UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE FÍSICA

PROCEDURAL CONTENT GENERATION IN A 2-D PLATFORMER

André de Melo Afonso Sistelo

Mestrado Integrado em Engenharia Física

Dissertação orientada por:
Prof. Dr. Luís Manuel Ferreira Fernandes Moniz

2022

Acknowledgments

A realização desta tese de mestrado não seria possível sem o constante apoio de múltiplas pessoas durante a minha vida académica.

Primeiramente agradeço ao Professor Luís Moniz por disponibilizar este tema de tese e por toda a paciência e apoio fornecido na forma de orientação do projeto, críticas construtivas e respostas prontas a todas as questões colocadas.

Agradeço também à Professora Guiomar Evans por possibilitar a realização desta tese e por ajudar e guiar não só a mim mas a todos os alunos do curso, que consequentemente facilitou os nossos percursos académicos.

Um obrigado a todos os meus colegas que durante estes cinco anos que contribuíram positivamente nos meus estudos e trabalhos de grupo: Luís Atayde, Nuno Taborda, Renato Alegria, Pedro Martins e Tiago Ferro. Entre estes destaco: David Pereira, Diogo Sucena, João Morgado, Matilde Santos e Romeu Abreu.

Finalmente quero fazer um agradecimento especial à minha família por toda a ajuda ao longo deste percurso. Destaco especificamente os meus pais por todo o apoio moral, por toda ajuda face múltiplos problemas e por toda a positividade fornecida. Sinto-me grato por toda a ajuda, pois sei que sem ela não estaria nas mesmas condições, por isso espero que esta tese seja digna da vossa admiração.

Abstract

Procedural content generation (PCG) is the process of automatically generating content through the use of algorithms, making it a concept that involves artificial intelligence (AI). This project suggests the construction of a Rhythm-Based Level generator with online dynamic difficulty adjustment (DDA) in a 2-D platformer, using the idea of the launchpad generator [1] as a basis. DDA is a technique that adjusts the difficulty of a game to the player's skill. The goal of this project is to contribute to the research in this area by either adding or improving methods of generating content.

To create the generator, a different approach to generate rhythms and actions is considered, where an action is defined as an input or a combination of inputs that allows the player to progress in the game, and a rhythm is viewed as a group of actions. These actions are converted into playable geometry that forms a level. An analysis of the properties of each type of action was made to ensure that the generated geometry is possible to complete and the difficulty is adequate as well as adaptable.

To validate the generated levels, an artificially intelligent player (AI agent) is used to set benchmark values for the DDA method and ensure that the difficulty of each generated level is discernible. Tests with this agent are conducted to verify the quality of the generated content. The evaluation function and geometry probabilities are constantly altered to reach results that suit the agent. The tests showed the average difficulty of the levels was converging to what was considered an adequate bound value.

Tests with real players are performed to validate the results of the agent's test and to obtain different opinions regarding the quality of the generated content, and the viability of the PCG and DDA methods. The results demonstrated that the players showed interest in the concepts explored in this study.

The project concludes with an analysis of the viability of these methods, the qualities of the work done, and what can be improved in the future.

Keywords: Procedural content generation, Dynamic difficulty adjustment, Rhythm, Action, Artificial Intelligence

Resumo

Este projeto veio no seguimento de outra tese com tema semelhante e teve o objetivo de implementar as melhorias propostas pela tese anterior. Para esta tese desenvolveu-se um gerador procedimental de níveis para um jogo de plataformas 2-D, em que a dificuldade dos níveis era adaptada consoante as habilidades do jogador durante o tempo de jogo. Um nível é fragmentado em múltiplos segmentos, sendo que os subsequentes elementos do nível eram formados à medida que o jogador progredia no jogo.

Geração procedimental de conteúdo é o processo de gerar automaticamente conteúdo através de algoritmos. Quando aplicada a video jogos, o conteúdo surge na forma de: níveis, modelos 3-D, imagens 2-D, objetos de jogo, etc. Este tipo de geração tem a vantagem de ser maioritariamente realizada por computador, reduzindo o tempo de trabalho que seria consumido a criar o respetivo conteúdo. Como o conteúdo é gerado aleatoriamente a sua qualidade está dependente das restrições colocadas no gerador, por isso a implementação deste tipo de métodos considera-se um processo complicado quando se pretende produzir bons resultados consistentemente. Como a qualidade do conteúdo gerado pode ser inconsistente, existe sempre um risco associado à sua implementação em jogos comerciais, assim este projeto pretende adicionar ou melhorar técnicas de geração procedimental de conteúdo para aumentar a viabilidade da sua execução em jogos atuais.

A técnica que permite o constante ajuste da dificuldade num jogo denomina-se ajustamento dinâmico de dificuldade. Estes tipos de métodos permitem alterar a dificuldade de um jogo consoante o progresso e as habilidades demonstradas pelo jogador, com o objetivo de tornar o jogo mais apelativo. Se um jogador sentir muita facilidade durante o jogo, a contínua exposição a este conteúdo com igual grau de dificuldade pode afectar negativamente a experiência do jogador. A mesma lógica aplica-se a conteúdo que é considerado muito difícil. Para evitar estes cenários, métodos que ajustem a dificuldade tem que ser implementados corretamente.

No projeto foi inicialmente realizada uma investigação de várias técnicas de geração procedimental de conteúdo e ajustamento dinâmico de dificuldade. Realizou-se uma análise dos diferentes tipos de técnicas e das suas propriedades, de forma a considerar várias metodologias que pudessem ser utilizadas no projeto.

Este projeto tencionou suceder o anterior, visto que o gerador de conteúdo desenvolvido tem na sua base o conceito de ritmos. Ritmos descrevem o tipo de geometria que pode ser gerada aleatoriamente num dado nível. Um ritmo é definido como um conjunto de teclas ou ações que ao serem executadas corretamente, permitem ao jogador completar o segmento do nível que é caracterizado por um conjunto de *inputs*. Quando a secção constituída pelo ritmo é concluída, um novo ritmo de dificuldade adaptada é gerado. Este ciclo contínua até o jogo terminar.

Primeiramente para criar o gerador de conteúdo, definiu-se a metodologia utilizada para a geração de ações. Nesta seção foi definido o que é uma ação, as suas propriedades, como é que uma ação varia consoante o tipo de jogo (sendo o nosso caso um jogo de plataformas), o que se deve considerar ao gerar uma instância de ação e os principais aspetos que afetam a dificuldade de uma ação. Com esta análise foi possível averiguar uma forma viável de gerar níveis para este projeto. Concluiu-se que seria benéfico gerar geometria que proporcionasse o jogador a pressionar as teclas (*inputs*) que correspondiam a essa geometria, em oposição a gerar tipos de teclas que posteriormente seriam associadas a uma geometria.

Foi realizada uma análise das mecânicas do jogo (limites físicos do jogo, e os eventos e

interações que são possíveis de executar ao pressionar conjuntos de *inputs*) para determinar os tipos de geometrias que correspondem a cada ação. Este estudo possibilitou associar a cada ação um nível de dificuldade que estava, diretamente relacionado com o conjunto de *inputs* e a forma de os executar.

Para desenvolver o método de dificuldade adaptativa foi necessário estabelecer métricas que caracterizam qualitativamente o progresso do jogador em cada ritmo. Cada parâmetro está correlacionado com o quão bem um jogador executa as ações associadas ao ritmo. Os valores obtidos em cada parâmetro permitiram avaliar a velocidade, precisão, tempo de reação e coordenação do jogador ao completar uma geometria. Cada métrica encontra-se relacionada com um peso que ajusta o impacto da variável na adaptação de dificuldade. Estas definições possibilitaram a construção de uma função de avaliação cujo valor, quantificava a qualidade da execução das ações, e consequentemente a dificuldade do ritmo seguinte.

A validação do gerador foi realizada através de testes com um agente de inteligência artificial. O agente tinha a função de obter tempos de referência para o método de dificuldade adaptativa. Ponderou-se que tipo de comportamento deveria ser usado pelo agente e concluiu-se que uma árvore de decisões seria um estilo comportamento adequado devido ao tipo de conteúdo que é gerado. O agente consegue identificar uma geometria que se encontra relacionada com uma ação, e executa-a automaticamente pressionando as teclas associadas à geometria apresentada.

Os testes com o agente consistiram na extração de dados usados na função de avaliação enquanto o agente jogava níveis criados pelo gerador com diversas dificuldades. Os objetivos destes testes foram, verificar se era possível distinguir a diferença na dificuldade dos níveis através da métricas utilizadas para avaliar a execução do agente, e se o agente convergia para um intervalo de dificuldade adequado. Teorizou-se que, para cumprir estes objetivos, o agente teria que obter valores superiores nas dificuldade mais baixas e valores gradualmente menores a medida que a dificuldade dos níveis aumentava. No entanto, para garantir que a função de avaliação atribuía valores adequados, nos testes iniciais não foi introduzida dificuldade adaptativa. Quando um teste terminava os resultados eram analisados e posteriormente eram executadas alterações nos pesos da função de avaliação e na probabilidade de gerar ações, com o objectivo de aperfeiçoar os resultados e a qualidade do conteúdo gerado.

Nos resultados dos testes foi evidenciada uma dispersão que pode ser considerada significativa, no entanto verificou-se resultados positivos que fundamentavam os argumentos e que compriam os objetivos do projeto. Esta dispersão era proveniente da aleatoriedade do gerador e do comportamento do agente.

Nos testes com dificuldade adaptativa, verificou-se que em média o agente obtinha mais progresso no jogo à medida que a dificuldade diminuía, ou seja obteve uma forma de distinguir os níveis de dificuldade. No que diz respeito a convergência do agente para um grau de dificuldade, o agente em média convergia para um intervalo de dificuldades de [5, 6.6] (os valores variam entre 0 e 10).

Para terminar a validação testou-se o gerador com jogadores reais. Um grupo de dez jogadores com experiências diversas jogou níveis criados pelo nosso gerador com e sem ajustamento dinâmico de dificuldade. Realizou-se um *quiz* para obter a opinião de cada um dos jogadores sobre a qualidade da adaptação de dificuldade, e a preferência individual entre o jogo com e sem ajuste na dificuldade. Adicionalmente foi posta em questão a relevância da reutilização dos conceitos de geração procedimental de conteúdo e dificuldade adaptativa em jogos atuais.

Os resultados do *quiz* mostraram que em geral os jogadores sentem que a adaptação foi

adequada e que ambos os conceitos de geração de níveis introduzidos no gerador são interessantes para serem explorados noutros jogos. Os poucos jogadores que discordavam com a qualidade da adaptação, tendiam para valores de dificuldades baixos (entre 0 e 2). Estas opiniões derivavam da geração de ações com dificuldades ligeiramente superiores nas dificuldades mais baixas. Embora estas geometrias exigiam uma execução de inputs que era considerada mais difícil de acordo com os parâmetros estabelecidos, estas alterações vieram no âmbito de aumentar o número de geometrias possíveis em dificuldades inferiores.

O projeto conclui-se com a análise dos resultados obtidos e uma discussão da viabilidade do uso das técnicas de geração procedimental de conteúdo e ajustamento dinâmico de dificuldade nos jogos comerciais. Finalmente o projeto termina mencionando algumas sugestões de inovações e reparos na abordagem utilizada.

Palavras-chave: Geração procedimental de conteúdo, Ajustamento dinâmico de dificuldade, Inteligência artificial (AI), Ritmo, Ação

Contents

Acknowledgments ii

Abstract iii

Resumo iv

List of Figures ix

List of Abbreviations xii

1 Introduction 1

1.1 Motivation . . . 1
1.2 Objectives . . . 2
1.3 Procedural Content Generation . . . 3
1.3.1 Taxonomies . . . 3
1.3.2 Some Approaches To PCG . . . 4
1.4 Procedural Level Generation . . . 6
1.4.1 Rhythm-Based Generation . . . 6
1.4.2 AI-Based Generation . . . 8
1.4.3 Combining Pre-Made Parts . . . 9
1.5 Dynamic Difficulty Adjustment . . . 9
1.5.1 Probabilistic . . . 9
1.5.2 Dynamic Scripting . . . 10
1.5.3 Hamlet System . . . 10
1.6 Research Considerations . . . 10
1.7 Document Structure . . . 11

2 Level Generator Implementation 12

2.1 Software Considerations . . . 12
2.2 Rhythm-Based Generation Adaptation . . . 13
2.2.1 Defining Actions . . . 13
2.2.2 Defining Rhythms . . . 23
2.2.3 Converting Rhythms Into Geometry . . . 26
2.3 Dynamic Difficulty Adjustment . . . 27
2.3.1 Defining Performance . . . 27
2.3.2 Calculating Performance . . . 29
2.3.3 Updating The Level . . . 33

2.4 Generator And DDA Considerations 33

3 Validation Of The Level Generator **35**
3.1 AI Benchmark Tools 35
3.2 AI Agent Behavior 36
3.2.1 Decision Tree 36
3.2.2 Non-Sequential Behavior 38
3.2.3 Sequences Of Inputs 39
3.2.4 Negative Aspects To Consider 41
3.3 Synopsis Of The Validation Process 43

4 Testing The Level Generator **45**
4.1 Testing With The AI Agent 45
4.1.1 Tests Without Adaptive Difficulty 46
4.1.2 Tests With Adaptable Difficulty 54
4.2 Testing With Real Players 67
4.2.1 Quiz Results And Data Comparison 68
4.3 Summary Of The Results Obtained 77

5 Conclusion **79**

Bibliography **82**

A Algorithms **84**

B Relevant Figures **85**

List of Figures

1.1 Comparison between some approaches: search based, generate-and-test and constructive approach [10]. . . . 4
1.2 Experience-driven approach's structure [12]. . . . 5
1.3 Learning-based procedure content generation (LBPCG) framework [13]. . . . 6
1.4 Representation of the various parameters of rhythms [1]. . . . 7
1.5 Representation of a rhythm that is based on two actions move and jump and has a length of 8.5 seconds [1]. . . . 7
1.6 A generated rhythm that can be associated with four possible levels [1]. . . . 8
1.7 An abstract example of a probabilistic graph showing the player's progression model in a level-based game [17]. . . . 10

2.1 Geometry made with Mario AI benchmark, that requires the player to move and jump. . . . 15
2.2 Geometry made with Mario AI benchmark that requires the player to run, move right and jump. . . . 15
2.3 Possible geometries of a jump action. . . . 17
2.4 Possible up action geometry. . . . 18
2.5 Possible duck action geometry. . . . 18
2.6 Possible move left action geometry. . . . 18
2.7 Wall jump action geometry. . . . 19
2.8 Possible run jump action geometry with different length properties making one is harder than the other. . . . 22
2.9 An example of a geometry with three highlighted cells. . . . 25
2.10 Possible run jump action geometry. . . . 32

3.1 A visual representation of the grid (the player character is in mode 0). . . . 36
3.2 Decision tree that describes the agent's main behavior. . . . 37
3.3 Game view with and without the grid. The order of the decisions in the main decision tree affects what tasks the agent will perform. . . . 38
3.4 Visual representation of the non-sequence movement decision tree. . . . 39
3.5 Run jump sequence decision tree. . . . 41
3.6 A scenario where two geometries of different actions can affect the performance of the agent. . . . 43

4.1 Box plots of Test 1's performance scores in each difficulty with the outliers. . . . 48
4.2 Average of Test 1's performance scores in each difficulty with the outliers removed. 49

4.3 Box plots of Test 2's performance scores in difficulty 5. 52
4.4 Box plots of the performance scores in each starting difficulty in Test 3 (for the first rhythm). 55
4.5 Average values of the length traveled for each starting difficulty in Test 3. 57
4.6 Box plots of the length traveled for each starting difficulty in Test 3. 57
4.7 Box plots comparing the starting and ending difficulty in Test 3. 58
4.8 Box plots of the length traveled for each starting difficulty in Test 4. 59
4.9 Average values of the length traveled for starting difficulty in Test 4. 60
4.10 Box plots comparing the starting and ending difficulty in Test 4. 61
4.11 Average values of the ending difficulty for each starting difficulties in Test 4. . . . 61
4.12 Box plots of the length traveled for each starting difficulty in Test 5. 63
4.13 Average values of the length traveled for each starting difficulty in Test 5. 64
4.14 Box plots of the length traveled in the second and final attempt in relation to the percentage total level length in Test 5. 65
4.15 Box plots comparing the starting and ending difficulty in Test 5. 66
4.16 Average values of the ending difficulty for each starting difficulties in Test 5. . . . 66
4.17 Type of answer in relation to the number of answers to question 1. 69
4.18 Type of answer in relation to the number of answers to question 2. 69
4.19 Type of answer in relation to the number of answers to question 3. 70
4.20 Player's 7 performance and difficulty in each rhythm. 70
4.21 Player's 8 performance and difficulty in each rhythm. 71
4.22 Player's 5 performance and difficulty in each rhythm. 72
4.23 Type of answer in relation to the number of answers to question 4. 73
4.24 A combination graph shows the answers to question 4 in relation to the difficulty that the players were placed in game type 2. 73
4.25 Type of answer in relation to the number of answers to question 5. 74
4.26 Player's 6 performance and difficulty in each rhythm. 75
4.27 Type of answer in relation to the number of answers to question 6. 76
4.28 Type of answer in relation to the number of answers to question 7. 77

B.1 A representation of the flow channel of a player [17]. 85
B.2 Comparison of the number of rounds played with and without DDA [18]. 85
B.3 Comparison of the time played with and without DDA [18]. 86
B.4 A Gantt chart for a Jump action. 86
B.5 A Gantt chart for a Up action. 86
B.6 A Gantt chart for a Duck action. 87
B.7 A Gantt chart for a Move Left action. 87
B.8 A Gantt chart for a Wall Jump action. 87
B.9 Average of Test 1's performance scores in each difficulty with the outliers. 88
B.10 Box plots of Test 1's performance scores in each difficulty with the outliers removed. 88
B.11 Average performance score obtained in the first rhythm of Test 3. 89
B.12 Box plots of Test 4's performance scores in each difficulty. 89
B.13 Average performance score obtained in the first rhythm of Test 4. 90
B.14 Box plots of the percentage of length travelled in the second attempt of Test 5. . 90
B.15 Box plots of the percentage of length travelled in the third attempt of Test 5. . . 91

B.16 Percentage of games won by the agent in Test 5. 91

List of Abbreviations

AI Artificial Intelligence.

CC Content Categorization.

DDA Dynamic Difficulty Adjustment.

EDPCG Experience-Driven Procedural Content Generation.

FPS First-Person Shooter.

GPE Generic Player Experience.

ICQ Initial Content Quality.

IP Individual Preference.

LBPCG Learning-Based Procedural Content Generation.

PCG Procedural Content Generation.

PDC Play-log Driven Categorization.

PLG Procedural Level Generation.

RPG Role-Playing Game.

SBPCG Search Based Procedural Content Generation.

Chapter 1

Introduction

This thesis arises from a previous project with a similar topic, in which the main goal was to create a level generator that could work in a variety of 2-D games. With this generator, it was possible to choose the difficulty of the game, and the level would be constructed accordingly using a rhythm (definition in subsection 1.4.1) to generate its geometry. A geometry is a chunk of the level composed of game objects. However, this generator created levels offline, meaning the level is built before the run time of the game. A possible improvement was proposed where these levels could be generated online, adjusting its difficulty/geometry while the player is playing the game. Another proposed improvement was a bigger range of difficulties for the game in question, allowing for a better adjustment of them to the player's skill. Therefore, the main objective of this thesis was to create an online level generator that uses rhythms to build the geometry of a level in a 2-D platformer, and contribute to the research in this area.

1.1 Motivation

Procedural content generation (PCG) is a group of methods capable of automatically generating content. This type of content generation is often applied to videogames to extend the player's playtime. Since there is more content for the players, they are inclined to spend more time playing, increasing the game's replayability. Therefore, the use of PCG in this industry allows the possibility of creating products/games where a lot of work can be done by an AI. This massively reduces the workload of videogame designers and artists, and helps create games with similar quality to those that only possess "human-made content". Therefore, by using PCG, a product with more longevity while consuming less time and money is achievable [2].

However, there are some negative aspects to PCG namely the controllability of the content produced when compared to human-made content. To have complete control of the generated content and its quality, various details need to be specified [2]. If it does not improve the player's experience then the use of PCG can be a hindrance to the success of a game, resulting in the content not meeting the player's expectations. Nonetheless, if the developers can control the generated content while ensuring its quality, PCG can have a positive influence on the player's experience enhancing the entertainment factor of the game. To sum up, PCG needs to be executed rigorously otherwise it will not benefit the final product.

The most common use of PCG in games is the offline creation of environments such as vegetation, like trees and bushes [3], but many games use PCG for other purposes. Some games use PCG to generate game objects such as levels/playable areas 3-D models and sprites, and

usable items in-game. One of the first games to introduce PCG was Rogue [4], in which all of the levels in the game were generated by an algorithm that made each one a different experience. This type of generation is what inspired various games that nowadays use PCG. When applied correctly, PCG can be an extremely useful tool to create a better and more replayable product. This has been proven by the success of multiple games like Minecraft [5] classified as the best selling videogame of all time. However, other games such as the series Civilization [6] also found success by using PCG to create environments and levels. Another famous series called Borderlands [7] also uses PCG but differently, instead of creating levels it uses an algorithm to generate a variety of weapons for the player. These examples help to identify ways of using PCG that found success in the videogame industry.

For this thesis, the main focus of this project is applying PCG to generate levels in a 2-D platformer. A platformer or a platform game is a videogame genre where the player has to maneuver a character by moving and jumping between suspended platforms of differing heights to reach the main objective of the game. To moderate the difficulty of the level, while the player is playing, a dynamic difficulty adjustment (DDA) method was implemented. These methods usually consist of the alteration of parameters and game behaviors that make a game easier or harder [8]. Thus, it allows the game to adapt its difficulty based on the player's skill set [9]. These methods guarantee that the content is neither too easy or too hard for the player (if so the player is considered to be in a flow state, a chart that represents this concept can be visualized in Figure B.1 in appendix B).

1.2 Objectives

The project had the following objectives:

- Create a rhythm-based level generator. This generator will use rhythms and their properties to recognize what type of geometry needs to be placed. To achieve a generator with these properties, these subgoals need to be met:
  - Define the actions, rhythms, and their respective properties for our game.
  - Implement a rhythm generator that allows us to create and manipulate multiple rhythms, that are then turned into playable geometry.
  - Implement a geometry generator that uses the rhythms as input to create the geometry of the level. The generated content is associated with the rhythm's properties.
- Implement online content generation with a DDA method. The generator should be able to evaluate the player's skill and generate more content in real-time based on the skill value attributed to the player (the DDA method requires us to understand and define what makes the generated content more difficult).
- Validating our generator to ensure the generated content is suitable for players. Conduct a statistical study that analyzes the results to formulate adequate conclusions.
- Test the game with real players to obtain their opinion, and based on the results, conclude if the type of content generation and DDA method used are worth revisiting in other platformers (if the players enjoyed the type of game and want to play more of it in the future).

1.3 Procedural Content Generation

1.3.1 Taxonomies

In PCG several types of algorithms vary in the way they generate content. In this segment, different aspects of these methods are summarized [10, 11]. An algorithm may possess the following properties:

- Input Parameters:
  - **Use of random seeds** – the algorithm receives a random number as input and it generates content based on that seed. If given the same seed the content generated will be the same.
  - **Use of parameter vectors** – the algorithm receives a multidimensional vector as input. Each vector contains various parameters that allow for a more controlled generation of content.
- Influence of parameters in the randomness of the content:
  - **Stochastic generation** - content generated with the same input parameters can differ.
  - **Deterministic generation** – content generated with the same input parameters will have the same result.
- Validity of the content:
  - **Constructive generation** – the content is only generated once. Before the content is finalized and put into the game, it goes through a verification process to ensure the validity of the generated content.
  - **Generate-and-test generation** – the content is first generated and evaluated by the chosen algorithm's criteria to validate the content. If the content is not valid it is discarded and regenerated. The process loops and terminates when the content is valid.
- Type of content:
  - **Necessary content** – content that needs to be generated due to it being crucial for the game to be playable.
  - **Optional content** – content that is not necessary to complete the game or a level.
- When is the content generated?
  - **Offline generation of content** – content is generated during the game's development.
  - **Online generation of content** - content is generated during the game run-time.

