may very well be doing the job it sets out to do, which is to generate content of the appropriate category for the target player, and in this respect it outperforms the other models. However, the results for the binary fun and simple metrics indicate that player satisfaction between the Balanced Model and Quake-IP Model are similar, regardless of this.

To analyse the results in more detail, we examined the states that the IP Model entered for each participant. We are primarily interested in how often the Quake-IP Model managed to enter the PRODUCE state for each player, as this indicates that the IP Model had detected that the player had a particular

preference for a category. The IP State transition table, and several other pieces of information, is given in Appendix E for each player whose IP Model entered the PRODUCE state. The Quake-IP Model entered the PRODUCE state for 10 out of the 22 participants. This means that it failed to match 12 of the players to any particular category. We suggest that this could have been caused by one or more of several problems: (a) the player not having a preference for a particular category, (b) the PDC threshold being set too high, meaning even if the PDC Model classified play-logs correctly it's classification was not taken into account or (c) the PDC Model not being accurate enough to detect if a player was having fun. In addition, we note that if the Quake-IP fails to enter the PRODUCE state, it will continue to present games ranked by controversy, as justified in Section 4.4.5. This potentially places it at a disadvantage to the Balanced Model, if the Balanced Model is "luckier" and chooses more appealing games from $\mathcal{G}_{GPE}$. Table 5.9 provides a summary of the level of success when entering the PRODUCE state for several of the players for which it did so. Note the analysis in this table is based on binary fun feedback from the player.

Player ID	**Observations**
Player 8	The Quake-IP Model also entered into the PRODUCE twice for this player. On the first occasion, it successfully determined that the player enjoyed two games of category 2 in a row. It then generated eight games for this category of which the player enjoyed half. Later on, the Quake-IP Model correctly identified that the player enjoyed two games of category 3 in a row, and produced two games, of which the player enjoyed one. The Quake-IP model correctly determined this and switched back to the CATEGORIZE state.

Player 10	The Quake-IP model moved into the PRODUCE state twice for this player. On the first occasion it correctly determined the player enjoyed two games of category 1 in a row and so shifted to the PRODUCE state, presenting four games of this category, of which the player only liked one. It correctly determined that the player disliked one of the games and re-entered the CATEGORIZE state. Later on, the Quake-IP model briefly entered the PRODUCE state again, after correctly determining the player enjoyed two games of category 3 in a row. However, it then detected the player disliked the first game presented to the player in the PRODUCE state, and so shifted back to the CATEGORIZE state.
Player 11	The Quake-IP Model incorrectly determined that the player enjoyed two games of category 3 in a row (they only enjoyed one). The Quake-IP model then presented four games of category 3, of which the player enjoyed three, before correctly determining the player wasn't having fun and switched back to the CATEGORIZE state.
Player 13	The Quake-IP Model only momentarily moved into the PRODUCE state for this player, for one iteration. The model correctly determined that the player enjoyed two games of category 0 in a row. However, it then incorrectly determined that the player disliked the next game it presented, and as such switched back to the PRODUCE state.

Player 15	The Quake-IP Model correctly determined that the player enjoyed two games of category 1 in a row, although it failed to see that the player also enjoyed two games of category 0 before-hand due to a low PDC confidence. It presented seven games for category 1, while in the PRODUCE state, before exiting back to the CATEGORY state. Of these games, only two were enjoyed by the player.
Player 16	The Quake-IP Model entered into the PRODUCE state twice for this player. On the first occasion, it incorrectly determined that the player enjoyed two games of category 1 in a row, even though the player only enjoyed one, it then also incorrectly determined the player didn't enjoy the first game it presented in the PRODUCE state and so switched back to the CATEGORIZE state. Later on, the Quake-IP incorrectly identified that the Player enjoyed two category 0 games in a row, and entered the PRODUCE state. Of the six games that were presented, four of these were enjoyed by the player.

Table 5.9: Quake-IP PRODUCE state analysis

We analysed the performance of the Quake-PDC Model on the evaluation surveys. Figure 5.35 shows the effect on the error rates of the PDC of sweeping the rejection threshold from 0 to 1 and Figure 5.36 shows the rejection rate resulting from sweeping the rejection threshold from 0 to 1. We observe from these two graphs that the rejection threshold was probably well chosen. Reducing the rejection threshold by a small amount would have had a noticeable effect on the error rates. Figure 5.37 shows the error rates as the class boundary is swept from 0 to 1. Recall from Section 5.8.4 that a class threshold of 0.61 was chosen as a result of studying Figure 5.24. We observe that the positive and negative error rates for the IP evaluation meet at a class boundary of approximately 0.525, suggesting that the class boundary chosen based on the results from the GPE surveys may have been too high, which could have increased the error rate of the Quake-PDC.

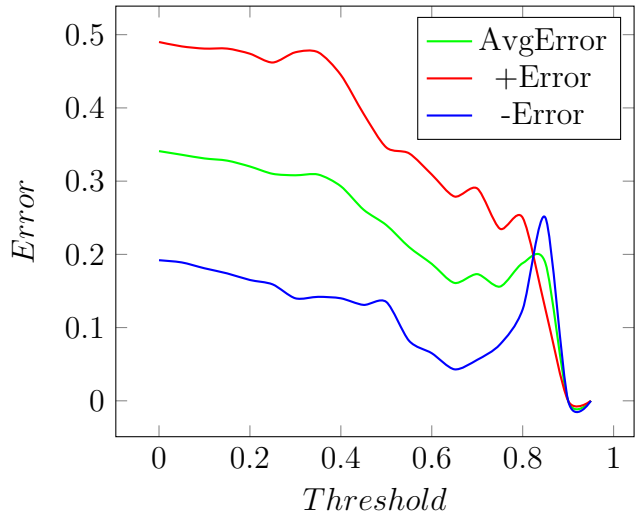

Figure 5.35: PDC Error on IP Evaluation Surveys vs rejection threshold

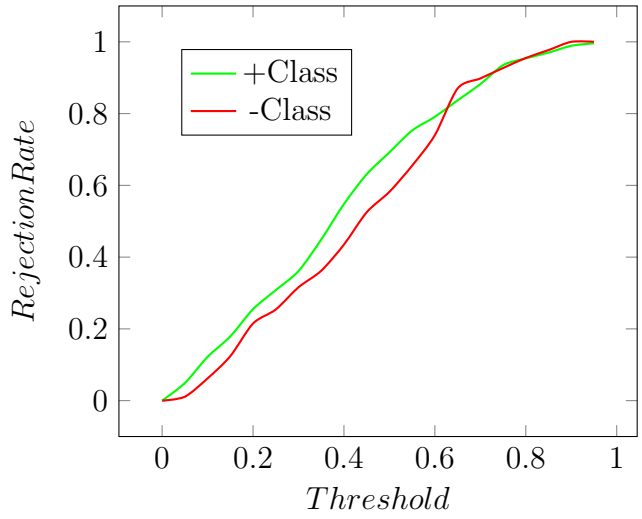

Figure 5.36: PDC Rejection Rates on IP Evaluation Surveys vs rejection threshold

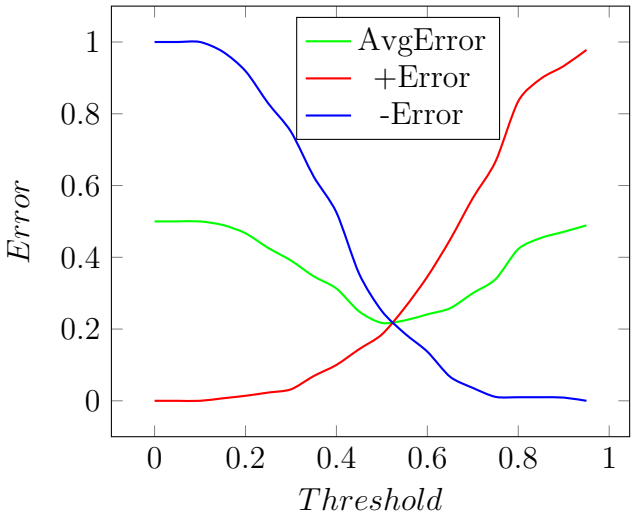

Figure 5.37: PDC Error on IP Evaluation Surveys vs class threshold

Since the Random Model uniformly sampled $\mathcal{G}$, this meant that it had the possibility of presenting games the Quake-ICQ model found unacceptable to the player. Therefore, this enabled us to verify whether the participants agreed with the ICQ Model's notion of whether content is acceptable or not. For every game produced by the Random Model, we assessed its acceptability using the ICQ Model, then determined the rate at which players labelled unacceptable games positively and negatively, by checking the response to question C. We found that players liked 32% of the games that the Quake-ICQ Model found unacceptable. This means that the Random Model was able to produce fun content that the Quake-LBPCG could not produce due to its initial filtering. This suggests a weakness in the training of the ICQ Model, either in the learning mechanism or the notion of acceptability envisioned by the developer, which affects the model as a whole.

