University of Derby
Department of Electronics, Computing &
Mathematics

A project completed as part of the requirements
for BSc (Hons) Computer Games Programming

entitled

AI-Evaluated Procedural Content Generation for Video Game Level Design

Daniel Millward

April 2017

University of Derby
Department of Electronics, Computing & Mathematics
BSc (Hons) Computer Games Programming

AI-Evaluated Procedural Content Generation for Video Game Level Design

by Daniel Millward

Abstract

Procedural Content Generation (PCG) has long been used in video games to produce content that would otherwise be restricted by time, cost, or memory limitations. A relatively new area of research within the field is that of Search-Based PCG, which uses a process inspired by biological evolution to evolve content over iterative generations. Expanding on this work, we present a prototype of AI-Evaluated PCG, which aims to generate content with as few human-authored rules as possible, by using an AI-driven Bot for content evaluation. Our working prototype focuses on level generation for a 2D platforming game similar to Nintendo's *Super Mario Bros.* We develop a bot which, through the use of an artificial neural network, learns from player behaviour across a number of training sessions, and replicates that behaviour to test-run and evaluate generated levels. Level evaluation is used to score potential level solutions, and a genetic algorithm employed to evolve our level content. Ultimately our prototype is able to successfully generate complete and playable levels, with room for improvement. We detail a list of recommendations for further research.

Contents

Chapter 1

Introduction

1.1 Rationale

Procedural Content Generation (PCG), defined as the creation of content through algorithmic means (Togelius et al., 2011), has been a popular technique used in video games for many years. PCG solutions can be used to extended player replayability, conserved memory, and produce large quantities of content with limited time and resources. These benefits alone make it a crucial area of study, particularly as video games continue to increase in scope and production cost. While PCG techniques provide a method of generating vast quantities of content beyond that which could be human-authored, a great deal of development time is usually still spent defining the rules and constraints that govern the content a generator can produce, and testing the generated content to refine those rules. For this reason, any new technique which could alleviate some of the work placed on designers and content creators would be hugely beneficial to the industry (Yannakakis and Togelius, 2011).

A fairly new and active area of research in the field of PCG is the Search-Based PCG paradigm, which focuses on generating content through the use of *evolutionary computation* (Togelius et al., 2011). These techniques use an algorithm inspired by biological evolution, where many potential content solutions are generated, evaluated, and then the desirable features combined through a breeding process, until an optimal solution is found (Russell and Norvig, 2009). Crucially, the evaluation stage of this paradigm relieves a great deal of testing work, assuming that the function used for evaluation accurately describes complete and optimal content. Evaluating content algorithmically can prove a difficult challenge however, one that already has many studies dedicated to it (Togelius and Shaker, 2016).

We propose a new technique, which aims to reduce the amount of additional work and human designed constraints required to a minimum. Our technique builds on the Search-Based PCG paradigm and introduces the concept of player-trained artificially intelligent (AI) Bots for content evaluation. Like many video game PCG research papers (Pedersen et al., 2010; Jennings-Teats et al., 2010; Kerssemakers et al., 2012), we will focus our study on the domain of 2D platformer games, similar to Nintendo's *Super Mario Bros.* (Nintendo, 1985). This domain not only has a large field of work to drawn comparisons to, but represents a form of content that must always be complete and playable. Should our hypothesis prove successful, we hope that the design and development process documented in this paper can provide the base work for a technique that has huge potential for the video game industry.

1.2 Aims and Objectives

To manage our development phase and ensure that our technique is properly evaluated, we have split our project into an aim and multiple objectives. Each objective must be met in order to conclude if our aim has successfully been achieved. The aim of our project is:

To develop a Search-Based Procedural Level Generator and prototype game, that uses player-trained AI Bots to evaluate generated content with minimal human-authored constraints.

To achieve this aim, the following objectives will be met:

1. A thorough literature review will be conducted into Search-Based Procedural Content Generation in general, prior academic attempts to generate game content through machine learning, and techniques used to build Artificial Neural Networks and Genetic Algorithms.

2. A simple prototype video game will be developed with several human-authored levels, and the ability to load in new levels, which will be used as the basis of our content generation experiment.

3. An Artificially Intelligent Bot will be developed, which learns entirely from the actions of a player during a play-through of the previously mentioned game. The Bot will be able to replicate player behaviour in order to traverse the levels of our game.

4. A Procedural Content Generator will be developed which uses our Bot for generated level evaluation, and evolutionary learning to output new levels for the game.

5. The finished product and a sample of generated levels will be evaluated to determine the effectiveness of the generator. A study will be conducted upon several human subjects, who will be asked to play our game and give feedback on each level via a questionnaire.

6. The development process will be evaluated to assess our design and implementation decisions, and the outcome of our hypothesis.

1.3 Hypothesis

Based on the aims and objectives provided in this section, the hypothesis of this paper is as follows:

An Artificial Neural Network controlled Bot can be trained to emulate a player's in-game behaviour, and can then be used to provide a fitness evaluation of levels generated by a Search-Based Procedural Content Generator, in order to produce level designs for a game that look and play like human-authored content with limited designed constraints.

Chapter 2

Literature Review

2.1 Introduction

2.1.1 Section Summary

In this chapter, we will assess the value of using Genetic Algorithms and Search-Based Procedural Content Generation for the creation of level content in Video Games. In order to do so, we will first review exactly what is meant by Procedural Content Generation, and how it has been used in Video Games dating back to *Rogue* (Epyx, Inc., 1980). We will evaluate prior and current research into Search-Based and Experience-Based techniques, and come to a conclusion as to how the combination of Artificial Neural Networks and Genetic Algorithms could be used to generate level data for a game.

2.1.2 Section Structure

- **2.2 Procedural Content Generation in Games:** We begin by exploring the background of Procedural Content Generation and its application in video games, as well as outlining a number of key terms and concepts, and what challenges they present.
- **2.3 The Content Creation Bottleneck:** This section introduces the concept of the "content creation bottleneck", the barrier to artistic and technological development caused by the rising time and cost requirements of video game production. We provide details of the increasing trend of "games as a service", and discuss a theoretical solution.
- **2.4 Genetic Algorithms:** In this section we temporarily leave the domain of video games and content generation to discuss *genetic algorithms*, and detail their uses and common implementation.
- **2.5 Artificial Neural Networks:** Continuing our discussion on artificial learning techniques, we outline the development and implementation of *artifical neural networks*, and how they can be trained through back-propagation.
- **2.6 Search-Based Procedural Content Generation:** Returning to video games and content generation, we discuss the more recent paradigm of *search-based* procedural content generation, and explore prior research in the field. We particularly focus on content evaluation functions using artificially intelligent bots.

- **2.7 Evolving AI Agents to Play Games:** Having discussed the use of AI bots to evaluate procedurally generated content, we explore prior research in the field of evolving AI bot behaviour, and discuss how an evolved bot could be beneficial to content evaluation.
- **2.8 Evaluating the Project's Worth:** Once we have presented our research within the fields of procedural content generation and evolutionary bot behaviour, we evaluate our argument for the project's worth to academia and the video games industry.
- **2.9 Conclusion:** Finally, we summarise the results of our literature review, and present our conclusions on the worth of the project.

2.2 Procedural Content Generation in Games

In their 2011 survey, Togelius et al. define Procedural Content Generation (PCG) as "creating game content automatically, through algorithmic means" (Togelius et al., 2011), where "game content" can refer to anything that would otherwise be the responsibility of a human designer; from textures and models, to plot and level design (Smith, 2015). Generating game content algorithmically has many potential benefits, with the most popular applications being to provide extended player replayability, conserve memory, and dynamically tailor content to the player. Some of the earliest examples of PCG in games can be found in 1980's *Rogue* (Epyx, Inc., 1980) and 1984's *Elite* (Acornsoft, 1984). Both titles use PCG to generate their world layout, and, while this has the benefit of providing the player with a new experience each time they play, their use of PCG initially arose from technical limitation: the need to conserve space on low memory machines (Smith, 2015). In this instance, PCG can be thought of as a data compression method, since the algorithm used to generate each game's content was a fraction of the size of the generated content itself.

As advancements in technology over the last few decades have resulted in an increase in processing power and memory storage, PCG has been employed less due solely to technical limitations, and more as a tool to aid developer or design limitations. Indie developed titles such as *Minecraft* (Mojang AB, 2011) and *Spelunky* (Mossmouth, 2008) use PCG to produce large quantities of playable content with a small development team, where crafting the same content by hand would be out of scope and budget. 2016's *No Man's Sky* (Hello Games, 2016) attempts to portray a near infinite number of planets, each with their own unique ecosystems, a design feat which would be impossible to achieve with hand-tailored content, but made possible through the use of PCG. Before exploring common PCG techniques and how they have been used in prior work, it is important to outline several key concepts which apply to all PCG algorithms. These concepts have to be considered when developing a PCG system, as they present different challenges depending on the system's application.

2.2.1 Online and Offline Generation

First is the concept of *online* verses *offline* PCG systems. In an *online* system, the Procedural Content Generator is executed at application runtime. In an *offline* system, the generator is executed during development time, and the content generated is then

saved and packaged with the game (Togelius et al., 2011). Offline generation is therefore more limited in terms of the content that it can create; since all content is generated and saved ahead of time, there is no reduction in memory cost, and the content generated is the same for all users. However, while online generation can potentially provide unlimited new content, it has two major drawbacks. The algorithm must be fast enough to execute at runtime, and the content it produces must be *reliably correct*, a term which means that the content is not intractable or unplayable (Togelius et al., 2011). Content that takes minutes to load, or content that makes it impossible for the player to progress is unacceptable.

2.2.2 Necessary and Optional Content

Next is the distinction between *necessary* and *optional* content. *Necessary* content is seen as all content which is required to complete the game, such as the structure of levels and the main plot. This is the content which should always be *reliably correct*. *Optional* content is that which the player does not rely on, and therefore can behave unexpectedly (Togelius et al., 2016). An example of *optional* content generation can be seen in Borderlands 2 (2K Games, 2012), where the weapons that the player acquires have procedurally generated properties. This combination of properties can sometimes be detrimental to the weapon's effectiveness, producing weapons that are undesirable to the player. However, since the game produces a vast number of weapons, those which are undesirable never hinder the player's progress.

2.2.3 Generic, Interactive and Experience-Based Generation

Generic content generation refers to all techniques which produce content without participation or interaction from a human participant, or other previously recorded dataset (Togelius et al., 2016). Content generated by a *generic* generator is governed entirely by the constraints and design of the original algorithm, and is not customisable by the player. In contrast, in *interactive* generation systems, the player or user of the system can directly interact with the generator, and influence the content that it creates (Smith, 2015).

This distinction can be split further into *parametrized* and *experience-based* generation. In *parametrized* systems, the user is able to manually set parameters to control content generation (Smith, 2015), whereas in *experience-driven* systems, the generator collects and analyses data based on the user's behaviour, and dynamically adjusts its output to produce more desirable content (Yannakakis and Togelius, 2011). An example of *experience-driven* PCG can be found in *Galactic Arms Race* (Evolutionary Games, 2012), which produces new, tailored weapons for each player in real time, with properties based on weapons that the player has preferred to use in the past (Hastings et al., 2009). This system is discussed in further detail in section 2.6.3.2.

2.2.4 Constructive and Generate-and-Test Patterns

Finally, PCG techniques can be split into two groups, those that are *constructive*, and those that incorporate a *generate-and-test* mechanism. A *constructive* algorithm is one that generates content in a single pass, and the results are output immediately without validation. Because of the lack of a testing phase, content created through *constructive*

techniques must either be optional, or the algorithm must be guaranteed to never produce incomplete content (Togelius et al., 2011). Since *constructive* techniques are based on a single pass of their generation algorithm, they usually have the benefit of predictable runtimes and results.

Generate-and-test patterns, on the other hand, perform in two stages: a generate or construct stage, and a test or evaluation stage. Content is generated in the construct stage based on a given ruleset. The content is then tested in the evaluation stage, according to a given criteria. If the evaluation fails, then some or all of the previously generated content is discarded, and the pattern returns to the construct stage. The pattern loops until the evaluation stage passes, and the content is deemed appropriate (Togelius et al., 2011).

2.3 The Content Creation Bottleneck

In their 2011 taxonomy of experience-based PCG, Yannakakis and Togelius describe the "content creation bottleneck" in regards to the recently increasing time and cost requirements for developing content for "triple-A", or "top-tier" video game titles (Yannakakis and Togelius, 2011). They note that as these requirements increase, developers can no longer afford to try out new and potentially risky concepts, creating a barrier to artistic, design, and technological development. They also point out that while the technology used to develop video games has advanced considerably since the days of *Rogue* (Epyx, Inc., 1980), the content creation process is still, for the most part, a manual process.

Furthermore, the recently increasing trend of "games as a service" has introduced a further complication to the content creation bottleneck. Games as a service refers to the consumer shift of media access across a range of devices, and, more importantly to the bottleneck, long-term engagement with a product (Batchelor, 2014). This means that even after a title is shipped, developers must consider and support the continuous delivery of additional new content. Games such as *Grand Theft Auto V* (Rockstar Games, 2013), *Trials Fusion* (Ubisoft, 2014), and *Super Mario Maker* (Nintendo, 2015) all include content creation tools packaged with the title, which allow users to create and share their own levels. The popularity of the communities which consume this user generated content proves the general desire for further game content, even after players have completed the base game. Consumers also use these tools to create unique and tailored experiences beyond that which was originally considered by the developers, resulting in new user experiences.

It is clear to see that any form of technology which can reduce the effect of the content creation bottleneck, and potentially aid in the creation of user tailored content would be hugely beneficial to the industry (Yannakakis and Togelius, 2011). The ideal solution would be one which interfaces well with the current development pipeline, and provides minimal additional work for designers and content creators. Although PCG provides a method of generating game content algorithmically, which indeed alleviates some of the work from content creators, developing a complex PCG system and ensuring the content generated is reliably correct adds to development time. Instead, we turn to evolutionary techniques, with the intention of training the content generator during development to produce additional content. Ideally, an evolutionary method could also then be applied post shipping, and, by observing the player's behaviour in the game, produce new content tailored to their play style.

2.4 Genetic Algorithms

Genetic algorithms, a term now almost synonymous with *evolutionary algorithms*, are a form of stochastic optimisation algorithm, initially inspired by the mechanisms of biological evolution. The theory behind genetic algorithms suggests that given a number of potential solutions and an appropriate amount of small changes to each, an optimal solution can eventually be reached for any problem (Russell and Norvig, 2009). In order to achieve this, two components are required: a data representation format within the domain of a problem's solution, called a *chromosome*, and a technique for evaluating the suitability of a potential solution, known as a *fitness function* (Whitley, 1994).

Genetic algorithms usually begin with a set of potential, randomised solutions known as a *population*. Each solution in the population, called an *individual*, is evaluated against the algorithm's fitness function, and given a fitness value. This value represents how close the individual is to the, or one of many, optimal solution, with higher values indicating a better solution (Russell and Norvig, 2009). Exactly how this value is calculated depends entirely on the problem domain.

Once all individuals have been assigned a fitness, all, or a select few (based on the highest fitness values), enter a phase of reproduction. Individuals are divided into pairs, and their chromosomes are *cross-bred*, producing one or more offspring, whose chromosome is made up partly of one parent's genetic material, and partly the other's (Whitley, 1994). Exactly how the genetic material is split and reallocated depends on the chromosome representation. For simplicity, many genetic algorithms choose to represent their chromosomes as binary or character strings, as they can easily be split and concatenated for cross-breeding (Russell and Norvig, 2009).

Finally, each individual is given a small chance of *mutation*, where a single element of the chromosome is randomly reassigned. The offspring then replace either the lowest fitness individuals, or the entire population, becoming the new *generation*, and the process repeats until an individual with an optimum fitness value is generated (Russell and Norvig, 2009). This process works by preserving useful genetic data between generations during crossover, allowing two independently evolved individuals with medium fitness to potentially produce an offspring with high fitness. Mutation introduces a further element of random exploration, and has the potential of introducing entirely new solutions to the gene pool (Russell and Norvig, 2009).

2.5 Artificial Neural Networks

In the early days of Artificial Intelligence, a prominent field of research was the creation of mathematical approximations of the workings of the human brain. Inspired by the hypothesis that the brain works by firing signals through interconnected networks of neurons, models were devised to create *Artificial Neural Networks* (Russell and Norvig, 2009). Simply put, these networks consist of a series of input values fed to one or more connected layers of artificial neurons, and finally to a resulting series of output values. One of the most interesting, and relevant, features of artificial neural networks today is their ability to "learn", and to therefore provide accurate output for previously unmet datasets (Russell and Norvig, 2009). Just as with genetic algorithms, this makes them ideal for attempting to find optimised solutions to problems where the precise equation is unknown. Exactly how optimal these results are depends on both the data representation format, and the optimisation function; whether that is the fitness evaluation of a

genetic algorithm, or the learning method of a neural network. A poor or inefficient data representation or optimisation function can result in convergence on an incorrect *local minima*, or solution which matches the search criteria, but does not fulfil the intended goal.

2.5.1 Artificial Neurons

Originally devised by McCulloch and Pitts in 1943, an artificial neuron is defined by several weighted input links along with a *bias weight*, an *activation function* applied to the sum of the weighted inputs, and a number of output links (McCulloch and Pitts, 1943). Input values are initially passed to the neuron along the input links, and are each multiplied by the weight associated with that link. The weighted inputs are then summed, along with a bias weight value associated with the neuron. The total value is then passed through the activation function, which can be thought of as a threshold which must be exceeded in order for the neuron to "fire" (Russell and Norvig, 2009). The result of the activation function is then passed as output to the connected nodes.