The objective was to create an online rhythm-based level generator with a DDA method [8] that controls the difficulty of the game as it is being played. Rhythms will be used as input parameters to generate levels that can be completed (use of parameter vectors to generate necessary content). Stochastic generation is considered because the same rhythm can generate different content.

1.3.2 Some Approaches To PCG

In the following subsections, a number of approaches to PCG are discussed [3, 10].

Search Based

Search based procedural content generation (SBPCG) is a special generate-and-test type of generation that mostly uses evolutionary algorithms. Nonetheless, other search methods such as heuristic/stochastic are also viable to generate content for games [10].

In these methods, when a candidate is generated, the algorithm not only accepts or rejects the content but also grades it using a test function. This function is denominated as a fitness function and its value defines the quality of the content. The most difficult part of this method is obtaining a good enough fitness function that grants the desired optimization for the content.

When generating content for a given candidate, the value of the previous instance (previous candidate) of the fitness function is put into consideration, meaning the next candidate is influenced by the previous. The goal of this dependency is to generate content with a higher fitness value. When the next candidate is generated, it will always differ from the previous due to a random mutation that occurs in the evolutionary algorithm. By constantly mutating candidates it is possible to create content with higher or lower fitness values than the previous. When the process ends the candidate that possesses the content with the highest fitness value is then generated.

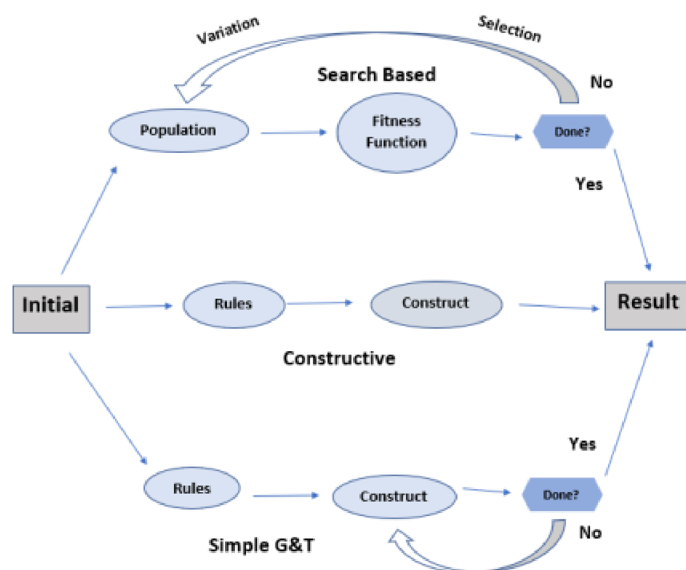


Figure 1.1: Comparison between some approaches: search based, generate-and-test and constructive approach [10].

Figure 1.1 shows a visual representation of a comparison between the search based, constructive and generate-and-test generations when it comes to the process of content creation and validation. In the constructive method, the content is only generated and evaluated once, while in the generate-and-test method the content can be evaluated multiple times before being placed into the game. Search based and generate-and-test differ in the way that content is re-generated, search based mutates the already existing population while generate-and-test creates completely new content.

Experience-Driven

Experience-driven procedural content generation (EDPCG) uses the information of a player's experiences as a tool to optimize its content [12].

In this method, the player's experience is modeled as a function of the game's content. The function describes the player by its playstyle and reactions to the gameplay, and the quality of the content is evaluated with the model of the player's experience. The content is also represented in a way that maximizes the efficiency and performance of the generator. Finally, the generator searches for the content that optimizes the player's experience according to the acquired model, and the process is repeated. In Figure 1.2 the representation of the EDPCG previous described loop can be visualized.

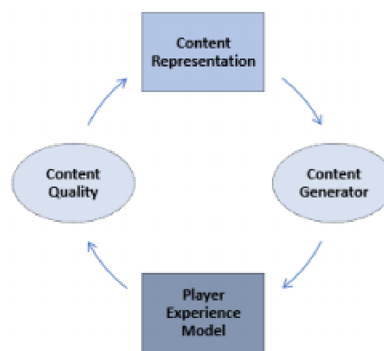


Figure 1.2: Experience-driven approach's structure [12].

Learning-Based

Machine learning is a data-driven methodology for knowledge modeling that has been applied to various fields of AI [13]. Learning-based procedural content generation (LBPCG) is a PCG framework that is based on this methodology [13].

This method uses information gathered from game developers and public testers, and it needs to generalize trained component models that allow the generation of adaptable content with minimal interruption to a player's experience. Therefore, LBPCG avoids the use of evaluation functions which limits the search to relevant content only and a minimized interface [13].

This process tries to mimic commercial videogame development and divides a videogame's life cycle into three stages [13]:

- **Development** - attempts to encode the developer's knowledge of the content by using the Initial Content Quality (ICQ) model that filters illegitimate and content of low quality,

and the Content Categorization (CC) model which separates the legitimate and acceptable content.

- **Public test** - models public player's experiences by using the Generic Player Experience (GPE) model that collects the feedback of the player, and the Play-log Driven Categorization (PDC) that models the cognitive/affective experience based on the tester's behavior and the category of game content that elicits the experience.

- **Adaptive** - generates content for the target players using the Individual Preference (IP) model that controls the generated content with the four models created in the development and the public test stages.

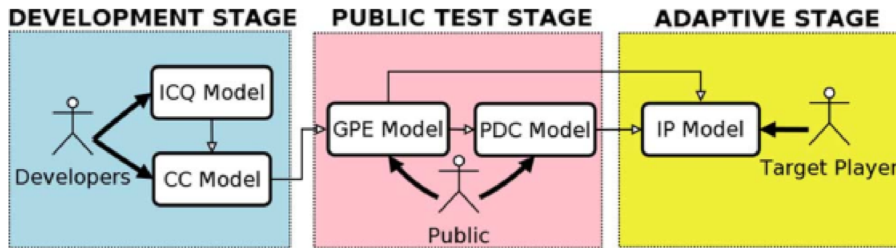


Figure 1.3: Learning-based procedure content generation (LBPCG) framework [13].

In Figure 1.3 a visualization of the game's life cycle and the interaction between the models used can be observed [13].

1.4 Procedural Level Generation

Since level generation is one of the main objectives of this thesis it is important to mention the technique that is going to be used to generate content, however other methods are also mentioned.

1.4.1 Rhythm-Based Generation

Similar to the previous thesis and the article from which this idea was based on [1], we intend to use rhythms in order to build levels. To explain this method, it is imperative to understand what a rhythm is in this context. In Launchpad, a rhythm can be described as a sequence of actions that a player can make in-game, and action is an input that allows the player to perform an event in the game that affects the player character. In this case, an input (or player input) is considered to be a key that the player can press to deliver the information of the corresponding action to the game. In this article [1], rhythms have properties that make them easy to distinguish from each other, these being: length, type, and density:

- **Length** - the time that the rhythm takes to complete.

- **Type** - the actions of a rhythm can be spaced differently along its length, this variation in space from action to action is defined as the rhythm's type. A rhythm's type can be, random, regular (even distribution), or swing.

- **Density** - defines the overall quantity of the actions in relation to the amount of time used to perform them.

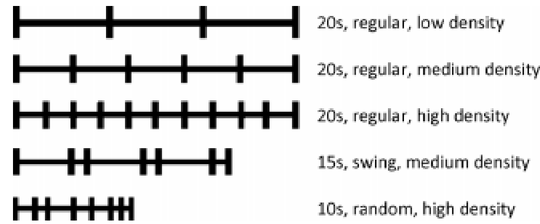


Figure 1.4: Representation of the various parameters of rhythms [1].

Figure 1.4 illustrates a representation of rhythms that simplifies their discernibility. The lines serve as a way to visualize the rhythms and their different properties. The text at the right of these lines enumerates the various parameters of each rhythm.

move	0	5
jump	2	2.25
jump	4	4.25
move	6	10
jump	6	6.5
jump	8	8.5.

Figure 1.5: Representation of a rhythm that is based on two actions move and jump and has a length of 8.5 seconds [1].

In Figure 1.5 another way of observing rhythms is shown. The image contains three columns that each represent the type of action that needs to be performed (the type of event that occurs), the action's starting point, and the endpoint in seconds. This rhythm has a length of 8.5 seconds and the player needs to perform actions of two different types move and jump. The type of rhythm in this instance seems to be regular since there is a new action every 2 seconds (the time between new actions is the same as seen in the first line of Figure 1.4).

According to [1], the rhythm-based method is a two-layered grammar-based approach to generating rhythms. This means that the process starts by generating rhythms based on the player's available actions in the game. Then these actions are associated with level geometry that is restricted based on the player character's physical constraints. The clearest way of showing its effect is with an example. In Figure 1.6 a rhythm and four possible levels can be visualized. The rhythm is represented in the same way as in Figure 1.5. By observing the rhythm and the generated levels it is possible to spot and associate each geometry with each action. For instance, at the beginning of **A**, there are three gaps that the player must jump over, these can be correlated to the three jump actions in the rhythm that initiate at 2, 4, and 6 seconds. The player is also moving during this time, meaning the move and jump action types intersect each other and serve as a means for the player to clear these initial geometries. The same can be observed in **B**, **C** and **D**, but instead of there just being gaps, it varies from small platforms, gaps, and enemies (red boxes). These geometries can be cleared by performing the same actions in the initial portion of the rhythm. After the beginning section in **A**, the next geometry is a moving platform. To clear it the player must wait 2 seconds and not input any actions, the wait time is noted due to the move action not being performed from seconds 8 to 10. The same

can be applied to the other levels with blocks that move vertically and a platform that moves horizontally. These examples show how the two-layered grammar-based approach works and how the geometry can vary even when using the same rhythm for geometry generation.

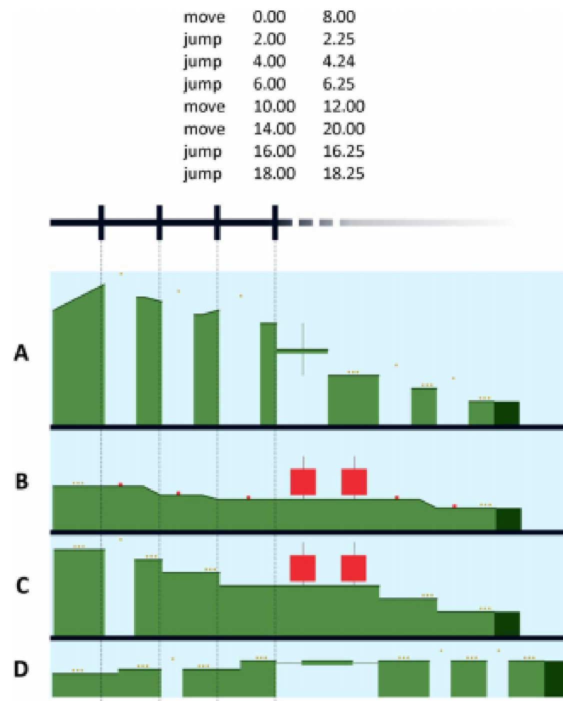


Figure 1.6: A generated rhythm that can be associated with four possible levels [1].

In our project, we intend to develop a rhythm generator that creates rhythms to generate levels with different difficulties. This means that the difficulty of a level is dependent only on the geometry produced by the generated rhythm. Therefore, according to these examples, and assuming all actions have the same difficulty, then rhythms with lower density could generate an easy level while rhythms with higher density could build harder levels. However, not all actions have the same difficulty, and thus using the density property to define what makes a level easy or hard is not a viable option.

The evaluation of the player's performance on a given level needs to be properly defined. For this, we need to implement an algorithm whose fitness function can evaluate the rhythm's difficulty and the performance of the player. Since the same rhythm can produce different geometries it is easy to assume that these levels have different difficulties, even though they were generated from the same rhythm (this issue is addressed in 2.2.1).

1.4.2 AI-Based Generation

AI-Based generation uses two types of AI in order to create a level that is possible to complete [14]. These are the level design AI and the AI player, they are responsible for creating the level and testing the level's playability respectively. Firstly, the level design AI creates a chunk of a level, then the AI player (who has all the information about the mechanics and physics of the game) confirms if the generated content is possible to complete. If the AI player checks the chunk and recognizes it as feasible then, the process repeats meaning the level design AI creates a new chunk, and the AI player tests it. If the AI player does not find the level possible, then

the level design AI creates a different new chunk to be tested. The process is repeated until the level is complete.

The randomness of the levels comes in the form of a generic design algorithm that uses the characteristics of a level as parameters to randomize and create different experiences. However, this method is known for not being efficient, for its versatility and its complexity [14].

1.4.3 Combining Pre-Made Parts

This is the most popular method of level generation due to it being the easiest. This approach uses pre-made parts (parts of levels that already exist), and combines them in a complex way to achieve a level chunk, and eventually a full level.

The method offers a decent diversity because of the number of combinations it can create. However, after an extended amount of time the human mind starts recognizing the patterns and the illusion of every level being unique quickly disappears [15].

1.5 Dynamic Difficulty Adjustment

DDA or dynamic difficulty adjustment is a technique that allows a method to adjust parameters in a game that affect its difficulty. These parameters differ based on the specific game the DDA system is implemented [16]. In this section, some DDA methods are mentioned. Most of this information can be found in this article [17].

1.5.1 Probabilistic

There have been multiple studies that used probabilistic algorithms to implement DDA methods or investigate possible improvements. The probabilistic method of the article [18] has the main objective of maximizing the player engagement throughout the game by using the modeled progression of the player on a probabilistic graph that maximizes engagement. In Figure 1.7 a visualization of a player progression model in a level-based game can be observed, where each state is represented by the blue spheres that identify the current level and the number of trials done to clear that level. The authors determine a reward variable that denotes the expected number of rounds (attempts) the player will play throughout the entire game. This reward is calculated using the probability of the player winning and losing. It is also used to determine the player's engagement and how the difficulty is going to be adapted. On the mobile level-based game, using this method, an A/B experiment was conducted with two groups of players, one played with DDA and another without. In this experiment, an analysis of the number of rounds and the time spent playing was done to compare player retention with and without DDA. The authors of [18] indicate to have achieved an increase up to 9% in playtime in relation to both groups. The results of this test can be seen in Figures B.2 and B.3 in appendix B.

Another DDA method was proposed by [19] where the authors suggested the use of parameter manipulation as a way to improve player experience in a game called Space shooter. To reach this goal a challenge function was created which uses probabilistic calculations to modify the in-game parameters that consisted of player health, weapons, behaviors, and more. A study was realized with 30 people with three different versions of a game: an easy version, a hard version, and a version with DDA. Their results showed that 60% of the players preferred the version with DDA, 10% preferred the easiest, and 30% preferred the hard version.

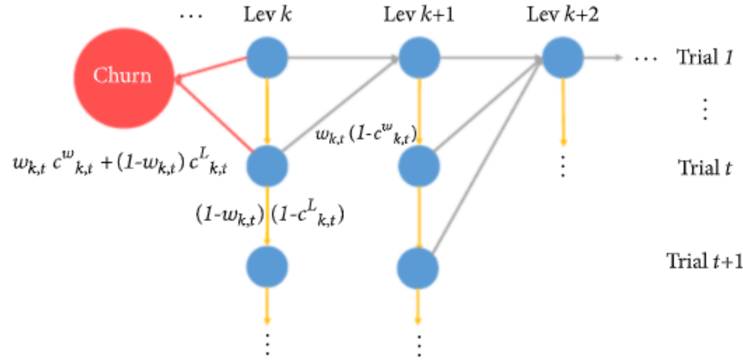


Figure 1.7: An abstract example of a probabilistic graph showing the player's progression model in a level-based game [17].

1.5.2 Dynamic Scripting

Dynamic scripting is an online unsupervised machine learning algorithm of DDA that uses rule-based to change the behaviors of enemies in a game. The viability of this method was studied by [20] and it mainly applies to RPGs (RPGs or Role-Playing Games are a game genre where generally a character undergoes a quest, with turn-based combat and a progression system that improves the abilities of the said character). The rule-based serve to create scripts that control the behaviors of each newly generated opponent, and the rules are designed manually using domain-specific information. A group of rules is extracted that corresponds to the type of opponent generated, which are then used to compose a script that controls that particular opponent. Each rule also possesses a weight that affects the probability of that rule being selected. These weights serve as a way to adapt the rule-base and are influenced by the success and failure rate of the rules that compose the script.

1.5.3 Hamlet System

Hamlet Systems built by [21] are used in games that possess an inventory mechanic, where a player can store items to aid it throughout the game. This system has been seen in games such as Half-Life [22] and it adjusts the supplies in-game, to manipulate a game's difficulty. Hamlet systems manage the game statistics in accordance with statistical metrics that are defined in advance, decide the adjustment tasks and rules, carries out those tasks and rules, present data and system settings, and created traces for playing rounds [21].

The method is constantly monitoring the information of the game and with the information, it attempts to anticipate when the player is struggling repeatedly and when it is near a state where it needs more resources.

1.6 Research Considerations

For the PCG methods researched some of them can be generalized to fit various game genres, however since this project is meant to be a sequence to the previous thesis, the PCG method that is considered to generate levels is rhythm-based.

When it comes to DDA methods most work best in specific game genres, for example, dynamic scripting is mainly applied to RPG games to control the behavior of the enemies, while

the hamlet systems can be applied to games that possess an inventory mechanic to manage the number of resources the player obtains in-game. Therefore, the best approach for this project was to create a DDA method from scratch to ensure that it fits our platform game. At the same time, a generalization of this method and what is being evaluated is done with the purpose of its implementation being possible in other game genres.

To determine what type of statistical study should be conducted, research was made to comprehend the type of tests that are usually executed and what is the most adequate for our project. When examining [18, 19, 23, 24, 25], it is clear that to understand the effects of a DDA method, A/B tests were conducted, where the results of the tests with and without DDA were analyzed to reach conclusions regarding its contribution to the player's experience. Therefore, it we decided to conduct a similar study. The idea was to test the generator with an AI agent and visualize how certain parameters of the DDA method would vary with the difficulty across multiple runs of the game and to observe if the change in difficulty was adequate for real players. To conclude the generator's validation it was imperative to test it with real players to understand how their experience varies when playing a game with and without DDA (A/B test).

1.7 Document Structure

The document structure is the following:

- **Chapter 2** - an explanation of our adaptation of the rhythm-based level generation is done by: mentioning how we define actions and rhythms, the methods used to generate them, what makes an action difficult, and how the online generation of content is applied, and the DDA method used is explained by describing what was being evaluated, what metrics were used and how much those metrics weight.

- **Chapter 3** - describes the method used to validate our generator. The method was developed in the form of an AI agent that plays the generated levels and sets relevant information for the DDA method (benchmark times).

- **Chapter 4** - a statistical study was conducted that analyzes the quality of the generator's content. Multiple tests with the AI agent explained in chapter 3 were done to ensure the discernibility of the difficulties, the adequacy of the content, and that the agent was converging to a bound value of difficulty. Afterward, the generated levels were played by real players, which provided another source of validation.

- **Chapter 5** - the conclusion of the project where we analyze the success of the implemented of the concepts in this thesis and suggest some ideas for future work.

Chapter 2

Level Generator Implementation

The main goal of this thesis is to test the viability of a game that has its levels procedurally generated using a rhythm-based method while their difficulty is altered with a dynamic difficulty adjustment (DDA) technique. To achieve this goal, we must implement these concepts in a level generator of a platformer game (since this project is a sequence to another thesis the game genre is already decided). This chapter focuses on explaining everything that was done to make our level generator.

2.1 Software Considerations

The first step in this project was deciding on what platform the level generator was going to be developed. Some ideas were considered immediately due to their easy accessibility, these were:

- **Mario AI Benchmark** - this is a benchmark software based on Infinite Mario Bros [26] that was used for an AI Competition [27] and already has a built-in generator that can be modified for this project [28]. The generator uses seeds to generate random levels with functions that can create geometries to fill the levels. The benchmark is a good option since, we want to create a generator that uses rhythms and converts them into geometry, and having many tools at our disposal facilitates this process.

- **Pygame** - alternatively an already existing game made in Python with the library Pygame could be used. If this option was chosen and no games that suited the project were found, then we would have to create our game engine from scratch which could take a long time to develop.

- **Unity** - Unity is a free game engine that allows anyone to make games. It makes various tasks easier when it comes to game development. However, to use this platform with the level of dept that might be needed for this project requires a lot of knowledge on how to use it properly, which takes more time than we have for this thesis. The project had an initial deadline of six months, making this option not viable.

For the reasons previously listed, we decided to use the Mario AI Benchmark. This benchmark already had a basis for our generator that was easy to modify. The built-in generator can create a random level by changing multiple parameters with a given random seed. However, the goal of this project is to implement a generator that uses rhythms as an input and converts

these into a level where its geometry matches that of the rhythm. Thus, we made alterations to the Benchmark's generator in order to achieve our goals. Meaning that the same Launchpad generator's two-tiered grammar-based method [1] was used to generate our levels. Even though the same concept is being used, some changes to the definitions of actions and rhythms were made they fit our project.

A DDA method was also implemented in the generator with the goal of adapting the difficulty of the game to the player's skill, thus turning the generated levels more interesting and fun for all players regardless of their previous experience with videogames. To implement the DDA method the actions' properties and what makes an action difficult must be defined as well.

The benchmark possesses level templates that can be filled and altered to create levels with the desired geometry. First, it is imperative to understand how the templates classify the space. Every level is constructed with two variables that determine its space in length and height. Considering a 2-D Euclidean space the level's length determines how long the level can go on its x axis (horizontal axis), while the height determines the size of the level's y axis (vertical axis). Length and height are the number of cells a level has in each line and column of the template respectively. Each cell is composed of 256 pixels in a shape of a square, with 16 pixels being the value of the sides. This information is relevant to classify the properties of actions and rhythms. The goal of the level generator is to fill the cells of the template with geometry that represents the rhythms that reflect the player's skill.

2.2 Rhythm-Based Generation Adaptation

2.2.1 Defining Actions

Actions are considered inputs or buttons that the player uses to perform events that allow it to progress in the game, and in this case complete levels. We consider that the player is progressing in a game when it is getting closer to achieving the end goal of the game. Although this goal varies from game to game, for this project the goal is finishing a level.

When considering the generation of instances of action, it is important to be able to distinguish different actions and know what geometries can be assigned to them. This creates a need to establish properties for each instance of action. These properties can be based on the example in Launchpad generator [1]. In Launchpad the different properties of actions that can be discernible are the following:

- **The type of action** - the event in-game that occurs by making the corresponding input.
- **The initial position where the action occurs** - the position where the action begins, normally represented in a euclidean space with (x, y) coordinates.
- **The duration of the action** - the time the corresponding input is read as true.

The type of action is highly dependent on the type of game that is being considered. In a platformer, a player might be capable of moving and jumping; in an FPS (First-person shooter is a shooter videogame from a first-person perspective) (shooters are a game genre that revolves around gun-based combat) a player might be able to shoot a weapon and move, while in a turn-based RPG a player might be able to command an attack on an enemy. By using the established Launchpad generator's [1] definition of actions, it can be inferred that even when considering

different game genres with non-identical styles of play, an action is still defined as an input that when performed creates an event in the game.

In this project, the focus will be on a platformer. Given most 2-D Mario games ([26]) as an example, the player can use singular inputs to move left and right, jump, run and duck. This move-set also applies to the benchmark we are using. In addition, an up input is considered to complete certain types of geometry (a geometry is considered completed when a player performs the correct inputs while in that geometry and progresses in the level).

Action Generation Considerations

When it comes to the generation of instances of action, it is necessary to understand all the actions that can be performed and what they can achieve in-game. Meaning it is possible to recognize how much progress each action can give to the player in every geometry. As mentioned before, in-game, progress is achieved when the player completes a geometry and gets closer to finishing the level. Depending on the geometry that is being considered, an input can have a positive or negative impact on progress. In most scenarios, for a player to advance in a level the player must combine certain complementary inputs (for instance, the move and jump inputs can be combined to complete some geometries). This means that, depending on the game genre, different inputs affect the player's progress differently making the generation of some individual inputs not beneficial. For example, if the goal of the game is to reach the right border of the level, then an input that makes the player move left without it being necessary to progress is redundant. Therefore, by identifying what actions can give the player progress, a significant number of inputs and input combinations that do not need to be generated can be disregarded.

To create the generator, the properties of the generated instances of action need to be considered. This information simplifies the process of choosing the type of geometry and where to place it in the level. However, this process can be complicated when dealing with groups of actions that possess simultaneous inputs. The authors of the Launchpad rhythm-based generator [1] generated instances of action as inputs, and so when two actions are generated their starting times differ. However, in some scenarios, actions can have starting times that can occur while another action is still being performed, which is what we consider simultaneous or overlapping inputs.