5.9 Summary

In this section we will make our parting conclusions with respect to the LBPCG-Quake framework. The LBPCG-Quake framework was implemented as per the requirement of the LBPCG framework in Chapter 4. A number of design choices were made in the implementation, such as: (a) the use of SVMs, extended with probabilistic output, for active learning in the Quake-ICQ and Quake-CC models, (b) the use of SVMs as the resulting classifier/regressor for the Quake-ICQ and

Quake-GPE models, (c) the use of random forests for the resulting classifier for the Quake-CC and Quake-PDC models. We built up the Quake-ICQ and Quake-CC using a single developer and built the Quake-GPE and Quake-PDC using a large number of anonymous people on the internet. Analysis of the Quake-ICQ and Quake-CC models seemed to suggest that they were successful to an extent, although the availability of models to contrast against was limited, so we cannot make conclusions as to whether other implementations would perform better. A final evaluation study using 10 participants was performed, which compared the LBPCG-Quake framework against two non-adaptive models. The data collected in this study helped us to assess the performance of the Quake-PDC and Quake-IP models. The results indicated potential weaknesses in the Quake-PDC Model, such as the choice of the class boundary. Three metrics were constructed which allowed us to rate the success of the each of the models in the evaluative study. Statistical analysis showed that the Quake-IP Model outperformed the non-adaptive models for one of the metrics used. For the other two metrics, the Quake-IP and one of the non-adaptive models outperformed the third model, but did not perform statistically significantly differently from each other. It was noticed that the Quake-IP Model failed to enter its PRODUCE state for over half of the players, which is important for its adaptive capabilities. Several possible explanations of this phenomena were offered. It was also found that the Quake-ICQ model may have been too restrictive in its filtering of content, hiding potentially fun content from the target player. In summary, we conclude the LBPCG-Quake framework, as a new approach to data-driven PCG, produced some promising results and we have highlighted how it could be improved in future research.

Chapter 6

Concluding Remarks

6.1 Summary

In this thesis we have introduced a novel framework for Procedural Content Generation (PCG) in video games. PCG allows content to be constructed in video games automatically, rather than by hand by developers of the game. PCG has a long history in video games, dating back to the 80s, and has many notable success stories in commercial applications. As video game development costs spiral, PCG has an important role to play in the future of video games and could be used for many advanced applications, including creating a sturdy base of content on which developers can create games or even individually tailoring content for players in a dynamic fashion.

A literature review of the current state of art in Section 3.3 revealed that there is much exciting ongoing research into PCG, with researchers aiming to redefine and improve how content is created for video games. Many of the current approaches fall under the description of Search-Based Procedural Content Generation (SBPCG) algorithms, which usually use evolutionary approaches with either a pre-defined or learnt fitness function to find desirable game content. While SBPCG shows promising results in several applications, it also poses many challenges that need to be overcome.

The main contribution of this thesis is a framework that aims to solve several of these problems, described in Chapter 4. The Learning-Based Procedural Content Generation (LBPCG) framework adapts content for an arbitrary target player using five models, each trained using learning-based approaches. The LBPCG itself is quite high-level, and while it recommends the use of several enabling

techniques, the exact details are left to the implementers of the system. We have provided a concrete implementation for the LBPCG using the game Quake in Chapter 5, which show promising results for the framework.

It can be stated that the LBPCG framework shows some promising results in solving the following challenges in the current state of the art research into PCG:

1. *Avoiding catastrophic content.* Unacceptable content, such as that which is unplayable or that which is of exceedingly poor quality should never be given to a target player. The Initial Content Quality (ICQ) model described in Section 4.4.1 is designed to filter out unacceptable content. An implementation of the ICQ was created for Quake as described in Section 5.7.1, which showed a level of success at identifying undesirable content in a test set as described in Section 5.8.1. In Section 5.8.5 the Quake-ICQ Model was found to filter our content that players did in fact enjoy, indicating that its performance could be improved on the Quake/OBLIGE content space. It also remains to be seen if the approach could generalize to other content spaces. The Individual Preference (IP) model described in Section 4.4.5, is designed in such a way that it immediately begins by presenting content that has been rated by the public, guaranteeing some level of quality. The results in Section 5.8.5 were not entirely conclusive, but showed that the Quake-IP model performed better than two non-adaptive models when performance was measured using one metric. The Quake-IP Model performed statistically equivalently to another model using the two other metrics, while at the same time performing better than the remaining model.

2. *Removing the requirement for pre-defined fitness functions.* Many existing SBPCG approaches use pre-defined fitness functions for exploring the solution space, although researchers have recently been using approaches that learn the evaluative functions required as detailed in Section 3.3. The advantages of using a learning-based approach are that the implementers of the PCG framework need less prior knowledge about the problem. Using a non-learning approach means that the developer has to be able to articulate their knowledge to form an evaluative function usually made up rules. The learning-based approach has more potential to capture knowledge of the fitness of content from the developer, perhaps discovering information they would be unable to encode in a rule-based way. All components of

the LBPCG are learning-based, which provides what we believe to be a much more robust and generalized approach. The results of each individual model in the LBPCG-Quake in Section 5.8 suggest that the learning-based approach to learning evaluative functions is potentially achievable.

3. *How to exploit potentially unreliable information acquired from the public.* SBPCG and similar approaches to PCG generally require that desirable content is found with the assistance of the public. The public can be difficult to work with, either because they provide poor feedback for malicious reasons or simply because their individual tastes are too unique. The General Player Experience (GPE) model component of the LBPCG is a crowd-sourcing approach designed to circumvent this problem by determining contributor reliability and using this information to form a robust model of general consensus. The results for the Quake-specific implementation of this model in Section 5.7.3 are given in Section 5.8.3 and suggest that this approach can work.

4. *Modelling the player without interrupting their experience.* In video games development, it goes without saying that making the player's experience enjoyable is of critical importance. Anything that detracts from that should be avoided. In many existing approaches to player modelling it is common to ask players questions about their experience as reviewed in Section 3.2. The Player-Driven Categorization (PDC) model along with the Content Categorization (CC) model, described in Section 4.4.4 and Section 4.4.2, are formed to avoid this problem by enabling the LBPCG framework to recognize different categories of content and recognizing whether a player is having a positive experience. Both models were implemented for the LBPCG-Quake with the results given in Section 5.8.4 and Section 5.8.2, respectively. The Quake-CC model indicated a level of success when it was tested on a test set, indicating that incorrect classifications often ended up in adjacent classes. When tested on the evaluation study in Section 5.8.5, the Quake-PDC indicated that there may be potential issues, in particular with the selection of a threshold value that it uses.

5. *Handling the volatility of player preference.* Video games players are volatile entities. At one point they may prefer one type of content and then later on something completely different. The IP model component of the LBPCG is

designed to tackle this problem of concept drift by monitoring the players experience over time and then changing the type of content it produces when a bad experience is detected. It was inconclusive from the results in Section 5.8.5 whether or not the Quake-LBPCG has the ability to recognize and solve concept drift and future work will need to address several issues found.

From the results for the LBPCG-Quake experiment, this thesis concludes that a learning-based approach to PCG has potential, but more research should be conducted to improve issues encountered. We propose that our results are a good baseline for comparison in future research within this area.

6.2 Limitations

The LBPCG framework is a new approach to PCG and has some limitations. In this section we divide the issue into two parts, firstly issues with the general LBPCG framework, and secondly issues encountered in the LBPCG-Quake.

6.2.1 LBPCG limitations

1. The LBPCG framework is dependent on the quality of each of its constituent models to have good performance. For example, if the PDC model has a low accuracy, then the IP model will be unable to recognize whether the player is having a good experience or not, resulting in the IP model producing generally acceptable non-optimized content, or in the worst case, presenting inappropriate content, although some safeguarding measures were taken in our IP algorithms. Another challenge is recognizing whether the models do have good performance or not given the fact that in most circumstances there is not a ground truth and feedback is noisy and subjective.

2. Our current enabling techniques for the LBPCG suggest that active learning can be used for the ICQ and CC models. Although these methods worked for the LBPCG-Quake, there is in fact no proof these approaches will converge and lead to satisfactory performance with a limited number of annotated examples. This can be mitigated to some extent by requiring

the models of the LBPCG to output confidence along with their classifications and using this information to ignore classifications that have a low confidence.

6.2.2 LBPCG-Quake limitations

1. It was found after informally speaking to participants after the evaluation of the Quake-LBPCG that several players didn't particularly have a preference for the difficulty of a game, rather they had preferences for what weapons were available and the type of monsters they encountered. The core concept of the IP Model is identifying a player with a category of content they like and then producing content of that category for them until their preference shifts. If the content categories themselves are ill-defined, then the LBPCG can potentially failed at its main objective. In the current designed for the LBPCG framework, the developers define the categories. However, a better approach may be to implicitly learn the categories of content using a learning-based approach. Since there are a lot of people with potentially very diverse preferences, it would be wise to enlist the help of the public to identify what the categories are, potentially implicitly. As such, the Quake-CC training process would, in addition to the Quake-GPE process, be something that is trained using members of the public. We believe this is an important vector of research moving forwards which could greatly improve the LBPCG.