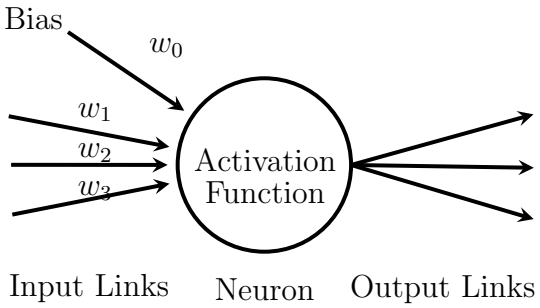


Figure 2.1: Example of an artificial neuron.

Activation functions typically employ either a hard threshold, or a logistic or *sigmoid* function (Russell and Norvig, 2009). Hard threshold neurons are the simpler of the two, and are usually referred to as *perceptrons*. Perceptrons work using binary values, and each have a "threshold" value associated with them. When the sum of the weighted inputs exceeds this threshold, the perceptron returns true; otherwise it returns false (Rosenblatt, 1958). In modern neural networks, the more common neuron model is the *sigmoid neuron*, which, unlike the perceptron, accepts any real number between 0 and 1. Instead of a hard threshold condition, the sigmoid function returns any real number between 0 and 1, and, when plotted, can be visualised as a smoothed step function (Nielsen, 2015). This results in a much wider range of input and output representation, and also has the added benefit of using a differentiable equation, which becomes extremely useful when we discuss learning algorithms further on (Rosenblatt, 1958).

2.5.2 Feed-Forward Networks

The main structure of an Artificial Neural Network's graph can be split into two categories: *feed-forward networks* and *recurrent networks*. Both are comprised of two or more layers of nodes, connected by edges. These layers are the input layer, optional "hidden" layers, and a final output layer, where every node in a layer is connected to every other node in the next. However, while *feed-forward networks* form directed acyclic graphs, *recurrent networks* allow outputs to be fed directly back into their own inputs (Russell

and Norvig, 2009). While this results in the benefit of short-term memory (as output values depend on prior input), recurrent networks can be potentially chaotic or result in oscillating output (Russell and Norvig, 2009). For this reason, we will be focusing on feed-forward network implementation.

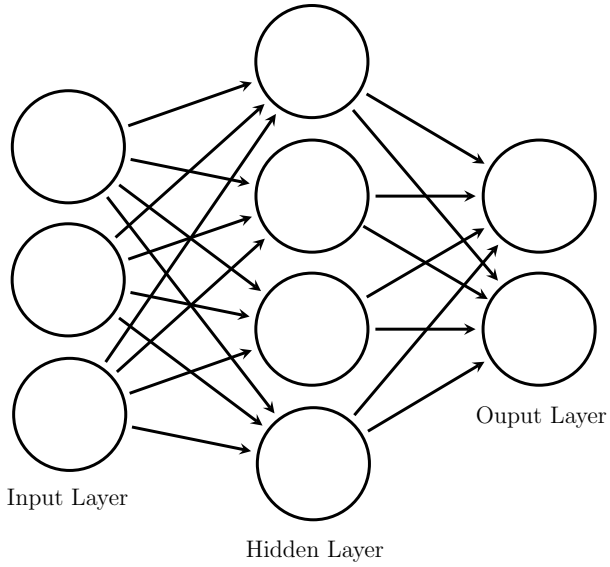


Figure 2.2: Example of a Feed-Forward network with a single hidden layer.

2.5.3 Back-Propagation Learning

As mentioned previously, one of the most useful features of modern day artificial neural networks is their ability to learn and evolve in order to produce "correct" output data. In most cases, a neural network will be initialised with randomised values for each link weight and bias, and as such, input data will at first produce equally random output data. The network must, therefore, be trained with a set of known input to output data. In feed-forward networks, the most common method used to train a neural network is through the process of *back-propagation*, a form of *supervised training* (Russell and Norvig, 2009). The feed-forward process is applied first with a known input. The differences between the outputs from the neural network and the known desired outputs, known as the *error*, are then calculated. The link weights and biases of each layer are then altered in proportion to the error, in respect to how much the specific input contributed to the incorrect output. This is usually calculated using a gradient descent learning rule known as the *delta rule*, which uses the derivative of the sigmoid function in respect to each neuron's individual inputs (Russell and Norvig, 2009).

The structure of a neural network (that is, the number of hidden layers, and number of neurons present in those layers) is vital to its ability to learn. Too few neurons and not enough variation will be present to represent all possible solutions accurately. Too many neurons, and the network will be subject to *overfitting*, the memorisation of training inputs. In this scenario, the network forms a lookup table of its training data, rather than generalising the relationship between the training input and output, and will therefore be unable to produce accurate results for input that it hasn't seen before (Russell and Norvig, 2009).

2.6 Search-Based Procedural Content Generation

Search-Based Procedural Content Generation is a more recent paradigm of PCG, first introduced by Julian Togelius in 2011 (Togelius et al., 2011). In search-based PCG, a set of initial content is created, and a stochastic search or optimisation algorithm is used to search the set for optimal features (Togelius and Shaker, 2016). This technique can be thought of as an abstraction of real-world design procedures, where a collection of initial designs are created, analysed, and the best parts of those designs carried forward into the second design iteration (Togelius and Shaker, 2016). Given a broad enough initial set of content, and an effective analysis technique, the paradigm should, theoretically, converge on an optimum solution. Because of its multiple pass and evaluation nature, the paradigm falls into the category of *generate-and-test*, and the evaluation and refinement stages identify the paradigm as an *experience-based* generator.

2.6.1 The Search Algorithm

The Search Algorithm forms the basis of a Search-Based PCG system. While many different search algorithms can be applied, by far the most common, and the one which has been proven to work well for a variety of problems, is the use of a genetic algorithm (Togelius and Shaker, 2016). The very nature of the search-based paradigm, that of creating an initial set, evaluating fitness, and producing an offspring population, parallels the genetic algorithm process nicely, as can be seen in figure 2.5. While implementing a simple search-based algorithm is trivial, two key components exist which are critical to the resulting success of the algorithm. These are a good *content representation*, and an efficient *evaluation* or *fitness function* (Togelius and Shaker, 2016).

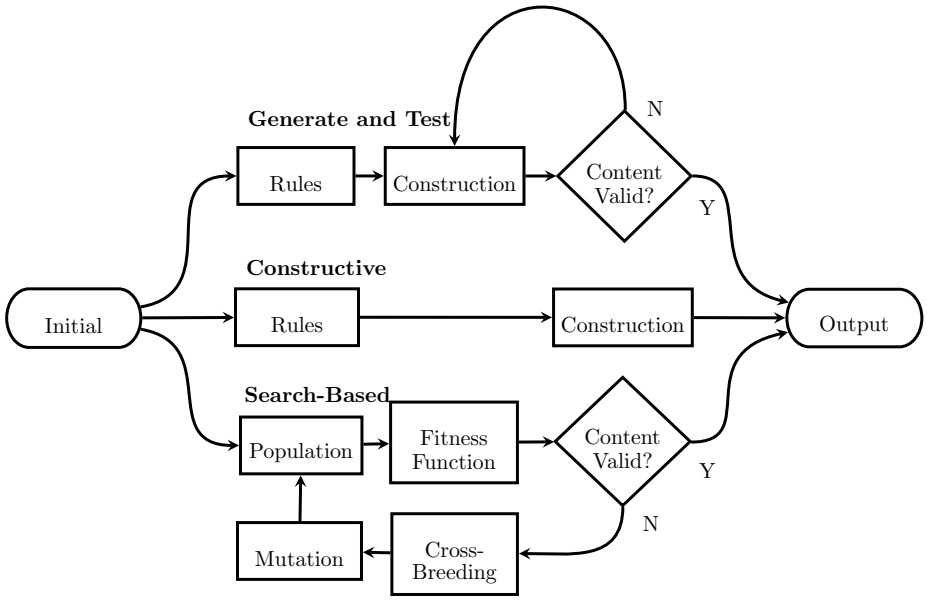


Figure 2.3: Examples of *constructive*, *generate-and-test*, and *search based* algorithms, as described in (Togelius et al., 2011).

2.6.2 Content Representation

Content representation refers to how the content to be generated is represented within the search space. This search space representation, called a *genotype*, can be thought of as a blueprint, which encodes the information needed to create the in-game content representation, known as a *phenotype* (Togelius and Shaker, 2016). The genotype in our search-based paradigm is also our genetic algorithm chromosome, and as such, should use a format which allows for cross-breeding and mutation, such as a character string.

A further consideration is the *genotype-to-phenotype mapping*, or the relationship between the two content representations. Mappings can be *direct* or *indirect*, indicating whether the genotype is linearly or non-linearly proportional in size to the phenotype, respectively. Linear mapping is the simpler of the two, as each gene in the genotype corresponds directly to a single specific part of the phenotype. Indirect mapping requires a usually complex mapping algorithm, and runs the risk of low *locality*, where a small change in the genotype can produce a much larger change to the phenotype, and by extension, the fitness value (Togelius et al., 2011). This can have chaotic or unpredictable results, as a small change to a single gene, either through mutation or cross-breeding, can result in an entirely different fitness value, harming that individual's chances of reproduction, and resulting in the loss of good genetic material.

Focusing primarily on level generation within games, genotypes have been represented in prior work in a multitude of ways, mostly indirectly or hybrid (Togelius et al., 2011). Direct representation, as previously stated, remains simple in terms of mapping, with two or three dimensional character or integer matrices, where each variable represents an exact tile, block, or object within the level. The difficulty, however, is the so-called "curse of dimensionality" (Togelius et al., 2011); since the genotype is proportionate in size to the level, analysing and organising the data can become difficult. Evaluating a directly mapped genotype also presents a challenge, which is discussed further in section 2.6.3.1.

Indirect representations present more abstract, yet manageable solutions. In their Compositional Dungeon Generator, Togelius et al. present a genotype which uses pre-defined variables for properties such as minimum number of steps to a room's exit, number of empty tiles, and range of enemy types, which are then fed through an *Answer Set Solver* in the mapping stage (Togelius et al., 2012). The Answer Set Solver provides a method for generating reliably correct content through the use of *facts*, pre-defined rules which must all be satisfied before generated content is deemed acceptable.

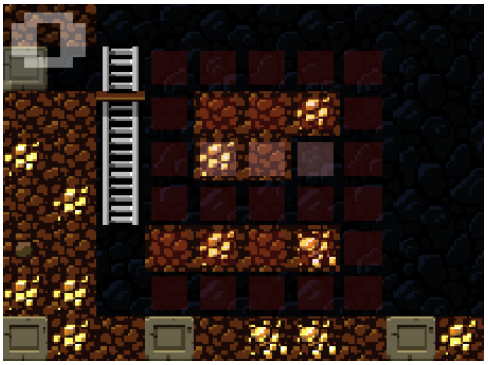
Pedersen et al. used an indirect vector of variables representing the number, positions, and lengths of gaps required to generate a level for Nintendo's iconic 2D platfomer *Super Mario Bros.* (Nintendo, 1985). These controllable features were selected based on feedback from video game design professionals, and represent the very lowest level features that define all games within the platforming genre (Pedersen et al., 2009). Hand-authored level "chunks" of tiles which fit the gap lengths are then inserted in the order specified by the genotype in the mapping stage.

Similar representations were used by Smith et al. (2009) and Jennings-Teats et al. (2010) in the development of their rhythm-based platform generators. Both genotype representations use abstractions of a level chunk's properties, such as the chunk's length, number of enemies and number of jumps required, which are then mapped to hand-authored chunks matching the desired properties (Jennings-Teats et al., 2010).

At the most indirect level, research has been carried out into *generic* content repre-

1	1	0	0	0	0	0	0	0	0
4	0	L	6	0	0	0	0	0	0
1	1	P	0	0	0	0	0	0	0
1	1	L	0	0	0	0	0	0	0
1	1	L	5	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1

(a) Level segment genotype. Each character either directly represents a phenotype tile, or in the case of the '5' and '6' characters, indirectly represents a randomly chosen level chunk.



(b) The resulting phenotype as it would appear in-game.

Figure 2.4: Example of genotype-to-phenotype mapping in *Spelunky*. Adapted from (Kazemi, 2013).

sentation in games: the concept that a single generic content generator can be created to author content for any number of different games and genres. These attempts focus on breaking down level designs into core *design elements*, the most basic composite building blocks which make up a level. Each element in the genotype represents one of these design elements, and are assigned parameters relevant to the element (Sorenson and Pasquier, 2010).

2.6.3 Evaluation Functions

Possibly the most crucial stage of search-based PCG, the evaluation function assigns a fitness value to each individual in the population. A good evaluation function must clearly identify desirable characteristics in an individual; if it does not, then potentially optimal material will be assigned a low fitness score, and will therefore be discarded. What these characteristics are and how they are identified presents an interesting challenge, one that usually possesses a unique solution per problem. Search-based PCG evaluation functions can be split into three categories, *direct*, *interactive*, and *simulation-based* evaluation (Togelius and Shaker, 2016).

2.6.3.1 Direct Evaluation

In direct evaluation functions, evaluation processing is carried out directly on the resulting phenotype. While this results in fast and computationally simple analysis, determining exactly what defines "good" content is difficult from simply observing the resulting level from a data perspective. One way of dealing with this issue is by defining *constraints*: rules which must be observed in the phenotype for it to be valid (Sorenson and Pasquier, 2010). Examples of constraints might be that objects do not overlap, or that there is a valid path from the start to the end of the level.

A second form of direct evaluation is based on the concept of building statistical models of a user's experience when playing a sample of test levels made out of level chunks. This approach has been trialled both through analysing a player's visual and behavioural

traits (Shaker et al., 2013), and through the use of experience questionnaires presented to the player (Pedersen et al., 2010). These techniques focus on rating level chunks and combinations of chunks before they are concatenated by the level generator. The ratings usually use subjective qualities such as "difficulty", "enjoyment", and "beauty" based on player feedback, and the combination of these values are used to approximate how optimal a resulting generated level is (Pedersen et al., 2010). The resulting evaluation is an approximation of the correlation between the generated content, and a player's affective or cognitive response (Togelius and Shaker, 2016).

2.6.3.2 Interactive Evaluation

For the generation of certain game content, such as necessary game levels, it quickly becomes apparent that the content must be sufficiently experienced before any meaningful evaluation of its characteristics can be made (Togelius et al., 2011). In order to achieve this, interactive and simulation-based techniques can be employed. In interactive evaluation, a human user is included "in the loop" as part of the runtime evaluation process (Togelius and Shaker, 2016). This process works by presenting the user with the current population, and asking them to manually provide fitness scores for each individual, based on desirable characteristics or an objectively good user experience. Interactive evaluation can be employed as a direct offline process, where the user is a content designer evolving content to be packaged with the final product (Kerssemakers et al., 2012), or as an indirect online process, such as in the weapon generation system developed for *Galactic Arms Race* (Evolutionary Games, 2012). Hastings et al. (2009) discuss how their system presents the current population of generated weapons to the player as pick-ups, and evaluates and scores each weapon based on the amount of time the player chooses to use them. This results in weapon characteristics that the player found desirable becoming more prominent in future generations.

2.6.3.3 Simulation-based Evaluation

Where interactive evaluation introduces the presence of a human user, simulation-based evaluation removes this requirement through the use of an artificially intelligent agent or *bot*, which, in the case of game content, is able to play the game to some degree. Exactly how efficient the agent must be at this task depends on the type of evaluation required. If the evaluation determines objective qualities, such as whether the generated terrain can be traversed from start to end, then the agent should be designed to excel in traversal (Togelius and Shaker, 2016). In their 2012 paper, Togelius et al. present a simulation-based evaluation function which uses two agents of differing ability to determine skill differentiation in a two-dimensional tile-based dungeon generator. One agent is designed to play smartly, avoiding dangers and collecting ability-enhancing pick-ups, while the other plays recklessly, attempting to take the shortest path to the exit, regardless of dangers in the way. The ratio between the damage sustained by the two agents is then used as part of the level's fitness score, creating an approximation of the difficulty of a level via the assumption that a skilled player should have more success than an unskilled one (Togelius et al., 2012).

However, if the evaluation determines subjective qualities, or is based on prioritising a particular player experience, then the agent should be designed to accurately mimic a human player's performance (Togelius and Shaker, 2016). In their 2007 paper, Togelius et al. present a technique used to generate race tracks for a driving game, tailored to the

ability of the player. Their technique used an artificial neural network controlled agent, trained on data collected at runtime of the player's progress around a series of test tracks. This agent was then used to test drive each generated track in the evaluation phase, with its success governing a fitness score for the track, and therefore an approximation of the track's suitability for the player (Togelius et al., 2007).

Work by Khalifa et al. on creating a framework for generic or *general* video game level generation used a AI agent for level evaluation, initially designed with super-human speed and efficiency, in order to complete any level as quickly as possible. In order to more accurately represent a human player's experience, discrepancies were introduced into the agent's decision making, such as repetition and wait times between actions. This ensured that generated situations which required beyond-human ability to overcome were discouraged in future generations (Khalifa et al., 2016).