As a result, it is relevant to consider not only the position and duration of that action but also what other actions can be performed at the same time. To analyze how to deal with the issue of overlapping inputs, an analysis of the possible geometries that frequently appear in this type of game genre was conducted. In 2-D Mario games ([26]), there are geometries in a level where the player is required to perform a jump over a gap, implying that at one point in time, the player must press the corresponding move and jump button at the same time to complete that geometry. In Figure 2.1 an example of this gap geometry constructed with the Mario AI benchmark can be observed. To complete this geometry, the player is required to press both the move right and jump buttons simultaneously. It is possible to complete this geometry with these inputs as shown in [b].

In this example, if the length of the gap is extended, it adds a third action and another requirement for the player. The player now needs to run, move right, and perform a jump to complete the geometry with the added momentum the character carries. In Figure 2.2 the example of a geometry with a wider gap than in Figure 2.1 can be observed. The increased speed

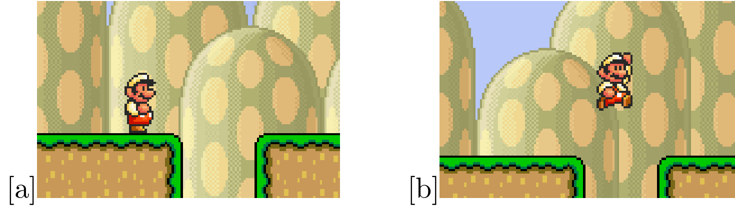


Figure 2.1: Geometry made with Mario AI benchmark, that requires the player to move and jump.

from the run input allows the player to complete the geometry as shown in [b]. Otherwise, if this combination of inputs is not performed it is impossible to progress in the game.

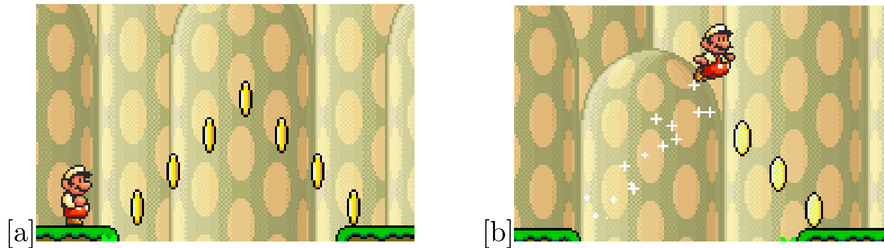


Figure 2.2: Geometry made with Mario AI benchmark that requires the player to run, move right and jump.

However, adding inputs can also negatively impact player progress, if a duck action is added to the previously mentioned sequence, the player is no longer able to complete the same geometry because the geometry of Figure 2.2 can only be completed by that sequence of inputs. Thus, it is possible to associate that geometry with the three previous inputs.

Some combinations of inputs create scenarios where there exists no geometry that can be associated with the combination and contribute to the player's progress. For example, if the player inputs to move right and left at the same time, the player character stops moving. Even though there are instances where this could be useful, for example, an action where the player must wait and stop his movement, the same can be accomplished by not inputting any actions at the correct time.

The initial idea for the project was to generate the inputs that constitute a rhythm and then generate the geometry accordingly, however, with this method, some issues start to arise. The resulting geometry that is associated with the inputs needs to be possible to complete. To solve this problem, there was a need to apply various constraints to our input generation. Even so, it was necessary to examine the density and the rate of each generated action. In most levels of 2-D Mario games ([26]), the end of the level is usually located at the most right part of the level (which is considered to be our main goal), meaning that if the goal of the game is to reach this end location, most of the generated inputs would have to be of type move right. Even if this was feasible, there was still a need to adjust the types of geometries presented based on the player's skill. Making it more complicated to modify the rate and density of the generated inputs, while trying to simultaneously create easy or hard geometries based only on inputs. To achieve this, it would require defining a different set of constraints in action probability and duration for each input and each level of difficulty. For these reasons, other methods were put into consideration.

Our Adaptation Of Action Generation

To guarantee that undesired sequences of actions were not generated in our rhythms, a simple change was proposed. Instead of generating each input individually, it was more practical to isolate each type of geometry and associate them with a specific combination of inputs. This facilitated the generation due to there being no need to determine the input density and duration. This solution also solved the problem of the generated geometry not being possible to complete, since it was easy to guarantee that each geometry was feasible (to complete) due to our understanding of the game's mechanics and the player character's physical limitations. In our case, the biggest concern was ensuring that the player character can reach the end of the level given its move-set, by being able to complete geometries with big gaps and high platforms. Finally, there was a need to ensure that the generated geometries require the player to perform specific input combinations that are vital to the geometry's completion. This way it simplified the differentiation of actions while increasing input variety in the gameplay.

Nevertheless, there will always be scenarios where inputs can be used without necessity and the player is still able to complete the geometry. An example of this type of geometry would be the one in Figure 2.1, where the player can move right and jump over the gap, yet a run input can be added even if it is not required. In the geometry of Figure 2.2 the player is required to input move right, run, and jump to complete the geometry. These geometries differ by the fact that they both require a move right and jump input, but only one requires a run input. Since one geometry requires the player to press and additional input, we can discern these as two separate actions even though the geometries are similar in appearance.

Since the association of geometries with input combinations was being considered, then a redefinition of action was required. In Launchpad [1] an action is considered to be an input that allows the player to perform an event in the game. However, for this project, an action is defined as an input or a combination of inputs that the player must perform to progress in the game. With this redefinition, the issue of generating undesired and overlapping inputs was solved due to an action being able to contain multiple inputs that are required for the player to achieve progress.

The next step was to search for geometries that could be associated with an individual or combination of inputs. To determine these correlations, as mentioned previously, a fundamental understanding of the game's mechanics, the character's physical limitations (more specifically the character's horizontal and vertical speed and acceleration), and the available geometry was required. By comprehending the character's move-set it was possible to associate combinations of inputs with events in the game and correlate these events with a set of geometries.

This way the number of possible geometries is only limited by the move-set of the player character and the game's mechanics, meaning that the only downside to this approach is the possibility of not having enough variety in geometries for every type of input combination. If there is a low diversity of geometries in a level, then presumably the game can get stale similarly to the issue of the combining pre-made parts method of level generation explained in subsection 1.4.3.

After examining the available tools in the benchmark, it was decided that this approach would be used to generate actions, due to it being feasible to create a variety of scenarios where the player would be forced to use the input combination associated with the geometry of that action. These combinations of inputs can be discerned by the different necessary inputs that the

player is forced to use to complete a geometry. Meaning that these necessary inputs are what characterize the type of action of each instance. If the player chooses to not use these inputs, the geometry is either impossible to complete or the player is penalized.

The action types created are identified by their necessary inputs, thus some actions that allow the player to progress in the level are the following:

1. **Move Right** - the type of action that corresponds to geometry that only requires the player to input moving right.

2. **Jump** - the type of action that correlates to the combination of a move right and jump inputs.

3. **Run Jump** - the type of action that requires the combination of a run input, a moving right input, and a jump input.

4. **Up** - the type of action that corresponds to the input of going up a ladder.

5. **Duck** - the type of action that corresponds to any combination of inputs that involve the player ducking are contained can be generated from this type.

6. **Move Left** - the type of action that generates geometries that require the player to move left. Inputs other than move left need to be present due to the position of the end goal.

7. **Wall Jump** - this type of action requires a wide variety of necessary inputs to be performed. This action type generates geometry that allows the player character to jump off a wall. The inputs required are the run, jump, and move left and right inputs. An example in detail is analyzed in this subsection.

8. **Wait** - the type of action that requires the player to make no inputs for a short period to achieve the most favorable outcome. However, due to the nature of the end goal, other inputs need to be used to progress.

Converting Actions To Level Geometry

After determining the types of actions that could be generated, there was a need to convert these instances of actions into playable geometries. Our idea was to generate instances of action and determine the corresponding geometry based on their properties.

In the examples of Figures 2.3 to 2.7, some of the possible geometries that can be generated with these types of actions can be observed. In addition, examples of possible Gantt charts for each action can be visualized in Figures B.4 to B.8 in appendix B.

Figure 2.3: Possible geometries of a jump action.

In Figure 2.3, examples of possible geometries for a jump action and the action being performed are visible. The action is performed with two inputs, move right and jump. In [a] there

is a simple platform with a higher elevation. In [b] and [c] the player is shown jumping over enemies with the same inputs. In the case of [a] and [b], the player is required to input a jump to proceed with the game due to an object blocking its path, yet in [c] its path is not blocked. Theoretically, it is not imperative for the player to use the jump action to process in the level, however in this scenario, if the jump input is not used the player is punished and can lose the game.

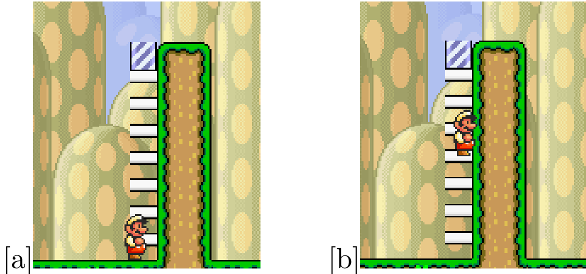


Figure 2.4: Possible up action geometry.

The images shown in Figure 2.4 represent a possible geometry for an up action and the action being performed with one input, move up. Figure 2.4 [a] and [b] show the player going up a ladder by pressing its corresponding input.

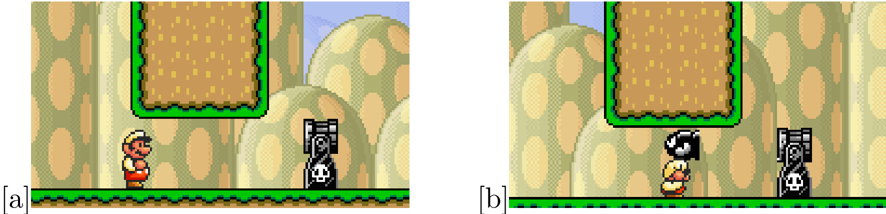


Figure 2.5: Possible duck action geometry.

In Figure 2.5, the images represent a possible geometry for a duck action and the action being performed. To complete the chunk of the level, the player should duck and avoid the incoming enemy. This geometry shows that if the player does not perform the corresponding action, the game will punish it by taking damage or ending the game.

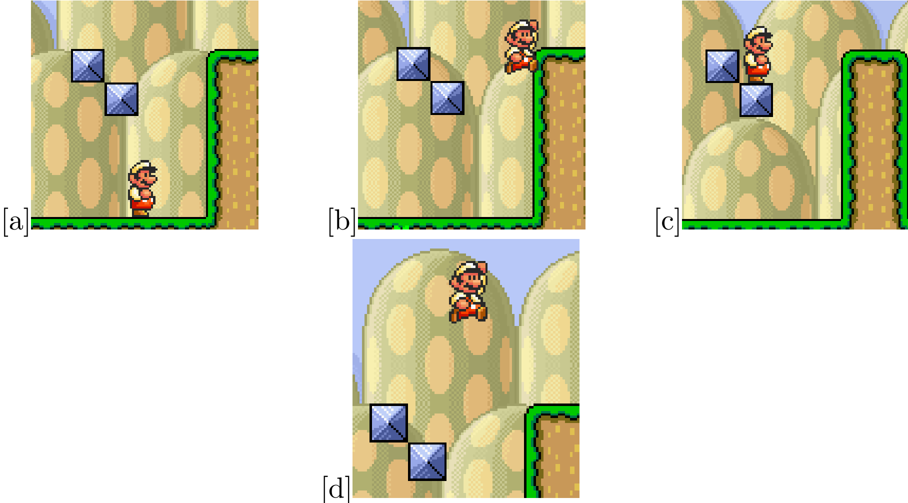


Figure 2.6: Possible move left action geometry.

The images in Figure 2.6 reveal a possible representation of the geometry generated by the move left action. Image [a] shows the beginning position of the action. To complete the left action geometry, the player must move left and jump onto the platform as shown in [c]. Afterward to progress in the level the player is required to perform another jump and move right as seen in [d]. Image [b] reveals what occurs if the player tries to complete this geometry without moving left. The player character has physical limitations that do not allow the player to jump and complete the geometry without moving left.

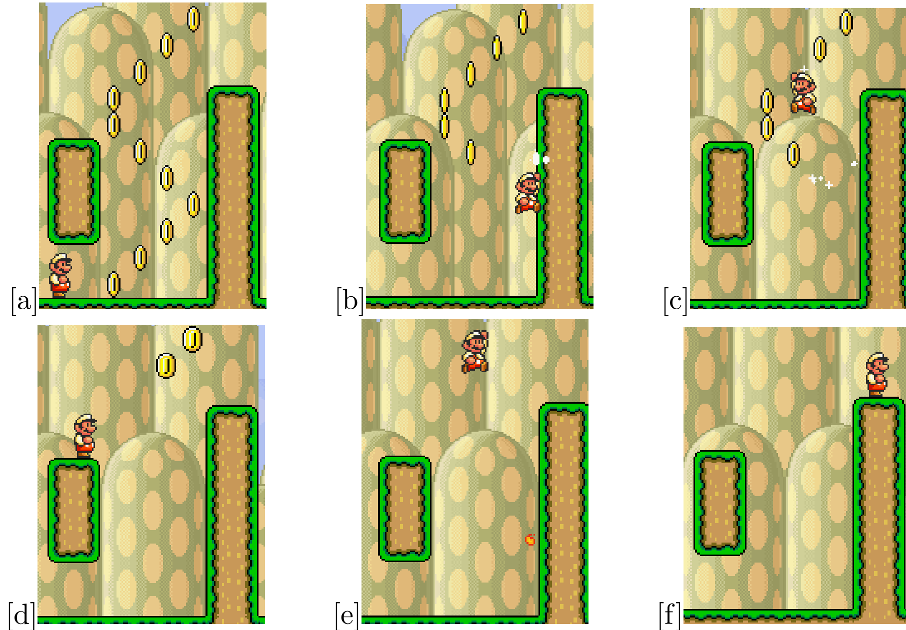


Figure 2.7: Wall jump action geometry.

The images in Figure 2.7 represent a possible geometry for a wall jump action. Image [a] shows the beginning position of the action. To complete this geometry the player must move right while running and then jump onto the wall as shown in [b]. While on the wall the player is required to hold the move right and run button and let go of the jump button. While holding the previous inputs, the player needs to press the jump button again and change directions (done by using a move left input, without inputting to move right). If executed correctly, this sequence allows the player to jump off the wall as seen in [c] to reach the platform as shown in [d]. To complete the geometry the player needs to reach the top of the left platform. Achieving this can be done with a simpler run jump combination, as seen in [e] and [f]. Since this is by far one of the hardest actions, it will be left for the highest difficulties. The game's mechanics allow this type of geometry to be feasible to complete, and so it was decided that this type of action would be viable to use in our generator due to it providing more difficult geometries that can be later used on the highest difficulties when the DDA method is implemented.

Accessing Action Information

The simplest way to differentiate instances of actions is by their properties. Since a redefinition of action was done, it was imperative to also redefine the properties of these instances. The following parameters define the properties of an action and contain the information necessary to generate instances of action that can be converted into geometry:

- **Type** - the type of geometry that is associated with one or a combination of necessary inputs.
- **Length** - how much of the level does the generated geometry occupy in cells.
- **Height** - the position where the geometry of the action ends (ends in the last horizontal cell) in the vertical axis.
- **Difficulty** - the difficulty of an action determines how easy or hard the type of action is going to be.

Through these properties, the information required to convert an instance of action into a playable geometry can be accessed. When compared to the action properties set by [1] it is clear that these new properties do not contain the action's duration. This is due to the instances of action being considered a combination of inputs that can vary in duration. Initially, it was thought that, in most cases, the length of the geometry could dictate the duration of the action, because it would be simple to calculate the action's duration based on the length of the geometry and the velocity and acceleration of our player character in geometries that only require the player to move in one direction. Nonetheless, there are problems with this approach. Firstly, the player is not always moving in the same direction, for instance, in the actions type move left and wall jump this method would be slightly more complex but still feasible. This complexity stems from the player having to change its direction multiple times. Another issue with this method is that the values of action duration would be used as a reference for real players (when applied to the DDA method), thus this method would be considering that players can obtain the best times in each geometry which is not feasible. Therefore, to better suit the adaptation of difficulty, this method would require an alteration of the duration of each geometry, thus determining when the player should increase or decrease its speed and at what rate. To solve this problem, it was decided that an AI agent would play every geometry isolated, and then the times the agent achieves (moving in both directions) would be used as a reference for the action duration property.

Determining Action Difficulty

The difficulty of performing every action is different and needed to be clearly defined if one of the goals was to apply a DDA method to the generated levels.

One method of discerning the difficulty of two actions is to use the rhythm definition, which infers that the action/geometry that requires the most number of inputs (more density in a rhythm) is harder. However, more metrics for determining an action's difficulty could be defined to obtain a more concrete group of properties that would later help with the DDA method. Using the number of inputs as the only metric to determine an action's difficulty is not the best option available.

A starting point for this analysis would be to consider the how difficult required to perform the correct sequence of inputs of an action. For this project, we consider the difficulty to be how arduous an action can be to perform for a given player. A player can struggle with input combinations due to various factors, the ones that were identified in this project are the following: the coordination needed to perform the action, the speed at which the action is performed, the precision of the inputs (the ability to press an input in a specific time frame), and the reaction

time of the player. The difficulty of a geometry is not associated with the punishment attributed to the player if it fails to perform it. For example, the geometry generated by the wall jump action type (visible in Figure 2.7) can be attempted by the player as many times as necessary without the player being punished (in terms of losing the game it can only happen if the time runs out). However, to complete this action, the player is required to press multiple types of different inputs and press inputs simultaneously (which requires some level of coordination), have the timing to jump off the wall (has to have good precision and reaction time or it falls) and press the inputs with the correct duration to not overshoot platforms.

Alternatively, a jump action type (as shown in Figure 2.1) can generate a hole in the ground that can be fatal if the inputs are not performed correctly (the player loses the game). When comparing this geometry to the one in Figure 2.7 the number of inputs required to complete each geometry differs drastically. In the geometry of Figure 2.1, the player needs to press fewer simultaneous inputs with a smaller duration (smaller jump compared to the previous wall jump action), and the timing required is correlated to the length of the gap, thus there is a clear difference in the difficulty.

Another example could be of two jump actions, one where a hole is generated and another where a platform with higher elevation is generated. Considering that, in these scenarios, the length and height of the required jump are the same, both actions demand an equal number of inputs (being two move right and jump) and duration, yet they differ in the fact that one of the generated geometries can end the game. This could imply that the player needs to have good timing to complete the geometry with the hole, meaning that the difficulty of the jump needed to complete the geometries is very similar. This signifies that these actions should be considered close when it comes to the level of difficulty, even if one is slightly harder.

With these examples, it is possible to comprehend that the difficulty of an action is not related to the punishment the player receives when failing to perform the action.

Some of the aspects that modify the difficulty of an action were identified in the previous examples, these included the number of inputs, simultaneous inputs, timing, and duration. By altering at least one of these characteristics, in one geometry type, it is possible to create level chunks with various degrees of difficulty. For instance, the duration of an input can be altered to increase an action's difficulty and creates a new geometry. Given the example of a run jump action where a huge gap is generated and the player is required to move right, run and jump to complete the gap. If two gaps with different length values are considered, then in the larger gap, the duration of the inputs is significantly higher than in the smaller gap. In this example, by increasing the length of the gap, the reaction time and the degree of precision needed to perform the action correctly also increases.

In Figure 2.8 there are two different gaps in [a] and [b] where one possesses a greater length than the other. These actions differ notably, in one the player does not need to be as precise, meaning it can jump earlier and still complete the geometry (Figure 2.8 [a]). Therefore, the characteristics that were listed (duration and timing) are indirectly connected through these geometries, since creating a geometry where the duration of inputs is increased influences the timing required to complete the geometry. In [b], if the player jumps as early as in [a], it can fail the geometry due to the gap being larger, signifying that for example, if the player is moving at maximum speed, there is a different interval of time, from [a] to [b], or area of the platform from which the player can perform a jump and complete the gap. The interval is more forgiving in [a] making this action easier.

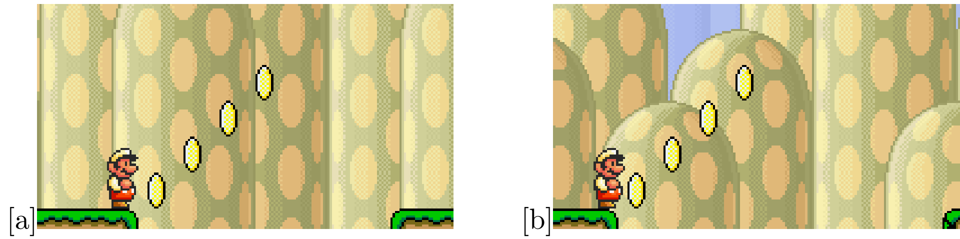


Figure 2.8: Possible run jump action geometry with different length properties making one is harder than the other.

The same logic can be applied to our move left action, the player is not punished for failing the jump in that geometry, but by increasing the length of the gap, the player's timing becomes key to succeeding in variants of geometries in the highest difficulties. This process of altering properties can be repeated for all action types expanding the number of total geometries with different difficulties.

However, there are some types of geometries that possess a different property than all the others. When considering the geometries that generate enemies, all the previous metrics can be used to evaluate the difficulty of these actions. However, the inputs that are required in these geometries constantly change with time due to the enemies changing their positions in each frame. For example, if an enemy is closer the duration of the jump input does not need to be as long when compared to an enemy that is farther away. The timing and types of inputs also differ with the enemy's current position, if the enemy collides with the player and the game continues, then the player needs to move left to defeat the enemy. In our generator, the number of enemies and their speed increases with the difficulties. The goal of this change is to alter the values of the difficulty's parameters to better suit higher levels of difficulty. More enemies implies an increase in the number of inputs (more jumps to be performed), and the increase in their speed, accelerates the process of inputs changing with time and decreases the time the player has to complete it successfully. For these reasons, the actions that generate enemies can be considered one of the hardest types of possible geometries.

In conclusion, the difficulty of actions can be distinguish by multiple parameters:

1. **Number of inputs** - how many buttons (including repetitions of the same input) does the player need to press to complete the geometry.

2. **Duration of inputs** - how much time does the player needs to hold certain buttons. The higher the duration does not necessarily indicate that the action is harder. Implying this metric that requires context to be correctly evaluated. The duration of the inputs can be affected over time (geometries with enemies). If the duration is modified the timing required for the player to complete a geometry can also be affected.

3. **Timing of the inputs** - how precise does the player need to be with his inputs to succeed.

4. **Amount of simultaneous inputs** - how any inputs are required to be used by the player at the same time.

5. **Influence of time in the inputs** - how does time influence the types of inputs in a given geometry.

By using these parameters as a basis, the difficulty of the created geometries can be discerned and associated with a certain degree of difficulty. A level can possess an action with slightly higher or the same difficulty as the rhythm they belong to, for example, the easiest version of the jump and run jump actions can appear in the same difficulty, even though one is easier than the other. This is done to increase the number of geometries in lower difficulties and reduce the monotony of the game (not using the same inputs constantly). Since the mechanics of the game are not very complex (only requiring six inputs to play), the number of input combinations that are associated with lower difficulties is smaller. To exemplify this association, consider the example of a wall jump action (Figure 2.7) and a move left action (Figure 2.6). In this case, the geometry generated by a wall jump action requires more inputs and more simultaneous inputs, thus this geometry is associated with a higher difficulty than the move left action. By applying this method to all action types and their variants, we intend to increase the engagement of multiple players with varied experiences.

Originally the range of the difficulties went from [0; 5], where 0 is the easiest and 5 is the hardest. Each difficulty also had its types of geometries, meaning that, in different difficulties, no geometry would be the same. Nevertheless, it was decided that the interval of difficulties would be extended to [0; 10], and the difficulties would be separated in pairs, and thus these would have similar geometries, but different enemy movement speed values. For example, difficulties 5 and 6 have similar geometries but the enemies in difficulty 6 move much faster and therefore are harder to deal with.