2. Machine learning is a data-driven methodology and this means that the training data must contain sufficient information. The public test surveys, in the case of the LBPCG-Quake, have no assurance in quality feedback or the inclusion of a sufficiently diverse set of player types. In contrast, the target players in the Quake-IP evaluation were better selected and hence they played all games seriously and were likely to have given their genuine feedback. The Quake-PDC, trained on the public test surveys, behaved in a somewhat different manner when test on the Quake-IP evaluation subjects. One explanation for this may be due to the members of the public not being varied enough in their play-style, or simply being too unreliable in their feedback. In conclusion, the public test surveys should be done on a set of more appropriate beta players, which may incur a cost in commercial

games.

3. Through the training of the Quake-ICQ, Quake-CC, Quake-GPE and Quake-PDC we chose a number of concrete implementations of the abstract approaches proposed in Chapter 4, using machine learning techniques such as random forests and SVMs. In addition, active learning was used on several occasions using a specific approach, uncertainty sampling. These implementations were not contrasted with other potential approaches, so we make no claims as to whether better methods exist. In future work we would like to see if more sophisticated algorithms could be deployed, which would potentially make the LBPCG-Quake more successful.

In conclusion, the LBPCG framework provides a fresh approach towards generating personalised game content. Our proof-of-concept prototype LBPCG-Quake showed some success, but also identified a number of issues ranging from the robustness of the LBPCG framework to its enabling techniques to be studied in future research.

Bibliography

[1] V. Ambati, S. Vogel, and J. Carbonell. Active learning and crowd-sourcing for machine translation. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, 2010.

[2] G. Andrade, G. Ramalho, A. S. Gomes, and V. Corruble. Dynamic game balancing: An evaluation of user satisfaction. In *The Artificial Intelligence for Interactive Digital Entertainment Conference*, pages 3–8, 2006.

[3] T. H. Apperley. Genre and game studies: Toward a critical approach to video game genres. *Simulation and Gaming*, pages 6–23, 2006.

[4] S. Asteriadis, N. Shaker, K. Karpouzis, and G. N. Yannakakis. Towards players affective and behavioral visual cues as drives to game adaptation. In *LREC Workshop on Multimodal Corpora for Machine Learning*, 2012.

[5] J. Babcock. Cellular automata method for generating random cave-like levels. Website, 2005. `http://roguebasin.roguelikedevelopment.org/index.php?title=Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels`.

[6] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[7] Y. Bjrnsson and H. Finnsson. Cadiaplayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1:4–15, 2009.

[8] L. Breiman. Random forests. *Journal of Machine Learning*, 45(1):5–32, 2001.

[9] C. Browne and F. Maire. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2:1–16, 2010.

[10] L. Cardamone, G. N. Yannakakis, J. Togelius, and P. L. Lanzi. Evolving interesting maps for a first person shooter. In *EvoWorkshops*, pages 63–72, 2011.

[11] C. Chang and C. Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:1–27, 2011.

[12] C. Chang and C. Lin. Libsvm – a library for support vector machines. Website, 2014. `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`.

[13] M. Csikszenthmihalyi. *Flow - the psychology of optimal experience*. Harper Perennial, 1991.

[14] J. Demšar. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research*, 7:1–30, 2006.

[15] J. Doran and I. Parberry. Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games*, 2:111–119, 2010.

[16] J. Dormans. Adventures in level design: generating missions and spaces for action adventure games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–8, 2010.

[17] J. Dormans and S. Bakkes. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, pages 216–228, 2011.

[18] A. Doull. The death of the level designer: Procedural content generation in games. Website, 2008. `http://roguelikedeveloper.blogspot.com/2008/01/death-of-level-designer-procedural.html`.

[19] A. Drachen, A. Canossa, and G. N. Yannakakis. Player modeling using self-organization in tomb raider: Underworld. In *IEEE Symposium on Computational Intelligence and Games*, pages 1–8, 2009.

[20] R. O. Duda, D. G. Stork, and P. E. Hart. *Pattern classification*. Wiley, 2000.

[21] M. S. El-Nasr, A. V. Vasilakos, C. Rao, and J. A. Zupko. Dynamic intelligent lighting for directing visual attention in interactive 3-d scenes. *IEEE*

Transactions on Computational Intelligence and AI in Games, 1:145–153, 2009.

[22] M. S. El-Nasr, J. Zupko, and K. Miron. Intelligent lighting for a better gaming experience. In *Proceedings of the Computer Human Interaction*, pages 1140–1141, 2005.

[23] ESRI (Company). Procedural pompeii. Website, 2014. `http://www.esri.com/software/cityengine/industries/procedural-pompeii`.

[24] G. Frasca. *The Video Game Theory Reader*, chapter 10. Simulation versus Narrative: Introduction to Ludology. Routledge, 2003.

[25] GameSpot. Gamespot. Website, 2014. `http://www.gamespot.com`.

[26] J. Gow, R. Baumgarten, P. A. Cairns, S. Colton, and P. Miller. Unsupervised modeling of player style with lda. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:152–166, 2012.

[27] S. Greuter, J. Parker, N. Stewart, and G. Leach. Real-time procedural generation of pseudo-infinite cities. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, 2003.

[28] E. A. A. Gunn, B. G. W. Craenen, and E. Hart. A taxonomy of video games and ai. In *Artificial Intelligence Simulation of Behaviour*, 2009.

[29] J. Han and M. Kamber. *Data Mining: Concepts and Techniques: Concepts and Techniques*. Elsevier Science, 2011.

[30] E. J. Hastings, R. K. Guha, and K. O. Stanley. Evolving content in the galactic arms race video game. In *IEEE Symposium on Computational Intelligence and Games*, pages 241–248, 2009.

[31] E. J. Hastings and K. O. Stanley. Interactive genetic engineering of evolved video game content. In *Proceedings of the FDG Workshop on Procedural Content Generation in Games*, pages 1–8, 2010.

[32] M. Hazewinkel. *Encyclopedia of Mathematics*. Kluwer Academic Publishers, 2002.

[33] R. Hunicke. The case for dynamic difficulty adjustment in games. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, pages 429–433, 2005.

[34] IBM Corporation. Commonsense reasoning with the discrete event calculus reasoner. Website, 2010. `http://decreasoner.sourceforge.net`.

[35] IGN. Ign. Website, 2014. `http://www.ign.com`.

[36] K. D. Jong. *Evolutionary Computation: A Unified Approach*. The MIT Press, 2006.

[37] M. Kerssemakers, J. Tuxen, J. Togelius, and G. N. Yannakakis. A procedural procedural level generator generator. In *IEEE Symposium on Computational Intelligence and Games*, pages 335–341, 2012.

[38] R. Koster. *A theory of fun for game design*. Paraglyph press, 2005.

[39] L. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley-Interscience, 2004.

[40] K. Lang and E. Baum. Query learning can work poorly when a human oracle is used. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pages 335–340, 1992.

[41] A. Laskov. Level generation system for platform games based on a reinforcement learning approach. Master's thesis, Computer Science Edinborough University, 2009.

[42] B. Lee. librf: C++ random forests library. Website, 2014. `http://mtv.ece.ucsb.edu/benlee/librf.html`.

[43] A. Liapis, G. N. Yannakakis, and J. Togelius. Adapting models of visual aesthetics for personalized content creation. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:213–228, 2012.

[44] H. Lin, C. Lin, and R. Weng. A note on platts probabilistic outputs for support vector machines. *Machine Learning*, 68:267–276, 2007.

[45] C. A. Lindley. The gameplay gestalt, narrative, and interactive storytelling. In *Proceedings of the Computer Games and Digital Cultures Conference*, pages 6–8, 2002.

[46] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General game playing: Game description language specification. Technical report, Stanford Logic Group, 2008.

[47] J. Ludwig. Procedural content is hard. Website, 2007. `http://programmerjoe.com/2007/02/11/procedural-content-is-hard/`.

[48] T. Mahlmann, A. Drachen, J. Togelius, A. Canossa, and G. N. Yannakakis. Predicting player behavior in tomb raider: Underworld. In *IEEE Transactions on Computational Intelligence and AI in Games*, pages 178–185, 2010.

[49] T. Mahlmann, J. Togelius, and G. N. Yannakakis. Modelling and evaluation of complex scenarios with the strategy game description language. In *IEEE Transactions on Computational Intelligence and AI in Games*, pages 174–181, 2011.

[50] T. Malone. What makes computer games fun? In *Proceedings of the Joint Conference on Easier and More Productive Use of Computer Systems.*, page 143, 1981.

[51] H. P. Martinez, K. Hullett, and G. N. Yannakakis. Extending neuro-evolutionary preference learning through player modeling. In *IEEE Transactions on Computational Intelligence and AI in Games*, pages 313–320, 2010.