2.7 Evolving AI Agents to Play Games

While AI agents, or bots, can be purely hand made, many studies have been carried out into evolving agent behaviour through the use of genetics and artificial neural networks. Agents that can learn and adapt on their own have two major benefits: they can learn to exhibit advanced, human-like behaviours without the need for developing complex logic, and they can mimic an individual's play style to allow for user tailored content development (such as in the race track generation work by Togelius et al. (2007)).

In their 2012 paper, McPartland and Gallagher outline an interactive training technique which allowed designers to train bots for a first person shooter game in real time. Their method used a lookup table of state-action pairs, where each state was a parameter list describing the bot's and its surrounding's state, each action was a behaviour that the bot could carry out, and each pair was assigned a weighted priority. Training was carried out during a real time simulation in three ways: through direct action guiding, reward-penalty allocation, and automatic rewarding. Through the use of these tools, state-action pairs could be created by the designer by specifying goals and actions that the bot should carry out. The bot's actions could then be rewarded or penalised, which modified the weight of the current state-action pair. The bot's actions were also rewarded automatically on the completion of specified goals such as collecting pick-ups, defeating enemies, and not taking damage. Results were ultimately successful, with the resultant bots possessing observably different behaviour patterns (McPartland and Gallagher, 2012).

At a more direct level, research has been carried out into training bots straight from recorded player input. In their 2004 study, Thurau et al. used recorded player input to train AI bots in the game *Quake II* (Activision, 1997), by extracting the player's input and game state information at that moment from the game's network traffic. Since this data was intended for syncing two client's game states over a network, the files included everything that was important to the gaming experience. As such, it could be assumed that the player's input was, to some degree, a direct result of the state data at that moment. Thurau et al. refer to this form of behaviour as *reactive behaviour*, where the player reacts as a result of their immediate state. A form of artificial neural network called a *self-organising map* was then used to train the bot's reactive behaviour using the network data's game state as training input (Thurau et al., 2004).

Similar work was carried out by Karpov et al., who attempted to produce believable

bot navigation in *Unreal Tournament 2004* (Epic Games, Inc., 2004) through the use of a large database of recorded player "traces". These traces were, again, formed from player input and state data taken from network packets, and indexed based on the player's position using an *octree*, a type of spatial partition graph. Karpov et al.'s navigation method used a combination of *Unreal*'s default *navigation graph* traversal, coupled with retrieval and playback of recorded traces. When the bot became stuck or strayed from its current path, traces indexed at the bot's current position, and which matched the bot's current state, were retrieved and the recorded input replayed, resulting in the bots exhibiting smooth and human-like behaviour (Karpov et al., 2013).

2.8 Evaluating the Project's Worth

Having discussed prior research and established terms within the fields of Search-Based PCG and Artificial Intelligence, and identified key content creation issues within the games industry, we now look to assess the value of developing our own PCG technique using AI Bots for fitness evaluation. We will do so by first evaluating our chosen techniques, and how they compare to prior work, and then our problem domain, that of a two-dimensional level generator for a platformer.

2.8.1 Prior Work Comparisons

Despite the recently increasing field of study surrounding new and adaptive PCG techniques (Togelius et al., 2011), the vast majority of commercial video games continue to use generic and constructive (see sections 2.2.3 and 2.2.4) PCG techniques (Mossmouth, 2008; Mojang AB, 2011; Hello Games, 2016). These techniques have their benefits: they are generally simple to implement from a code perspective, and tend to offer the most control over the generated content, due to their predictable runtime results. This content control is particularly appealing, as it guarantees the generated content is reliably correct once the PCG algorithm has been finalised. However, generic constructive techniques can take a large amount of development and design time to reach an iteration that produces the desired output, and usually requires human involvement to play through generated levels to check for inconsistencies and assess general level quality.

Generate-and-test (see section 2.2.4), and by extension, search-based PCG (see section 2.6) offer techniques to alleviate much of this direct human evaluation stage, with search-based techniques also favouring evolutionary learning over hand tailored generation rules (Togelius and Shaker, 2016). Of course, the most vital component of search-based PCG is the fitness function. An ineffective evaluation technique will result in the loss of good genes, and convergence on local minima. Work by Togelius et al. (2007), Togelius et al. (2012) and Khalifa et al. (2016) proved that an effective technique for evaluation was the use of AI bots designed to play through the generated levels just as a human player would, and provide analysis to determine a level fitness rating. The bots could not only test if a level was reliably correct by successfully navigating from the start to the end of a level, but, in the case of Togelius et al.'s (2012) work, also determine how difficult a level was by comparing the success of two bots of differing ability. Togelius et al.'s 2007 paper demonstrated how a neural network could be used to train an evaluation bot to drive an in-game vehicle, by learning from a real player's driving. This technique has two major benefits: the bot has the potential to learn human-like behaviour beyond that which could be easily hand-tailored, and, if employed in an online environment, could

learn an individual player's mannerisms, allowing for the development of content tailored to the ability of the individual.

Several studies have been carried out into the development of genetically engineered AI bots, namely those of McPartland and Gallagher (2012), Thurau et al. (2004) and Karpov et al. (2013). While McPartland and Gallagher's interactive training technique (see section 2.7) produced interesting and noticeably different resultant behaviours (McPartland and Gallagher, 2012), the technique required further human interaction over a series of iterative design sessions. Since we ultimately aim for limited additional human interaction in our proposed technique, we look to more direct learning methods, primarily that of neural network evolution, proven successful by Togelius et al.'s (2012) work. In order to evolve a neural network bot controller, a known set of input to output training data is required (see section 2.5.3). The use of "traces", or a sensory breakdown of the surrounding world and player state mapped to the input buttons pressed by the player in that state, has been proven by Thurau et al. (2004) and Karpov et al. (2013) to be an effective method of collecting training data. Bots trained with recorded player data sets exhibited smooth and human-like movement, although it should be noted that this form of training can only reliably replicate *reactive behaviour*, behaviour which is directly tied to the player's immediate state. More complicated behaviour, such as that which requires tactical planning, or visualisation of future events, is much harder to learn, as it requires deeper understanding of how the world functions, and in many cases, *short term memory* (Thurau et al., 2004). For these, and other reasons, our proposed technique is limited to the problem domain described below.

2.8.2 The Problem Domain

PCG has been used to generate levels and maps within many video games, dating all the way back to some of the very first PCG applications in games such as *Rogue* (Epyx, Inc., 1980). The widespread use of various PCG techniques for level generation, as well as the community dedicated to level generation research, proves how useful further research within this problem domain will be. As games become more demanding in terms of content creation, new techniques to alleviate time and cost constraints will be highly beneficial to the industry (Yannakakis and Togelius, 2011). Furthermore, the problem of level generation in particular introduces some challenges which make it more difficult than generating some other game content. Game levels are almost always necessary content, and as such, they must be reliably correct (see section 2.2.2). They also contribute greatly to how a game is actually played, and how interesting, challenging, and fun the experience is for the player.

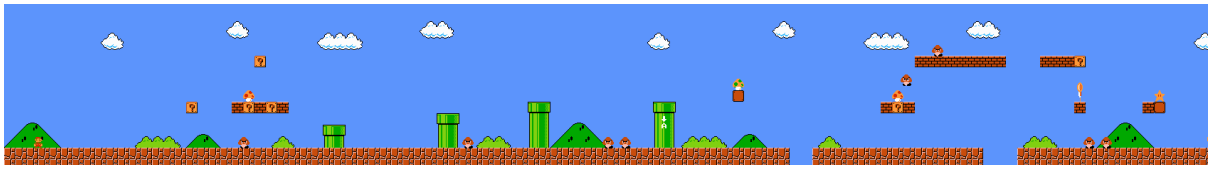


Figure 2.5: The first section of World 1-1 from *Super Mario Bros.* (Nintendo, 1985). Taken from (Albert, 2010).

A great deal of PCG research, including that of Pedersen et al. (2010), Jennings-Teats et al. (2010), and Kerssemakers et al. (2012), has focused on the generation of two-dimensional levels for platformers, particularly for *Super Mario Bros.* (Nintendo, 1985)

or Markus Persson's public domain clone *Infinite Mario Bros.* (The original source code for *Infinite Mario Bros.* has since been removed by Persson). Kerssemakers et al. (2012) note that since the design of *Super Mario Bros.* has been so influential to gaming, and its mechanics so well established, it is the perfect benchmark for testing and comparing the suitability of different PCG techniques. The game's simple level representation of a two-dimensional matrix of "blocks", a representation common in two-dimensional games as discussed in section 2.6.2, simplifies the search domain (Kerssemakers et al., 2012). The two-dimensional view reduces the number of behaviours that must be learnt by our fitness bot, by simplifying camera and orientation control down to left and right movement. Furthermore, the level design of the earlier *Mario* titles focuses almost entirely on reactive mechanical skill; jumping over obstacles, landing on enemies' heads to defeat them, collecting coins, all whilst moving from the start position on the left, to the end goal on the right. This focus on reactive behaviour is perfect for feed-forward neural network training using recorded player state and input mapping (see section 2.8.1). While we will not be using *Infinite Mario Bros.* due to its lack of support, we shall be developing a prototype which features gameplay heavily inspired by *Super Mario Bros.*, in order to gain the same benefits which made *Infinite Mario Bros.* such a popular benchmark.

2.9 Conclusion

In this section we have presented and evaluated Procedural Content Generation concepts and techniques, focusing on Search-Based PCG for level generation in video games. We have discussed prior work in the field, and presented evidence to back up our claim that there is both academic and industry worth in pursuing our concept of AI bot evaluated content generation. As such, the following conclusions can be drawn.

Firstly, there is worth in developing new PCG techniques for the industry. While search-based PCG has an active research community, the vast majority of procedural content generation in triple-A titles is constructive. While these techniques can result in reliable content generation, a large amount of design and programming time is still required, particularly when testing the content generated. With the rapidly increasing cost and time requirements in video game development (the so called "content creation bottleneck" outlined in section 2.3), and the relatively new concept of "games as a service", any techniques which can reduce the number of human participants required would be highly beneficial to the industry (Yannakakis and Togelius, 2011). The ideal technique would be one which complements the current content development pipeline and requires minimal additional effort for the content developers. As such, an evolutionary technique which learns how to generate content from hand tailored examples would not only have the potential to alleviate work load, but also develop user tailored content online to complement the "games as a service" experience.

Secondly, there is worth in developing a search-based PCG solution using a hybrid genetic algorithm and neural network trained evaluation bot. Genetic algorithms share many similarities with the search-based paradigm, mainly that of their population generation and fitness evaluation stage (Togelius and Shaker, 2016). Several search-based PCG papers (Togelius et al., 2007, 2012; Kerssemakers et al., 2012), have proven the effectiveness of using genetic algorithms for various game content generation. While these generators employ a variety of fitness evaluation functions, a technique which removes the need for direct human involvement is the use of an artificially intelligent bot, which

interacts with and evaluates the generated content based on its experience (Togelius and Shaker, 2016). While the evaluation bots of Togelius et al.'s 2012 work were hand tailored, the evaluation bots themselves could be evolved using data collected from human interactions with hand tailored content. Such bots have been previously developed for games such as *Quake II* (Activision, 1997), using neural networks trained on player behaviour data (Thurau et al., 2004). Using an evaluation bot such as this could save a great deal of development and testing time, and, if the content generator was employed in an online environment, could also provide means of generating content tailored to an individual player's play style.

Finally, there is worth in applying our proposed PCG technique to the problem domain of two dimensional level generation. Level generation is one of, if not the most used applications for PCG in video games. Game levels contribute greatly to the interest and challenge of a player's experience within the game, and their generation offers some interesting challenges in that their content must almost always be reliably correct. A large amount of research has been conducted in the field of video game level generation, particularly in *Super Mario Bros.* (Nintendo, 1985), and two-dimensional platformers in general (Pedersen et al., 2010; Jennings-Teats et al., 2010; Kerssemakers et al., 2012). This is due to both the highly influential design of *Super Mario Bros.* to video gaming in general (Kerssemakers et al., 2012), and the simplified search domain and content representation (see section 2.8.2). The highly reactive and mechanical nature of titles inspired by *Super Mario Bros.* also makes evolving bot neural network behaviour simpler. Overall, we hope that by applying our PCG technique within an established active research field, comparisons as to the effectiveness of the technique can be made.

Chapter 3

Methodology

3.1 Introduction

3.1.1 Section Summary

In this chapter we will explore the design, development and final implementation of our proposed Bot-Driven PCG technique. As was outlined by our objectives in section 1.2, part of our project implementation requires the development of a prototype two-dimensional platform game, inspired by *Super Mario Bros.*, several human-authored levels for said game, and a content generator which can generate new levels purely from input recorded from human play-throughs. We will be discussing the design and implementation of all of these components, and how they interact with each other to produce our final working prototype. We will detail the design and research methods employed throughout the project, and outline how we will be collecting and analysing our data in the following results chapter.

3.1.2 Section Structure

- **3.2 Design Methods:** We begin this chapter by outlining the design methods employed on our project; how the Agile Software Development principles helped us to rapidly prototype and re-evaluate our progress, and how the use of Unified Modelling Language and flowcharts helped us to plan our application structure.
- **3.3 Technical Overview:** We then begin our technical discussion by providing an overview of the different components of the project, and how they communicate.
- **3.4 Prototype Game Implementation:** Before delving into the content generator itself, we discuss the development of our prototype game, and the principles which will govern our level design. We discuss the tools used to create our human-authored levels, and consequently the file format that we chose to support.
- **3.5 Evaluation Agent Implementation:** In this section we detail the development of the first component of our generator: our evaluation agent. We discuss the artificial neural network used to determine the agent's reactions, and how we collect and use player input to train it.
- **3.6 Content Generator Implementation:** Following on from the creation of the evaluation agent, we discuss the implementation of our search-based PCG tech-

nique, how we create an initial population from our human-authored content, and how we use the agent to determine a generated level's fitness score.

- **3.7 Collecting and Analysing Data:** Having discussed the full generator implementation, we outline the techniques we will be using to determine the success of our prototype in the following chapter.

- **3.8 Graphical User Interface Design:** Finally we discuss the implementation of our application's graphical user interface, and present examples of our final working product.

3.2 Design Methods

Before development of our technique could begin, it was important to decide on a set of design methods and principles which complemented the project. Adhering to a set of established design methods ensures that the project remains focused and manageable. For the development of our technique, we chose to employ *Agile Software Development* principles (Beck et al., 2001), which favour adaptive development and rapid response to change. In standard Agile development, development time is split into "timeboxes" of determined length (usually one to four weeks), with a set of goals specified for each timebox. At the end of a timebox, the project is evaluated, and goals for the next adjusted to accommodate any issues or developments that have occurred. The experimental and iterative nature of this project meant that this constant re-evaluation of progress was crucial for effective time management. Agile development states that working software is the main measure of progress, and therefore each timebox should result in some working component of the project where possible. This philosophy is particularly important for our project, which contains multiple encapsulated components. While each component provides some functionality on its own, it is the interaction of these components which defines our finished technique. The use of Agile planning allowed us to focus on the development of each component in order and ultimately ensured that each component was functional before connecting them.

A vast number of techniques exist for outlining the structure and communication of components within a project, the most common of which are *Unified Modelling Language* (UML), *data flow diagrams*, and *flowcharts*. For our project we decided to use both UML, a standardised general purpose modelling language for software design, and flowcharts. UML defines two categories of diagram types, *structural* and *behavioural* diagrams, where each diagram outlines a project's implementation from various points of view. We chose UML not just for its universal recognition and legibility, but also because specific diagrams such as the communication and component diagrams are particularly apt for describing the interaction between the various components of our project. For describing more generalised high-level behaviour, we chose to use flowcharts for their simplicity. We found that planning out the intended behaviour of a system beforehand using flowcharts was a good method for determining the sequence of different events, and worked well for communicating our ideas in an easy to follow format.

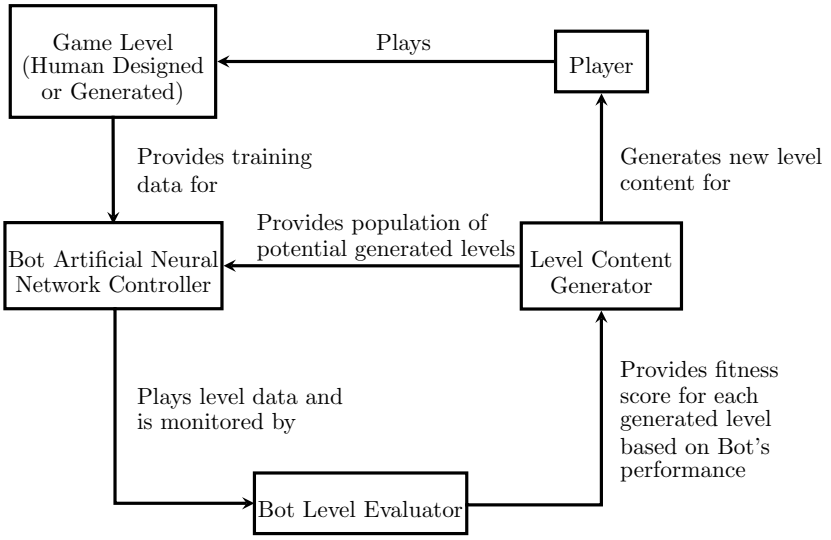


Figure 3.1: Diagram showing the relationships between the core components of our implementation.