2.2.2 Defining Rhythms

The authors of [1] define a rhythm as a set of actions that the player must complete in order to complete a level. The authors also defined the characteristics of a rhythm those being length, action probabilities, density, and type. In our case, we decided to simplify these properties due to the changes made to our actions. Therefore, in this project, the properties of a rhythm are the length and the difficulty. A rhythm's length remains the same, which is how long a group of actions in a rhythm takes to complete and consequently how much space of the level it occupies. Action probabilities differ based on the difficulty of the rhythm, meaning easier actions appear more often in a rhythm with a small degree of difficulty. The opposite applies to a rhythm of higher difficulty.

However, unlike the other characteristics, the density is indirectly affected by the other parameters these being the rhythm's difficulty and action probability. Meaning that, in this project, the density of a rhythm is not set to be low, medium, or high, as seen in [1] (based on the number of actions in the rhythm). Instead, as mentioned previously, the odds of generating an action and the rhythm's difficulty define the density of our rhythm. For example, a move right action requires one input, and a jump action requires two inputs (move right and jump). Therefore, if we consider a rhythm where a move right action is followed up by a jump action, then this section of the rhythm only requires two inputs. However when considering a section with a single wall jump action, the density of the wall jump action is greater than the previous example due to the number of required inputs being superior. This example reveals how the difficulty can affect the density since by the parameters stated in subsection 2.2.1 a wall jump action is considered more difficult and thus is only generated in the highest difficulty. Therefore, in some cases, the increase in the difficulty can cause the density to also increase. In short, in

higher difficulties, harder types of actions are generated more, which often leads to the generation of rhythms that required more inputs from the player (denser rhythms). Finally, the rhythm's type is similar to the density because it is influenced by the odds of generation an action. In [1] a rhythm can be considered regular, swing, or random. A consistent rhythm with an even number of inputs that are spread along the level is called regular. In a swing rhythm, actions are separated into small groups, meaning that there are sections of the rhythm that are denser than others. In any other case, the rhythm is considered random. In our generator, regular rhythms usually only happen in very low difficulties due to the type of actions that are generated. Swing rhythms never occur since the generator places the geometry of its instance of action immediately after the previous geometry, and thus if the rhythm is not regular or swing then it is random.

Accessing Rhythm Information

Similar to the methodology applied actions, it was also possible to differentiate each rhythm by its properties. These properties contain the information needed to generate our levels, and initially, the idea was to store every characteristic of the rhythm just as in the Launchpad level generator [1]. However, most of the information contained in their properties (in the rhythms generated by the Launchpad level generator) did not benefit our project, meaning that to discern our rhythms there was only a need to define two properties, the rhythm's length and difficulty. The length of the rhythm is the number of cells the rhythm occupies in the level, while the difficulty defines how easy or hard the action in that rhythm will be, in short:

- **Difficulty** - the difficulty of the actions that are generated later are set by this value, which also influences the rhythm's density and type.
- **Length** - the number of cells (space in game) the rhythm occupies.

Generating Rhythms

With rhythms and actions defined, the next step was to generate a group of actions that represent our rhythm in-game. Firstly, a reference for a starting point in the level needed to be established to determine the positions of the geometries. Afterward, it was imperative to set the length value of the level (how long our level is going). Modifying the rhythm's length, allows us to choose how many rhythms should be generated for a given level. Since DDA was going to be applied, a method to generate content up to a certain point in the level was required. With this in mind, a method was made to create a list of instances of individual actions. The following itemization simplifies this method:

- **Step 1** - first we create a new empty list to contain all the instances of individual actions that will be generated. Then an instance of type begin is added to this list (this begin action has the same required inputs as a move right action). The function of this action is to give the player some time before the evaluation starts.
- **Step 2** - then we verify if the maximum playable length has been covered, if so we set an end action after the last action and end the method.
- **Step 3** - in case the length has not been reached, we check if the current height is equal to the maximum height, if so an action that reduces the current height of the geometry

is added, otherwise, we create a new instance of action with randomized height, length, and type of action, which is then added to the list (randomizing the properties requires restrictions to ensure that the geometry is playable, these were explained in 2.2.1). Return to Step 2.

By using this method, it is possible to obtain a list of instances of individual actions where the information of each property can be used to generate playable geometries.

Even though this method is simplified in the previous itemization, its complexity lies in the number of restrictions that are mandatory for the level to be playable. For example, when the properties are randomized, there is a need to ensure that the length and height properties are set in specific intervals due to the player character's physical limitations. The height property of an action can be defined as the playable row of cells in the generated geometry where that action ends. Meaning that when generating instances of action, it is necessary to guarantee that the player can achieve this value of height at the end of each geometry. For example, considering two instances of individual actions denominated A_1 and A_2, where A_1 is the instance that was generated before A_2, and their height values are respectively y_1 and y_2, where $y_1 < y_2$. In Figure 2.9 an example of a generated geometry for A_2 can be observed. In this Figure, three cells are highlighted with colors, cells 1, 2 and k of coordinates (x_1, y_1), (x_2, y_2) and (x_k, y_k) respectively, where 1 and 2 are the last playable cells of the lowest y value of each generated geometry. To complete this geometry, the player needs to reach cell 2, however, the player character cannot reach cell 2 by inputting to move right and jump (the player character can jump a maximum of four cells vertically). This implies that before reaching cell 2, the only viable option is to first reach cell k. Therefore, when generating the height property, in these scenarios where there are other platforms, it is mandatory to consider that the y_k position needs be at least $y_1 + J_{Height} \leq y_k \leq y_2 - J_{Height}$ where J_{Height} is the maximum jump height. However, this example just demonstrates the restrictions of generating instances of actions when considering one property. In the example of Figure 2.9, the length property affects the space between x_k and x_2. This distance needs to be close enough for the player character to be able to reach cell 2 because there is a limit of cells the player character can move horizontally while jumping before it starts to lose height. Implying that the geometry can be impossible if $y_2 - y_k \leq J_{Height}$, due to the distance between x_k and x_2 not allowing this jump to be feasible.

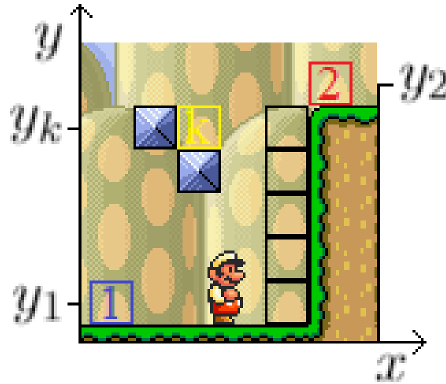


Figure 2.9: An example of a geometry with three highlighted cells.

Nevertheless, even if it was possible to randomize these action properties and achieve geometries that were feasible to complete, there was still a necessity to consider how these should

change with the increase of difficulty. Therefore based on the criteria of subsection 2.2.1, more restrictions were made to ensure the generated geometries get considerably harder as the difficulty increases.

However, even after implementing these restrictions, the previously mentioned method can reliably generate multiple different geometries with the same instances of individual actions.

2.2.3 Converting Rhythms Into Geometry

At this point in the project, the rhythms and actions were clearly defined and the information of their properties was easy to obtain. To create a level, the last step was converting these instances of individual actions into playable geometry. To achieve this, modifications to an already existent algorithm in the Mario AI Benchmark were executed, which we will refer to it as algorithm B. This algorithm was previously used to generate levels based on a seed (random number) that would change the characteristics of a level. Therefore, instead of creating a new algorithm from scratch, it was decided that this algorithm B would be altered to better suit this project. After modifying it, the algorithm permitted the construction of the previous examples of the geometry shown in this chapter.

Algorithm B begins by creating a level template with the length property of the rhythm and a height value that was preset for the game to be playable. This level template corresponds to an empty level of $length \times height = cells$. An empty level possesses no geometries, therefore the second step was to fill the cells and create a playable level. To determine the types of geometries that should compose a level, the list of instances of individual actions was used to obtain the information of each action and generate the geometry accordingly. However, for this to be feasible, the algorithm should place geometries that differ based on the types of inputs (type of action) and the difficulty. For this reason, more algorithms were created to generate an appropriate level chunk with the information of the current instance of action. This process was simple due to the benchmark already possessing some tools to create geometries in chunks such as floors, blocks, ladders, enemies, and coins. After the geometry is placed in the level, the last step of B consists in setting an endpoint for the game, thus a cell in the most right part of the level is chosen, and when the player has the same position as this cell, the game ends and it counts as a win.

When it comes to the algorithms used to build the geometry in the second step of B, these can generate different geometries even with the same instance of action. A simple example would be an instance with the type jump that can generate an enemy, a tube, coins, or a terrain that requires the player to jump over it. However, since we were associating the geometries to the types of inputs, before creating these algorithms, there was a need to comprehend the possible outcomes that could be generated while ensuring the geometries were possible to complete with the restrictions made to the action's properties. The second step of this algorithm can be seen in algorithm 1 in appendix A. In this step, it is clear that the type of geometry generated is depended on the type of action, and that the information of each instance of action is used in all the algorithms that build the geometries.

2.3 Dynamic Difficulty Adjustment

As seen in subsection 1.5, the approaches for DDA methods vary for different game genres. In this project, we decided to generalize and create a simple method that can be easily implemented in other games.

To accomplish this, it was necessary to define some objective and subjective aspects of our game. The goal was to change the game's difficulty mid-gameplay, but there was a need to determine when to increase or decrease the difficulty. The best way is by evaluating the performance of the player. Player performance can be defined as how well a player can progress in the game, what it can achieve, and the way it does it. In subsection 2.2.1, we consider that the player is progressing when it is getting closer to the end goal of the game, and so the objective (goal) is to reach the end of the level. Implying that it was imperative to establish how well the player is performing until it reaches its objective. To determine performance, we need to account for subgoals. These can be classified as secondary tasks that do not always give the player progress but can complete to increase its performance. In a 2-D Mario game ([26]), there are specific tasks that increase the overall score of the player these are usually the collecting of coins and defeating enemies.

Once the performance value is obtained, it is associated with an increase or decrease in difficulty, which directly influences the next generated rhythm.

2.3.1 Defining Performance

In this project, the player performance is considered to be a value (or score) that can be calculated based on the comparison of certain parameters.

First, it was necessary to determine what was being evaluated, in this case, we are evaluating how well a player can complete a rhythm. Completing the rhythm requires the player to have certain types of inputs and timing to perform every action correctly. This means that we considered the following variables for the evaluation of the player: the time the player takes to complete the rhythm, the subgoals it completes, backtracking in the level (considered a mistake when not necessary), taking damage from enemies, and losing the game. In some 2-D Mario games ([26]), the score stat is affected by most of the variables mentioned. At the end of the level, the time that was not used, the coins picked up and enemies defeated all convert into score. In this project, the score attributed to each metric used to evaluate a player's performance is referred to as an individual performance score. The following metrics were used to determine the performance:

- **Time to complete a rhythm** - can indicate how fast a player was at completing the rhythm.
- **Time Moving Backward** - this metric shows if the player made mistakes due to its theoretical unnecessary need to backtrack.
- **Coin collecting** - the number of coins picked up can determine if the player is passing through specific locations. If these are placed in spots where the player needs to be as precise as possible, it can be a way of determining the precision and timing of the inputs.
- **Enemies Defeated** - the enemies are a subgoal for the player that can increase its score.

However, they are constantly moving which forces the player to have good timing and input adjustment.

- **Damage taken** - if the player takes damage (by colliding with an enemy) or loses the game it is an indication that the intended combination of inputs was not performed correctly.

Evaluating Performance

To evaluate a player's performance, we have to compare the values it obtains with the expected values of each level. When it comes to the coins picked up and enemies defeated, we determine how many coins and enemies exist and how many the player has respectively picked up or defeated. But when it comes to the time it took to complete a level, the time spent backtracking, and the damage taken during the game, the comparison becomes more complex. Initially, we thought of calculating the fastest possible time to clear a certain length of the level and use that as a reference for most actions. However, not every player plays perfectly, meaning that we either add some time to compensate or we adjust the values extracted. Both options add bias due to us directly affecting these values. Even if these values are not modified, this option is not viable because it is assumed that it is feasible for a player to maintain maximum velocity throughout the entire level, which is not possible. To solve these issues, as mentioned previously it was decided that an AI player (AI agent) would be used to obtain reference times in all the possible geometries. These times are then compared to the player's values.

However, the damage taken metric cannot be evaluated in the same manner. This benchmark has a specific health system that is connected to the power-ups the player character currently possesses. A power-up is a game object that, when picked up, alters the state or mode of the player character, which influences the number of times the player can make mistakes (in the benchmark these states are denominated as modes, and thus we refer to them as such). In this benchmark, the player character (or Mario) has three states or modes and two power-ups:

- **Mode 0** - the default mode, if the character takes damage from an enemy while in this mode, the game ends on a loss.

- **Mode 1** - it can be reached by obtaining a mushroom, if the character takes damage it reverts to mode 0.

- **Mode 2** - it can be reached by obtaining a fire-flower, if the character takes damage it reverts to mode 1.

Each power-up gives the player an extra health point, meaning that getting hit by an enemy while having at least one power-up, does not end the game on a loss. This implies that, if the player is not allowed to get power-ups in the middle of the rhythm it is easy to track how many times the player got hit by checking the beginning mode and the end mode. To estimate good values for this metric, some tests need to be made with the AI agent. Still, a logical starting point of this individual performance metric would be to attribute a positive score when the player remains in the same mode at the end of the rhythm (does not take damage), and a negative score otherwise, increased by the number of times it took damage.

If the player loses, the evaluation needs to be executed differently. If the level is not complete, there needs to be another method other than time to determine the player's performance. In

this case, we decided to use the percentage of length that was traveled in the rhythm. The time and time moving back metrics are replaced by this length and not put into consideration when evaluating the performance. If the player ends the game early due to a loss, the time obtained can be lower than the estimated time. This results in the individual performance metric of time and time spent backtracking being greater than they should be.

Tracking Evaluation Metrics

This subsubsection explains the process used to track these metrics. To achieve this, we track the relevant events that occur in every frame of the game. The number of frames is used to check how long it takes to complete any given geometry (the frames per second are 24 regardless of the device used), and by analyzing the number of frames that the player has negative velocity, the time that it spent backtracking is obtained. To track the number of coins picked up, a coin pick-up counter that was already implemented in the benchmark was used.

When it comes to the tracking of enemies defeated, there is was a slight issue that needed to be considered. Even though a counter for the enemies defeated exists, it was not adequate for this project. In the game, some enemies are impossible to defeat if certain conditions are not met, some enemies can only be defeated with the fire-flower power-up, and others are completely invincible. Another enemy type can be defeated repeatedly due to their intended infinite spawning system. For these reasons, a new enemy defeated counter was created that is only affected by enemies that were always able to be defeated regardless of the game state. This allows the player to obtain the best possible performance scores in any starting character mode (or Mario mode) and it also prevents players from abusing the enemies that are constantly spawning for more performance score.

To check the number of times the player took damage, the Mario mode at the beginning and end of the rhythm is verified. The subtraction of these modes is described by:

$$n = M_{End} - M_{Begin} \tag{2.1}$$

The value of n in equation 2.1 is used for the evaluation of the performance. Where n is the number of times the player took damage, and M_{End} and M_{Begin} are the Mario mode at the end and beginning of the rhythm respectively.

Finally, the length traveled can be determined by either the maximum number of cells reached (cell column) or the number of pixels in the x axis, the second option was chosen. This value is updated when the current length of the player is greater than the maximum length reached at that point in the game.

2.3.2 Calculating Performance

To calculate the performance, a method that uses the previously mentioned individual performance scores as input was created. This method calculates every quotient between the values the player obtains and the ones that are estimated to be achieved, therefore that the scores are based on how close the player is to these estimated values. In the case of the time variables, since the fastest possible time is not being used, it is possible to go faster. If a player obtains a shorter time than the estimate, the highest possible score is equivalent to 100% or 1 even if the player was faster.

This process affects the difficulty of the next rhythm, thus the performance score needs to be a number that reflects that change. If the performance value is defined as a percentage, then the increase or decrease in difficulty per rhythm should be carefully chosen. To start, a maximum change of 3 was implemented, which implies that after finishing a rhythm, the difficulty of the next rhythm will fluctuate between $[D - 3; D + 3]$, where D is the difficulty of the previous rhythm.

Calculating Individual Performance Values

The next step was to define how the performance score affects the difficulty. First, there was a need to determine what metrics were going to be put into consideration for the evaluation, because as mentioned in subsection 2.3.1, if the player loses, different metrics are used to evaluate the performance. Another case that needs to be considered is the lowest difficulty since there are no enemies, thus this metric cannot be used to calculate the performance score.

Afterward, we compare how close the player is to the estimated value and attribute an individual performance score (based on the quotient) for that parameter.

Initially, a discrete method was briefly tested with the values that were somewhat evenly spread along with various intervals. However, since increasing or decreasing the difficulty by 3 can affect the geometry drastically the interval required to achieve this score was considered small.

- For percentages in range of [90; 100]% a +3 performance value is obtain.
- For percentages in range of [89; 75]% a value of +2 is obtained.
- For percentages in range of [60; 74]% a value of +1 is obtained.
- For percentages in range of [40; 59]% a value of 0 is obtained.
- For percentages in range of [25; 39]% a value of −1 is obtained.
- For percentages in range of [10; 24]% a value of −2 is obtained.
- For percentages in range of [0; 9]% a value of −3 is obtained.

Even so, it could be argued that different difficulties should have different performance benchmarks for our values. For example, difficulty 0 requires a percentage of above 60% to reach a value of +1, but in difficulty 5 it could require less percentage, due to it being considered harder. However, it provides an inaccurate determination of the performance of a player, due to it becoming easier to reach increasing higher difficulties, and the possibility of the generated content being inadequate for the player.

This option of determining the performance was proven unfavorable after a few tests with the AI agent. The problem with this discretization is that reaching the limits of our intervals is completely irrelevant to the performance score. For instance, if there are two players and one achieves a value of 75% and another achieves a value of 74%, they both receive different changes in difficulty even though their percentages only differed by 1%. For this reason, a continuous option was chosen. The alternative and definitive option was to use the following equation:

$$P(x) = \frac{(x - 50) \times 3}{50} \tag{2.2}$$

Equation 2.2 represents how the individual performance score varies based on the given percentage x. This way we attain a linear equation that could be considered a slight improvement, due to its simplicity when compared to the method that used the discrete intervals. However, the previous issue is still present in this new method, being continuous signifies that decimal numbers are obtained, while the actual difficulty values are considered integers, therefore if we chose to round up the values, there are instances where, for example, the two players can obtain values of 1.4 and 1.5 respectively and gain different increases in difficulty. Still, it was decided that the project would be processed with this approach.

This method works for most metrics, but it cannot be applied to our measurement of individual performance value of damage taken. When it comes to the Mario modes, there only exist three states, thus the only viable solution is to make this parameter discrete.

Nevertheless, there is another aspect to examine about Mario modes. If the goal is to determine how well a player is doing based on the damage taken, then all the possible cases need to be considered, due to it being feasible for players to lose in any of these modes. For example, the generator can create geometries with gaps that the player needs to jump over, however, if a player fails to perform this jump and falls, the game ends regardless of the current state of the health system. Therefore, a player who did not collide with any enemies but fell in the gap should be considered better than a player who collided with two enemies and still fell in the gap (when considering that in the other metrics the scores that both scores obtained were the same). For these reasons, the individual performance score of the damage taken metric in a rhythm is determined by the following method:

- If the player loses and:
  - did not get damaged, obtains a performance value of 0.
  - got damaged once, obtains a performance value of -1.5.
  - got damaged twice, obtains a performance value of -3.
- If the player clears a rhythm and:
  - did not get damaged, obtains a performance value of 3.
  - got damaged once, obtains a performance value of -1.
  - got damaged twice, obtains a performance value of -2.

Determining Weights

The next step was to determine the weights of each parameter. A weight determines how valuable a metric is when calculating performance. For example, if the coins have a greater weight than everything else, then a player can take more time, take more damage, backtrack more and defeat fewer enemies, and still obtained a high performance score if it picked up every coin.

One way to dictate how relevant these are variables is by using the definition of a rhythm (sequence of actions that must be completed to finish a level) to get an idea of how well a player is completing the said level. This implies that there is a need to judge how fast and precise a player is during a rhythm.

The precision can be studied with the placement of subgoals (coin and enemy placement). In our geometries, coins and enemies were placed in a way that allows the player to perform the

correct combination of inputs and complete the objective at the same time. Implying that, if the player collects every coin in a segment, it had precise inputs while performing the action. For example, two different players can go through the same chunk of the level and complete it with different precision. The player that gathered more coins is considered more precise due to the inputs required to attain that number of coins. This becomes clear when observing a geometry similar to Figure 2.10, if the player does not jump at the edge of the platform the coins will not be collected. This method can be a way of judging the timing and precision of the player.

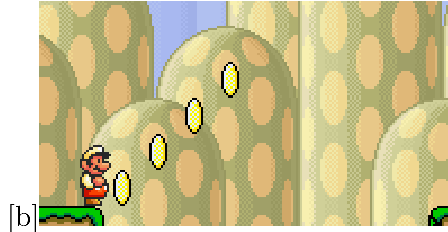
[b]

Figure 2.10: Possible run jump action geometry.

When it comes to enemies, these move toward the player, implying that at one point in time, these eventually collide with the player and move on to a previous geometry in the level. To achieve the lowest amount of time and defeat the enemies, the player needs to jump in a certain interval of frames to hit and defeat the enemy, before the player is forced to backtrack, taking more time in the process. This becomes even harder the higher difficulties due to the increase in enemy movement speed and number of enemies. For these reasons, the number of enemies defeated can be another way of studying the player's precision, since missing enemies implies that the intended inputs were not performed.

In short, the weights can be changed based on how we define player performance. For instance, if the speed and precision of the player have equal values then the option of putting similar weights can be considered. However, if speed is valued more than precision then the weights can be modified to benefit players who complete levels faster.

Even though we established ways to evaluate the player's performance, the weights, and estimated individual performance score, these are ultimately determined by the AI agent's behavior. Since there is a need to judge the generator's content, the weights must be constantly altered to suit our agent, thus in the initial test, it was decided that all the evaluation metrics should have the same weight.

Calculating The Performance

To calculate the performance, the individual performance scores are multiplied by their respective weights and are added up afterward. The final score is then added to the current difficulty.

If the player wins:

$$P = W_1 \times T_F + W_2 \times T_B + W_3 \times C + W_4 \times E + W_5 \times M \tag{2.3}$$

If the player loses:

$$P = W_1 \times C + W_2 \times E + W_3 \times L + W_4 \times M \tag{2.4}$$

Where in equation 2.3, P is the total performance score, T_F is the individual score attributed

to the time metric, T_B is the score for the parameter of time spent backtracking, C is the score achieved by picking up coins, E is the score acquired by defeating enemies and M is the score associated with the damage taken. W_1 to W_5 are the respective weights of each individual performance score (although they began with the same value they were altered during the test phase).

Equation 2.4 removes the time and time moving backward parameters and introduces L which is the performance value of the distance traveled. W_1 to W_4 are also the respective weight of each metric.

2.3.3 Updating The Level

One of the main goals of this project was to implement a level generator that generates geometry as the player is playing. At this point in the project, we defined every requirement and constructed the tools to accomplish this goal. The actions and rhythms were defined, and algorithms were created to generate them and their associated geometries for different difficulties. The player's performance could also be calculated in each chunk of the level. Therefore, the last step was to update the level as the player progresses, more specifically when a rhythm is completed.

Updating the level while in the game required us to check where the player is in the level in relation to the rhythms. When the player reaches the length that is equivalent to the last action of that rhythm a new rhythm needs to be generated. To accomplish this the first step was to calculate the player's performance by using the DDA method. The method only evaluates the player when it leaves the geometry associated with the beginning action and finishes when it reaches the last action. Afterward, by using the algorithms mentioned in subsections 2.2.2 and 2.2.3 a new list of individual instances of actions is generated and these are converted into the geometry of the next rhythm. Lastly, each rhythm needs to be evaluated separately, thus our solution was to implement a checkpoint system. In a rhythm, we consider a checkpoint to be the first column of cells of the last action. This location is associated with its corresponding rhythm, therefore when the player finishes a rhythm its performance score, and the length traveled are stored for future use, and the counters used for the player's evaluation are reset.

To evaluate the player and our AI agent, there was a need to establish how many rhythms should be played before finishing the game. The length value of the level's template affects the number of rhythms that are possible to place in each level, thus it was decided that a player should play 10 rhythms. This way, it should give the DDA method enough attempts to converge any given player to a bound value of difficulty, without making the game too short nor too long.