[52] MedCalc. Medcalc. Website, 2014. `http://www.medcalc.org/`.

[53] T. M. Mitchell. *Machine learning*. McGraw-Hill, 2010.

[54] M. J. Nelson and M. Mateas. Recombinable game mechanics for automated design support. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 84–89, 2008.

[55] M. Nitsche, C. Ashmore, W. Hankinson, R. Fitzpatrick, J. Kelly, and K. Margenau. Designing procedural game spaces: A case study. In *FuturePlay*, 2006.

[56] T. Ong, R. Saunders, J. Keyser, and J. J. Leggett. Terrain generation using genetic algorithms. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1463–1470, 2005.

[57] F. Paul. Factbox: The $60 billion global games industry. Website, 2010. `http://www.reuters.com/article/idUSTRE65D4Y020100614`.

[58] C. Pedersen, J. Togelius, and G. N. Yannakakis. Modeling player experience for content creation. *IEEE Transactions on Computational Intelligence and AI in Games*, 2:54–67, 2010.

[59] K. Perlin. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, volume 21, pages 681–682, 2002.

[60] D. Pizzi, M. Cavazza, A. Whittaker, and J.-L. Lugrin. Automatic generation of game level solutions as storyboards. *IEEE Transactions on Computational Intelligence and AI in Games*, 8:96–101, 2008.

[61] D. Plans and D. Morelli. Experience-driven procedural music generation for games. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(3):192–198, 2012.

[62] J. C. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *Advances in Large Margin Classifiers*, pages 61–74, 1999.

[63] L. Plunkett. Only 20% of games make a profit. Website, 2008. `http://kotaku.com/5098356/only-20-of-games-make-a-profit-+-eedar`.

[64] M. Prachyabrued, T. Roden, and R. G. Benton. Procedural generation of stylized 2d maps. In *Advances in Computer Entertainment Technology*, volume 203, pages 147–150, 2007.

[65] V. C. Raykar, S. Yu, L. H. Zhao, G. H. Valadez, C. Florin, L. Bogoni, and L. Moy. Learning from crowds. *The Journal of Machine Learning Research*, 11:1297–1322, 2010.

[66] B. Settles. Active learning literature survey. Technical report, University of Wisconsin, 2009.

[67] N. Shaker, S. Asteriadis, G. Yannakakis, and K. Karpouzis. Fusing visual and behavioral cues for modeling user experience in games. *IEEE Transactions on System Man and Cybernetics*, 2012.

[68] N. Shaker, M. Nicolau, G. N. Yannakakis, J. Togelius, and M. O'Neill. Evolving levels for super mario bros using grammatical evolution. In *IEEE Conference on Computational Intelligence and Games*, pages 304–311, 2012.

[69] N. Shaker, G. N. Yannakakis, and J. Togelius. Towards automatic personalized content generation for platform games. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment*, 2010.

[70] N. Shaker, G. N. Yannakakis, and J. Togelius. Feature analysis for modeling game content quality. In *IEEE Conference on Computational Intelligence and Games*, pages 126–133, 2011.

[71] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52:591–611, 1965.

[72] D. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC Press, 2000.

[73] A. M. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, 2011.

[74] G. Smith and J. Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–4, 2010.

[75] R. Snow, B. O'Connor, D. Jurafsky, and A. Y. Ng. Cheap and fast—but is it good?: Evaluating non-expert annotations for natural language tasks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 254–263, 2008.

[76] N. Sorenson, P. Pasquier, and S. DiPaola. A generic approach to challenge modeling for the procedural creation of video game levels. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):229–244, 2011.

[77] C. R. Strong and M. Mateas. Talking with npcs: Towards dynamic generation of discourse structures. In *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference*, 2008.

[78] P. Sweetser and P. Wyeth. Gameflow: a model for evaluating player enjoyment in games. *Computers in Entertainment*, 3(3):1–24, 2005.

[79] J. Togelius, R. D. Nardi, and S. M. Lucas. Towards automatic personalised content creation for racing games. In *IEEE Symposium on Computational Intelligence and Games*, pages 252–259, 2007.

[80] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbck, G. N. Yannakakis, and C. Grappiolo. Controllable procedural map generation via multiobjective evolution. *Genetic Programming and Evolvable Machines*, 14(2):245–277, 2013.

[81] J. Togelius and J. Schmidhuber. An experiment in automatic game design. In *IEEE Symposium on Computational Intelligence and Games*, pages 111–118, 2008.

[82] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 2011.

[83] S. Tong and D. Koller. Support vector machine active learning with applications to text classification. In *Journal Of Machine Learning Research*, pages 45–66, 2001.

[84] L. van der Maaten and G. E. Hinton. Visualizing high-dimensional data using t-sne. *The Journal of Machine Learning Research*, 9:2579–2605, 2008.

[85] H. Wallop. Video games bigger than film. Website, 2009. `http://www.telegraph.co.uk/technology/video-games/6852383/Video-games-bigger-than-film.html`.

[86] P. Welinder, S. Branson, S. Belongie, and P. Perona. The multidimensional wisdom of crowds. In *Neural Information Processing Systems Conference (NIPS)*, 2010.

[87] P. Wonka, M. Wimmer, F. X. Sillion, and W. Ribarsky. Instant architecture. *ACM Transactions on Graphics*, 22(3):669–677, 2003.

[88] G. N. Yannakakis and J. Hallam. Towards capturing and enhancing entertainment in computer games. In *Advances In Artificial Intelligence*, pages 432–442, 2006.

[89] G. N. Yannakakis, M. Maragoudakis, and J. Hallam. Preference learning for cognitive modeling: A case study on entertainment preferences. *IEEE Transactions on Systems, Man, and Cybernetics*, 39(6):1165–1175, 2009.

[90] G. N. Yannakakis and J. Togelius. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing*, 2(3):147–161, 2011.

[91] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[92] Zillions Development Corporation. Zillions of games. Website, 2010. `http://www.zillions-of-games.com/`.

Appendix A

OBLIGE parameters

Parameter	Effect	Values
strength	Frequency of types of monsters that will appear	weak, medium, tough, crazy
health	Frequency of health packs	none, scarce, less, normal, more, heaps
ammo	Frequency of ammo	none, scarce, less, normal, more, heaps
mons	Frequency of monsters	none, scarce, less, normal, more, heaps, nuts, prog, mixed
powers	Frequency of power-ups	none, less, normal, more, mixed
theme	Theme for the level	original, mixed, jumble, psycho, quake_base, quake_castle
size	Size of the level	tiny, small, regular, large, extreme, prog, mixed
outdoors	Frequency of outdoor areas	none, few, some, heaps, always, mixed
traps	Frequency of traps	none, few, some, heaps, mixed

Continued on next page

Table A.1 – continued from previous page

Parameter	Effect	Values
secrets	Frequency of Secret areas	none, few, some, heaps, mixed
dog	Rottweiler Frequency	none, scarce, less, plenty, more, heaps, insane
fish	Rotfish Frequency	none, scarce, less, plenty, more, heaps, insane
grunt	Grunt Frequency	none, scarce, less, plenty, more, heaps, insane
enforcer	Enforcer Frequency	none, scarce, less, plenty, more, heaps, insane
zombie	Zombie Frequency	none, scarce, less, plenty, more, heaps, insane
knight	Knight Frequency	none, scarce, less, plenty, more, heaps, insane
death_kt	Death Knight Frequency	none, scarce, less, plenty, more, heaps, insane
scrag	Scrag Frequency	none, scarce, less, plenty, more, heaps, insane
tarbaby	Spawn Frequency	none, scarce, less, plenty, more, heaps, insane
ogre	Ogre Frequency	none, scarce, less, plenty, more, heaps, insane
fiend	Fiend Frequency	none, scarce, less, plenty, more, heaps, insane
vore	Vore frequency	none, scarce, less, plenty, more, heaps, insane
shambler	Shambler frequency	none, scarce, less, plenty, more, heaps, insane
ssg	Double Shotgun frequency	none, scarce, less, plenty, more, heaps, loveit