3.3 Technical Overview

Our project can be broadly broken down into three components: our prototype game, Evaluation Agent, and Level Content Generator. Figure 3.1 outlines the high-level relationship between these components. Use of the finished prototype can be split into two phases: the Training Phase, where the user plays through the human-designed levels of the game and indirectly collects training data for the bot (see section 2.7), and the Generation Phase, which is modelled on the Search-Based PCG paradigm (see section 2.6) and uses the newly trained bot for level fitness evaluation. In the following sections we will detail the design and implementation of each component, and how they interact with one another.

3.4 Prototype Game Implementation

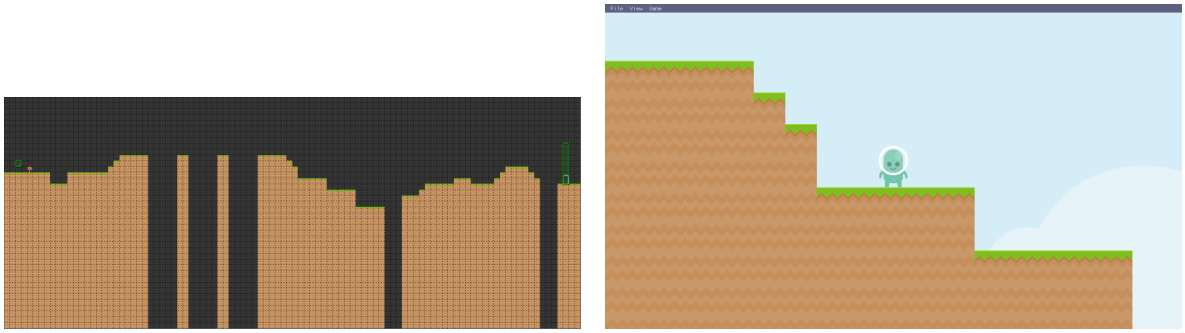
As previously stated, one of our first objectives was the development of a simple video game, inspired by *Super Mario Bros.*, that would provide the problem domain for our generator. As outlined in section 2.8.2, *Super Mario Bros.* and similar two-dimensional platformers provide a good benchmark for procedural level generators, due to their tile matrix data representation, and simplified traversal space (the player can usually move left or right, and jump) (Kerssemakers et al., 2012). To that end, our game incorporates the following simple design principles:

- Each level features one entrance and one exit point. The entrance point is on the far left of the level, while the exit is on the far right. The player starts at the entrance point, and must reach the exit point to successfully complete the level.
- The player can move left, right, and jump a set height. They can move while in the air.
- Each level contains a series of platforms that the player can tread on, and a set of gaps which the player must jump over. Falling through a gap results in the player

failing the level.

- Each level will contain only a single path from start to end.

Our level data is represented by a tile matrix, plus some additional data for game objects such as entrance and exit points. This data is stored in an XML file, specifically in the TMX file format, a format developed for the map editor software *Tiled* (Lindeijer, 2008). Tiled was chosen for our human-authored level creation due to its wide range of two-dimensional level creation tools. It therefore made sense for both our game and our generator to support the data format, so that an additional file conversion step was not required, and so that generated levels were produced in a widely supported format. Our sample game consists of five human-authored levels, which we designed following the principles outlined above. These levels provide both the training ground for our evaluation bot (see section 3.5.1), and the content for generating our initial population (see section 3.6.1).



(a) A hand authored level from our game as it appears in *Tiled*.

(b) Part of the same level as it appears in our game.

Figure 3.2: Example of a hand authored level from our game prototype.

3.5 Evaluation Agent Implementation

Before discussing the generator itself, it is important to establish the behaviour of our evaluation bot, as the bot's ability to play through a level of our game is crucial to determine the fitness of a level. As outlined in section 2.6.3, many different evaluation methods have been explored in prior work, including the use of questionnaires presented to the user after playing through a generated level (Pedersen et al., 2010), and monitoring player statistics during play to determine the type of content favoured (Hastings et al., 2009). The use of trained AI bots to evaluate fitness is of particular interest to our study, however, as it removes the need for human interaction during testing, while still emulating human-like behaviour. This removal of direct human evaluation would alleviate a great deal of the work load placed on developers of procedurally generated games.

To develop our bots, we will be using an artificial neural network, similar to that developed by Togelius et al. (2007) for their procedural race track generator, with a training method inspired by the work of Thurau et al. (2004). Both studies produced favourable results regarding the use of artificial neural networks for learning situation dependant reactive behaviour. Thurau et al.'s use of recorded game state mapped to player input for bot training is of particular relevance to this project, as this data could

easily be collected during normal play sessions of hand authored content. This could be implemented as an offline process, where data is collected from play-testing during development, or online in a shipped product, where the player's own input is recorded.

3.5.1 Game State to Input Mapping

The first stage of our bot development is the collection of our training data. Our training data collector can be activated at any time during a play session, and, when activated, records a new entry of data each time the player moves into a new tile while a button is pressed, or when a new button is pressed. Each data entry is a tuple, mapping an abstraction of the world around the player (referred to as the "State") to the button that was pressed (referred to as the "Reaction"). Our State is a binary vector representing all tiles in a specified radius around the player that are passable (0) or impassable (1). Our Reaction is simply a string representing one of our three responses (move left, move right, jump). The same State vector can be mapped to multiple Reactions, allowing different player behaviour in the same State to be represented across multiple play-throughs. If the entries are recorded consecutively, as shown by the second and third entry in figure 3.3, then the player pressed multiple buttons at the same time in a given situation. In this scenario, the Reactions will be concatenated, which we will discuss further in section 3.5.2. The recorded training data is output to a simple CSV (comma-separated values) file.

```
000111111110001011111111000101111111111111011111111,MoveRight
001111111100011111111110001111111111111111111111111,MoveRight
001111111100011111111110001111111111111111111111111,Jump
000000000000000111111110001111111111000111111111111,MoveRight
```

Figure 3.3: The format of our training data entries (State vector length reduced for space).

Before we can map our State vector, we must decide on a tile radius. This value represents how many tiles in each cardinal direction the bot can "see"; as the radius increases, so too does the total number of tiles recorded with each data entry. As will be explained in section 3.5.2, the number of tiles in our State vector will become the number of neurons in the input layer of our neural network. As such, this value will directly affect how quickly and efficiently the bot can learn from the training set. If the radius is too small, then not enough variation will be present to represent all possible solutions. However, if the radius is too big, then we risk our network becoming subject to overfitting, as discussed in section 2.5.3. For our experiment, we found that a radius of 6 provided a good amount of variation.

To map our State vector, we first determine the tile position in our tile map that the player currently occupies. In our game, that is as simple as dividing our player pixel position (where the origin of our character is at the bottom centre of our sprite) by the size of a tile in pixels, and rounding the result down to the nearest integer. We then define a min and max extent in both the x and y axis as the player's tile position plus or minus the tile radius. Starting at the min extents, we iterate through the surrounding tiles, one row at a time, and write the passability of each tile to our State vector, as demonstrated in figure 3.4.

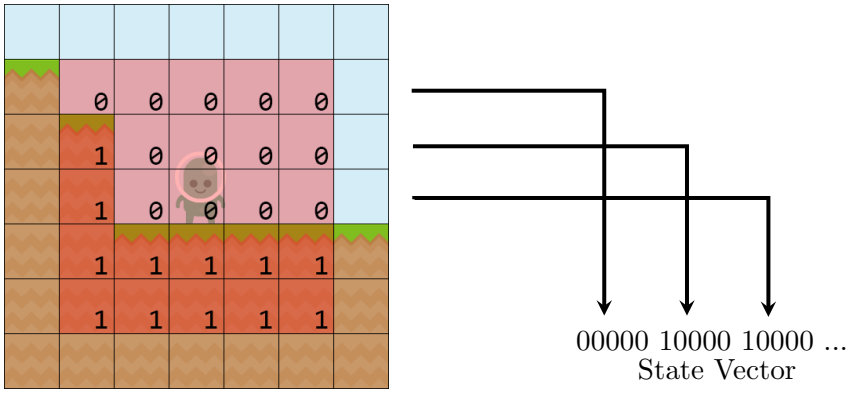


Figure 3.4: Mapping the abstracted world state to a State vector with a tile radius of 2.

3.5.2 Artificial Neural Network

While there are many different potential implementations of artificial neural networks, we have chosen to use a feed-forward network with one hidden layer. The added benefit of short-term memory provided by recurrent networks was deemed an unnecessary addition for learning purely situational reactive behaviour, particularly given the added complications of potentially chaotic output oscillation that arises from cyclic neural networks. We chose to use one hidden layer in our testing, as the relationship between our State and Reaction data is non-linear, and more than one hidden layer has been proven for the majority of simple problems to increase the chances of arriving at a local minima (Russell and Norvig, 2009).

Our input layer features one neuron for each tile in our State vector; with our tile radius of 6 tiles, each row and column contains 6 tiles in both the positive and negative directions, plus 1 for the tile currently occupied by the player, resulting in $13^2 = 169$ neurons. Our output layer contains just three neurons, one for each of our three responses. As discussed earlier in section 2.5.3, the number of neurons in our hidden layer is crucial to the effectiveness of the network's learning capabilities. As such, we used the approximation

$$N_h = \frac{N_s}{\alpha(N_i + N_o)} \tag{3.1}$$

where N_h is the number of hidden neurons, N_s is the number of training data State-Reaction tuples recorded, N_i and N_o are the number of input and output neurons respectively, and α is an arbitrary scaling factor between 2 and 10. This equation represents a general rule-of-thumb for calculating the upper bound of hidden neurons that can be used without resulting in overfitting. The scaling factor was adjusted through trial and error to arrive at a value which produced favourable results for our simulation. With a training set of $N_s = 6520$ entries, $N_i = 169$ and $N_o = 3$, and our chosen scaling factor of $\alpha = 5$, we arrive at our upper bound of hidden neurons $N_h \approx 7.6$. As such, we chose to use 7 neurons in the hidden layer.

Each link between our neurons was assigned a weight, and each layer, excluding the input layer, a bias modifier. These modifiers were all initially assigned randomised values between 0 and 1. While our weights modify the steepness of our sigmoid function, the bias shifts the sigmoid along the x-axis, allowing a wider range of output representation.

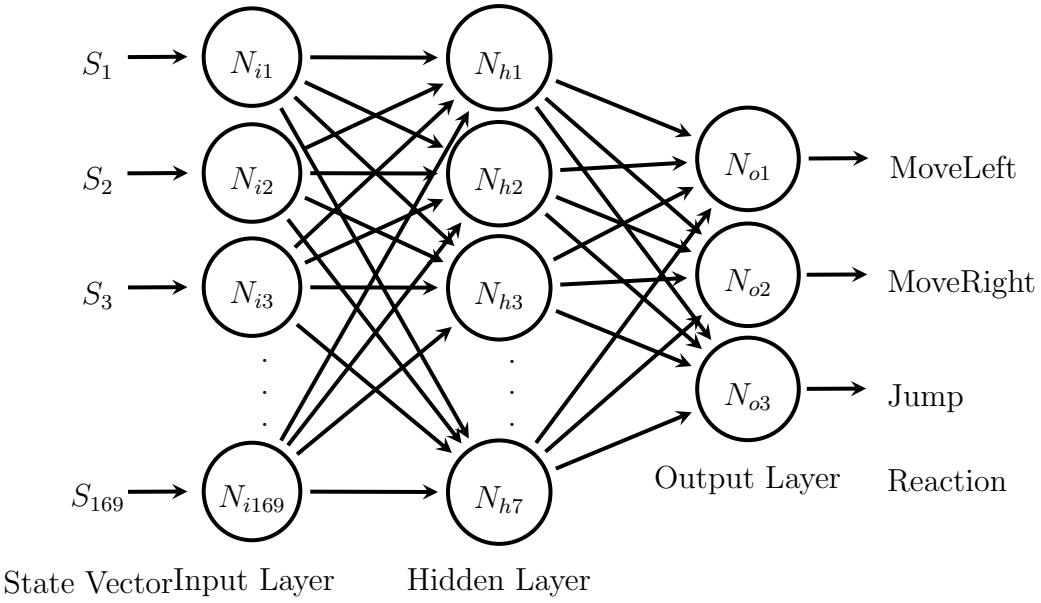


Figure 3.5: The structure of our State Vector to Reaction neural network implementation, with $N_i = 169$, $N_h = 7$, and $N_o = 3$.

To train our neural network, we used a process of back-propagation learning, which was considered appropriate given our collection of known input to output sample data. To make our back-propagation algorithm simpler, our neuron activation function needed to be differentiable. We chose to use a sigmoid function, which gives us the added benefit of a wide range of data representation. The sigmoid function is defined as

$$S_t = \frac{1}{1 + e^{-t}} \tag{3.2}$$

where t is our input value and e is the mathematical constant. Since the sigmoid function results in a non-binary output, but our actual reactions are binary (a button can either be pressed or not pressed), values in the output layer are rounded up to 1 above a certain threshold value (0.8), and down to 0 below that threshold. Values are only rounded when output is requested: in order for back-propagation to work as intended, the actual values are still stored internally for future calculations.

Before a training session begins, each entry in the specified training data is converted to two arrays of floating point values, where the State array specifies the input values for our neural network, and the Reaction array specifies the "correct" output values for that input. Although these values are floating point, only values of 0 or 1 are used at this point, with the State vector mapping directly to the State array, and a value of 0 or 1 used to represent each Reaction which was or was not carried out by the player respectively. The Reaction array therefore contains the same number of entries as the character has player-controllable reactions. In our implementation, this means three entries, where the first entry represents our "move left" reaction, the second "move right", and the final "jump". Importantly, concurrent entries with matching State vectors have their Reactions combined at this point, so that reactions which were triggered at the same time are correctly represented. Figure 3.6 demonstrates this process.

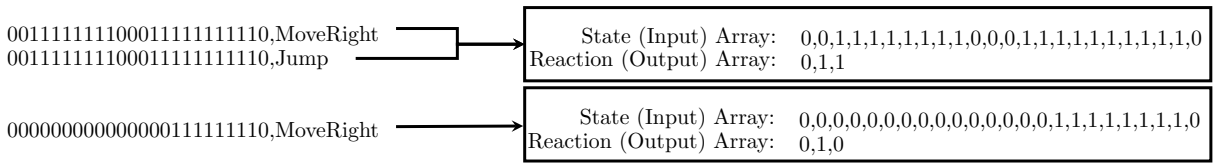


Figure 3.6: A diagram explaining how recorded tuples are converted for neural network training.

With the training data converted, we begin the training session by specifying a *learning rate*, *epoch number*, and *batch size*. Our learning rate is a scaling factor which determines how much a neuron's weight should be changed during back-propagation; too small a value and the network will take too long to train, while a large value will result in oscillation around the correct value. For our work, we found that a learning rate of 0.15 produced favourable results. The epoch number determines how many batches of training data should be used to train the network, while the batch size determines how many entries should be selected in a batch. While these parameters are left editable by the user, we found that 16000 epochs with a batch size of 50 was enough to train reasonable reactive behaviour.

For each epoch, the training data set is shuffled and a group of training tuples, specified by the batch size, selected. For each tuple, the network is initially fed-forward with the tuple's State array as input, and then back-propagated against the corresponding Reaction array. With each back-propagation pass, the network's weights are adjusted by the calculated error between the fed-forward output and the desired output specified by the tuple's Reaction array, scaled by the learning rate. The result is a trained network capable of outputting a predicted Reaction to any given State input. The trained neural network data is output to an XML file which specifies the number of layers, the size of each layer, and the weight and bias values of each neuron connection.

3.5.3 Bot Controller

To translate our neural network output into actual character behaviour, we developed a Bot Controller which interfaces with our in-game character. Every frame, the Controller queries the tiles surrounding the player's character (using the same State vector query technique outlined in section 3.5.1 and figure 3.4), and feeds the State vector into the trained neural network. The resultant Reaction output then determines which of the character's three reactions are triggered. For example, if our Reaction output was "0,1,1", our character's "move right" and "jump" functions would be called. Since our Reaction output only specifies which buttons should be pressed, and not those which should be released, we release all buttons at the start of each frame.

3.6 Content Generator Implementation

With our prototype game and evaluation Bot developed, the final core component of our project is the Content Generator itself. The generator ties all of our work up until this point together. It operates the following inputs:

- The trained neural network file that we wish to use for our Bot's behaviour.

- The list of human-authored level files that will provide our generator its starting content.
- The size of a level population.
- A Mutation Rate between 0 and 1.
- The number of levels to generate.

The generator saves and outputs the file locations of each level generated. As previously discussed, our implementation is modelled on the search-based PCG paradigm, and features a population of potential levels, a fitness evaluation function provided by our Bot, and a cross-breeding and mutation pass. The generator creates an initial population of randomised potential levels using data from the specified human-authored content, and proceeds to present each to the Bot in turn. The Bot attempts to play through each level, and assigns a fitness value based on its success or failure to complete it. Once all potential levels in a generation have been analysed, the population is cross-bred and mutated to form the next generation. If a level scores a perfect fitness of 1, the generator terminates its current cycle and saves the level data. This process is outlined in figure 3.7. To describe the process in detail, we will break the implementation down into its key parts.

3.6.1 Generating the Initial Population

As we discussed in section 2.6.2, one of the first considerations when creating a search-based PCG system is the content's genotype-to-phenotype mapping. The tile matrix used in our prototype game implementation is an example of linearly proportional mapping; each value in our TMX tile array directly corresponds to a different tile type and its position in game. While linearly proportional mapping is desirable, working directly with our TMX file format for content generation presents some difficulties, as we have no defined way of cross-breeding or mutating the files. As such, we split our level data (hereby referred to as our level "chromosome" in keeping with genetic algorithm terminology) into chunks (our chromosome's "genes"). Representing a game's level data by chunks is a common technique, which we outlined in more detail in section 2.6.2.