2.4 Generator And DDA Considerations

To advance in the thesis, some aspects of this chapter need to be kept in mind. In the Launchpad article [1], actions are defined as an input that allows the player to perform an event in the game. However, when analyzing the rhythm-based approach, it was clear that this methodology could be simplified in order to create similar or better outcomes while maintaining the integrity of what makes a rhythm and an action. Thus, action was redefined to be an input or a combination of inputs that are associated with a specific group of geometries and that, when performed correctly, allow the player to progress in the game. Individual instances of actions have various

properties but are mostly discerned by their action type. However, the same instance can generate different geometries and the other action properties also influence their associated level chunk. To ensure the geometries were possible to complete, the player character's limitations were studied to determine what type of restrictions should be placed when generating the action properties. To generate these geometries, some already existent algorithms were altered and multiple were created using the tools provided by the benchmark. Thus, when the game begins or when the player finishes a rhythm, these individual instances of action are used to create a possible to complete level chunk based on the player's skill.

To implement a DDA method there was a need to comprehend what elements make an action difficult. Some parameters that might influence the difficulty of an action were able to be identified, these were: the number of inputs and the number of simultaneous inputs required, the duration and timing of these inputs, and the influence of time on the inputs. These parameters helped determine how difficult the necessary combinations of inputs is to execute. Therefore, the geometries considered for this project were associated with a level of difficulty based on these parameters.

Finally, the DDA method allowed us to evaluate the player's performance based on the same concepts used to determine the difficulty of the geometries (speed and precision). The speed of a player is determined by the number of frames, and its precision can be judged based on the completion of sub-goals (coins and enemies). The scores attributed to the player in the time metrics are compared with the values obtained by the AI agent obtains (benchmark values). However, these expected times have a bias since we decide how the agent behaves.

Chapter 3

Validation Of The Level Generator

After creating the level generator and the adaptive difficulty method, the next step in the project involved the recognition of the quality of the created levels. Therefore, it was necessary to use a tool that could validate the generated content. The goal of this validation process was to determine if the generated levels had the right degree of difficulty and if the DDA method was adequate at determining the player's performance. Thus, if both these conditions were met, we consider that the generator is generating appropriate geometry for the player while changing the difficulty at a rate that is acceptable.

To achieve this goal, it was decided that an AI agent would be used for level validation. Meaning the agent would play multiple generated levels while being evaluated on its performance or on other metrics suited to determine the adequacy of the content. With the information of the agent's individual performance scores (e.g., time scores), it is possible to reach conclusions in regard to the current state of the generator. For example, if the agent obtains an individual performance score that is consider inferior to the intended value, then modifications could be made to our DDA method's evaluation function or to the weights of the individual performance scores. To ensure the adequacy of the DDA method, the different difficulties should be able to be discerned by the agent's data and the agent should on average converge its ending difficulty (achieved by winning or losing) to an adequate bound value (e.g. [6; 8]).

However, the issue with this approach is that the results have an innate bias toward our agent. Therefore, it is necessary to understand how the agent is operating to comprehend the context of the results. By understanding the AI's behavior and the context behind the results, it is possible to form conclusions in regard to the adequacy of the generated content.

Initially, the idea was to use an AI that was already developed to conduct the tests. However, after a long search and multiple attempts to find a suitable agent that would satisfy our needs, no agent was qualified to fulfill this purpose. Most of the agents were not compatible with the version of the benchmark used and modifying the version would imply changing multiple aspects of the project done up to this point. Thus, it was decided that we would create our agent.

3.1 AI Benchmark Tools

The Mario AI Benchmark was used for an AI agent competition, so for this reason, it comes equipped with useful tools to build an AI agent. The first tool is a grid that is constantly attached to the player character represented in Figure 3.1. This grid is by default made up of 19 by 19 squares with its center on the character in mode 0 (character mode explained in section

2.3.1). These squares have the same size as a cell, implying that each one is made up of 256 pixels. This grid is used to obtain the information that is contained in the center pixels of each square in each frame. The information is then used by an algorithm (second tool) to associate it with a game object (e.g., ground, enemies, coins, nothing). This generalization allows us to clearly identify what is inside each square and consequentially what type of geometry surrounds the agent at any given time. Therefore, to stay in tune with the type of PCG used, we can define the agent's behavior to be the combination of actions that need to be performed to complete the generated geometry that is currently surrounding the agent.

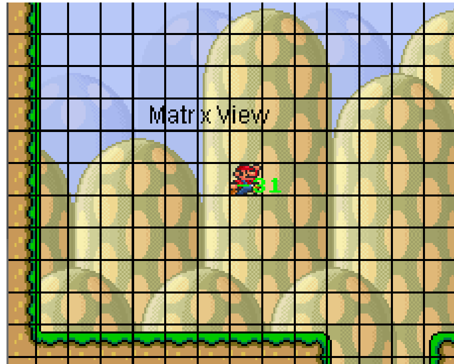


Figure 3.1: A visual representation of the grid (the player character is in mode 0).

Lastly, there is another algorithm that changes the inputs the agent performs in every frame, thus by modifying this algorithm, it is possible to construct behaviors for the agent that alter the inputs executed in each frame.

The combination of these tools creates a system that works similarly to a group of sensors. With these, it is possible to discern what is happening around the AI agent in every frame and make it respond adequately.

3.2 AI Agent Behavior

3.2.1 Decision Tree

When it comes to what method should be used to define the agent's behavior, it is imperative to determine the goals of the behavior. As mentioned previously, we wanted the agent to identify the type of geometry that surrounds it and make it perform the combination of inputs that are associated with that geometry. To execute this methodology, it was decided that, for this project, a decision tree could be used. A decision tree is a decision-making system that is made up of connected decision points [29]. The tree has a starting decision and for each decision one of a set of ongoing behaviors is chosen (an example can be seen in Figure 3.2). This type of decision-making is simple to implement since the game that is being used has a low number of different inputs which create unique events. Due to the low variety of inputs, there are multiple groups of geometries that can be completed with the same combination of inputs, thus the number of decisions necessary to correlate a geometry with a sequence of inputs can be considered low. Therefore, a decision tree was chosen as the decision-making system of the agent due to the effectiveness of the methodology and the simplicity of the algorithm.

Nonetheless, the agent is required to obtain benchmark values for the DDA method, which are later used as a comparison for real players. Meaning that, for these values to exist, the agent

needs to be capable of completing any geometry in an isolated system (the level is composed of only the specific geometry the agent is completing). Once the agent can complete all the possible geometries, we extract the number of frames taken to complete it, the number of frames the agent was moving with negative velocity, and associate these with their corresponding geometry.

Our idea for the decision tree is to break it down into a section of sequences and a section of non-sequence movement. For this project, we consider a sequence to be a sequence of inputs that the agent can perform to complete a group of geometries. However, sequences also manage scenarios where movement restrictions need to be executed to cover instances in case the agent ever gets stuck (nevertheless these are still sequences of inputs). The non-sequence is a behavior with not sequential inputs that the agent can execute to complete simple geometries. The non-sequence only triggers when the agent does not spot any geometries related to the sequences (this fact is visible in Figure 3.2, if none of the checks are made, then the agent chooses to perform the non-sequence behavior), thus it is used to complete specific geometries correlated to instances of actions with the types move, jump, and duck.

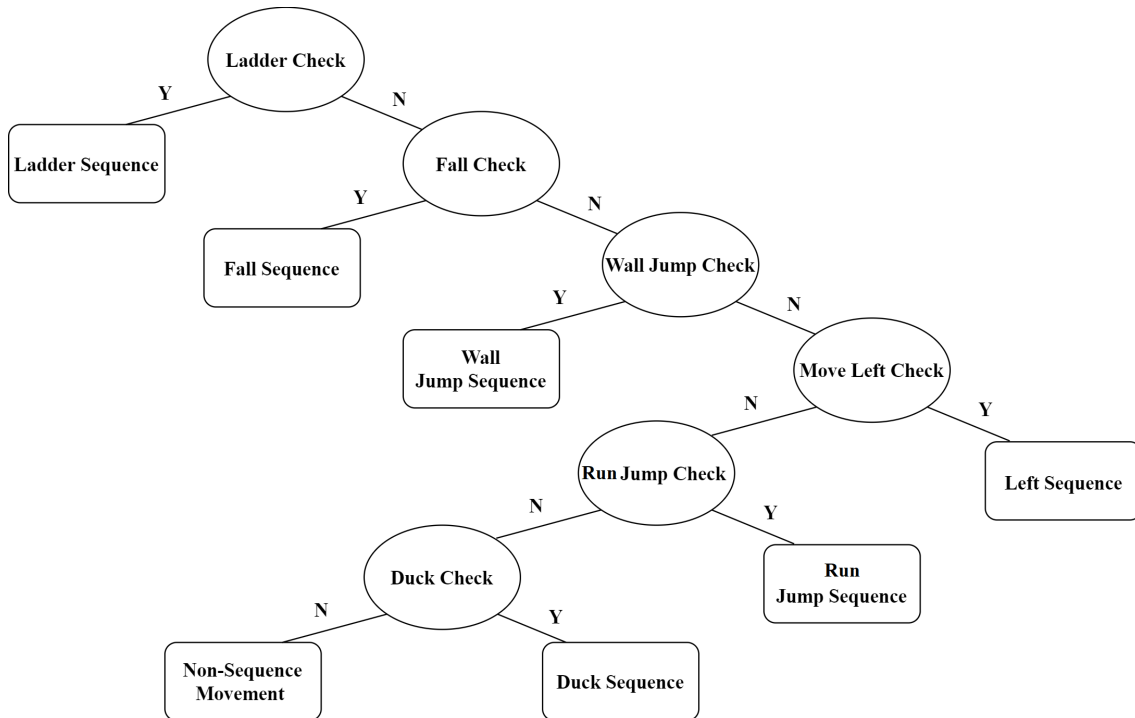


Figure 3.2: Decision tree that describes the agent's main behavior.

Figure 3.2 reveals a simplified version of the main decision tree. The circle-shaped blocks represent a decision and the outcome of this decision is based on the geometry the agent detects with the use of the grid and the previously mentioned algorithms. The rectangle-shaped blocks represent the previously explained sequences and non-sequence. Each sequence and the non-sequence have their own decision trees. Sequences can end when the agent has completed the geometry, a small fraction of time has passed after the geometry was completed, or the set time for the agent to complete the geometry ended. To establish how long the agent was in each sequence and to determine how long some inputs would be pressed the computer clock was used as a variable. However, by controlling the time spent inputting certain combinations of keys, each run of the game needed to be done in real-time, which limited the number of tests that could be done. Nevertheless, after running the tests, we considered that this restriction did not hinder the

conclusions made during the test phase (since the goals set were achieved). Therefore, to make the most out of each test, a variety of changes were made to ensure that adequate conclusions could be drawn. Another point to note is that understanding the context behind the agent's behavior and what should be changed was essential to produce positive results.

Figure 3.2 also shows the priority of each sequence. The order of the sequences needs to meet certain criteria. Because the agent analyzes its surrounding using the grid, it is possible, in some instances, that it checks two or more sequences simultaneously. In these cases, if the priority is not implemented correctly, then the agent does not perform the correct sequence and fails to complete the geometry.

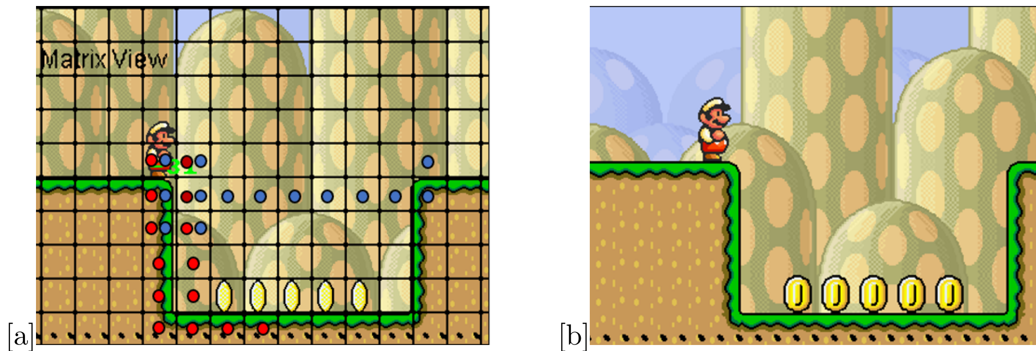


Figure 3.3: Game view with and without the grid. The order of the decisions in the main decision tree affects what tasks the agent will perform.

Figure 3.3 shows a scenario where the order of the sequences is extremely important. In this case, two different sequences can be detected, these being the **Fall sequence** and the **Run Jump sequence**. The tiles that are required to be verified for the agent to begin a **Fall sequence** or **Run Jump sequence** contain red or blue circles respectively. In Figure 3.3 [a] there are tiles with both red and blue circles, therefore the priority in the main decision tree determines what sequence the agent should perform and, in this case, the agent performs a **Fall sequence**.

When the priority is executed poorly, the agent can either ignore certain geometries and subgoals or even lose the game. Figure 3.3 shows an example of a geometry where a subgoal would be ignored if the priority of the sequences was inverted. If the jump is prioritized then the agent ignores the coins below and loses performance.

3.2.2 Non-Sequential Behavior

As mentioned previously, the non-sequence movement is the type of behavior the agent uses when it does not detect any geometries that are related to sequences. This behavior handles the agent's simple movements such as: moving forward, jumping over small gaps, jumping to platforms, defeating one enemy by jumping, and ducking to avoid enemies. Therefore, when the agent decides to use this behavior, the inputs it uses are not sequential, thus it was denominated as a non-sequence movement.

In this non-sequence behavior, when performing a jump action, the agent always inputs the highest jump possible with the exception of the instances of action that generate an enemy (by varying the duration of the input jump the character can jump to different heights). By allowing the agent to jump the highest possible even when it is not necessary, we ensure that it

can complete any of the gaps or platforms that appear in front of it without impacting the time of completing the geometry.

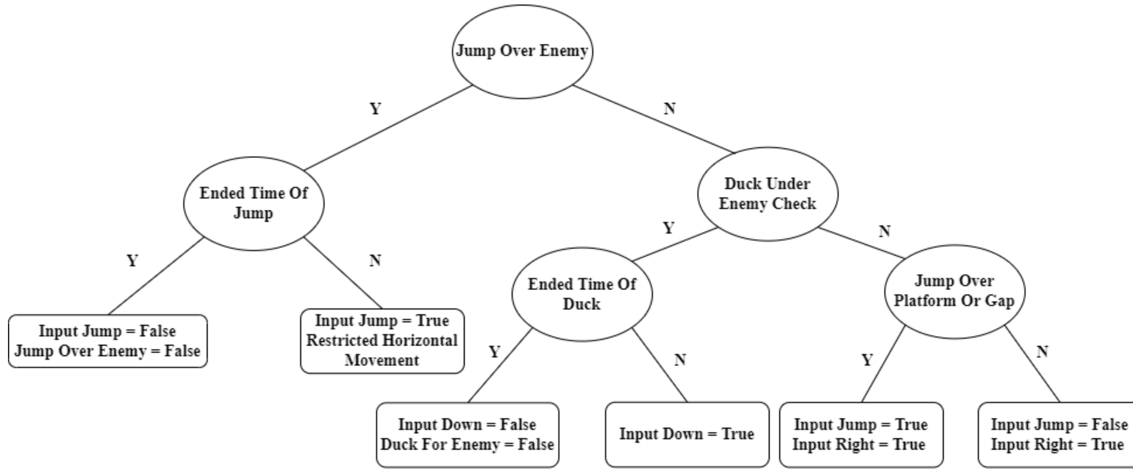


Figure 3.4: Visual representation of the non-sequence movement decision tree.

In Figure 3.4 a simplified representation of the decision tree of the agent's behavior when it using the non-sequence movement can be observed. The agent uses this behavior to complete geometries that are considered easier based on our criteria in subsection 2.2.1. The number of squares in the grid that need to be checked for the agent to perform any action in this type of behavior is lower when compared to the sequences. For this reason, these actions can be used in the same decision tree without negatively affecting the agent's behavior. The priority of the decisions is not as crucial as in the main decision tree because the generated geometries do not have checks that overlap similarly to Figure 3.3.

3.2.3 Sequences Of Inputs

The sequences are a sequence of inputs that allow the agent to complete a geometry. Sequences are differentiated by the type of geometry (and consequentially by the type of action) they can handle, therefore, in most cases, sequences can be used to complete most variations of possible geometry that correspond to an action type. However, some sequences were made specifically to ensure the agent does not get stuck.

The sequences that the agent can use are the following:

- **Ladder sequence** - deals with any kind of ladder (up actions).
- **Fall sequence** - handles scenarios where the agent needs to fall to reach a geometry (move actions).
- **Run Jump sequence** - allows the agent perform jumps over large gaps (run jump actions).
- **Left sequence** - allows the agent to complete geometries where it was to move in the opposite direction of the end goal (move left actions).
- **Duck sequence** - manages the behavior of the agent in some of the geometries that require the agent to duck (duck actions).

- **Coin sequence** - made to correct the **Fall sequence** (move action correction).
- **Wait sequence** - handles geometries where the agent has to wait to complete them (wait actions).
- **Wall Jump sequence** - meant to deal with geometries that involve jumping from one wall to another (wall jump action).
- **No Momentum Jump sequence** - allows the agent to complete specific geometries where it has no space to move, but it needs to perform a greater jump (jump actions and jump action correction).

The **Ladder sequence** commences when the agent spots a ladder and it focuses on making sure the agent can go up the ladder. The agent's momentum needs to be put into consideration, so the idea is to slow it down first and let it climb up the ladder. While climbing the agent needs to shift its direction when necessary to stay on the ladder. This adjustment needs to be made carefully to ensure the process does not take a long time (obtain better times to performance measure). The agent can also, leave the ladder early if necessary or perform a jump at the top if needed to reach other geometries.

The **Fall sequence** handles scenarios where the agent needs to reduce its height by falling and not ignore geometries. If the priority of this sequence is poorly managed it can conflict with the **Run Jump sequence** due to their similarities when checking for the respective scenarios. The agent uses the grid to identify the space ahead that it must fall, managing its speed based on that space.

The **Run Jump sequence** allows the agent to jump over large gaps. This sequence begins by allowing the agent to move back and afterward, it turns around to start a run. When it reaches the edge of the platform a jump is executed. For the agent to not move back the grid needs to increase massively. However, while running the character can jump over 10 cells in length if it lands on a platform with the same height as the initial platform. Thus, this behavior allowed us to construct many variations of the same style of geometry which requires the agent or the player to perform these types of inputs (run, move and jump inputs).

The **Left sequence** begins once the agent observes a wall it cannot jump over, and so it starts and moves back. Afterward, two checks need to be made, first, the agent must identify the platform on the left to jump on. Once the agent reaches this platform (this being the second check), it moves in the opposite direction and jumps to the top of the previous unreachable wall. In higher difficulties, the jump is bigger requiring the agent to also input a run action to complete the geometry.

The **Duck sequence** is similar to the **Run Jump sequence**. The geometry of these actions consists of a wall that can only be traversed by ducking under it. First, the agent moves backward to clear some distance, then it turns around and begins to run. Once it is close to the wall, the agent inputs a duck action and performs a slide to complete the geometry.

The **Coin sequence** has the function of correcting any sequence or non-sequence behavior that makes the overshoot coins. An example would be a jump action where the agent jumped past a few coins and turns around to collect them. The sequence starts when the agent detects a coin immediately behind it and finishes once no more coins are detected.

The **Wait sequence** takes care of geometries where the agent has to wait or make no inputs to complete the geometry without being punished (by losing or taking damage). The sequence

begins once the agent spots an enemy that it cannot avoid. When the agent does not detect the enemy, the geometry is now possible to complete, and the agent proceeds.

The **Wall Jump sequence** was previously explained in subsection 2.2.1, but to reiterate, the sequence begins when the agent encounters a geometry with a platform on the left and a wall on the right. To complete this geometry, the agent needs to jump off the wall to reach the platform, and then, to achieve progress it needs to execute a jump while running similar to the **Run Jump sequence**.

The **No Momentum Jump sequence** is used to handle scenarios where the agent detects a jump that needs to be made, but it is currently located in a confined space where it will not have enough momentum to make the jump unless a run input is made.

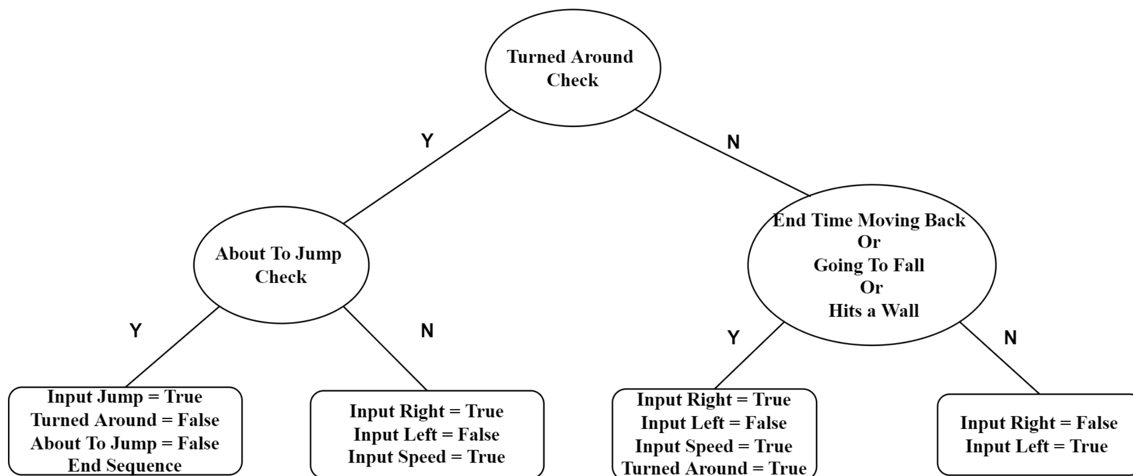


Figure 3.5: Run jump sequence decision tree.

3.2.4 Negative Aspects To Consider

Even though careful precautions were taken to guarantee the agent could complete most actions presented to it, this behavior still has negative aspects that need to be mentioned, and these are relevant to understand the results achieved in the tests. Firstly, the agent can complete all the possible geometries, if they are isolated. Meaning that if the agent's starting point in the game is the same as the geometry, and there is nothing else in the area that occupies that geometry (conflicting geometries), then the agent will complete that level chunk without being punished by the evaluation criteria explained in subsection 2.3.1. It was imperative to achieve a way of determining the time values for every geometry to evaluate a player's performance, and this was the only option.

However, as the agent was being created and experimented with, it was evident that it struggled with some combinations of these geometries (when a geometry would directly influence another one). For this reason, some sequences were changed and some were added to ensure the agent could handle the generator's content (an example of an added sequence is the **No Momentum Jump sequence**). Nevertheless, there are some rare instances where the agent still fails and loses the game.

The problem with this approach is that, if the agent can complete every geometry without failing then it is complicated to discern the difficulty of a level based on the performance score. Thus, we needed to decide in what situations the agent should fail while being able to use it to

retrieve the data that is used as a benchmark. Therefore, excluding scenarios where we want the agent to fail, the agent should in theory always attain the best possible scores in the time variables. This is not only due to the agent completing most geometries but also because we are comparing the time the agent achieved in isolated geometries with the time it achieves in a normal level. However, even if we are able to acquire the benchmark values and decide where the agent should fail, the agent should achieve a lower performance score when the specific group of geometries that should make him lose is generated, making our results biased for those specific scenarios.

The final decision was to make the agent have the hardest time when facing enemies. Because of the way we are evaluating player performance, the enemies are in theory the hardest obstacle to overcome that can affect the performance score (subsection 2.2.1). Even though failing a gap that can end the game, the player is not timed to perform an input and can take time to perform the jump. However, enemies move toward the player forcing it or the agent to act before it takes damage or loses the game. Another important detail is that the enemy's position is constantly changing, thus the time the player needs to be pressing these inputs changes from frame to frame (input duration). Since the performance value is dependent on the enemies defeated, it is important for the player to be able to defeat them to obtain a good score. In this case, the agent still has a bias when it comes to performing poorly, but it is easier to manipulate the results of our generator by changing the weights of each metric. To improve the results of the test, the odds of an enemy being generated in higher difficulties can be increased, and its behavior can be modified to be harder to avoid and defeat. The only issue that arises with this approach is that the validation of other geometries might not be as good as it is for enemies due to the agent mostly failing in these scenarios. For example, using extracted data (performance score) it is hard to discern the difficulty of two levels if one only contains gaps (the agent rarely fails) and another that only contains enemies (the agent fails mostly on these geometries). Even though these geometries are constructed with the data of the same type of action, the level with gaps is significantly easier for the agent.