Continued on next page

Table A.1 – continued from previous page

Parameter	Effect	Values
nailgun	Nailgun frequency	none, scarce, less, plenty, more, heaps, loveit
nailgun2	Perforator frequency	none, scarce, less, plenty, more, heaps, loveit
grenade	Grenade Launcher frequency	none, scarce, less, plenty, more, heaps, loveit
rocket	Rocket Launcher frequency	none, scarce, less, plenty, more, heaps, loveit
zapper	Thunderbolt frequency	none, scarce, less, plenty, more, heaps, loveit
bridges	Level structure	mixed, none, few, some, heaps
barrels	Level structure	mixed, none, few, some, heaps
beams	Level structure	mixed, none, few, some, heaps
big_room	Level structure	mixed, none, few, some, heaps
big_juncs	Level structure	mixed, none, few, some, heaps
cages	Level structure	mixed, none, few, some, heaps
caves	Level structure	mixed, none, few, some, heaps
crates	Level structure	mixed, none, few, some, heaps
crossovers	Level structure	mixed, none, few, some, heaps
cycles	Level structure	mixed, none, few, some, heaps
hallways	Level structure	mixed, none, few, some, heaps
lakes	Level structure	mixed, none, few, some, heaps
liquids	Level structure	mixed, none, few, some, heaps
odd_shapes	Level structure	mixed, none, few, some, heaps
pictures	Level structure	mixed, none, few, some, heaps
pillars	Level structure	mixed, none, few, some, heaps
scenics	Level structure	mixed, none, few, some, heaps
symmetry	Level structure	mixed, none, few, some, heaps
teleporters	Level structure	mixed, none, few, some, heaps

Continued on next page

Table A.1 – continued from previous page

Parameter	Effect	Values
windows	Level structure	mixed, none, few, some, heaps
room_shape	Level structure	mixed, none, L, T, U, S, H

Appendix B

Quake Play-log format

Play-log feature	**Description**
Completion	Monsters killed / total monsters
Total ticks	Total number of loops of the game engine
Total distance	Total distance travelled
Total distance/ticks	Average distance travelled per tick
Total Input Events	Total number of input events
Total Input Events/ticks	Average input events per tick
Input Event Speed	How quickly input events occur
Mouse X mean	Average mouse movement in x-axis per window
Mouse Y mean	Average mouse movement in y-axis per window
Mouse X std dev	Standard deviation of mouse movement in x-axis
Mouse Y std dev	Standard deviation of mouse movement in y-axis
Mouse X sum	Total sum of movement of mouse in x-axis
Mouse Y sum	Total sum of movement of mouse in y-axis
Mouse X sum/ticks	Average mouse movement in x-axis per tick
Mouse Y sum/ticks	Average mouse movement in y-axis per tick
Key x Count	For each interesting key (5 total), average movement

Continued on next page

Table B.1 – continued from previous page

Play-log feature	Description
In Damage	Total damaged received
In Health	Total health received
In ammo	Total ammo received
Axe Usage	Frequency of axe use
Shotgun Usage	Frequency of shotgun use
Supershotgun Usage	Frequency of super shotgun use
Nailgun Usage	Frequency of nailgun use
Supernailgun Usage	Frequency of super nailgun use
Grenade Launcher Usage	Frequency of grenade launcher use
Rocket Launcher Usage	Frequency of rocket launcher use
Lightning Gun Usage	Frequency of lightning gun use
In Damage/ticks	Damage received per tick
In Health/ticks	Health received per tick
In ammo/ticks	Ammo received per tick
Axe Usage/ticks	Axe use per tick
Shotgun Usage/ticks	Shotgun use per tick
Supershotgun Usage/ticks	Super shotgun use per tick
Nailgun Usage/ticks	Nailgun use per tick
Supernailgun Usage/ticks	Super nailgun use per tick
Grenade Launcher Usage/ticks	Grenade launcher use per tick
Rocket Launcher Usage/ticks	Rocket launcher use per tick
Lightning Gun Usage/ticks	Lightning gun user per tick
Monster x Damage	For each monster x, total amount of damage received
Monster x Damage/ticks	For each monster x, amount of damage received per tick
Monster x killed	For each monster x, amount killed
Monster x killed/ticks	For each monster x, amount killed per tick
Out Damage	Total damage given
Out Damage/ticks	Damage given per tick
Monsters killed	Total monsters killed

Continued on next page

Table B.1 – continued from previous page

Play-log feature	**Description**
Monsters killed/ticks	Monsters killed per tick
Total monster spawns	Total monsters spawned
Total monster spawns/ticks	Monsters spawned per tick
Player Death count	How many times played died
Player Death count/ticks	Player deaths per tick

Appendix C

LBPCG Software Project

In this appendix we provide an overview the software developed to implement the LBPCG for the target game Quake in Chapter 5. We refer to the software used to drive LBPCG-Quake as the LBPCG Software Project (LSP). The LSP consists of 5 software libraries and 11 executables written in C++, which are illustrated in Figure C.1 and Figure C.2, respectively. Packages coloured red and green indicate software libraries and executables that are part of the LSP, respectively, and packages coloured white are external components. Arrows indicate communication and dependency. Table C.3 gives a summary of the executables. Table C.6, Table C.7, Table C.5, Table C.8 and Table C.4 gives details of the libraries.

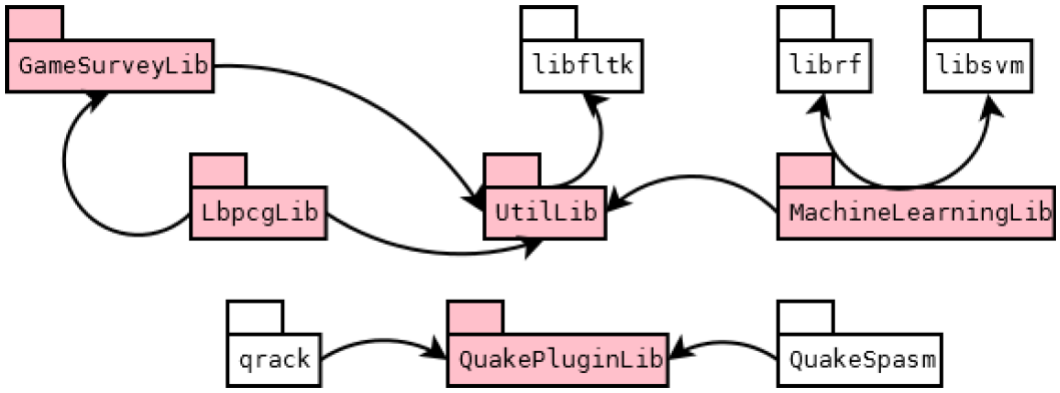

Figure C.1: LSP Libraries.

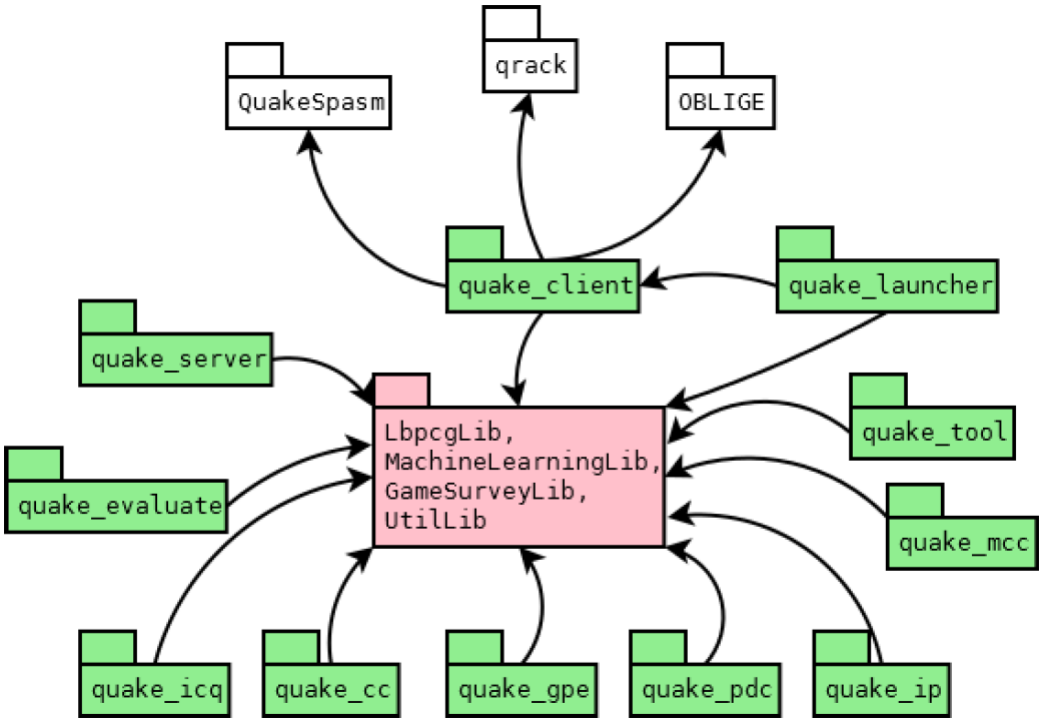

Figure C.2: LSP Executables.

quake_icq	Quake-ICQ model
quake_cc	Quake-CC model
quake_gpe	Quake-GPE model
quake_pdc	Quake-PDC model
quake_ip	Quake-IP model
quake_mcc	Quake-MCC model
quake_server	Server software designed to collect surveys from the public for the games in $\mathcal{G}_{GPE}$. Ensures that each game is distributed evenly amongst participants. Each communication packet is hashed using a secret sequence to prevent malicious interaction.
quake_client	Client software designed to collect surveys from the public. Communicates with quake_server using the encoding format provided by UtilLib. Uses OBLIGE to generate games, then launches either QuakeSpasm or qrack dependant on platform. Collects feedback via dialogs.
quake_launcher	Downloads new versions of quake_client to the participants computer and launches quake_client. This is the executable distributed via the web.
quake_evaluate	Launche quake_ip and collects feedback via dialogs. No network communication. Files needs to be loaded from participants computer afterwards.
quake_tool	Tool for miscellaneous operations such as interacting with data.