In our implementation, we ensured that all of our human-authored levels were the same size (100 x 40 tiles), and on loading the generator, each level is split vertically into 10 equally sized chunks. These chunks contain the tile matrix contained within their specified bounds, as well as any game objects they include (such as an entrance or exit point), and a variable indicating whether the chunk is a "start", "middle", or "end" chunk, referred to as the chunk "Sequence Type". This differentiation ensures that each generated level begins with exactly one start chunk, concludes with a single end chunk, and contains 8 randomised middle chunks between the two. An example of a human-authored level split into chunks can be seen in figure 3.8.

This process of chunk creation is repeated on all human-authored levels, and each chunk saved to our "gene pool". This pool contains all possible genes a level chromosome can be built from. Our randomised initial population is now created, the size of which can be specified by the user. We found that a population size of 20 chromosomes was a good amount to represent a varied initial population. To create a new chromosome, the generator randomly selects 10 chunks from the pool, ensuring the first is a start chunk and the last an end chunk.

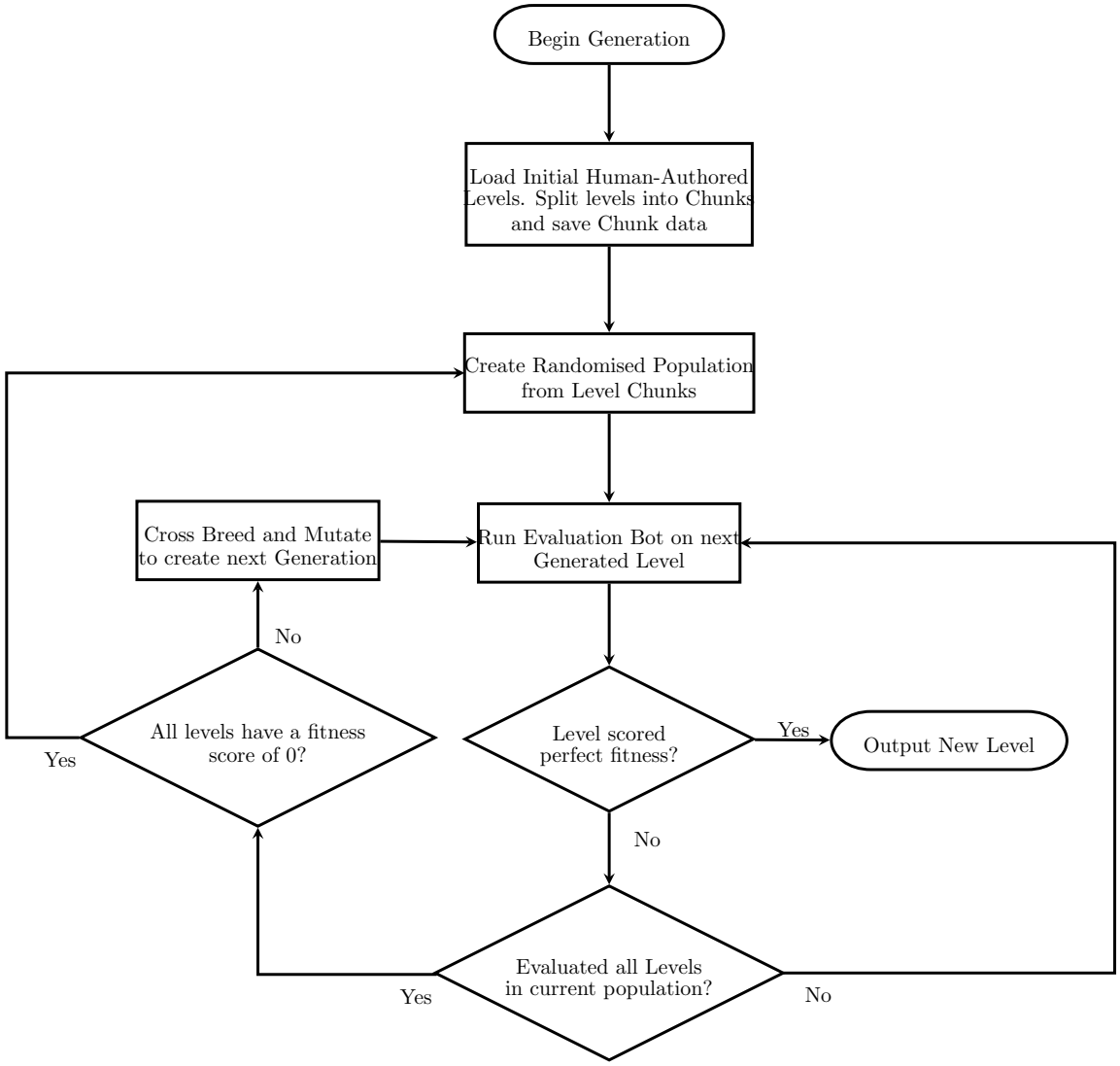


Figure 3.7: A flowchart describing the level generation process of our Content Generator.

3.6.2 Chromosome Fitness Evaluation

The first stage of evaluation is to convert our current population into the TMX file format, so that each potential level can be opened and evaluated in the game engine. To do this, we wrote a simple function to concatenate the tile matrix of each chunk in the chromosome in order, and save it to a new TMX file. These files are stored in a temporary folder, and are deleted at the end of each generation. Once saved, the potential levels are then opened in sequence within our game, and an evaluation bot spawned at the location of the start object. The evaluation bot's progress is monitored and statistics recorded as it attempts to play through the generated level. The most important statistics are:

- The time spent stuck and unable to move.
- The time spent backtracking or simply not progressing through the level.
- The distance the bot has currently progressed through the level.

Statistics such as the time spent stuck and time spent backtracking are used to break out of the current level early if they reach a specified time (3 seconds). It is assumed that

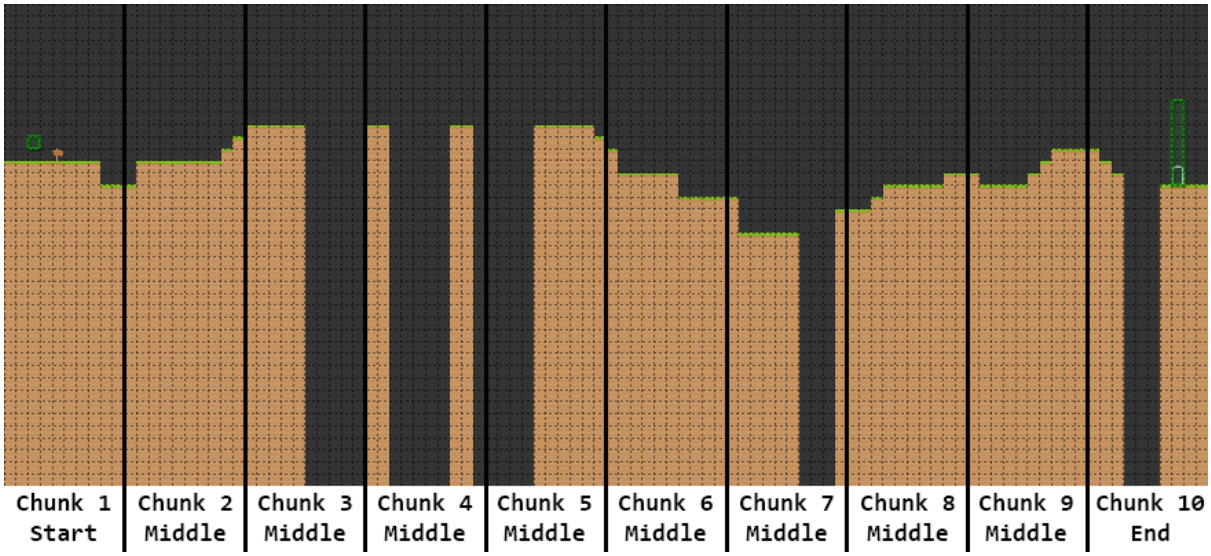


Figure 3.8: Splitting a human-authored level into chunks and specifying the Sequence Type.

if the bot has not progressed within this time that the level is unplayable and testing can be terminated. Likewise, if the bot dies by falling out of the world, the level is once again assumed unplayable and testing terminated. Once evaluation of a level has been completed, either by the bot reaching the exit, or by one of the aforementioned termination conditions, the level is awarded a fitness value F. This value is defined as

$$F = \frac{D_{achieved}}{D_{total}} \tag{3.3}$$

where $D_{achieved}$ is the furthest distance through the level achieved by the bot, and D_{total} is the total distance between the start and exit of the level. Distance is measured as the number of tiles between two points horizontally, with the vertical component discarded, as height is unimportant in determining how much of our level the bot completed. This results in the calculated fitness being directly proportional to the bot's progress, and a perfect score of 1 only being awarded when the bot successfully completes the whole level. Assuming the level was not awarded a perfect fitness score, the evaluation process repeats until every chromosome in the current population has been assigned a fitness.

3.6.3 Cross-breeding and Mutation

Once each chromosome in the current population has been evaluated, the population is cross-bred and mutated to produce the next generation (see section 2.4). In our cross-breeding stage, we want to ensure that chromosomes with a higher fitness value have a greater chance of reproducing compared to those of a lower fitness. Our cross-breeding process begins by forming a *mating pool* of chromosomes. The mating pool is populated by $N_{instances} = 100F$ (rounded to the nearest positive integer) instances of each chromosome, where F is the chromosome's fitness value. This ensures that our probability distribution is directly proportionate to the fitness distribution across the population. Our two parent chromosomes are then randomly selected for reproduction from the mating pool.

The child chromosome is formed by randomly selecting either the first or second parent's genetic material for each level chunk in the chromosome, resulting in a combination of the two parents. Next, mutation is applied. Each chunk in the child chromosome is given a chance to mutate, specified by the generator's Mutation Rate. We found that a rate of 1% introduced a good amount of variation to the system. If a chunk is chosen for mutation, that chunk is replaced with a randomly selected chunk of the same Sequence Type from the gene pool. A number of children equalling that of the previous generation are created, thus becoming the new population. In the event that every chromosome in the population is assigned a fitness value of $F = 0$, the entire population is discarded and a new randomised population generated from human-authored level chunks in its place. In both scenarios, the fitness evaluation process is then applied once more on the new population, and the cycle is repeated until a chromosome of appropriate fitness is generated.

Importantly, we do allow the same chromosome to be selected for use as both parents. The result of this (assuming no chunks in the chromosome are mutated) is that the child will be an exact copy of the parent. This gives chromosomes with a high fitness rating a good chance of returning in the next generation, and preserves good genetic material for breeding in the following generation. If an unmutated chromosome of identical parents is created, it is automatically assigned the same fitness score as its parents, and is not re-evaluated in the next generation to save time.

3.7 Collecting and Analysing Data

In order to determine the value of our proposed technique, we must analyse both the content generator itself, and the level content that our generator can produce. A common method for level generator analysis is to determine the generator's *expressive range* (Smith and Whitehead, 2010) through statistical evaluation. This technique is designed to evaluate the impact of the generator's parameters and any constraints imposed on the system, and to determine the overall range of content that a single generator can produce. While we shall be borrowing certain aspects of this technique to analyse our solution, running our generator took too long on average to generate a wide enough range of content to justify using this method in full. Instead, we will be splitting our analysis into two sections. For the first, we will be analysing the process of generating a level, by recording and plotting data related to the training of our evaluation bot, and the evolution of the level itself. Then we will analyse the success of the content generated, by presenting our final product to a sample user base, and collecting and analysing their opinions on our levels.

Since the original aim of our project was to develop a content generator which could build on existing human-authored levels to extend a game's lifetime, we thought it best to include a sample of the potential target audience in the final testing stage. To this end, we decided that we would present our product to a selection of like-minded gamers, fifteen in total, and ask them to play through five levels of our game. Only two of these levels would be selected from the human-authored levels of our game, while the other three would be produced by our generator. After each level, the player would be asked to fill in a questionnaire on their experience. Our goal was to compare the player's experience between human-authored and generated content. To keep each player's opinions non-biased, we decided not to inform them of how each level was created, and instead opted to

ask each participant whether they believed the level to be generated or human-authored. The rest of our questions, which involve scoring each level a value of 1 (low) to 5 (high) for its difficulty, predictability, and enjoyment, were inspired by the work of Pedersen et al., who used similar questionnaires to model a player's experience in *Super Mario Bros.* (Pedersen et al., 2009). The final questionnaire design can be seen in figure 3.11.

To analyse the success of our solution, we will plot the results of our user experiment and compare the modelled player experience between the generated and human-authored levels. We will also present and discuss a selection of sample levels created by our content generator, and compare their features to the human-authored levels used to generate them.

3.8 Graphical User Interface Design

We developed a Graphical User Interface for our application, which allowed us to easily manipulate the various parameters of our generator, and to display important statistics. A user interface is also important for player testing, in which a questionnaire is presented to the user on completion of a level. While there are many third-party graphical user interface libraries to choose from, we decided upon *dear imgui* (also known simply as *ImGui*) (Cornut, 2017). ImGui is a free, open-source, immediate-mode user interface library for C++. We chose ImGui as it is completely self-contained, favours fast iteration and simplistic design, and has bindings for a wide range of languages and frameworks. Using ImGui, we were able to rapidly prototype, implement, and update the tools we needed without getting caught up in the usual complexities of user interface programming.

3.8.1 Generator and Evaluator

Figure 3.9 shows an annotated screenshot of our main user interface elements. These user interface panels allow us to control all aspects of our generator, as well as providing a way to test individual components. The individual elements as numbered in figure 3.9 are outlined below.

1. **Level Settings**

   General settings for loading and playing a level in the game engine. Allows the user to specify the level they wish to load by file path, and toggle whether the user or a bot should take control of the player character. Being able to observe a bot's progress through any level is a useful feature to have when it comes to analysing the bot's performance. This is also where the user can set up a training data recording session, and specify the file to save the training data to.

2. **Neural Network Settings**

   Allows the user to train a bot's neural network by specifying a set of recorded training data, and an optional starting network to train further. The parameters which control our back-propagation algorithm (see section 3.5.2) are all exposed here for the user to adjust accordingly. A progress bar is included to provide a visual representation during particularly long training sessions.

3. **Generator Settings**

The main controls for starting our Level Generator. The user can specify the evaluation bot they wish to use, the number of levels to generate, an optional max number of generations (which, if reached, will discard the current generation and generate a new one from randomised human-authored chunks), the size of each population, and a mutation rate (see section 3.6).

4. **Graphs**

   Displays a visualisation of two key statistics recorded from the generator; the average fitness value of each generation, and the time elapsed during the generation of each level.

5. **Generator Statistics**

   Displays further numerical statistics recorded from the generator. The current generation number and current population member in evaluation are both shown, as well as the total number of levels generated in this session, and the average fitness of the current generation.

6. **Evaluation Statistics**

   Statistics from the current evaluation session relating to the bot's progress through the level. The termination tracking and fitness evaluation statistics described in section 3.6.2 are all detailed here.

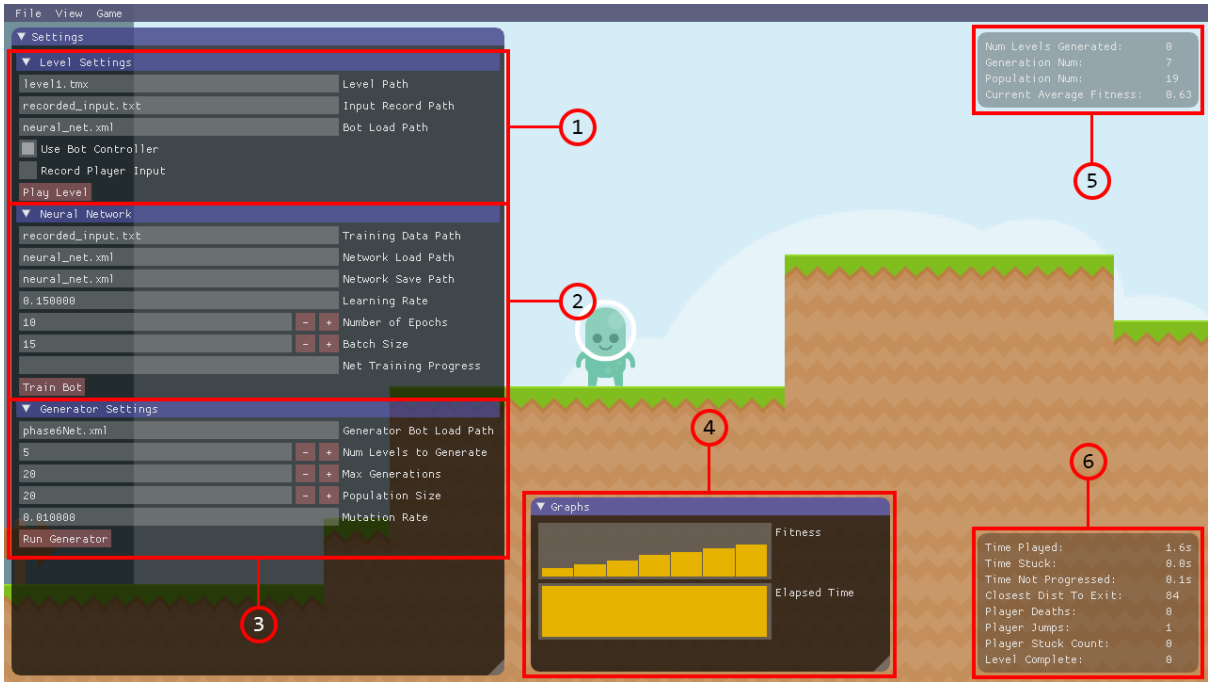


Figure 3.9: Annotated breakdown of our main graphical user interface.