Another possible way of discerning the difficulty of the game is with the generation of a random number that entails if the AI agent completes or fails a geometry. The problem with this approach is that the results will again have a bias toward the odds chosen for each sequence. Therefore, what makes the game easy or hard is only defined by the odds and not by the character's lack of behaviors to complete a geometry. The goal was for the agent to make mistakes because it does not know how to handle a given scenario instead of it being random.

Nonetheless, the agent still has another problem with its behavior. If the agent is in a sequence its behavior will not change until an end sequence condition is achieved. This implies that there are scenarios where the agent is using the behavior of a sequence, but it needs to adopt a new behavior to handle what it is currently facing. In these cases, the agent keeps performing the same behavior regardless of its surroundings. This mostly occurs when there is a combination of subsequent actions with an enemy involved. The enemies move in the initial player's direction and do not stop, meaning that sometimes they can end up influencing other geometries.

Figure, 3.6 shows four images of a possible group of geometries that the agent has a harder time completing. [a] shows the initial state of the geometries where there is a small space where the agent can duck under with enough momentum and slide to the other side, and afterward jump over the bullet enemy (a duck followed by a jump action). In the **Duck sequence** the

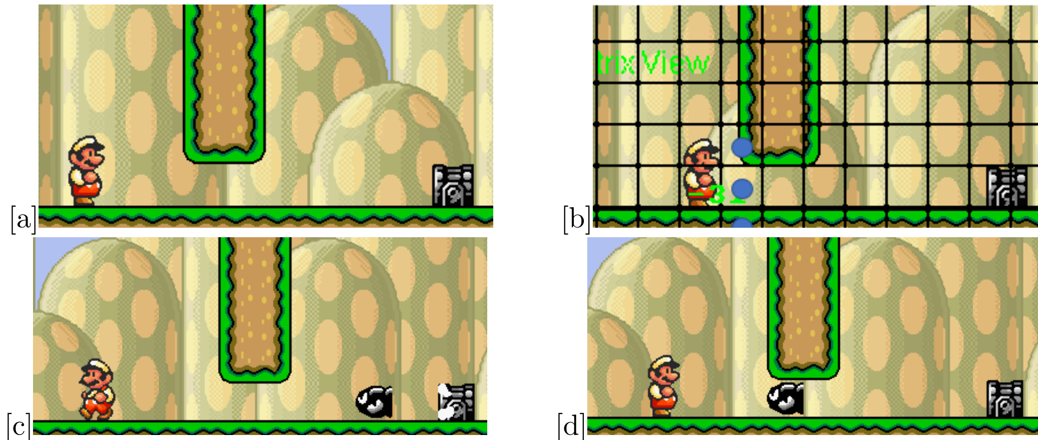


Figure 3.6: A scenario where two geometries of different actions can affect the performance of the agent.

agent checks that there is a space where it can duck under, turns around to get enough space to generate momentum, and executes the slide. The agent only finishes the sequence if it runs out of time, or it reaches the other side of the duck geometry. This means that if the agent is slow, the enemy will reach it before the agent executes all the actions. In this case, the agent will be punished by either losing or taking damage. [b] represents the turning point of the **Duck sequence**, the grid is active to show the point where the sequence is triggered (blue circles). The center of the two squares in front of the character's sprite needs to have the information of those corresponding blocks (game objects) to start the sequence. [c] shows the character moving back to get momentum and the enemy moving toward it. [d] demonstrates the character trying to complete the geometry but the enemy blocks its path. If the agent ignores the enemy it gets hit by it and loses performance by taking damage or losing. In this instance, the problem is that the agent is too slow while completing the geometry and consequently it gets hit by the enemy. A possible solution for this behavior would be to detect the information for a square farther away from the character. In this case, the agent does not have to turn around since it has the distance to initiate the run. However, due to other conflicting geometries, this was not possible.

Therefore when it comes to the tests, it might be difficult to discern where and why the agent was failing, and thus it was imperative to understand the context of its behavior to interpret the results, and to be able to correctly adjust the weights and the probability of each geometry being generated to ensure the agent was able to converge to an adequate bound value of difficulty.

3.3 Synopsis Of The Validation Process

For a better understanding of the next chapter, there are some aspects of the validation process that need to be considered as context to comprehend the results obtained in the tests. Firstly, the validation process needed to ensure that the generator was generating adequate content before we initiated the tests with real players. To realize this goal, we created an AI agent that played our generated levels while being evaluated by the DDA method. This agent was also used to set benchmark values for the said DDA method. To guarantee the adequacy of the content, we consider that the content is acceptable when the difficulties can be discerned, and when the agent can on average achieve a suitable bound value of ending difficulties (e.g. [4; 6]). For its

behavior, the agent uses the decision-making system of a decision tree which allows it to identify the geometry surrounding it and execute a combination of inputs to complete the geometry. This decision tree is composed of sequences and a non-sequence. Sequences are sequences of inputs that allow the agent to complete more complex geometries, while the non-sequence is the behavior used when the agent does not detect any geometries that are related to sequences. This decision-making system allows the agent to complete every geometry isolated (the agent's starting point in the game is the same as the geometry and there is nothing else in the area that occupies that geometry) while making it simple to establish the benchmark values for the DDA method.

However, the negative aspects of this approach were clearly identified. First, the agent can only fail in specific geometries (geometries with enemies), and when performing a sequence, the agent is locked into that sequence until all sequential inputs have been performed or it ran out of time. Thus, the agent adds a bias to the test results since we determined where the agent should lose. The agent being locked into a sequence was not considered as significant as the previous issue, seeing as the probability of it occurring is low. However, this sequence lock could have made it complicated to interpret why the agent was losing during the tests.

Chapter 4

Testing The Level Generator

With the AI agent finished, the project was ready to proceed to its test phase. Before initiating the analysis of the results, there was a need to establish how the testing process was going to occur while justifying the approach used. In this testing phase, three goals needed to be accomplished, these were the discernibility of the difficulties, the conversion of the agent's ending difficulty to a bound value, and ensuring that the change in difficulty was adequate. Once results were obtained, they were analyzed to determine if the goals were achieved, however, when the results proved inadequate, the parameters of the generator and of the DDA method were altered to improve the results. This method was repeated until a generator with the desired properties was obtained. When the generator possessed these properties, a test with real players was conducted to acquire the players' information and opinions regarding the generator's content.

In the agent's tests, we decided to execute a methodology similar to the ones used in the studies mentioned in section 1.6. In our case, we decided to evaluate the fluctuation a few metrics with the change in difficulty over multiple runs of the game. These methods frequently use an adequate sample size to reach the expected value due to the quality of the convergence being correlated with the number of samples. Thus, to obtain reasonable results and formulate adequate conclusions in regard to the generator's content, a adequate number of runs needed to be performed. For the initial test, we consider that given the space complexity, 100 runs per difficulty would be sufficient cover up the search space. By examining the results of the agent and the answers given in the quiz (while taking into consideration the context of the player's test results) it is possible to validate the generator.

4.1 Testing With The AI Agent

Evaluation Metrics

To test the AI agent, there was need to establish evaluation metrics that were able to discern the difficulty of the levels. In subsection 2.3.1, some parameters were used to determine the performance of a player. Our initial idea was to observe these metrics and attempt to distinguish the difficulties through the performance while analyzing if the adaptation was adequate. Ideally, the performance score would indicate how well the agent was playing the game and the change in difficulty, thus based on that change, it would be possible to discern if the agent was in a low or high difficulty. Another idea was to extract and analyze any metrics that had a noticeable relation with the performance score. Nonetheless, even if some of these parameters were not

used to validate our generator, it was mandatory to check if any could be used to reach our goals. The metrics used in the DDA method chosen to be extracted were the following:

- **Total Performance/Performance score** - the performance value obtained at the end of a rhythm.
- **Time to complete a rhythm** - the number of ticks the agent takes to complete a rhythm in comparison to the expected value.
- **Time spent moving backward** - the number of ticks the agent had negative velocity (moved back) after completing a rhythm in comparison to the expected value.
- **Coins picked up** - the percentage of the coins picked up during rhythm.
- **Number of enemies defeated** - the percentage of enemies defeated in the rhythm.
- **Damage taken** - the number of times the agent was damaged during the rhythm (this value can be 0, 1 or 2 depending on Mario starting mode 2.3.1).
- **Length traveled** - the percentage of length traveled by the agent in the rhythm (or the level depending on the test).

Due to the possibility of the generated geometry of each rhythm being drastically different, we decided to observe other parameters that could have a relation with the performance score. However, it seemed logical to find metrics that were correlated with the type of geometry since the agent's behavior cannot handle certain level chunks. Even though some rhythms might have the same difficulty value, the agent struggles when completing geometries that contain enemies (3.2.4), therefore, if a rhythm contains various geometries of this type, then its performance score should be inferior when compared to other rhythms. For this reason, the following metrics were extracted:

- **Number of dangerous geometries** - the number of geometries in the generated rhythm that can punish the player by damaging him or ending the game.
- **Percentage of dangerous geometries** - the percentage of dangerous geometries (in relation to the total number of geometries).
- **Number of geometries with enemies** - the amount of geometries in the rhythm that generate enemies.

4.1.1 Tests Without Adaptive Difficulty

Test 1

In the first test, we wanted to analyze how the performance score of the agent would vary when trying to complete multiple levels that were generated with the same rhythm (ensuring the stochastic generation explained in subsection 1.3.1, while distinguishing the difficulty). This test could indicate that, if different geometries were being generated when using the same rhythm, then these could be distinguished by the performance value obtained. Test 1 also allowed us to observe the fluctuation of the extracted values with the increase of the difficulty.

Thus, the following test was proposed, for each difficulty, 10 different rhythms would be generated. For each rhythm, the agent would play 10 levels, and after these 10 levels, a new rhythm would be generated. This process is repeated 10 times allowing for 100 runs per difficulty and 1100 runs in total. Once the agent finishes a rhythm or loses, the variables are extracted for observation, and the game is reset for another run. In every run, the AI agent starts with Mario mode 2, therefore it is possible to detect when it takes damage at least two times before ending the game (Mario mode is explained in subsection 2.3.1).

To grasp the state of the generator, it is imperative to choose from the established metrics which ones were suited to distinguish the difficulty of the levels, and to evaluate the content's adequacy. To discern the difficulty, the most logical parameters to study were the performance score and the length traveled in the level. The performance score was chosen due to it being considered an estimate of how well the player is playing and directly impacting the change in difficulty. In a game run where the length of the level is considerably large, the length traveled by the agent could indicate how hard a level is since it determines how far the agent went in a given level. For example, if two different levels are considered and the agent goes farther in one of them, then it could mean that that level is easier. In theory, both these metrics should decay as the difficulty increases due to the game progressively becoming harder for the agent, thus it is more difficult to achieve progress without committing mistakes. Ideally, the difficulties should be discerned through data analysis two game runs and by comparing their performance and the length traveled. Thus, by observing the results of the two games, it should be clear which one possesses the greater value. Nonetheless, the innate randomness of the generator complicated this process since a rhythm that is considered easy for the agent could still be generated in the highest difficulties (despite the odds being low). To ensure an adequate interpretation of the results, multiple runs per difficulty were executed, therefore, as mentioned previously, in Test 1 the agent played 100 levels per difficulty.

However, this evaluation process of the generator contains some biased that is related to the weights applied to the individual performance scores (explained in subsection 2.3.1) and the AI agent's behavior. If the agent has a difficult time in a specific geometry and that geometry is generated more often, then the agent might lose earlier (travel less length) and achieve an inferior performance score than it would otherwise have in another rhythm with the same difficulty. An example where the agent's behavior can influence the results is the following, consider two rhythms R_1 and R_2 composed of the same actions and with the same difficulty value. However, R_2 has its first action A_F swapped with its last A_L, and A_L generated a geometry that the agent has a harder time completing. Additionally, consider that the agent loses if it makes one mistake. Therefore, in R_1 the agent will travel more distance than in R_2 even though the actions in these rhythms have collectively the same difficulty. In this scenario, both rhythms have the same number of actions with the same type and the same difficulty, but the agent loses sooner in R_2 and, thus obtains a worse score and travels less length. Nevertheless, this specific scenario can only occur if the agent starts the rhythm in Mario mode 0 (making one mistake loses the game), which in this test never happened due to the agent only playing one rhythm per run. However, if two equal rhythms are considered and both have their hardest geometries placed at the end of the level, then if we change the position of these geometries in one of the rhythms, the agent will attain different scores and length values in both rhythms even if it starts with mode 2 (can commit two mistakes before ending the game). Nonetheless, these scenarios should only occur when multiples level chunks with enemies are generated at the beginning of a level.

When it comes to the weights of the individual performance scores (explained in subsection 2.3.1), there will always be a bias that is introduced, because we decide their weights. Initially, the weights of each parameter began with equal value, however as more tests were conducted, changes were performed to better suit our agent and improve the generator's content.

When observing the extracted data some issues were spotted. Firstly, as expected, not all of the extracted metrics had a clear connection to our performance and length values. Since there was no visible relation between these parameters, some evaluation metrics were discarded for this study. For example, no correlation was found when comparing the number of dangerous geometries, and the number of enemies to the total performance score and length traveled.

Another issue is that, since we are reusing rhythms there is some inconsistency in the results that without context can be misinterpreted, which can lead to incorrect conclusions about the state of the generator. By using the same rhythms, different results in geometry and scores were achieved, which means that different content is being generated with the same rhythm. However, even if the geometry differs the performance score was either very similar, or it drastically changed in the same rhythm. This inconsistency in the test's results was presumably caused by the randomness of the generator's content and the agent's behavior. For example, in cases where a very easy rhythm for our agent is generated, the performance and length values were greater than in the other nine rhythms of the same difficulty, skewing 10% the data to be higher than the average of that difficulty. Therefore, the randomness and the reuse of rhythms can inflate the score in some runs. Thus, to better discern the difficulty, it is imperative to conduct tests where the current rhythm is constantly changing (to remove rhythm bias).

To observe and interpret the extracted data, a box plot graph was constructed which reveals the performances scores obtained in each (visible in Figure 4.1).

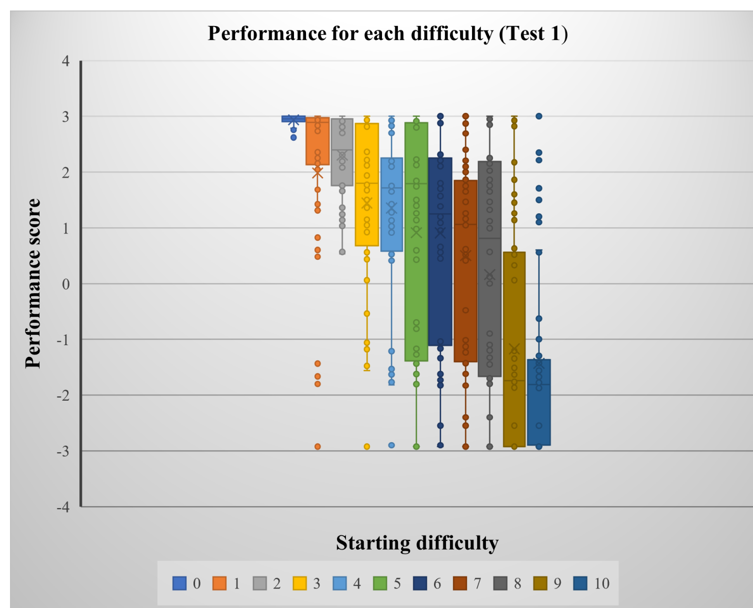


Figure 4.1: Box plots of Test 1's performance scores in each difficulty with the outliers.

In Figure 4.1, despite the dispersion in scores, it can be inferred that the median, the lower whiskers, and the averages in Figure 4.2 decrease as the difficulty increases (with the exception of difficulties 3's average). Figure 4.2 was constructed without outliers, however a version of this graph with the outliers can be seen in Figure B.9 in appendix B (the box plots without the

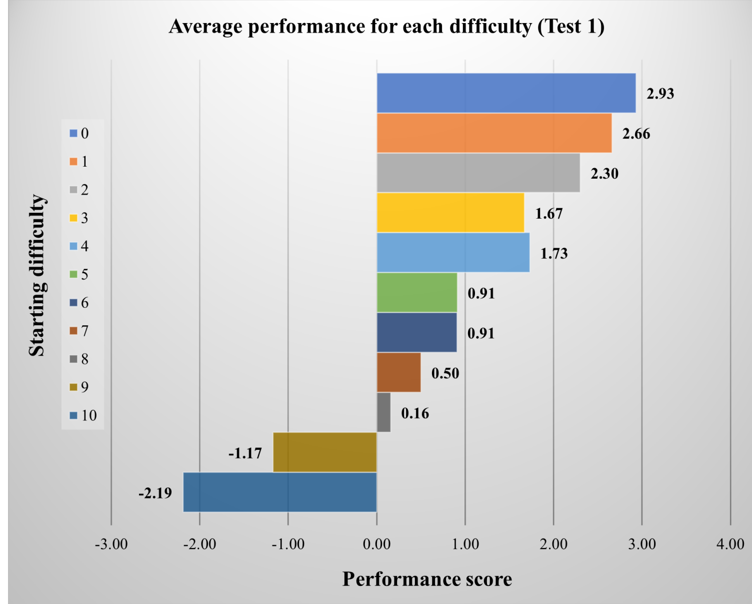


Figure 4.2: Average of Test 1's performance scores in each difficulty with the outliers removed.

outliers can also be observed in Figure B.10 in appendix B). Still, the dispersion is apparent when observing the positions of the quartiles since these do not decrease accordingly. In the first 5 difficulties (from 0 to 4) the position of the quartiles seem in tune with what was theorized, but in the rest of the plots (from 5 to 10), their positions are inconsistent. Another point to note is that there exist some outliers in difficulties 1 and 10, nonetheless, these appear in line with what was expected. For difficulty 1, these outliers are presumably due to difficult geometries (for the agent) being generated at the beginning of the level, and for difficulty 10, geometries that the agent finds complicated were not generated, therefore the agent is able to complete the levels (influenced by the rhythm bias). For example, the position of the upper whisker only changes in difficulty 10, which likely stems from this issue. The lowers whiskers of difficulties 5 to 10 reach a score of -3, implying that the agent lost several times at the beginning of the game. The dispersion of the results, it severely noted in difficulties 5 to 9 due to the whiskers enclosing the entire interval of scores of [-3; 3]. However, when it comes to the standard deviation σ of these difficulties, when compared to the other plots, these only differ by what can be considered a small margin. For example, the difficulty 7's standard deviation is $\sigma_7 = 1.73$ while difficulty 1's is $\sigma_1 = 1.75$. The value of σ_1 stems from the outliers present in difficulty 1's plot. However, by removing these outliers a value of $\sigma_1 = 0.47$ is obtained, which is a better representation of the dispersion of this plot. The same logic can be applied to difficulty 10 seeing as $\sigma_{10} = 1.66$, but by removing some outliers a value of $\sigma_{10} = 0.66$ is acquired. Despite the plots of difficulties 3 and 4 only containing a few outliers, their σ values are similar to rest of the results these being $\sigma_3 = 1.60$, $\sigma_4 = 1.49$ (and $\sigma_3 = 1.29$, $\sigma_4 = 1.00$ without the outliers). The exceptions in this pattern are difficulty 0 due to its high precision and difficulty 2 ($\sigma_0 = 0.07$, $\sigma_2 = 0.72$, which do not contain outliers). Difficulty 0's results are expected since it is impossible to lose in its generated geometry. When it comes to difficulty 2, it is assumed that easy rhythms were generated, and therefore this plot has the highest precision (when compared to the plots of difficulties 1 to 10). To sum up, when it comes to the standard deviation, the results show that our generator could be considered to have high variance (σ^2) results despite the positive behavior of the average.

Another method of studying the data dispersion is through the analysis of the plot's median, quartiles, and whiskers. In difficulties 5 to 10, when observing the positions of the quartiles and the whiskers, it can be inferred that in various runs the agent consistently achieved greater performance scores than in the difficulties beneath, which ideally should not occur. The opposite also transpires in difficulties 0 to 4, where there are instances of inferior scores when compared to the difficulties above. Difficulty 5's plot reveals both these phenomena. In this plot, the AI agent was able to consistently achieve runs where the performance score is greater than in difficulty 4. This is evident by observing the position of the whiskers and the quartiles in both box plots. In the area between difficulty 5's third quartile and upper whisker, the interval of scores is [2.88; 3] and for difficulty 4 the interval is [2.25; 3], therefore in the agent's best 25% runs of difficulty 5 it attained a minimum score of 2.88, while in difficulty 4 this minimum score was 2.25. The same process can be applied to other areas of the plot, for example, when comparing the interval of values between the median and third quartile of both difficulties ([1.72; 2.25] and [1.79; 2.88] for difficulties 4 and 5 respectively), it is evident that the agent frequently obtained greater scores in difficulty 5. However, when comparing the area between the median and the lower whisker of both plots, the agent often acquired inferior scores in difficulty 5, as intended. For example, in the regions between the lower whisker and first quartile (the 25 runs with the worst scores) the agent obtained intervals of [0.58; -1.77] and [-1.39; -3] for difficulties 4 and 5 respectively. In difficulty 6, the same region encapsulates the interval of values of [-1.11; -3], which means that in the same 25% worst runs, the agent obtained inferior scores in difficulty 5. To summarize, in difficulty 5, the agent frequently obtained greater and inferior scores when compared to difficulty 4, and it often attained worse scores than in difficulty 6, which is presumably due to a combination of the randomness of the generator and the AI agent's behavior, as mentioned previously.

One factor that could explain the sudden drop in the position of difficulty 5's first quartile is the probability of an enemy being generated. In difficulty 5, the geometries that include enemies become too complex for the agent to handle since its behavior is not optimal to complete these instances. Thus, when an enemy is generated, it is more likely for the agent to take damage or lose the game resulting in inferior scores. Nonetheless, there was no way to determine if this was the case, due to there being no relation between the performance score and the number of geometries that had enemies. The type of actions where the agent lost were also extracted and no relation with the performance score was spotted.

The inconsistency of the values is also apparent in other difficulties. Difficulty 6 to 8 have similar third quartiles and upper whiskers to difficulty 4, yet their first quartile is more akin to difficulty 5's first quartile. However, when comparing their medians, difficulties 6 to 8's are considerably beneath 4 and 5's. Still, in 6 and 8, the agent achieved performance scores above 2 with similar consistency as in difficulty 4 and obtained inferior scores to -1 as often as in difficulty 5.

When it comes to the plots for difficulties 9 and 10, these seem consistent with what was expected. Both plots have the lowest performance scores out of all the difficulties, with 10 having inferior scores more consistently than in 9, evident by the position of their third quartiles. In difficulty 10, the agent obtained scores between [-1.36; -3] in 75% of the runs (with the lowest scores), while in difficulty 9 these values differ between [-0.56; -3]. The first quartile of both plots show that the agent achieved a score around -3 in 25% of the runs. Still, in difficulty 9, the agent was able to attain scores above 0.5, 25% of runs, which as mentioned previously is

probably due to the randomness of the generator.

Some problems with the test are identified. First, by using the same rhythm several times, there is a biased for that specific rhythm, therefore if a rhythm is easy regardless of the difficulty (generated multiple actions that are easier for the agent), then the agent completes the level with a high performance score. In the opposite case, if the rhythm is harder, then the agent completes the level by taking damage or loses and obtains a score that is significantly inferior to the average. If the rhythm's difficulty is judged only by the performance score, then a level is easy or hard based on the number of times the agent fails to complete a geometry successfully and the location of the difficult geometries. If these geometries are placed earlier in the level, then these have a more substantial effect on the score, due to the agent making mistakes sooner and traveling less, and therefore, it achieves an inferior value of length traveled. Since one rhythm occupies 10% of the data of one difficulty, if there are multiple easy rhythms, then our data is skewed to have performance scores that match these easy rhythms.