Figure C.3: LSP Executables.

Name	UtilLib
Purpose	A helper library providing basic utility functionality to the rest of the LSP.
Functionality	(a) An encoding API to store all classes used throughout the LSP on disk and enable them to be transferred by network connection (b) MD5 support for hashing (c) Latex API for creating .tex files, compiling them using external utilities and generating graphs (d) Compression support for data, including network packets (e) Logging support (f) GUI support for dialogs
Dependencies	(a) Standard Template Library (STL): For lists, vectors, strings and other basic utilities (b) zlib: For compression (c) FLTK: For GUI features (d) Sockets: For network transportation
Important Classes	Serial: Base class inherited by any other that needs encoding support
Important Files	(a) compress.h/compress.cpp: Contains functions classes for compression (b) encoding.h/encoding.cpp: Contains classes and functions for encoding (c) graph.h/graph.cpp: Contains functions for creating LATEX and GNUPLOT graphs (d) latex.h/latex.cpp: Contains functions for generating LATEX files (e) log.h/log.cpp: Containins functions for logging (f) network.h/network.cpp: Contains functions and classes for network support (g) ui.h/ui.cpp: Contains classes and functions for GUI support (h) utils.h/utils.cpp: Contains many miscellaneous helper functions

Figure C.4: LSP: UtilLib

Name	MachineLearningLib
Purpose	To provide machine learning algorithms to the rest of the LSP
Functionality	(a) Random Forests (b) Support Vector Machines (SVM) (c) k-medoids (d) An abstract format for machine learning data (e) t-SNE (f) Bagging support for SVMs and random forests (g) Multi-class random forest support (h) Support for various error formats, including regression error, multi-class error and binary class error
Dependencies	(a) librf: For random forests (b) libsvm: For SVMs (c) UtilLib: For basic functionality
Important Classes	(a) Model::Training: Class that represents machine learning data, which can be fed into the APIs for the machine learning algorithms. Consists of a list of vectors of floating point inputs and outputs. (b) Normaliser: Helps to maintain a normalized version of training data, where new data can be added resulting in re-normalization (c) Svm: Base class for SVM functionality (d) RSvm: SVM-R support (regression) (e) CSvm: Binary classifier SVM support (f) MSvm: Multi-class SVM support (g) RForest: Random forest support (h) MultiClassRandomForest: Multi-class random forest support, achieved by maintaining an ensemble of random forests and checking their probabilistic outputs (i) RegressionError: Class for reporting regression error (j) ClassError: Class for reporting binary classification error (k) MultiClassError: Class for reporting multi-classification error and confusion matrices

Figure C.5: LSP: MachineLearningLib

Name	GameSurveyLib
Purpose	To provide an abstract representation of games and feedback surveys
Functionality	(a) Specifying content features (b) Randomizing game description vectors (c) Storing feedback from developers and the public (d) Representing play-logs
Dependencies	(a) UtilLib: For basic functionality (b) Machine-LearningLib: To enable surveys, games and play-logs to be converted into a generic machine learning format
Important Classes	(a) EventLog: Basic representation of an event-log generated by a player (b) Game: Representation of a game, stores a Parameters object (c) GameClusters: A class used to hold clusters of games after a clustering algorithm has been applied to the content space (c) Games: A list of Game objects with extra functionality to convert them into various formats suitable for learning (d) Parameters: A class representing the various parameters that can be set for a game e.g. OBLIGE parameters (e) PlayLog: Generated from an EventLog object, can be converted into a suitable format for machine learning (f) Survey: Generic class representing feedback from either a developer or member of the public. Can store multiple feedback entries in one record e.g. Difficulty, Acceptability, Fun. (g) Surveys: Storage class for Survey objects. Has functionality, for example, for converting all the surveys into a format for machine learning

Figure C.6: LSP: GameSurveyLib

Name	LbpcgLib
Purpose	Implements all models in the LBPCG-Quake framework
Functionality	(a) Quake-ICQ model (b) Quake-CC model (c) Quake-GPE model (d) Quake-PDC model (e) Quake-IP model (f) Quake-MCC model
Dependencies	(a) Armadillo maths library for manipulation of vectors in implementation of Crowd-EM algorithm (b) UtilLib: For basic functionality (c) GameSurveyLib: For manipulation of surveys provided by developers and members of the public (d) MachineLearningLib: For learning algorithms
Important Classes	(a)-(e) IcqModel/CCModel/GpeModel/PdcModel/IpModel: Implementations of learning and classification functionality for each LBPCG-Quake model (f) SvmCrowdEM: Implementation of Crowd-EM

Figure C.7: LSP: LbpcgLib

Name	QuakePluginLib
Purpose	A plugin library for qrack and QuakeSpasm (Quake implementations) that allow them to communicate with the other LBPCG libraries. Written to be C compatible.
Functionality	(a) Event creation and saving
Dependencies	None
Important Files	quakeplugin.h: Code that can be included in Quake for generating events

Figure C.8: LSP: QuakePluginLib

Appendix D

Play-log feature rank table

Rank	Feature	Score
0	Completion (Monsters Killed/Total Monsters)	6.36389303
1	Player Death count/ticks	5.66099739
2	Grunt killed	2.22189641
3	Total Input Events	2.14130044
4	Fiend spawns	1.08697462
5	Monsters killed	0.979292214
6	Total distance	0.794655621
7	Rottweiler killed	0.646714807
8	In Health/ticks	0.570297718
9	In Health	0.501151502
10	Total ticks	0.498960942
11	Rottweiler spawns	0.434645385
12	Grunt killed/ticks	0.425062567
13	Rottweiler spawns/ticks	0.401339918
14	Out Damage/ticks	0.351460725
15	Total Input Events/ticks	0.329764009
16	Ogre spawns	0.265760571
17	Total monster spawns	0.19812569
18	Out Damage	0.171154276
19	Rottweiler killed/ticks	0.160433486
20	In Damage	0.151273817
21	Fiend spawns/ticks	0.118670218

22	Grunt spawns	0.0514944792
23	Total monster spawns/ticks	0.0418163836
24	Damage to Knight	0.037143372
25	Input Event Speed	0.019054858
26	Damage to Scrag	0.00519630685
27	Grunt spawns/ticks	-0.000385532388
28	Game Category	-0.033378467
29	In ammo	-0.0413112752
30	In Damage/ticks	-0.0445075855
31	Scrag killed/ticks	-0.0499573499
32	Supernailgun Usage	-0.0721520185
33	Shotgun Usage/ticks	-0.0887285322
34	Knight killed/ticks	-0.128696725
35	Shotgun Usage	-0.136336938
36	Damage to Scrag/ticks	-0.139845476
37	Axe Usage	-0.140687734
38	Damage to Shambler	-0.15862149
39	Damage to Fiend/ticks	-0.163362622
40	Damage to Grunt	-0.166941941
41	Monsters killed/ticks	-0.168565825
42	Damage to Grunt/ticks	-0.174115255
43	Knight killed	-0.188326776
44	Lightning Gun Usage/ticks	-0.191952005
45	Scrag spawns	-0.194354117
46	Damage to Knight/ticks	-0.201093554
47	Ogre spawns/ticks	-0.20419924
48	Supershotgun Usage/ticks	-0.210053176
49	Damage to Rottweiler/ticks	-0.213232994
50	Fiend killed/ticks	-0.224306539
51	Shambler spawns/ticks	-0.226584211
52	Nailgun Usage/ticks	-0.230327412
53	Shambler killed	-0.230972141
54	Rocket Launcher Usage/ticks	-0.236424536

55	Scrag spawns/ticks	-0.238995433
56	Lightning Gun Usage	-0.242695928
57	Rocket Launcher Usage	-0.244883463
58	Scrag killed	-0.245931849
59	Damage to Zombie/ticks	-0.246905729
60	Player Death count	-0.247356415
61	Supernailgun Usage/ticks	-0.247871488
62	Knight spawns/ticks	-0.248428971
63	Damage to Fiend	-0.253079534
64	Axe Usage/ticks	-0.266998738
65	Supershotgun Usage	-0.270154864
66	Nailgun Usage	-0.274748832
67	Ogre killed/ticks	-0.274985939
68	Damage to Ogre/ticks	-0.278800994
69	Zombie spawns	-0.282825083
70	Zombie spawns/ticks	-0.283434957
71	Damage to Shambler/ticks	-0.291206509
72	Damage to Enforcer	-0.2993626
73	Damage to Enforcer/ticks	-0.2993626
74	Damage to Fish	-0.2993626
75	Damage to Fish/ticks	-0.2993626
76	Damage to Vore	-0.2993626
77	Damage to Vore/ticks	-0.2993626
78	DeathKnight killed	-0.2993626
79	DeathKnight killed/ticks	-0.2993626
80	DeathKnight spawns	-0.2993626
81	Enforcer killed	-0.2993626
82	Enforcer killed/ticks	-0.2993626
83	Enforcer spawns	-0.2993626
84	Enforcer spawns/ticks	-0.2993626
85	Fish killed	-0.2993626
86	Fish killed/ticks	-0.2993626
87	Fish spawns	-0.2993626