3.8.2 User Testing Mode and Questionnaire

Finally, we developed a second graphical user interface for use when running our application in User Testing Mode. This mode was designed for collecting the data discussed

in section 3.7, a key part of which is the Questionnaire. When running in User Testing Mode, the user is initially greeted by a welcome screen, shown in figure 3.10, which explains the key concepts of the game, and the experiment that they will be taking part in. The user is then prompted to play through a selection of levels, some human-authored and some created by our level generator. They are given three attempts to complete each one, and on completion, or failure, they are presented with the questionnaire show in figure 3.11. The data entered into each questionnaire is saved to an XML file, along with the evaluation statistics recorded for the player's behaviour in that level (across their potential three attempts). We then analysed this data, the results of which we will discuss in chapter 4.

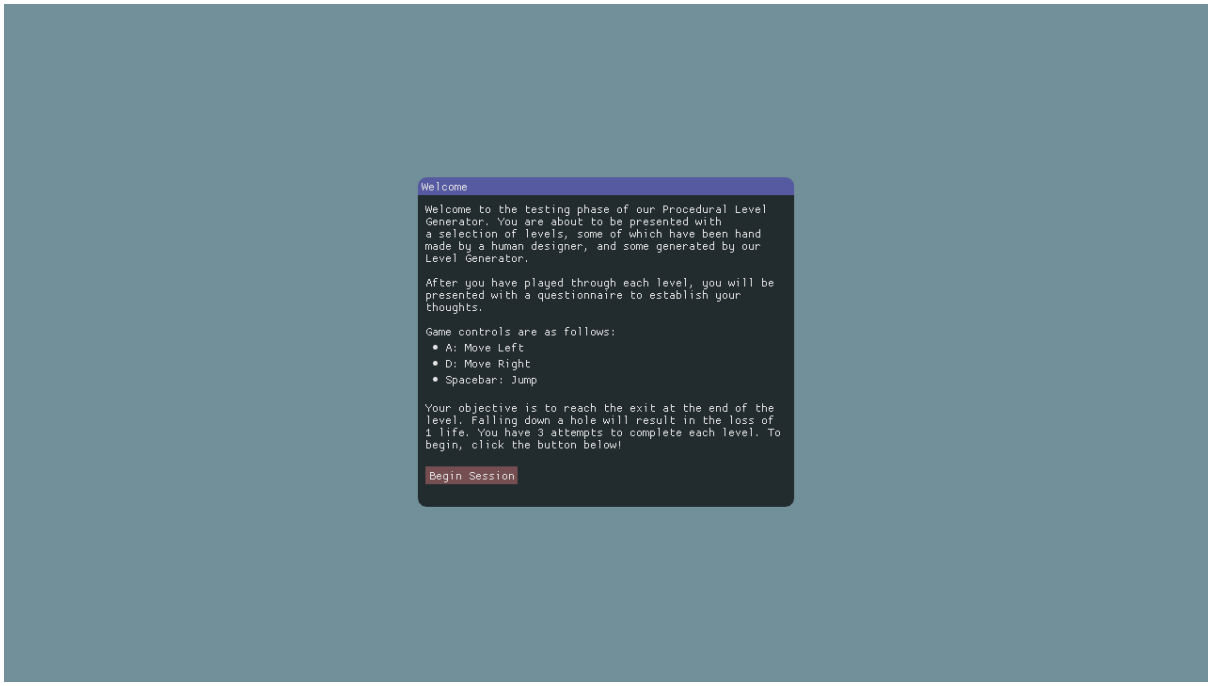


Figure 3.10: The welcome screen that greets the user when running our application in User Testing Mode.

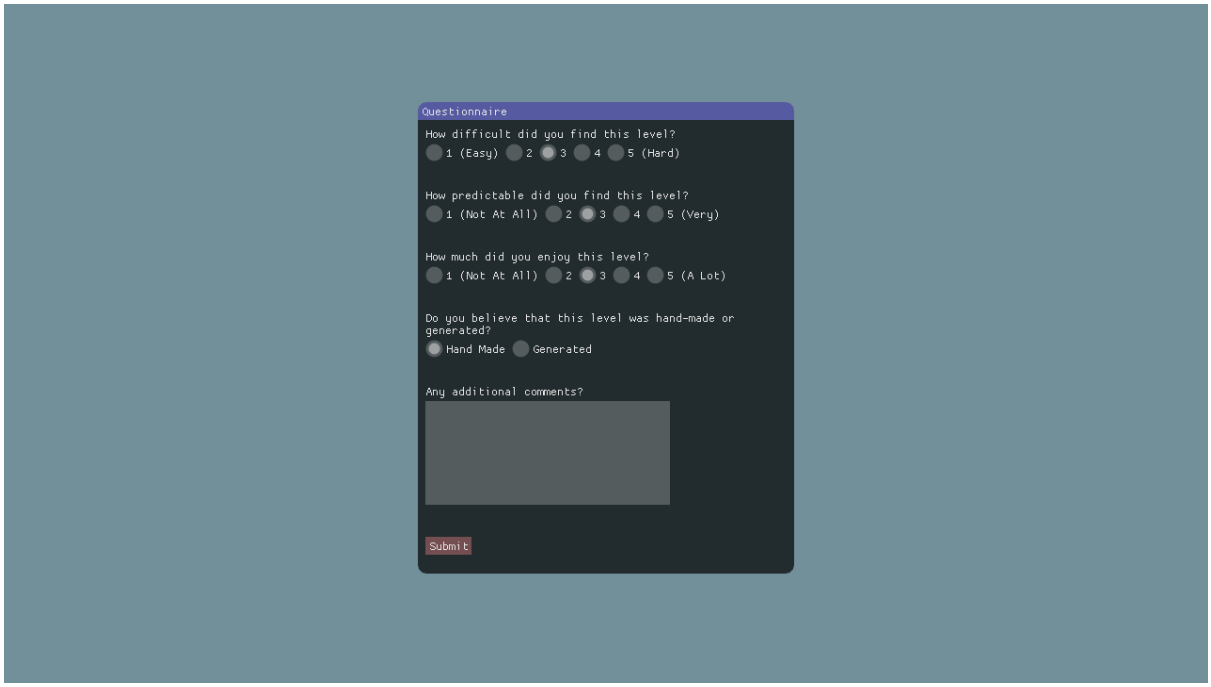


Figure 3.11: Our Questionnaire user interface as it appears in our game.

Chapter 4

Results, Analysis and Discussion

4.1 Introduction

4.1.1 Section Summary

Having discussed the design and implementation of our project, we will now present the results of both our experiment and our user case study outlined in section 3.7. We will analyse and discuss our development process, the sample of levels created by our content generator compared to our human-authored set, and the results of our study. In doing so, we will evaluate the efficiency of our solution and provide grounds for our conclusions and recommendations in the following chapter.

4.1.2 Section Structure

- **4.2 Evaluation Agent Analysis:** We begin this chapter by presenting the results of our bot training sessions, and analyse the training process. We identify several issues in our implementation, and outline potential improvements.
- **4.3 Content Generator Analysis:** In this section, we present the results of our generation sessions, and outline the successes and failures of our implementation. We discuss improvements to the termination speed.
- **4.4 Content Analysis:** After discussing the generation process, we present the results of our user testing, and use those results and our own observations to analyse our generated content.
- **4.5 Discussion on Hypothesis:** Finally, we discuss whether or not we met our aim and objectives, how successful each objective was, and conclude whether we proved or disproved our hypothesis.

4.2 Evaluation Agent Analysis

The first component of our project that we will analyse will be our Evaluation Agent. As discussed in section 3.5, our bot implementation was crucial to the overall success of our search-based PCG algorithm, as it provides the basis of our fitness function. As such, it was important that our bot was able to both learn reactive traversal behaviour

from recorded player input, and use that behaviour to reliably play through both human-authored and generated levels, without further human interaction.

4.2.1 Training Results

As we discussed in section 3.5.2, several variables control our training process, namely the number of hidden neurons, epoch number, batch size, and learning rate. In our testing, we found that the values outlined in section 3.5.2 provided the fastest convergence on optimum behaviour, taking approximately 1 minute 22 seconds to train a new bot, although the resulting behaviour was not perfect. Figure 4.1 shows a plot of the bot's normalised progress through each of our five human-authored levels at training intervals of 1000 epochs. Below 5000 epochs, the bot simply learnt to move right, and progressed only as far as the first jump or obstacle in each level. From 5000 epochs, the bot made considerably more progress across every level, although progress oscillated between seemingly optimum values until 16000 epochs, where our bot appeared to reach a local optimum. As is evident from the graph, our bot only consistently learnt how to complete level 1, and the majority of level 2. It failed to complete more than half of level 3, and less than a third of levels 4 and 5. This behaviour was observed across multiple training sessions, even when different values were used for our training parameters.

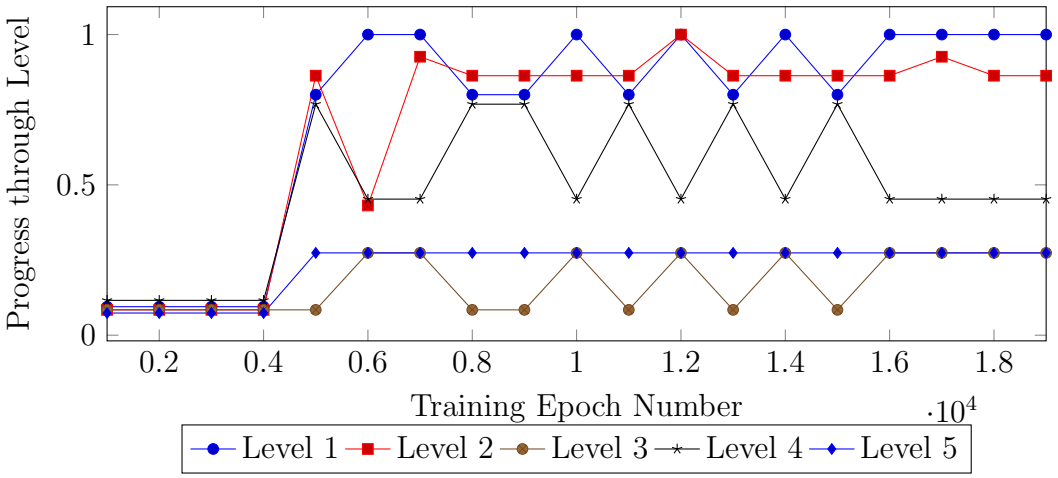


Figure 4.1: A plot of the bot's normalised progress through each level at training intervals of 1000 epochs, a batch size of 50, and a learning rate of 0.15.

4.2.2 Analysing the Training Process

The first step when analysing the training process is to consider what differed between each sample level that affected the bot's ability to complete them. All levels had the same amount of training data recorded for them, and all conformed to the design principles we outlined in section 3.4. Even so, levels 1 and 2 proved to be far easier for the bot to learn than the other three. We believe a prime factor of this is due to the level's *flow*. Both levels 1 and 2 can be completed by holding the "move right" button for the duration of the level, and timing jumps correctly. No further movement is necessary to complete them. Furthermore, these jumps are consistently sized, and either have plenty of room to land, or are spaced in such a way that the full extent of the player's jump is required. The remaining three levels, however, feature jumps which require the player

to either stop, reposition in mid-air, or make jumps which are dissimilar to others that occur far more frequently in the other level designs. This leads us to why our bot was unable to learn how to traverse every level. After analysing the structure of each level and the bot's behaviour whilst playing them, we conclude that there were three main issues with our bot implementation. These were the size of our neural network's hidden layer (complicated by limited training data), the resolution of our State vector, and our lack of button release Reactions.

4.2.2.1 Neural Network Hidden Layer Size

We believe that a greater number of neurons would be required in the hidden layer of the bot's artificial neural network in order to more successfully generalise the traversal behaviour. As outlined in section 3.5.2, the upper bound of our hidden layer is proportional to the number of training data entries, and due to time constraints, we were limited as to how much training data we could record. Two aspects of the bot's behaviour can be accredited to too few hidden neurons. The first is the bot's inability to react correctly when a minority of tiles in the State vector indicates behaviour different to the norm, and the second is the oscillation observed in training between 9000 and 16000 epochs. An example of the first scenario is a single tile "step" on an otherwise flat terrain (see figure 4.2). In this example, the bot attempts to "move right" through the step instead of jumping, and becomes stuck. This is more than likely because only a single tile (the step) in the State vector indicates that the jump reaction is required over simply moving right. More hidden neurons would be required to generalise this sort of fine control. While we initially believed that the oscillation during training was caused by using too greater value for our learning rate, we observed the same behaviour with a learning rate ten times smaller (although the training process obviously required a far greater epoch number and took much longer to complete). This led us to believe that the oscillation was also caused by too few hidden neurons. We concluded that the limited number of hidden neurons meant that there were too few weights to accurately represent the full range of behaviours, and on each back-propagation pass, the randomly selected sample set would adjust the same weights, causing oscillation in the resultant behaviour.

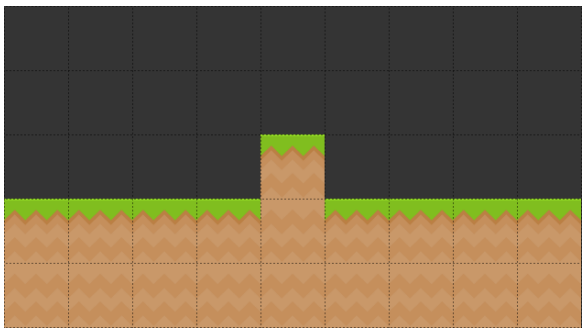


Figure 4.2: A "single step" tile placement in a generated level.

4.2.2.2 State Vector Resolution

The relatively low resolution of our State vector resulted in the introduction of error when recording, and therefore replicating, traversal behaviour. Our Reactions were limited by

the format of our State vector, which, as outlined in section 3.5.1, is a binary array indicating which tiles are passable or impassable in a radius around the player. Since our in-game character can move independently of the tile map, intricacies of the character's exact position when the player's input is recorded are lost. When the bot controller reproduces this behaviour (see section 3.5.3), the mapped Reaction is carried out immediately upon entering a tile, introducing a positional error up to the tile's size (in our prototype, 70 pixels) in each axis. This is a considerable amount for some of the finer platforming control, resulting in the bot incorrectly timing or being unable to make certain jumps. This issue could be improved by increasing the resolution of the State vector, splitting each tile into smaller quadrants. Alternatively, the player's exact position within the tile could also be recorded in the State vector, providing the bot with a finer indication of exactly when the required button press should occur. This additional State data would have an effect on the neural network, however, making it harder to train and potentially requiring much more training data.

4.2.2.3 Recording Button Release

When recording player training data, new entries are only saved when a button is pressed, or when the player moves into a new tile with a button held (see section 3.5.1). We do not record training data when the player releases a previously held button. The result of this is that no button release reactions are factored into the bot's training, and the bot therefore learns to almost exclusively hold down the "move right" button. While this is not an issue when traversing levels which only require well timed jumps (such as level 1 and 2 in our sample set), it does result in the bot misjudging jumps which require the player to adjust their speed before or during the jump. We attempted to introduce button release Reactions, but found that they caused the bot to occasionally stop completely. We judged this partly due to the error introduced by the resolution of the State vector, as discussed above, and partly due to the fact that many situations which required the player to slow down saw the player release, then almost immediately repress, the button during the same State vector. Because of the nature of our data format, the sequence that these buttons were pressed in to achieve the change in speed is lost when training the bot's neural network. A potential solution for this would be to change the Reaction data to reflect the in-game character's reactions, rather than those of the player; in this case recording the character's velocity instead of the buttons pressed to achieve it. This would of course also affect the implementation of the bot controller.

4.3 Content Generator Analysis

Despite the bot's inability to complete all of our human-authored levels, its application as our content generator's fitness function was met with success. Our generator proved capable of producing multiple generated levels, all of which were content complete and playable by a human. Figures 4.3 and 4.4 show plots of the average fitness of a population of potential levels recorded each generation during two generation sessions. Figure 4.3 shows the lower bound of our generator's generation time, displaying a session which took 20 minutes to generate a new level, whereas figure 4.4 shows the upper bound, displaying a session which took 2 hours and 14 minutes to generate. Our average generation time across a sample of 11 levels was 49 minutes. The majority of levels terminated around 30 minutes. During this time, the average fitness of each generation tended to show a

strong positive trend. In the case of levels which took upwards of an hour to generate, the average fitness tended to level out and remain fairly constant for many generations, before finally generating a level with a fitness of 1, as is the case in figure 4.4.