Another possible issue is the dispersion in the results. In Figure 4.1 in difficulties 5 to 9, the area between the whiskers covers the entire scope of possible scores. Assuming that the goal is to discern each difficulty by using the performance, then having results with an inferior σ can facilitate this process. This variance is presumably caused by the generator's randomness and the agent's behavior, therefore a possible solution is modifying the odds of generating actions that are more difficult for the agent. With this change, the dispersion should be reduced making it easier to distinguish each difficulty. Another point to note is that, when it comes to the dispersion, the standard deviation does not provide as much information in regard to the state of the generator, thus the box plot analysis appears to be better suited for this study.

Nevertheless, when observing Test 1's results the bias does not seem to be negatively affecting the decrease of the performance scores with the difficulty (the discernibility was considered decent but it could be improved). However, to better comprehend the generator, another similar test was conducted with the goal of observing more results and conclude with more certainty this bias is an issue. If the results were consistent, then reusing rhythms does not hinder the quality of the results. On the other hand, if the results were not analogous, then rhythms should not be reused if the goal was to discern difficulty. The rhythms were only reused to ensure the stochastic generation was implemented correctly, and by observing the different scores represented by the outliers in difficulties 1 and 10, it is clear that the same rhythm can generate different geometries.

If the number of runs was increased, the discernibility of difficulty may become easier. However, by increasing the number of runs the computation time of each test is also increased. Each test takes a long period of time to execute due to the agent having to play the level in real-time, thus, to make the most out of every test, cautious changes were performed before conducting further tests. However, for Test 2 no changes to the generator or the DDA method were made.

Test 2

To further understand consistency of the dispersion in the results, another test was conducted, but instead of changing difficulties, it was decided that only difficulty 5 would be studied. This new test consisted of 500 runs where the data of each segment of 100 runs would be used to construct one box plot graph (a sub-test). For every 10 runs of a sub-test, a new rhythm was generated, which equates to a total of 10 rhythms per sub-test and 50 rhythms for the entire study.

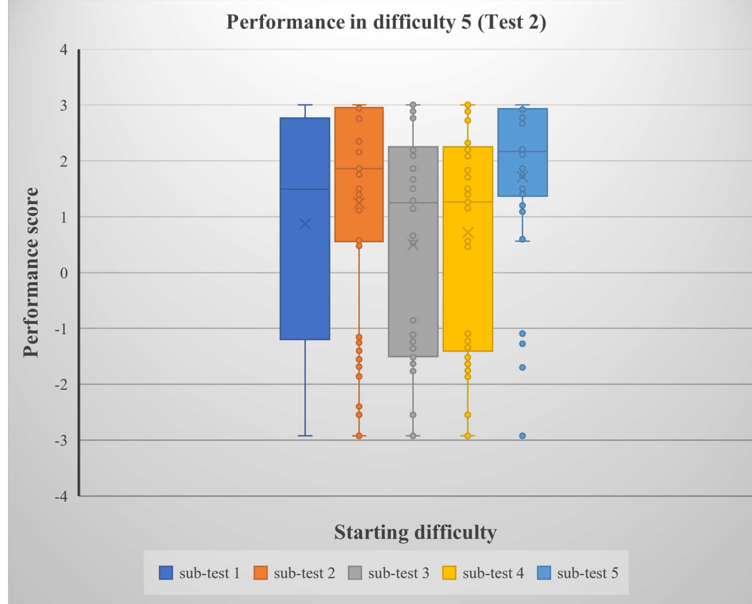


Figure 4.3: Box plots of Test 2's performance scores in difficulty 5.

The results of the box plots in Figure 4.3 confirm that these tests have a rhythm bias due to the inconsistency of the dispersion. In the difficulty 5's sub-tests 1, 3, and 4, the results appear consistent. These plots have their upper whisker around 3 and their lower whisker at -3. However, sub-tests 2 and 5 differ from this group by the position of the quartiles, and 5 also has its lower whisker significantly higher than the other plots.

In sub-tests 3 and 4, by observing the area between the upper whiskers and the third quartile, it can be affirmed that the agent achieved scores between [2.25; 3]. When compared to the other sub-tests, this zone, which consists of 25% of its best scores, covers a larger interval of values. For example, sub-test 1 first quartile has the value of 2.76, thus, in the other plots, the agent frequently attains greater scores than in sub-tests 3 and 4 in its best 25 runs of each sub-test.

The first quartiles of sub-tests 1, 3, and 4 appear to be similar, seeing as their values are respectively -1.20, -1.51, and -1.41. However, when observing sub-test 2's first quartile (0.56), it can be inferred that the agent acquired negative scores less frequently in this sub-test. Therefore, when compared to sub-tests 1, 3, and 4 the agent achieved greater scores in its worse 25 runs in sub-test 2. Similarly, sub-test 5's first quartile and lower whisker have considerably greater values than the other plots (1.37 and 0.59 respectively), thus this sub-test has significantly less dispersion than the rest of the plots due to the agent obtaining higher scores than the other sub-tests more often. This inconsistency in the dispersion is again attributed to the randomness in the actions, the reuse of rhythms, and the agent's behavior. Another point to note is that the outliers of sub-test 5 represent runs where the agent lost. These might stem from the generated rhythms being drastically easier for the agent except for one rhythm (represented in the scores as the outliers).

When it comes to the medians, in sub-tests 3 and 4 this value is 1.25, while in sub-test 1's the median is 1.49. Therefore, the agent obtained better greater more frequently in sub-test 1, however, the median seems to increase with the position of the first quartile, thus it is apparent that in regard to the median these plots are similar.

When it comes to the standard deviation of the plots, these have values of $\sigma_1 = 1.98$, $\sigma_2 =$

1.87, $\sigma_3 = 2.19$, $\sigma_4 = 2.01$ and $\sigma_5 = 1.53$ (without outliers $\sigma_5 = 0.73$). These σ values encompass what can be considered a sizeable interval of scores and indicate the magnitude of the dispersion without adding more relevant information. Therefore, it appears that, similarly to the previous test, using the standard deviation as a means to interpret the data is not the most adequate method of evaluating the generator's content.

However, sub-tests 2 and 5 show less dispersion due to the agent attaining consistently greater values of performance when compared to the other sub-tests, presumably due to the generated rhythms being easier for the agent, and thus the value of sub-tests 2 and 5 third quartiles is considerably greater at 2.95. Therefore, when comparing this value to the third quartile of sub-test 1, it is assumed that easier rhythms were more frequently generated in sub-test 1 in comparison to sub-tests 3 and 4, but not to the same degree as sub-tests 2 and 5.

The results of Test 2 show that, if the goal is to study the adequacy of the generator's content, then it is not viable to generate the same rhythm multiple times. Even though it was noted that the same rhythms can generate different geometry, which leads to different results in performance, this methodology is inconsistent when it comes to distinguishing difficulties. Generating different rhythms could shift the results to more favorable outcomes since the rhythm bias is removed. Nonetheless, with more rhythms, it was assumed that the considerably high dispersion of the results would still be present due to the generator having more opportunities to generate geometries of different difficulties for the agent, thus obtaining various high and low performance scores.

However, with Test 2 it was possible to observe and confirm that the agent adds some bias to our generator. Since the dispersion was expected to remain, some precautions were considered to facilitate the discernibility of the difficulties. Our first option was to further increase and decrease the odds of generating harder and easier geometries respectively as the difficulty goes up. Implying that results should be skewed allowing for fewer runs of high scores when it comes to difficulties 5 to 10. However, due to the generator creating more geometries that are harder for the agent (in the higher difficulties), the variety of types of action in a rhythm is reduced (the geometries presented will differ less when it comes to the inputs that need to be performed). Another option was to modify the DDA method by changing the amount of performance acquired with each metric and their weights to better suit our agent's behavior, which should alter the position of the median, quartiles, and whiskers of the box plot graphs. In this case, the metric of the number of enemies defeated could be increased (hardest geometries for the agent) while decreasing other metrics.

The negative aspect of this adjustment is that certain playstyles (the manner in which the player plays the game) are punished with less score. For example, a player that is fast at completing levels, but often misses enemies and does not defeat them receives less score if this change is implemented. Therefore, if the enemies defeated metric has a sizeable weight when compared to everything else, then the score this player attains is inferior, even though his time taken metric is considered good. Still, our idea was to try to reduce the dispersion by carefully changing the generator with both options.

4.1.2 Tests With Adaptable Difficulty

Test 3

For this new test, it was decided that the adaptive difficulty method (section 2.3) would be used and that no change to the action probability and DDA method should be performed. Presumably, by modifying the odds of generating certain actions that are more difficult for the agent, the problem of generating a complicated action earlier in the level which makes the agent lose or be punished should appear less frequently in the lowest difficulties. However, since the rhythms are not being reused, it was decided that it would be beneficial to study the results with the rhythm bias before changing other parameters.

The previous results showed that generally, the performance scores is decreases as the difficulty increases. Even if the dispersion of the results was considered an issue, this concept still applies. Seeing that, our main goal was to create a generator that adapts the difficulty to the player's skill, a methodology that helps confirm this fact is the observation of the agent's ending difficulty value. If it converges to an adequate bound value, the generator's DDA method is considered acceptable. However, another goal is to be able to discern difficulties through data analysis. To distinguish the difficulties, other metrics were considered for this study, mainly the length traveled in the level. For example, when comparing two levels, if the agent goes farther in one of them, it could mean that that level is easier for it to complete.

In this test, when a run began, a new rhythm and its geometry were generated. If the agent completed the first rhythm, a new rhythm was generated, however, the geometry beyond the first rhythm was influenced by adaptive difficulty. The difficulty of subsequent rhythms was determined by the difficulty of the previous rhythm added to the rounded-up performance score obtained in the previous rhythm. A game run, in this test, consisted of the agent attempting to finish the game by completing rhythms that were procedurally generated. A game run ended when the agent completes 10 rhythms or when it lost. In Test 3, 1100 runs were executed, in which, for every 100 runs, the game began with a different difficulty. Therefore, there were 100 runs where the agent started in each difficulty.

When it comes to what metrics should be analyzed, it was predicted that the most useful metrics would be the performance score in the first rhythm, the length traveled, and the difficulty where the game ends. The agent's performance in the rhythms beyond the first does not seem logical to evaluate because its Mario mode (or health system) can differ from the first rhythm if the agent takes damage. Therefore, if the agent takes damage in the first rhythm, then it has fewer chances to make mistakes in subsequent rhythms (has less health), which in turn may cause it to lose earlier in a rhythm (travel less length), and obtain a worse performance score in the process. Thus, this score is less likely to be correlated to the average score of that difficulty (making data analysis inconclusive). In addition, in cases where the agent begins a rhythm in mode 0 (if any damage is taken the game ends), and travels an arbitrary percentage of length and loses, it can receive a greater score than if it started with mode 2 (can take damage twice without losing). If the agent begins in mode 2, continues beyond that percentage of length, and gets damaged a total of three times resulting in a loss, it obtains an inferior score than if it started with mode 0 due to the damage taken metric. For these reasons, it is not logical to compare a box plot graph of rhythms beyond the first with the plots of Test 1.

Observing the performance scores of the first rhythm allowed us to conclude how the performance varied with different rhythms and if was possible to distinguish the difficulties with this

metric.

In Test 1, the length traveled was not directly studied since the goal was to discern the difficulties through data. Since the length is correlated with the performance score, it is likely that the runs where the agent attained the highest scores were the runs where it traveled more length. However, in Test 3, the agent was allowed to go beyond the first rhythm, thus by observing the total length traveled it was possible to analyze the DDA method adequacy. Since there was a low number of mistakes the agent was allowed to make before losing, it should travel more length when starting in a lower difficulty and less when starting in a harder difficulty.

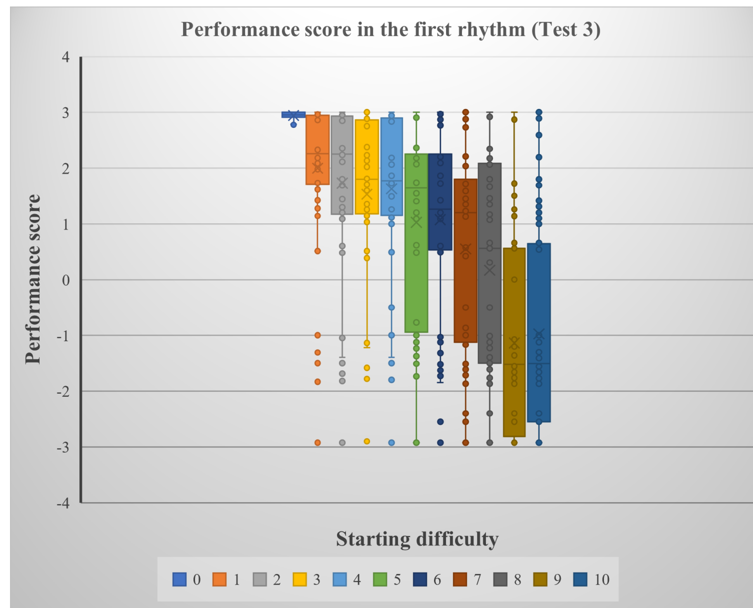


Figure 4.4: Box plots of the performance scores in each starting difficulty in Test 3 (for the first rhythm).

In Figure 4.4, a box plot of the performance scores of the first rhythms generated in each run for each difficulty can be visualized. When it comes to difficulties 0 and 1, these appear to have similar results to Test 1 (Figure 4.1), however when comparing difficulties 2 to 4 the position of their quartiles and whiskers are very similar unlike Test 1. Still, the median of difficulties 1 and 2 are around 2.25, meaning that the agent obtained scores between [2.25; 3], 50% of its best runs. When it comes to the median of difficulties 3 and 4, its value is around 1.80, thus in both these difficulties, the agent acquired a score between [1.8; 3], in the same 50% best runs. Therefore, despite their lower whiskers being inconsistent, these difficulty groups are discernible through the median. Even so, the constant generation of new rhythms seems to have affected the results of difficulty 4. When compared to Test 1, the agent achieved greater scores more often in Test 3, thus the position of its first quartile changing from 0.58 to 1.18. The intervals of scores of these groups are similar due to the biggest difference between difficulties 1 and 2, and 3 and 4 being the increase in enemy movement speed. In these difficulties, despite this change, the agent can still defeat them without any issue, thus the results have some similarities.

Nevertheless, these results are not ideal since there only is a small difference in scores between difficulties 2 to 4 and their lower whiskers are slightly inconsistent (difficulty 3 as its lower whisker is closer to -1 than difficulties 2 and 4), still, these results are considered very positive. Another point to note is that the outliers and the lower whiskers of these plots indicate that the agent

lost in less than 25% of the runs. However, with this data, it is not possible to conclude if these generated levels would have the same difficulty for a human player.

As mention previously, outliers are present in difficulties 1 to 4 but no outliers can be seen in difficulty 10, presumably due to Test 3 not reusing rhythms, thus the agent could acquire greater and lesser scores more frequently. Therefore, the dispersion was increased and outliers can be seen which presumably occur due to the agent losing earlier in the game akin to Test 1.

The plots of difficulties 5, 7, and 8 in Test 3 seem to have similar results to Test 1. Their whiskers, median, and quartiles are similar to Test 1, with the exception of difficulty 5's third quartile which has an inferior value than in Test 1, more akin to the plots of sub-tests 3 and 4 of Test 2. The inconsistency of the results is more prevalent in difficulty 6 where the first quartile changed from -1.11 to 0.53. This new value is greater than the first quartile of difficulties 5, 7, and 8, which again shows the randomness of these results. Since the geometries of difficulties 5 and 6 are similar, it is likely that with more samples, these results could have more similarities, however, this was not feasible due to the computational resources required to perform a test of this magnitude. It is again assumed that all these incongruences in the results stem from the actions that the agent finds the most difficult being generated less often in difficulty 6. Another relevant point to note is that, as mentioned previously, the agent begins to struggle with geometries in difficulties 5 and above, thus the dispersion is naturally greater in these plots.

When it comes to difficulties 9 and 10, it is evident that the previous results of difficulty 10 in Test 1 had a bias toward the generated rhythms and that there is no clear difference between both difficulties, when it comes to the performance score. In this case, difficulty 9 has lower scores than difficulty 10 which is again attributed to the agent's behavior and the randomness of the rhythms.

Even though it is complicated to discern difficulties 5 to 10, there were other metrics to consider for this test, mainly the length traveled. This metric could not be used to evaluate the generator's content in Test 1, due to it being complicated to distinguish how difficult a level is if its length is considerably small. However, this metric is more suited to study the generator's adaptation. By having more than one rhythm, it is possible to determine how far the agent could go in the level when starting in a given difficulty. Thus, the percentage of the total length traveled by the agent was studied with the goal of observing the quality of the adaptation and the possibility of it being beneficial when it came to discerning the difficulties.

In Figure 4.5, it is evident that generally the percentage of the average length traveled along the level decreases with the starting difficulty, as it was intended. On average when starting at difficulty 0, the agent completes three rhythms, when starting at difficulties 1 to 4 it clears two and the number of completed rhythms decreases as the difficulty the increases (each rhythm is composed of 100 length which equates to 10%). The difficulty groups of 5 and 6, and 9 and 10 do not follow the same pattern but differ by what can be considered a small margin. Thus, it can be inferred that the average length traveled appears to be a good metric when it comes to distinguishing the difficulties.

In Figure 4.6, it is possible to observe the dispersion in the percentage of length value across all starting difficulties. By visualizing these plots, it can be inferred that despite the dispersion, the difficulties can, in most cases, be discerned,with the exception of the difficulty groups mentioned previously (5 and 6, and 9 and 10). The most consistent value that decreases with the difficulty is the third quartile. Even so, even if the whisker and the first quartile do not follow this pattern perfectly, these results are still considered positive. This dispersion and

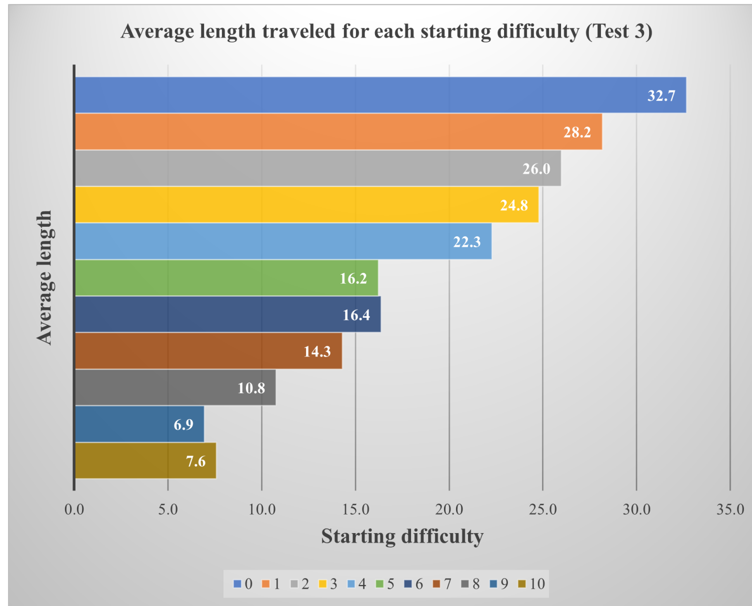


Figure 4.5: Average values of the length traveled for each starting difficulty in Test 3.

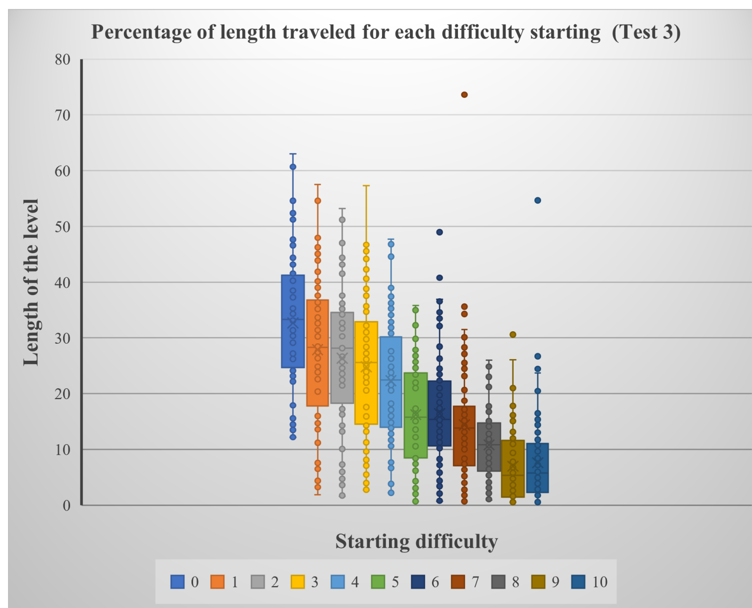


Figure 4.6: Box plots of the length traveled for each starting difficulty in Test 3.

inconsistency are again due to the type of geometry generated. Since our agent has a harder time dealing with enemies, it is likely for it to lose while in a geometry that contains them. In some runs, more enemies might be generated, and therefore the agent travels less length. The other possibility is that the agent is losing to a group of interfering sequences which can also have affected the results, as explained in subsection 3.2.4. On the other hand, the first test showed that there was no clear relation between the number of enemies and the performance score, so that data was not studied.

Since the position of the whiskers, quartiles, and medians of plots in Figure 4.6 are in tune it what was is expected (a decrease of length with the starting difficulty), the only possible issues with this data are the dispersion and some inconsistencies in the pattern. Nonetheless, if these

factors are not considered a problem, then the results seem to be positive.

In both Figures 4.5 and 4.6, it is apparent that the agent never finished the game, even in the lowest possible difficulties (never reaches 100%). The generator was making the game harder as the agent completes each rhythm, however, the agent was unable to complete some of the geometries, and so it lost before finishing any levels. There could be two reasons for this outcome, either the level was too long, and it needed to be shortened or the changes in difficulty were too abrupt. The second option was easy to validate, however, due to the time it took to perform these tests, a variety of changes needed to be implemented to improve the quality of the next test's results.

The last metric to study was the difficulty where the agent ended the game, and for this evaluation, the same box plot and average graphs were constructed.

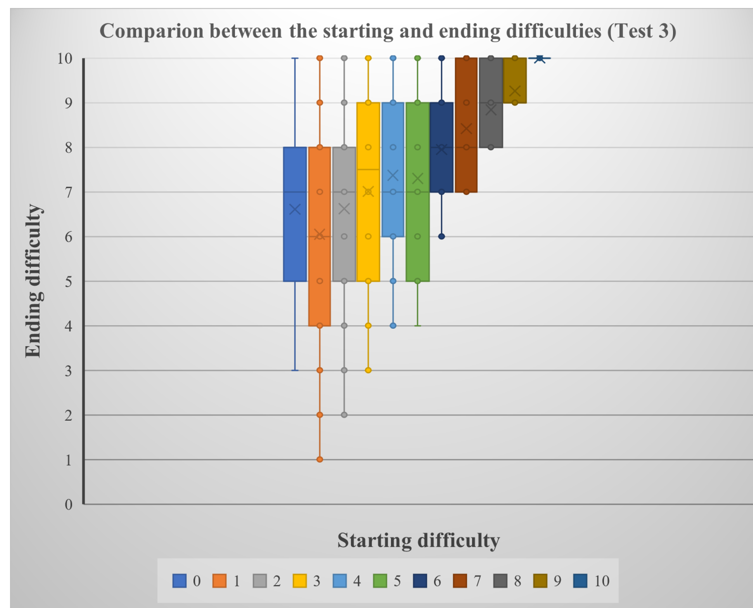


Figure 4.7: Box plots comparing the starting and ending difficulty in Test 3.

In Figure 4.7, it is evident that the end difficulty increases with the starting difficulty. Since the test only allows the agent the possibility of losing once, the difficulty of the game never decreased, with the exception of difficulty 5, visible by its lower whisker (25% of the runs ended either in difficulty 4 or 5). This implies that it was very rare for the DDA method to attribute the agent with a negative score unless it lost, thus making it improbable that the difficulty reduced when the agent completed a rhythm. When the agent lost, it always received a negative score which would decrease the difficulty if the agent kept playing. Another factor that could contribute to these results was the abrupt changes in difficulty. If the changes were too extreme then the agent would be placed in a level with geometries that were drastically more difficult, resulting in it losing.

One aspect to note about Figure 4.7 is that in the starting difficulty 1 the agent obtains lower end difficulty scores than in difficulty 0, due to the agent always receiving a score that rounds up to 3 in the first rhythm. In difficulty 0, it is impossible to lose (unless the time runs out), thus the lowest end value of the starting difficulty 0 is 3.

From these results, it seems that the generator was adapting too fast and not punishing the agent as much as it should. Therefore, in the next test, changes to the DDA method needed to

be implemented to improve the results.