88	Fish spawns/ticks	-0.2993626
89	Key ' ' Count	-0.2993626
90	Key 'a' Count	-0.2993626
91	Key 'd' Count	-0.2993626
92	Key 's' Count	-0.2993626
93	DeathKnight spawns/ticks	-0.2993626
94	Mouse X mean	-0.2993626
95	Mouse X std dev	-0.2993626
96	Key 'w' Count	-0.2993626
97	Mouse X sum	-0.2993626
98	Mouse X sum/ticks	-0.2993626
99	Mouse Y mean	-0.2993626
100	Mouse Y std dev	-0.2993626
101	Mouse Y sum	-0.2993626
102	Mouse Y sum/ticks	-0.2993626
103	Vore killed	-0.2993626
104	Vore killed/ticks	-0.2993626
105	Vore spawns	-0.2993626
106	Vore spawns/ticks	-0.2993626
107	Damage to DeathKnight	-0.2993626
108	Damage to DeathKnight/ticks	-0.2993626
109	Damage to Zombie	-0.300134003
110	In ammo/ticks	-0.300640106
111	Grenade Launcher Usage	-0.30311954
112	Ogre killed	-0.30668819
113	Damage to Ogre	-0.311606795
114	Knight spawns	-0.311769724
115	Damage to Rottweiler	-0.31335175
116	Shambler spawns	-0.313557893
117	Shambler killed/ticks	-0.315817714
118	Fiend killed	-0.321126133
119	Zombie killed	-0.333991498
120	Grenade Launcher Usage/ticks	-0.338178664

121	Zombie killed/ticks	-0.429573596
122	Total distance/ticks	-0.653832018

Table D.1: PDC Model Feature importance

Appendix E

IP Evaluation Data

E.1 Player 8

1. Q: What kind of gamer would you describe yourself as?
2. A: I play games regularly
3. Q: How would you rate your skill level at action video games?
4. A: Good

E.1.1 IP State Transitions (Player 8)

Iteration	State	Target Category	PDC Output	PDC Conf.	Player Binary Fun
0	CATEGORIZE	0	0	0.620913029	No
1	CATEGORIZE	1	0	0.368943989	No
2	CATEGORIZE	1	0	0.551352024	No
3	CATEGORIZE	2	1	0.204161003	Yes
4	CATEGORIZE	2	1	0.768211007	Yes
5	CATEGORIZE	2	1	0.731657028	Yes
6	PRODUCE	2	1	0.736346006	Yes
7	PRODUCE	2	1	0.340023994	Yes
8	PRODUCE	2	0	0.038511999	No
9	PRODUCE	2	1	0.187472999	No

10	PRODUCE	2	1	0.366742015	Yes
11	PRODUCE	2	0	0.385650992	No
12	PRODUCE	2	0	0.344686002	Yes
13	PRODUCE	2	0	0.761400998	No
14	CATEGORIZE	3	1	0.937162995	Yes
15	CATEGORIZE	3	1	0.635923982	Yes
16	PRODUCE	3	0	0.215125993	Yes
17	PRODUCE	3	0	0.626323998	No
18	CATEGORIZE	4	0	0.884727001	No
19	CATEGORIZE	0	0	0.443284005	No

Table E.1: IP State Transitions (Player 8)

E.2 Player 10

1. Q: What kind of gamer would you describe yourself as?
2. A: I play games regularly
3. Q: How would you rate your skill level at action video games?
4. A: Average

E.2.1 IP State Transitions (Player 10)

Iteration	**State**	**Target Category**	**PDC Output**	**PDC Conf.**	**Player Binary Fun**
0	CATEGORIZE	0	1	0.390008003	No
1	CATEGORIZE	0	1	0.569269001	No
2	CATEGORIZE	0	1	0.459095001	No
3	CATEGORIZE	0	1	0.441112012	No
4	CATEGORIZE	1	1	0.165254995	Yes
5	CATEGORIZE	1	1	0.575468004	Yes
6	CATEGORIZE	1	1	0.411365002	Yes
7	CATEGORIZE	1	1	0.545356989	Yes

8	CATEGORIZE	1	1	0.738659024	Yes
9	PRODUCE	1	1	0.281372994	Yes
10	PRODUCE	1	1	0.536148012	No
11	PRODUCE	1	1	0.196021006	No
12	PRODUCE	1	0	0.639014006	No
13	CATEGORIZE	2	1	0.345277011	No
14	CATEGORIZE	2	1	0.266259998	Yes
15	CATEGORIZE	3	0	0.0980189964	No
16	CATEGORIZE	3	1	0.538091004	Yes
17	CATEGORIZE	3	1	0.892060995	Yes
18	PRODUCE	3	0	0.750014007	No
19	CATEGORIZE	4	0	0.646501005	No

Table E.2: IP State Transitions (Player 10)

E.3 Player 11

1. Q: What kind of gamer would you describe yourself as?
2. A: I play games regularly
3. Q: How would you rate your skill level at action video games?
4. A: Very good

E.3.1 IP State Transitions (Player 11)

Iteration	**State**	**Target Category**	**PDC Output**	**PDC Conf.**	**Player Binary Fun**
0	CATEGORIZE	0	0	0.438519001	No
1	CATEGORIZE	0	0	0.298756003	No
2	CATEGORIZE	1	1	0.165935993	Yes
3	CATEGORIZE	1	1	0.369623989	No
4	CATEGORIZE	2	1	0.229450002	Yes
5	CATEGORIZE	2	0	0.595052004	No

6	CATEGORIZE	3	1	0.0842669979	Yes
7	CATEGORIZE	3	1	0.687744021	Yes
8	CATEGORIZE	3	1	0.517071009	No
9	PRODUCE	3	1	0.813212991	Yes
10	PRODUCE	3	0	0.392215997	Yes
11	PRODUCE	3	1	0.969971001	Yes
12	PRODUCE	3	0	0.635760009	No
13	CATEGORIZE	4	0	0.568572998	Yes
14	CATEGORIZE	0	1	0.638808012	No
15	CATEGORIZE	0	0	0.301896006	Yes
16	CATEGORIZE	0	0	0.0254280008	Yes
17	CATEGORIZE	1	1	0.278035015	No
18	CATEGORIZE	1	0	0.136187002	Yes
19	CATEGORIZE	2	1	0.907438993	Yes

Table E.3: IP State Transitions (Player 11)

E.4 Player 13

1. Q: What kind of gamer would you describe yourself as?
2. A: I play games sometimes
3. Q: How would you rate your skill level at action video games?
4. A: Average

E.4.1 IP State Transitions (Player 13)

Iteration	**State**	**Target Category**	**PDC Output**	**PDC Conf.**	**Player Binary Fun**
0	CATEGORIZE	0	1	0.0681329966	No
1	CATEGORIZE	0	1	0.119361997	No
2	CATEGORIZE	1	0	0.392589003	Yes
3	CATEGORIZE	1	1	0.137170002	Yes

4	CATEGORIZE	2	0	0.800689995	Yes
5	CATEGORIZE	3	0	0.0857639983	Yes
6	CATEGORIZE	3	0	0.678871989	Yes
7	CATEGORIZE	4	0	0.843699992	Yes
8	CATEGORIZE	0	1	0.544448972	Yes
9	CATEGORIZE	0	1	0.502516985	Yes
10	PRODUCE	0	0	0.660826027	Yes
11	CATEGORIZE	1	0	0.296741009	Yes
12	CATEGORIZE	1	1	0.262616009	Yes
13	CATEGORIZE	2	1	0.451296985	Yes
14	CATEGORIZE	2	1	0.88695699	Yes
15	CATEGORIZE	2	0	0.0768970028	No
16	CATEGORIZE	2	0	0.449748009	No
17	CATEGORIZE	3	1	0.727337003	Yes
18	CATEGORIZE	3	0	0.357080996	Yes
19	CATEGORIZE	3	1	0.767158985	Yes

Table E.4: IP State Transitions (Player 13)

E.5 Player 15

1. Q: What kind of gamer would you describe yourself as?
2. A: I play games regularly
3. Q: How would you rate your skill level at action video games?
4. A: Good

E.5.1 IP State Transitions (Player 15)

Iteration	**State**	**Target Category**	**PDC Output**	**PDC Conf.**	**Player Binary Fun**
0	CATEGORIZE	0	1	0.112305999	Yes
1	CATEGORIZE	0	0	0.0104360003	Yes