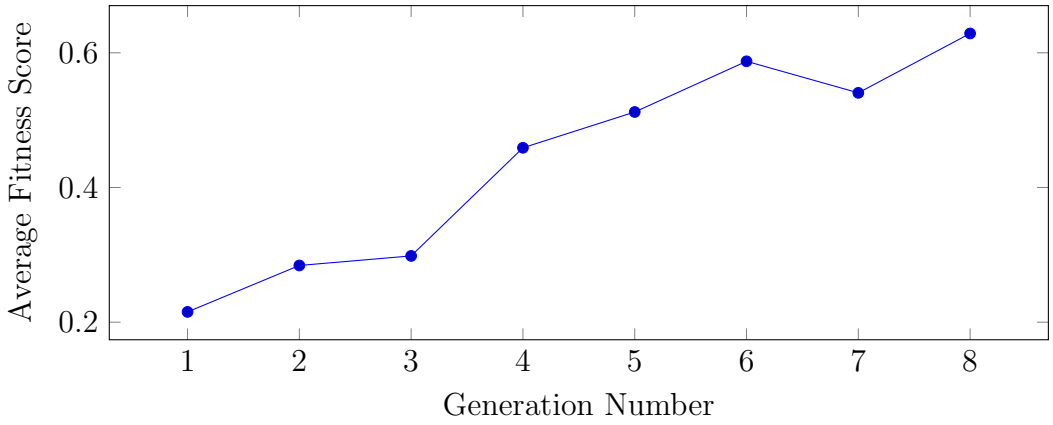


Figure 4.3: A plot of the average fitness score of each generation of chromosomes when creating a level. In this example, a level with fitness 1 was created in generation 8, and took 20 minutes to generate.

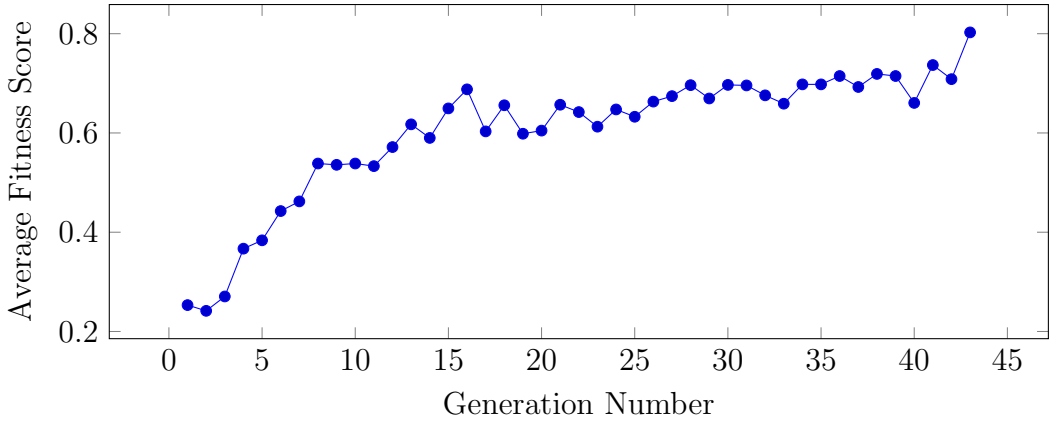


Figure 4.4: A plot of the average fitness score of each generation of chromosomes when creating a level. In this example, a level with fitness 1 was created in generation 43, and took 2 hours and 14 minutes to generate.

An average generation time of 30 minutes is far from ideal for our simple prototype game, and means that our solution is currently non-viable for use in an online environment. While our evaluation technique requiring full play-throughs of each potential level is timely in nature, we believe the technique could be improved. To understand why our generator takes so long to create a complete level, and what causes the outlying generation sessions of upwards of an hour, we must analyse and critique our implementation. One obvious contributing factor to the speed and efficiency of the generator is the evaluation bot's inability to learn how to traverse certain level features. This can result in the loss of good genetic material early on in the generation session, directly affecting how quickly the algorithm can reach termination. However, even with a fully functioning bot, there are further implementation details of our generator which should be discussed. These implementation details can be broken down into unnecessary content re-evaluation, and optimisations to cross-breeding and mutation.

4.3.1 Content Re-evaluation

A large portion of the evaluation time is wasted re-evaluating content that the bot has already successfully traversed. Early on in the development process, we realised that levels cross-bred from two instances of the same parent with no mutation did not need to be evaluated; they were exact genetic copies of their parent, and as such, could immediately be assigned their parent's fitness (see section 3.6.3). This cut down on the number of levels which needed to be evaluated each generation, especially at high generations, where the chances of a child having a single parent are greater. However, further optimisations are possible.

Since the layout of our individual level chunks stays constant, and only the sequence and interconnectivity of the chunks changes, the bot's progress through a specific chunk sequence could be cached and queried if that sequence occurs again. If we knew that the bot had been unable to traverse that sequence before, then evaluation could immediately terminate. If, however, the bot had successfully traversed the sequence, then it could be skipped, and the bot placed on the last chunk of the sequence to resume evaluation. This process could eventually be used to create a database of usable interconnected chunks, which could be queried for rapid content generation in an online system.

4.3.2 Cross-breeding and Mutation Bias

As is the case in many genetic algorithm implementations, we opted to choose randomly between either parent's genes when cross-breeding the child's chromosome, and also randomly choose which chunks would mutate based on the mutation rate (see section 3.6.3). This meant that our selection was non-biased, and every chunk from each parent had an equal chance to be selected or mutated. A potential optimisation to this process would be to bias chunks in sequences which the bot had already successfully completed, ensuring that sequences which are already playable are not mutated further, and that their individual chunks are selected together when cross-breeding. This would ensure that only chunks the bot had failed to reach would be discarded and replaced with new potential material. This technique of biasing chunks is similar to the concept of "dominant genes" in biology, where affected genes have a higher chance of passing from parent to child. This optimisation would pair well with the chunk sequence optimisation discussed in section 4.3.1, as the start of each potential level would no longer need to be evaluated every generation.

Of course, this optimisation relies on the fitness of a level being dictated purely by the bot's successful traversal distance. If more complex fitness scoring was introduced, such as considering the time taken to complete a level, or how many jumps the bot had to make, for example, then chunk sequences would also have to be assigned an individual fitness. This fitness could then be used to determine whether a sequence should be cross-bred or mutated in order to generate an overall higher level fitness.

Another factor to consider is the mutation rate itself. We observed that in sessions that took upwards of 30 minutes to generate content, many chromosomes had evolved into extremely similar patterns by around generation 15. This behaviour was responsible for the long period of relatively unchanging fitness shown in figure 4.4 between generations 16 and 42, which caused the delay in termination. This local minimum was caused by a lack of variation across the population, with only mutation introducing new genetic material. To solve this, we could introduce a scaling mutation rate, which increases as time goes on with little change in average generation fitness.

4.4 Content Analysis

Our last set of results concerns the generated level content itself, and how it was received by our test users. We will be analysing both the results of our user questionnaires, and discussing observations that we made in regards to the quality of the content produced by our generator.

4.4.1 User Testing Results

As outlined in section 3.7, we selected a sample of levels, two human-authored and three generated, and asked our 15 test users to play through each level. On completion of a level, each user was presented with a questionnaire to determine their thoughts on the level's design. Importantly, the users were not told how each level was created, and instead were asked to identify which levels they thought were generated and which were not. The levels were presented to each user out of order. We chose the levels randomly from each pool of human-authored and generated levels, although every user was asked to play the same five levels. The test user-base was made up of primarily video game players and computer science students.

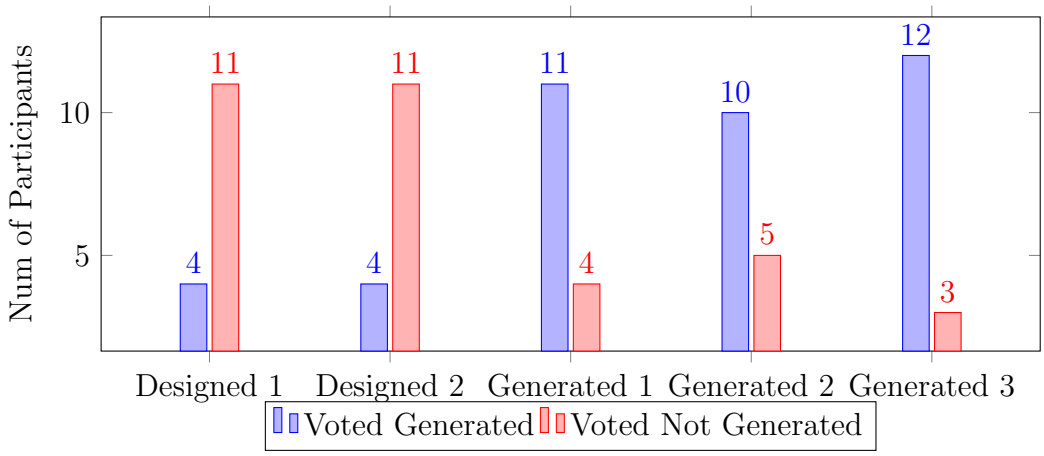


Figure 4.5: A bar chart displaying how many participants in our study correctly determined which levels were generated and which were human-authored.

Figure 4.5 displays the results of the "is level generated" vote for each level. Results showed that across our sample of five levels, the majority could correctly determine which levels were generated and which were human-authored. It should be considered these results may have been skewed by the percentage of participants who had some understanding of computer science and PCG in general. It should also be noted that the sample of 15 people may not be enough to infer any meaningful conclusions on the population as a whole. Users that fell within this category were better at correctly determining the origin of a level than users with less understanding of PCG. Even so, certain features of our generated levels were identified by all users as artefacts of our generation process, particularly in level "Generated 3". These artefacts will be discussed later in this section.

The second part of our questionnaire aimed to model the player's experience of the level. Figure 4.6 shows the results of our three player preference questions. On average, users found our human-authored levels easier and more predictable than those generated, but player enjoyment was consistent across all levels. The low difficulty and high

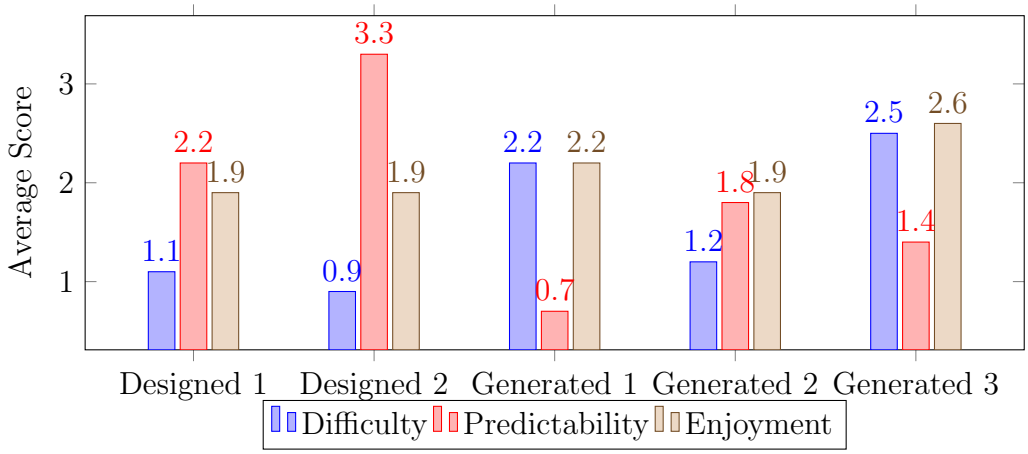


Figure 4.6: A bar chart displaying the average values of difficulty, predictability and enjoyment of each level as voted by our participants.

predictability of our designed levels was expected; as we discussed in section 4.2.2, this is more than likely due to level *flow*. These human-authored levels were designed with the timing of the player's movement and required jumps in mind, in order to create a rhythmic gameplay experience. Level flow did not appear to translate as well into our generated levels, and we will analyse why this is the case later in this section.

4.4.2 Level Rhythm and Flow

A clear observation that can be made from both the results of our user testing, and by simply comparing a generated and human-authored level side by side, is that our generated levels were far less predictable in their design. When discussing level predictability, we refer to the concept of "level rhythm and flow", outlined by Smith et al. in their 2009 paper. Smith et al. believe that level flow is a "key underlying idea behind 2D platformer level design", and describe it as that which is felt by the player when they successfully traverse a level through a series of perfectly timed and executed actions (Smith et al., 2009). While our human-authored levels were all created to promote good level flow, our generated levels frequently featured level design that required the player to stop, reposition, or make jumps which disrupted their overall rhythm. This is partly due to the generation artefacts which we will discuss in section 4.4.3, and partly due to our fitness function.

To preserve the concept of level rhythm in our generated levels, we would need to expand the evaluation logic of our bot. Instead of simply awarding fitness based on the distance that the bot successfully traversed, we could compare the time between, and quantities of, bot Reactions to those observed during a human play session, and modify the fitness accordingly. This would require us to track additional data regarding the rhythm of a player's Reactions during a training data collection session. Several studies have been carried out into collecting player rhythm data for level geneartion (Smith et al., 2009; Jennings-Teats et al., 2010), and we believe applying these processes to our evaluation bot logic could have favourable results.

4.4.3 Generation Artefacts

Across multiple generated levels, we observed two types of artefacts caused by our generation process. The first is mostly visual, odd collections of tiles or sharp drops or rises that seem out of place compared to the rest of the level design. These artefacts are the result of our chunk generation process. As we discussed in section 3.6.1, our level chunks are not designed for purpose, but rather created by splitting each of our human-authored levels into equally sized chunks. This resulted in certain designed features of each level being split across multiple chunks, which, when reassembled out of order, resulted in strange terrain. While this was an intentional step of our hypothesis to minimise the amount of additional human interaction required by the generator, visual artefacts would be reduced by using specially designed chunks. Additional logic could also be utilised to ensure the start of each chunk was positioned at the same height as the last, in order to flatten out any erroneous height changes.

The second artefact is more heavily gameplay focused, and involves the player jumping to platforms which are off screen, usually below the player's character. These "blind jumps" are not a problem for our bot, which was obviously able to complete the jump during the level's evaluation state, but they do not provide a good experience for a human player. As previously discussed, this situation could be alleviated with a more complex fitness function, which considers the distance between the start and end points of a jump to determine whether it is feasible for a player to make. In the case of blind jumps, a vertical jump distance greater than half the height of the screen would result in a penalty to the fitness score. Both types of artefacts were observed by many of our users, who highlighted their observations in the comments section of our questionnaire. Many users quoted these artefacts as their main reason for deciding that a level was generated, especially when the artefact directly affected gameplay, as with the blind jump featured in "Generated 3".

4.5 Discussion on Hypothesis

Throughout this chapter we have presented and discussed the results of our project, with the intention of either proving or disproving our hypothesis, which we outlined in section 1.3. Over the course of this paper, we have presented evidence that each of our objectives, and ultimately our aim (as discussed in section 1.2) have been met. We believe our proposed Procedural Content Generation technique, and therefore our hypothesis, to be successful, as we have demonstrated a working prototype which can generated new level content by using AI bot evaluation. Our technique not only generated levels that were playable, but did so with very little additional human interaction besides the normal human play-throughs carried out during a gameplay session, or during the testing phase of development.

Our prototype video game, which served as our problem domain, can be seen as successful. We developed a game which replicated the core gameplay observed in 2D platformer games, a genre which we outlined as a good benchmark for level generation research in section 2.8.2. The domain proved a good choice, as our bot was able to fairly successfully learn how to traverse our levels, and the 2D tile matrix format performed well when splitting our levels into chunks, and during genotype-to-phenotype mapping. Ultimately, a larger selection of human-authored levels would have been ideal, as it would have provided more variation for recording training data, resulting in more accurate bot

behaviour. Additional gameplay mechanics, such as collecting coins or avoiding enemies, would have added further requirements to our evaluation function, determining if our bot was capable of learning more complex behaviour.

Our evaluation bot implementation was fairly successful. While it was only able to successfully complete two of our five human-authored levels, it proved capable of providing an accurate distance-based fitness score for many levels in our population pool, and ultimately allowed our Search-Based PCG algorithm to successfully generate content-complete levels. As discussed throughout this chapter, there are many improvements that could be applied to our implementation to allow our bot to learn my accurate behaviour. It should be noted that the simplicity of our prototype game greatly reduced the type and amount of behaviour that our bot had to learn. Since our game's design only required the player to move or jump based on the level's terrain, and featured no backtracking or game state alteration, the bot's learnt behaviour was entirely reactive. A more complex game or environment would require further work or potentially an entirely different solution for behaviour training.

Compared to the near instant generation times of many constructive (and, indeed, some other generate-and-test) solutions, our generation technique at first appears unsuccessful. However, we would argue that with some of the improvements we have outlined applied to both our bot and our genetic algorithm, the average generation time could be dramatically reduced. Furthermore, while other faster PCG techniques exist for our chosen domain, they tend to require a great deal of development time to create tailored algorithms for the individual game. Our technique, on the other hand, provides a way of expanding on human-authored content by learning from human play sessions. The only requirements on the developer are then to provide a data format for player State to Reaction mapping and a set of training data, and either a selection of level chunks, or a method of splitting and reusing human-authored content. Of course, further research is required in order to determine whether our technique can be applied to more complex domains, and whether player behaviour can always be generalised accurately.

Chapter 5

Conclusions and Recommendations

5.1 Introduction

5.1.1 Section Summary

In this final chapter, we will be revisiting and reaching a conclusion on the success of the project as a whole. We will outline the achievement of our objectives and ultimately our aim, and conclude on whether our work proved or disproved our hypothesis. We will summarise the issues and limitations we encountered, and the limitations of the final product. These issues and limitations will provide grounds for our recommendations, and suggestions for future research.