2	CATEGORIZE	1	1	0.507088006	Yes
3	CATEGORIZE	1	1	0.65684998	Yes
4	PRODUCE	1	0	0.277242988	No
5	PRODUCE	1	1	0.255997002	Yes
6	PRODUCE	1	1	0.204125002	Yes
7	PRODUCE	1	0	0.307186007	No
8	PRODUCE	1	1	0.210003003	No
9	PRODUCE	1	1	0.209531993	No
10	PRODUCE	1	0	0.63637203	No
11	CATEGORIZE	2	0	0.788820982	No
12	CATEGORIZE	3	1	0.155102	No
13	CATEGORIZE	3	0	0.0833500028	Yes
14	CATEGORIZE	4	0	0.636707008	Yes
15	CATEGORIZE	0	1	0.152757004	Yes
16	CATEGORIZE	0	0	0.419456005	Yes
17	CATEGORIZE	1	1	0.283681005	Yes
18	CATEGORIZE	1	0	0.477816999	Yes
19	CATEGORIZE	2	1	0.271456987	No

Table E.5: IP State Transitions (Player 15)

E.6 Player 16

1. Q: What kind of gamer would you describe yourself as?
2. A: I don't play games much - if at all
3. Q: How would you rate your skill level at action video games?
4. A: Average

E.6.1 IP State Transitions (Player 16)

Iteration	**State**	**Target Category**	**PDC Output**	**PDC Conf.**	**Player Binary Fun**

0	CATEGORIZE	0	1	0.428667992	No
1	CATEGORIZE	0	1	0.226069003	Yes
2	CATEGORIZE	1	1	0.817843974	Yes
3	CATEGORIZE	1	1	0.594521999	No
4	PRODUCE	1	0	0.675516009	Yes
5	CATEGORIZE	2	0	0.649600983	No
6	CATEGORIZE	3	1	0.0805289969	Yes
7	CATEGORIZE	3	1	0.107128002	Yes
8	CATEGORIZE	4	0	0.702085018	No
9	CATEGORIZE	0	1	0.537397027	No
10	CATEGORIZE	0	1	0.513351023	No
11	PRODUCE	0	0	0.0531920008	No
12	PRODUCE	0	1	0.37501201	Yes
13	PRODUCE	0	1	0.256669998	Yes
14	PRODUCE	0	1	0.109323002	Yes
15	PRODUCE	0	1	0.312775999	Yes
16	PRODUCE	0	0	0.546459019	No
17	CATEGORIZE	1	0	0.00957500003	Yes
18	CATEGORIZE	1	1	0.714644015	Yes
19	CATEGORIZE	1	1	0.36313799	Yes

Table E.6: IP State Transitions (Player 16)

E.7 Player 18

1. Q: What kind of gamer would you describe yourself as?
2. A: I play games regularly
3. Q: How would you rate your skill level at action video games?
4. A: Average

E.7.1 IP State Transitions (Player 18)

Iteration	State	Target Category	PDC Output	PDC Conf.	Player Binary Fun
0	CATEGORIZE	0	1	0.406421989	Yes
1	CATEGORIZE	0	1	0.373145014	Yes
2	CATEGORIZE	1	1	0.767575026	Yes
3	CATEGORIZE	1	1	0.508132994	No
4	PRODUCE	1	0	0.616512001	No
5	CATEGORIZE	2	0	0.576898992	No
6	CATEGORIZE	3	0	0.149935007	No
7	CATEGORIZE	3	1	0.621451974	No
8	CATEGORIZE	3	1	0.692376018	Yes
9	PRODUCE	3	0	0.449270993	No
10	PRODUCE	3	0	0.638375998	Yes
11	CATEGORIZE	4	0	0.708742976	No
12	CATEGORIZE	0	1	0.651836991	Yes
13	CATEGORIZE	0	1	0.61076498	Yes
14	PRODUCE	0	0	0.672994018	No
15	CATEGORIZE	1	1	0.454356998	Yes
16	CATEGORIZE	1	0	0.166938007	No
17	CATEGORIZE	2	1	0.580411971	Yes
18	CATEGORIZE	2	1	0.891815007	Yes
19	PRODUCE	2	0	0.636524975	No

Table E.7: IP State Transitions (Player 18)

E.8 Player 19

1. Q: What kind of gamer would you describe yourself as?

2. A: I play games regularly

3. Q: How would you rate your skill level at action video games?

4. A: Good

E.8.1 IP State Transitions (Player 19)

Iteration	State	Target Category	PDC Output	PDC Conf.	Player Binary Fun
0	CATEGORIZE	0	1	0.437638015	Yes
1	CATEGORIZE	0	1	0.858783007	Yes
2	CATEGORIZE	0	1	0.590866983	Yes
3	PRODUCE	0	1	0.646411002	Yes
4	PRODUCE	0	1	0.323325992	Yes
5	PRODUCE	0	1	0.683916986	Yes
6	PRODUCE	0	1	0.174156994	Yes
7	PRODUCE	0	1	0.681186974	Yes
8	PRODUCE	0	1	0.636089981	Yes
9	PRODUCE	0	1	0.696941972	Yes
10	PRODUCE	0	1	0.459930986	Yes
11	PRODUCE	0	0	0.31931901	Yes
12	PRODUCE	0	0	0.229060993	Yes
13	PRODUCE	0	0	0.381471008	Yes
14	PRODUCE	0	0	0.525040984	Yes
15	CATEGORIZE	1	1	0.54586798	Yes
16	CATEGORIZE	1	1	0.592835009	Yes
17	PRODUCE	1	0	0.254368991	Yes
18	PRODUCE	1	1	0.458236992	Yes
19	PRODUCE	1	1	0.62291199	Yes

Table E.8: IP State Transitions (Player 19)

E.9 Player 20

1. Q: What kind of gamer would you describe yourself as?
2. A: I'm a hardcore gamer
3. Q: How would you rate your skill level at action video games?
4. A: Good

E.9.1 IP State Transitions (Player 20)

Iteration	State	Target Category	PDC Output	PDC Conf.	Player Binary Fun
0	CATEGORIZE	0	1	0.449721992	Yes
1	CATEGORIZE	0	0	0.0986630023	Yes
2	CATEGORIZE	1	0	0.0745590031	Yes
3	CATEGORIZE	1	1	0.397621989	Yes
4	CATEGORIZE	2	0	0.434058994	Yes
5	CATEGORIZE	2	1	0.73664403	Yes
6	CATEGORIZE	2	1	0.892782986	No
7	PRODUCE	2	1	0.160027996	Yes
8	PRODUCE	2	1	0.329775989	Yes
9	PRODUCE	2	1	0.448424995	Yes
10	PRODUCE	2	1	0.748813987	Yes
11	PRODUCE	2	0	0.407593995	Yes
12	PRODUCE	2	1	0.768700004	Yes
13	PRODUCE	2	1	0.343199015	Yes
14	PRODUCE	2	0	0.431199998	Yes
15	PRODUCE	2	0	0.312193006	No
16	PRODUCE	2	0	0.0103900004	Yes
17	PRODUCE	2	0	0.358013988	Yes
18	PRODUCE	2	1	0.429695994	Yes
19	PRODUCE	2	0	0.577466011	Yes

Table E.9: IP State Transitions (Player 20)

E.10 Player 21

1. Q: What kind of gamer would you describe yourself as?
2. A: I don't play games much - if at all
3. Q: How would you rate your skill level at action video games?
4. A: Average

E.10.1 IP State Transitions (Player 21)

Iteration	State	Target Category	PDC Output	PDC Conf.	Player Binary Fun
0	CATEGORIZE	0	1	0.736769021	Yes
1	CATEGORIZE	0	1	0.339435995	Yes
2	CATEGORIZE	0	1	0.333685011	Yes
3	CATEGORIZE	1	0	0.00334000005	Yes
4	CATEGORIZE	1	1	0.125159994	No
5	CATEGORIZE	2	0	0.085326001	Yes
6	CATEGORIZE	2	0	0.327980995	Yes
7	CATEGORIZE	3	1	0.0698999986	Yes
8	CATEGORIZE	3	0	0.36786899	Yes
9	CATEGORIZE	4	0	0.314502001	Yes
10	CATEGORIZE	4	0	0.526748002	Yes
11	CATEGORIZE	0	1	0.718445003	Yes
12	CATEGORIZE	0	1	0.507511973	Yes
13	PRODUCE	0	0	0.485612988	Yes
14	PRODUCE	0	0	0.230024993	Yes
15	PRODUCE	0	1	0.332740009	Yes
16	PRODUCE	0	1	0.680931985	Yes
17	PRODUCE	0	0	0.361384004	No
18	PRODUCE	0	1	0.355194986	Yes
19	PRODUCE	0	1	0.366937995	Yes

Table E.10: IP State Transitions (Player 21)