5.1.2 Section Structure

- **5.2 Aim and Objectives Conclusion:** We begin this section by summarising the work carried out to fulfil each of our objectives, and ultimately our aim. We conclude which objectives have been met successfully.
- **5.3 Issues and Limitations:** In this section we outline several key issues and limitations across the project, and the limitations of the final product.
- **5.4 Recommendations and Future Work:** Based on the limitations we discussed, we outline a number of recommendations for future work.
- **5.5 Project Summary:** Finally, we summarise the project as a whole.

5.2 Aim and Objectives Conclusion

Throughout the project, we have worked to achieve six core objectives, which we initially outlined in section 1.2. The first was to carry out a thorough literature review into Search-Based PCG, and two machine learning techniques which we would be utilising to develop our prototype. In chapter 2, we presented the results of our literature review, starting with a detailed background of PCG and PCG techniques, particularly focusing on its application in games. We introduced the concept of the "content creation bottleneck", which we outlined as one of the driving factors behind our project's rationale. We then discussed the broader concepts of genetic algorithms and artificial neural networks, which provided background knowledge for our review of the Search-Based PCG paradigm. We

diverted briefly to discuss research into the creation of AI agents for playing games, then linked this concept with Search-Based PCG to discuss the reasoning behind our hypothesis. Following an analysis of prior attempts to generate level content through AI Evaluated Search-Based PCG, we concluded that there was worth in our proposed technique, and our chosen problem domain of 2D platform games.

Our second objective was to develop a prototype video game and a selection of human-authored levels within our problem domain. We discussed the development of our game and the design requirements we chose to enforce in section 3.4. We kept the game as simple as possible to reduce the number and types of behaviour our evaluation bot would have to learn, choosing a 2D tile matrix level data format which complimented our genetic algorithm. We discussed how our game was successful in section 4.5, outlining that it, and our designed levels, provided a good domain for our testing.

Objective three focused on developing an AI bot which would be used to evaluate content generated in our game. Our bot implementation was fairly successful; despite failing to learn how to play all five of our human-authored levels, its application as our generator's fitness function was met with success. Our bot was ultimately capable of providing accurate feedback in order to generate content-complete levels. We documented the development process of our bot in detail in section 3.5, describing how we developed our training algorithm, how we collected the required training data, and how the bot interpreted its learnt behaviour to traverse a level. To test our bot's ability to traverse our levels, we conducted a test to determine how far through each level the bot progressed at sequential training intervals. We presented and outlined the results of this test in section 4.2, and, along with a detailed analysis of our implementation, we discussed three improvements that could be made.

Our fourth objective, to develop a PCG solution using our bot to evaluate generated levels, was met with success. Our generator was capable of producing levels which were all content-complete, without using human designed rules or constraints to determine what should or should not be generated. The newly generated levels were all output to file, and could be loading into the game to allow a user to play them. In section 3.6, we described in depth how we designed and implemented our generator, our implementation of the Search-Based PCG paradigm, and how our evaluation bot was used to accurately assign a fitness score to each potential level in our level population. We analysed the content generator's performance with data collected from multiple generation sessions in section 4.3, concluding that performance could be inconsistent, and session times were too long for online content generation. We proposed that performance was affected by three major issues: the bot's inability to learn about certain level features, the generator re-evaluating content unnecessarily, and non-biased cross-breeding and mutation. We then discussed several improvements to combat these issues.

To meet objective five, we developed a testing application and questionnaire, and carried out a user test study with a sample of fifteen people. The test application aimed to determine the user's thoughts on our generated levels, by presenting them with a selection of generated and human-authored levels, and asking them to rate their experience, and attempt to distinguish which levels were generated. Our test user size and demographic unfortunately meant that we could not infer any meaningful results on the population as a whole. Within our sample, however, it was shown that the vast majority of users could tell which levels had been generated, and rated these levels with a lower predictability and higher difficulty than the human-authored ones. These results coincided with our own observations of our generated levels. In section 4.4, we discussed our findings, identifying

several areas that our implementation could be improved to provide more accurate results.

Finally, we evaluated our development process and our design and implementation decisions in order to meet objective six. This was detailed throughout chapter 4, resulting in the conclusion that we had proved our hypothesis, by developing a successful AI-Evaluated Search-Based PCG technique for level creation. We discussed the implementation of each component of our project in detail, providing data from our bot training sessions, and identified a number of areas for improvement. Recommendations were made based on these areas so that further research could be carried out in the future. With our objectives completed successfully, we can conclude that we have met our aim, which was, as outlined in section 1.2, *to develop a Search-Based Procedural Level Generator and prototype game, that uses player-trained AI Bots to evaluate generated content with minimal human-authored constraints.*

5.3 Issues and Limitations

In this section we will summarise the issues and limitations faced during the development of our solution, and that are present in the final product.

5.3.1 Number and Complexity of Designed Levels

A key part of our project relied on us producing a selection of human-authored levels, which would provide both the grounds for collecting bot training data, and the initial content for our generator to work from. Unfortunately our prototype lacked the input of an actual level designer, and as such, our levels do not represent the full potential of a 2D platformer game. Furthermore, time constraints meant that we could only produce a total of five level designs, which ultimately did not feature enough variation or mechanics to create a unique experience for each level. As a result, we have been unable to determine the full scope of levels our generator could potentially create.

5.3.2 Limited Training Data

Partially due to the limited number of human-authored levels that we had to record training data on, and partially due to time restrictions, we were only able to collect 6520 training data entries, which limited our bot's potential ability to learn (see section 4.2.2.1). A greater selection of training data would have meant we could have increased the number of neurons in the hidden layer of our bot's neural network, potentially resulting in more generalised behaviour, and ultimately a faster and more accurate evaluation process.

5.3.3 Speed of Generator Termination

Our completed generator took an average of 30 minutes to produce a level, with outliers of over 2 hours. This means that not only is the technique non-viable for online generation, it takes longer than most other offline PCG solutions. In sections 4.2 and 4.3, we analysed the implementation of our bot and generator respectively, and concluded that the long termination times were a result of both the bot's inability to learn certain level features, and issues in the generator's algorithm. We discussed potential solutions for both issues.

5.3.4 Generalising Beyond Reactive Behaviour

As a direct result of the simplicity of our prototype game, our bot only ever learnt "reactive" behaviour (see section 2.7). While this was a deliberate choice to reduce the scope of the project, it made it hard to draw conclusions as to whether a similar technique to ours could be applied to a more complex game, with, for example, backtracking, changes to the world state, or puzzles. Applications of our technique will be limited by the types of behaviour that can be generalised by an artificial neural network or other machine learning technique.

5.3.5 Level Genotype-to-Phenotype Mapping

Just as our technique requires a way of generalising player behaviour, it also requires an abstraction technique to convert the in-game phenotype to a genotype format compatible with our genetic algorithm, and vice versa. While this was a relatively simple process in our implementation, given the fact that our level data was simply a 2D matrix of tiles (see section 3.6.1), it becomes more complex in 3D environments, and in levels not designed in a grid format. Our technique for automatically creating chunks from human-authored levels in particular would be far harder to replicate in a game that was not designed with this limitation in mind.

5.4 Recommendations and Future Work

Having identified the key issues and limitations of our implementation, we will present several recommendations to improve our prototype, and outline potential future work within the scope of this paper.

5.4.1 Optimise Bot and Generator Behaviour

As discussed throughout sections 4.2 and 4.3, there is room for many improvements and optimisation of our prototype, which should reduce the generator's termination time and result in more diverse levels. Optimisations such as improving the resolution and mapping of the bot's State vector, and increasing the size of the hidden layer of the bot's neural network, could dramatically improve the bot's traversal abilities, in turn improving the generator's variety and termination speed. The generator itself could be improved by reducing the amount of content the bot has to re-evaluate, by caching successful level chunk sequences and using a lookup table to query them, should they reappear in another potential level layout. Further optimisations could also be carried out on the cross-breeding and mutation process, to bias level chunks which have already been marked as successful. These recommendations are detailed in greater depth in sections 4.2 and 4.3.

5.4.2 Introduce New Mechanics and Level Design

As mentioned in the previous section, our prototype game was designed to be simplistic in order to reduce the scope of the project. Further work is required to introduce more mechanics and varied level design into our prototype game, to determine both the extent of our bot's learning capabilities, and whether our technique performs as well with more

complex terrain. Multiple paths through a level, harmful moving enemies, collectable items, and backtracking are only a few level design features common in 2D platform games, but as of yet untested by our technique. Discussed in section 4.4.2, an improvement to the generator that should be considered is the replication of level flow. This would require building on our fitness function to evaluate not just the distance traversed by the bot, but more behavioural evaluation, such as the rhythm and timing of player Reactions.

5.4.3 Apply Generator to Different Game Domains

In addition to building on our prototype project, further work is needed to apply AI-Evaluated PCG to other games and game genres. As we stated in section 5.3.4, our technique is limited by the player behaviours that can be generalised by our bot. More research is required to determine what behaviours can accurately be learnt and reproduced from State-Reaction data, and whether those behaviours can be translated into a suitable fitness function. A good initial domain may be that of a top-down 2D grid-based game, which introduces more complex player movement and potentially more complex State data. 3D and non-grid based games should also be trialled, to determine if our technique can be expanded for games with a higher degree of freedom.

5.4.4 Towards a General Games Level Generator

We believe that our technique, if developed further, could have the potential for providing an off-the-shelf general content generation experience, such as that proposed by the General Video Game Level Generation Competition (Khalifa et al., 2016). At its core, the generator only requires three external components unique to the individual game to function: a method of recording State-Reaction mapped player behaviour, a fitness scoring equation, and a method of mapping the in-game level phenotype to a genotype, suitable for cross-breeding and mutation. With additional work, we believe there is also potential for training a bot purely through machine learning, reducing the number of unique components which would need to be tailored for each game. A general content generator could have huge implications for the industry, allowing developers to expand their game's content with minimal additional effort required.

5.5 Project Summary

We have presented a functioning prototype of AI-Evaluated Procedural Content Generation for level design in video games. Our finished technique, although not without flaws, can generate content-complete levels for a 2D platformer game similar to *Super Mario Bros.*, without human defined rules or constraints. Our technique uses the Search-Based PCG paradigm and an evaluation function provided by an AI bot, whose behaviour is learnt from recorded player input. Over the course of this paper we have discussed the design and development of our technique in detail, and presented the results of our testing. Although our generator was ultimately slow by most PCG standards, and generated levels were distinguishable from human-authored ones, all content was complete and playable. We followed up our results with an extensive evaluation of our implementation, and provided a number of key steps to improve our generator's behaviour. These optimisations ranged from improving our bot's learning capabilities to introducing selection bias to our

generator's breeding process. We believe that with these improvements implemented, our prototype's performance would improve dramatically. There are, of course, many areas of future study within the scope of this project, including applying our technique to more complex games and game genres, and working towards a General Games Level Generator. This sort of technique has a great deal of potential for the industry, as it could eventually be used to expand the content of a game with minimal additional work required from developers. The work carried out in our study is just a small step towards truly automated PCG for games, but it introduces an interesting new technique with scope for future research.

List of Figures

Bibliography

2K Games (2012), 'Borderlands 2', [Microsoft Windows, PlayStation 3, Xbox 360].

Acornsoft (1984), 'Elite', [BBC Micro, Acorn Electron].

Activision (1997), 'Quake II', [Microsoft Windows, Mac OS].

Albert, I. (2010), 'Super mario bros. maps'.
URL: *http://ian-albert.com/games/super_mario_bros_maps/*

Batchelor, J. (2014), 'Games as a service: What does it mean for indies?'.
URL: *http://www.develop-online.net/interview/games-as-a-service-what-does-it-mean-for-indies/0189880*

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mallor, S., Shwaber, K. and Sutherland, J. (2001), The Agile Manifesto, Technical report, The Agile Alliance.

Cornut, O. (2017), 'Bloat-free immediate mode graphical user interface for c++ with minimal dependencies'.
URL: *https://github.com/ocornut/imgui*

Epic Games, Inc. (2004), 'Unreal Tournament 2004', [Microsoft Windows, Mac OS, Linux].

Epyx, Inc. (1980), 'Rogue', [DOS].

Evolutionary Games (2012), 'Galactic Arms Race', [Microsoft Windows].

Hastings, E. J., Guha, R. K. and Stanley, K. O. (2009), 'Automatic content generation in the galactic arms race video game', *IEEE Transactions on Computational Intelligence and AI in Games* **1**(4), 245–263.

Hello Games (2016), 'No Man's Sky', [Microsoft Windows, PlayStation 4].

Jennings-Teats, M., Smith, G. and Wardrip-Fruin, N. (2010), Polymorph: A model for dynamic level generation, *in* 'Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment', AAAI Press, pp. 138–143.

Karpov, I. V., Schrum, J. and Miikkulainen, R. (2013), Believable bot navigation via playback of human traces, *in* 'Believable Bots', Springer, pp. 151–170.

Kazemi, D. (2013), 'Spelunky generator lessons'.
URL: *http://tinysubversions.com/spelunkyGen2/*

Kerssemakers, M., Tuxen, J., Togelius, J. and Yannakakis, G. N. (2012), A procedural procedural level generator generator, *in* 'Computational Intelligence and Games (CIG), 2012 IEEE Conference on', IEEE, pp. 335–341.

Khalifa, A., Perez-Liebana, D., Lucas, S. M. and Togelius, J. (2016), General video game level generation, *in* 'Proceedings of the 2016 on Genetic and Evolutionary Computation Conference', ACM, pp. 253–259.

Lindeijer, T. (2008), 'Tiled map editor'.
URL: *http://www.mapeditor.org/*

McCulloch, W. S. and Pitts, W. (1943), 'A logical calculus of the ideas immanent in nervous activity', *The bulletin of mathematical biophysics* **5**(4), 115–133.

McPartland, M. and Gallagher, M. (2012), Interactively training first person shooter bots, *in* 'Computational Intelligence and Games (CIG), 2012 IEEE Conference on', IEEE, pp. 132–138.

Mojang AB (2011), 'Minecraft', [Microsoft Windows].

Mossmouth (2008), 'Spelunky', [Microsoft Windows].

Nielsen, M. A. (2015), *Neural Networks and Deep Learning*, Determination Press.

Nintendo (1985), 'Super Mario Bros.', [Nintendo Entertainment System].

Nintendo (2015), 'Super Mario Maker', [Wii U].

Pedersen, C., Togelius, J. and Yannakakis, G. N. (2009), Modeling player experience in super mario bros, *in* 'Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on', IEEE, pp. 132–139.

Pedersen, C., Togelius, J. and Yannakakis, G. N. (2010), 'Modeling player experience for content creation', *IEEE Transactions on Computational Intelligence and AI in Games* **2**(1), 54–67.

Rockstar Games (2013), 'Grand Theft Auto V', [Xbox 360, PlayStation 3].

Rosenblatt, F. (1958), 'The perceptron: A probabilistic model for information storage and organization in the brain.', *Psychological review* **65**(6), 386.

Russell, S. J. and Norvig, P. (2009), *Artificial intelligence: a modern approach (3rd edition)*, Prentice Hall.

Shaker, N., Asteriadis, S., Yannakakis, G. N. and Karpouzis, K. (2013), 'Fusing visual and behavioral cues for modeling user experience in games', *IEEE transactions on cybernetics* **43**(6), 1519–1531.

Smith, G. (2015), 'Procedural content generation: An overview', *Game AI Pro 2* pp. 501–518.

Smith, G., Treanor, M., Whitehead, J. and Mateas, M. (2009), Rhythm-based level generation for 2d platformers, *in* 'Proceedings of the 4th International Conference on Foundations of Digital Games', ACM, pp. 175–182.

Smith, G. and Whitehead, J. (2010), Analyzing the expressive range of a level generator, *in* 'Proceedings of the 2010 Workshop on Procedural Content Generation in Games', ACM, p. 4.

Sorenson, N. and Pasquier, P. (2010), Towards a generic framework for automated video game level creation, *in* 'European Conference on the Applications of Evolutionary Computation', Springer, pp. 131–140.

Thurau, C., Bauckhage, C. and Sagerer, G. (2004), Imitation learning at all levels of game-ai, *in* 'Proceedings of the international conference on computer games, artificial intelligence, design and education', Vol. 5.

Togelius, J., De Nardi, R. and Lucas, S. M. (2007), Towards automatic personalised content creation for racing games, *in* 'Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on', IEEE, pp. 252–259.

Togelius, J., Justinussen, T. and Hartzen, A. (2012), Compositional procedural content generation., *in* 'PCG@ FDG', pp. 16–19.

Togelius, J. and Shaker, N. (2016), The search-based approach, *in* N. Shaker, J. Togelius and M. J. Nelson, eds, 'Procedural Content Generation in Games: A Textbook and an Overview of Current Research', Springer, pp. 17–30.

Togelius, J., Shaker, N. and Nelson, M. J. (2016), Introduction, *in* N. Shaker, J. Togelius and M. J. Nelson, eds, 'Procedural Content Generation in Games: A Textbook and an Overview of Current Research', Springer, pp. 1–15.

Togelius, J., Yannakakis, G. N., Stanley, K. O. and Browne, C. (2011), 'Search-based procedural content generation: A taxonomy and survey', *IEEE Transactions on Computational Intelligence and AI in Games* **3**(3), 172–186.

Ubisoft (2014), 'Trials Fusion', [Xbox 360, Xbox One, PlayStation 4, Microsoft Windows].

Whitley, D. (1994), 'A genetic algorithm tutorial', *Statistics and computing* **4**(2), 65–85.

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