Test 4

To improve the results of Test 3, the following changes were implemented:

- **Total performance score** - the interval possible scores was changed from [-3; 3] to [-2; 2]. The goal of this change was to slow down the adaptation process.
- **Weights of the parameters** - the weights of the parameters of each individual performance score were change to better suit our agent. By slightly increasing the weights of the enemies defeated and of the damage taken metrics, and decreasing the both times and the coins picked up metrics. In cases the where the agent fails, the length score was also changed to be lower since its weight is given by the sum of both time weights.
- **Odds of generating geometries** - the odds of generating actions that were in theory harder for the agent such as jump actions in higher difficulties were increased slightly (specifically the odds of an enemies being generated were increased).

Since the length traveled proved effective when it came to discerning the difficulty, and the results of the performance score were inconsistent, we decided to not study the performance and focus on other metrics. Therefore, the length traveled and the difficulty at which the agent ended the game were analyzed and compared with the previous test.

Test 3 required an unreasonable amount of computational resources to complete, thus it was decided that the next test would be shorter, and consequently, a smaller sample size of data would be gathered. Test 4 consisted of 30 runs of the game for each difficulty where all the generated rhythms were random. A run ended when the game was completed, or if the agent lost. For this test, the main goal was to observe if the data acquired could reveal if the changes done to the generator were an improvement to the adaptability of the generator.

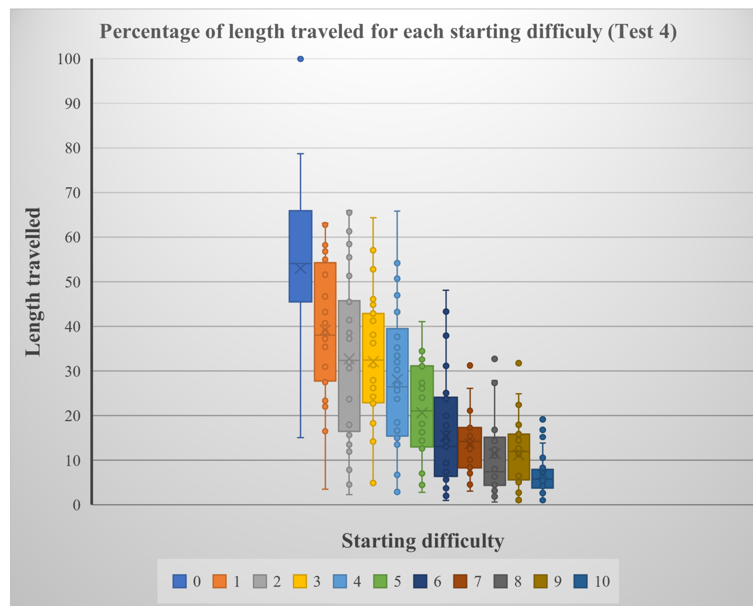


Figure 4.8: Box plots of the length traveled for each starting difficulty in Test 4.

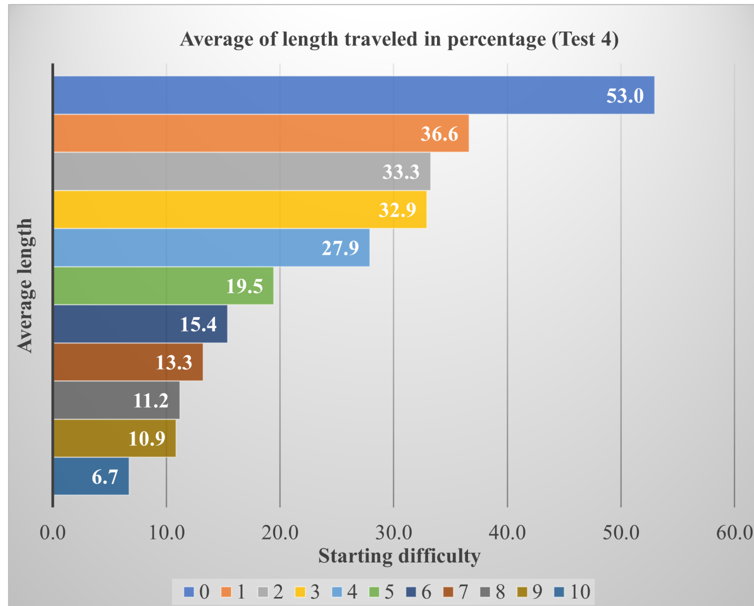


Figure 4.9: Average values of the length traveled for starting difficulty in Test 4.

In Figures 4.8 and 4.9, the results of the length traveled in percentage and its average can be observed. It is immediately evident that the average length results are more consistent than in Test 3, seeing as the length traveled decreases as the starting difficulty increases. It is also clear that the percentage of length traveled seems to have increased in the lowest difficulties when compared to Test 3. This increase is due to the modifications made to the interval of possible scores affecting the difficulty changes from rhythm to the rhythm being reduced to $\pm$ 2, and the odds of generating harder actions in lower difficulties being slightly decreased. Thus, the agent is able to complete more rhythms and go farther in the level. Difficulty 0 stands out due to it being impossible to lose in its first rhythm unless the time runs out. Therefore, the agent always has at least 10% of the level completed before reaching difficulty 1 or 2, depending on its performance in the prior level chunk.

When observing Figure 4.8, it is clear that the effective changes complicate the discernibility of the difficulties, evident due to the significant reduction in length in the highest difficulties and the position of difficulty 2 first quartile. Specifically, the positions of the upper whisker of difficulties 7 and 8 seem to have been lowered slightly, which again probably stems from the changes made to action generation. Therefore, the changes made for Test 4 did not improve the discernibility of the difficulties when taking into account the dispersion. Still, these results were considered acceptable when analyzing the other metrics.

It is possible that, if the number of samples was increased that better results would be achieved, however, this data is still in tune with what was expected.

In Figure 4.8 the dispersion of the results in the higher difficulties seems to have been reduced, but the changes also increased the dispersion in the lower difficulties. This decrease, as mentioned previously stems from the increased chance of the generator placing a geometry that is hard for the agent, making it more difficult to traverse the level in the highest difficulties, thus the agent loses sooner and travels less length.

In Figure 4.10 the plots show the comparison of the ending difficulties when the starting difficulty differs, while in Figure 4.11 the averages of the ending difficulty can be seen. From

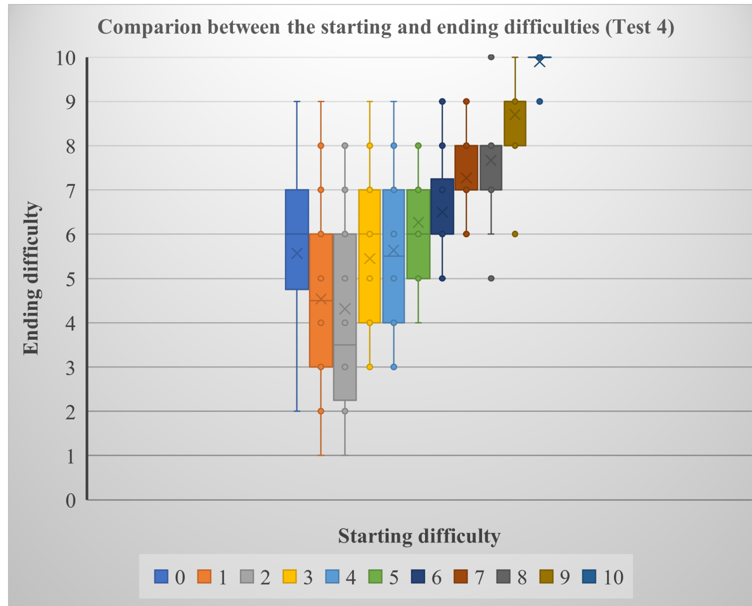


Figure 4.10: Box plots comparing the starting and ending difficulty in Test 4.

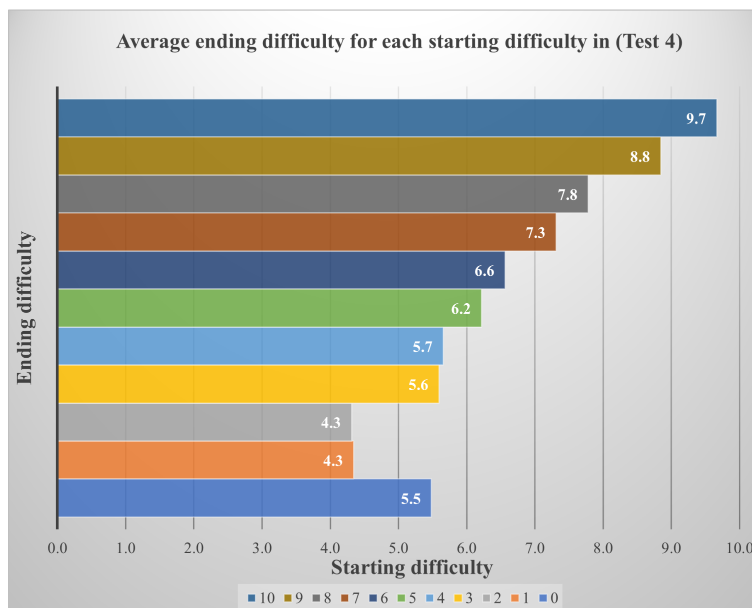


Figure 4.11: Average values of the ending difficulty for each starting difficulties in Test 4.

these results, it is evident that the DDA method was attributing more negative scores to the agent when it finished a rhythm since there are various occasions where the end difficulty is lower than the starting difficulty. For instance, this fact can be observed in the lower whiskers of difficulties 2 to 9, with the exception of difficulty 3, and the outlier in difficulty 10. Even so, the difficulty only decreases by 2 in difficulties 8 and 9, meaning the agent might need more attempts to fail for the game to adapt adequately. In difficulty 10, the value rarely changes, due to the agent losing before completing a rhythm. Another aspect to note is that, in difficulties 4 to 8, the agent fails more often leading to smaller differences between start and end value. Thus, the results are considered an improvement to Test 3 even with the slightly worse results in the length box plots.

Nonetheless, these values reveal that the changes to the DDA method were successful in increasing the difficulty of the game (traveled less length in Figure 4.8), in reducing the dispersion test the highest difficulties, and in slightly improving the adaptation of the generator when the agent does not lose but performs poorly. The last point is evident when comparing these results of Figures 4.7 and 4.10.

When observing the averages values of the ending difficulty in Figure 4.11, it is clear that, with the exception of starting difficulty 0, these keep increasing as the starting difficulty increases. However, in difficulties 0 to 7, the end value always exceeds starting value, which is a product of the number of times the game allows the agent to make mistakes before ending the game on a loss. In difficulties 8 to 10, the agent attains averages that are lower than their respective starting values, but still, the biggest difference between the initial value and the average in these difficulties is 10's -0.3, which stems from the impossibility of obtaining a greater difficulty value. Thus, when starting in difficulty 10, if there is at least one run where the agent completes a rhythm and achieves a negative performance that is $\leq$ -0.5, then the average will never be greater than the initial value. Therefore, to attain a more adequate adaptation, the agent required more attempts to commit mistakes.

These results show that the game got significantly harder and that the generator attempted to adapt to the difficulty when it was needed, yet these results could be improved. Test 4 did not let the agent continue if it lost, thus the performance value of the last rhythm was never accounted for the adaptation of difficulty. For this reason, another test was conducted to analyze scenarios where the agent is given more attempts while trying to improve the results by modifying the parameters of the DDA method and action generation.

Test 5

In Test 5 the agent would again play 30 runs of the game for each starting difficulty. However, when the agent lost the game, it continued playing in a new level with adapted values of length and difficulty based on the previous level's total length traveled and its performance. Additionally, upon losing, the game restarts and the agent returns to Mario mode 2 grating it more chances to make mistakes before ending the game. Therefore, a run allowed the agent to lose twice before finishing a run. With Test 5, the goal was to again study the adaptation of the generator, thus the length traveled and the end difficulty were observed. Ideally, on average every starting difficulty should have similar values of ending values, meaning that it would be possible to associate a bound value to our agent. The parameters of the DDA method and action probability were also altered to try to improve the results.

- **Weight of the parameters** - the weights of each parameter were changed in the same manner similarly to Test 4. Since the agent frequently came close to the expected time and it has difficulty dealing with enemies, the weights were changed:
  - **Time to complete the rhythm** - 10%.
  - **Time moving backward** - 10%.
  - **Coins picked up** - 10%.
  - **Enemies defeated** - 30%.
  - **Damage taken** - 40%.

  - **Length traveled** - 20%.

- **Required score to attain a change in difficulty** - the percentage to obtain a given score was also increased. Previously a score of 50% grants the player a change of 0, now to obtain the same score 75% is required. In addition, a minimum percentage to get a score above -2 was also added. This change was implemented to improve the converging of the ending difficulty by making it harder to attain an increase in difficulty upon completing a rhythm. The formula is given by equation 4.1.

$$P(x) = \begin{cases} \frac{(x-75)\times 2}{75}, & \text{if } x \geq 40 \\ -2, & \text{otherwise} \end{cases} \tag{4.1}$$

These changes would ideally slow down the adaptation even further and make it more complicated for the agent to reach the highest difficulties.

To formulate adequate conclusions, the data must be interpreted with the correct context, thus it is imperative to consider that, in this test, the agent possessed three attempts at clearing a level. For instance, if the agent begins in difficulty 10 and loses, then the difficulty can decrease to 8 and presumably the agent will travel more length in this second attempt due to this change. The same logic can be applied to the lower difficulties, for example, if the agent begins in difficulty 0, then after finishing a rhythm, the difficulty can increase to 2, making it possible for the agent to lose and travel less length.

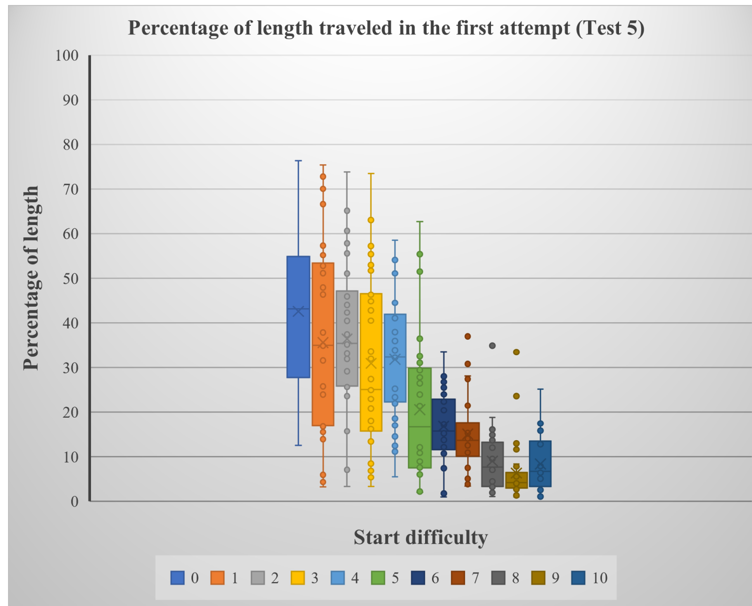


Figure 4.12: Box plots of the length traveled for each starting difficulty in Test 5.

Figures 4.12 and 4.13 represent the box plots and the average of length traveled in the first attempt of each run. As expected, due to the change in the required score, the results of average length were reduced when compared to the data of Test 4 (Figure 4.9), however since the agent has more attempts to finish a level, thus it as more chances to win. In Figure 4.12, the results appear more inconsistent than in Test 4, seeing as the first quartile, the median and the dispersion do not consistently reduce with the increase of the difficulty, even if these values

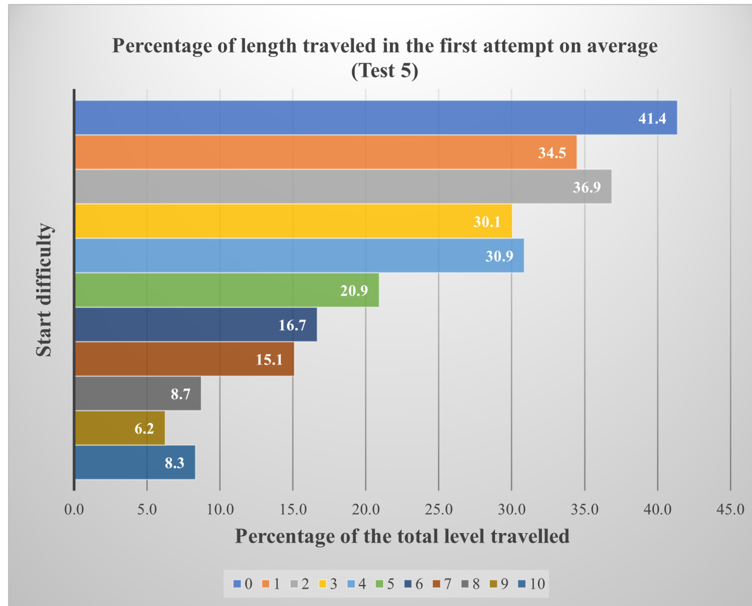


Figure 4.13: Average values of the length traveled for each starting difficulty in Test 5.

generally decrease. The pattern breaks in difficulties 2, 4, 5, and 9, presumably due to the agent traveling less length than in the difficulties below them more frequently, thus it obtain an inferior first quartile in these difficulties when compared to the ones above. It cannot be inferred with complete certainty since there is innate randomness to the results, but this incongruence must stem from the reduced sample size and the modifications implemented in the generator. However, when observing the upper whiskers and third quartiles of all plots, it is clear that their values, in the majority of cases, decrease as the difficulty increases which is considered a positive result. Therefore, it is again assumed that the inconsistency in the first quartiles is attributed to instances where the agent lost early in the game to a hard action. It is possible that, with more samples, the discernibility would improve, however for this test, we decided that these results were sufficient when considering the results later obtained for the ending difficulties.

In Figure 4.14, there is a noticeable difference in the value of the length traveled for the highest difficulties in [b] when compared to Figure 4.13. For example, by observing the difficulty 10's data in both Figures, it is clear that, in the first attempt, the agent reached an average value of 8.3%, however, in the second and third attempts, the values increase to 11.2% and 16.3% respectively. This is due to the game being adjusted when the agent loses, thus the difficulty was decreasing which allowed the agent to clear more length. In Figure 4.14 [b], it is apparent that the results are converging to a set value between [16.3; 24.8]%, which is a positive result considering that it could indicate that the difficulty of the level was also converging. In [a], the length traveled seems to generally decrease with the increase of difficulty, however, the values are considerably more consistent and close to each other than in Figure 4.13. In [b] the results appear to be converging but conclusions are difficult to make, seeing as, in difficulties 0 to 6, there were instances where the agent won and the game ended. Therefore, it would be possible to obtain greater values of length if the game continued (the agent is not able to win the game after a certain starting difficulty, in this case, 7). Another point to note is that adding the percentages of the graphs in Figure 4.14 [a] and [b], and in Figure 4.13, will never result in a value of 100% because the total length of the levels in attempts 2 and 3 is adapted based on the

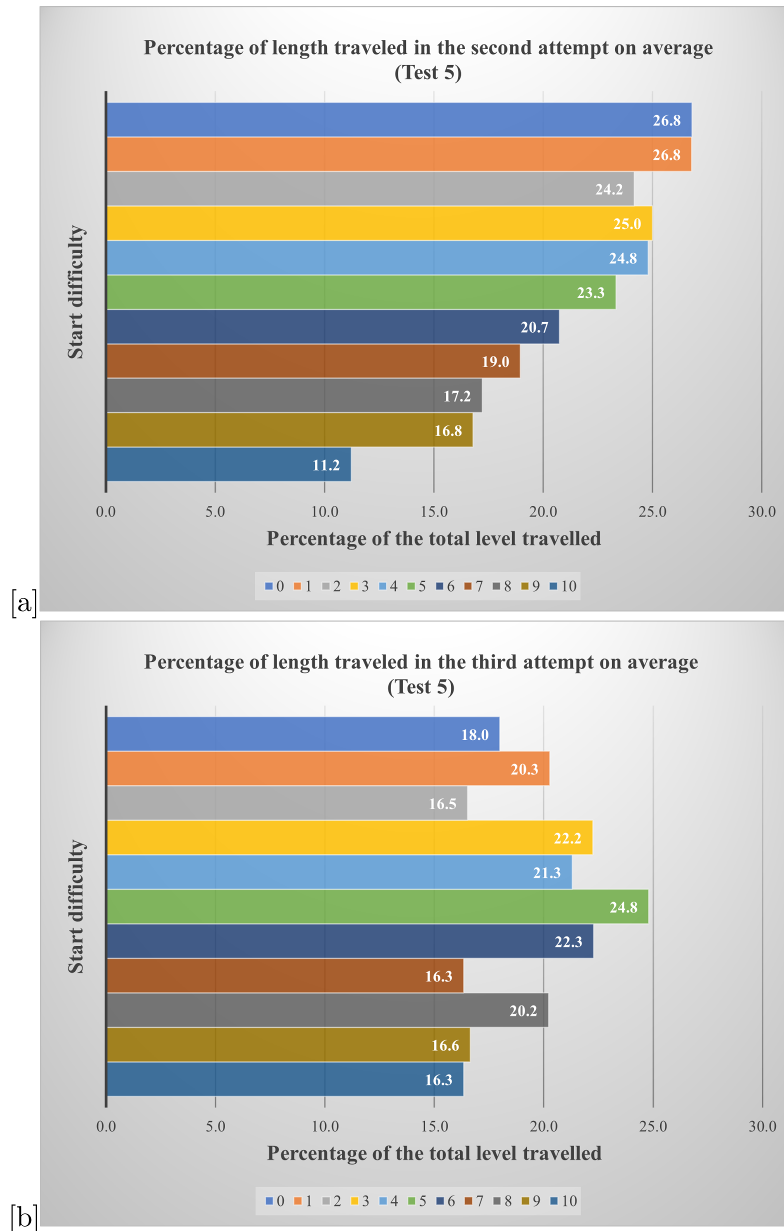


Figure 4.14: Box plots of the length traveled in the second and final attempt in relation to the percentage total level length in Test 5.

number of rhythms evaluated in the previous attempt.

Since the agent was able to win the game in some runs of Test 5, its success rate was studied (visible in Figure B.16 in appendix B). The results showed that the agent could complete the game 57% of the time starting at difficulty 0, and from difficulties 1 to 6 it achieved values between 40% and 13%. Therefore, there were multiple instances where the average length could have been increased if the game continued. Yet, it is unknown in what Mario mode the agent finished the game (how much health it had), thus it is uncertain if the agent had conditions to increase this value significantly.

In Figures 4.15 and 4.16, the relation between the start and end difficulty is studied using box plots and the average. In this test, it is evident that on average the ending difficulty is converging to values between [5; 6.6], with the average increasing with the starting difficulty. Test 5 allowed for a better adaptation of the difficulty seeing as the agent had more attempts

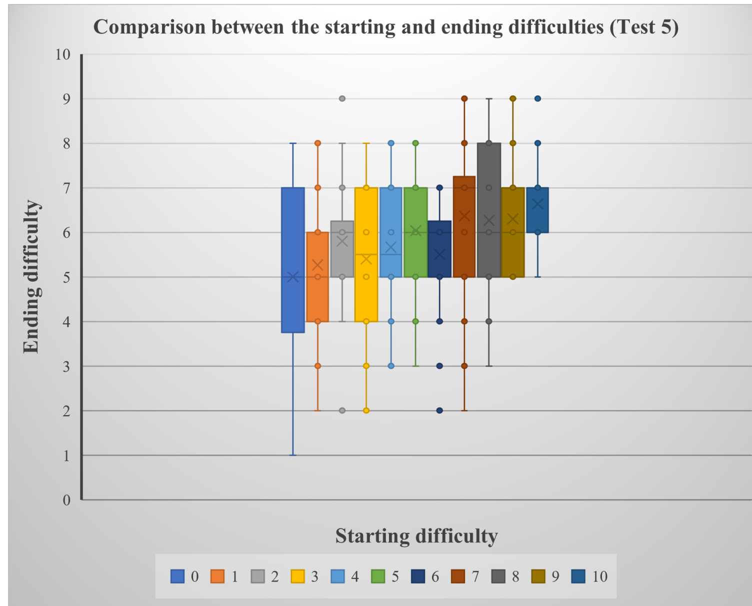


Figure 4.15: Box plots comparing the starting and ending difficulty in Test 5.

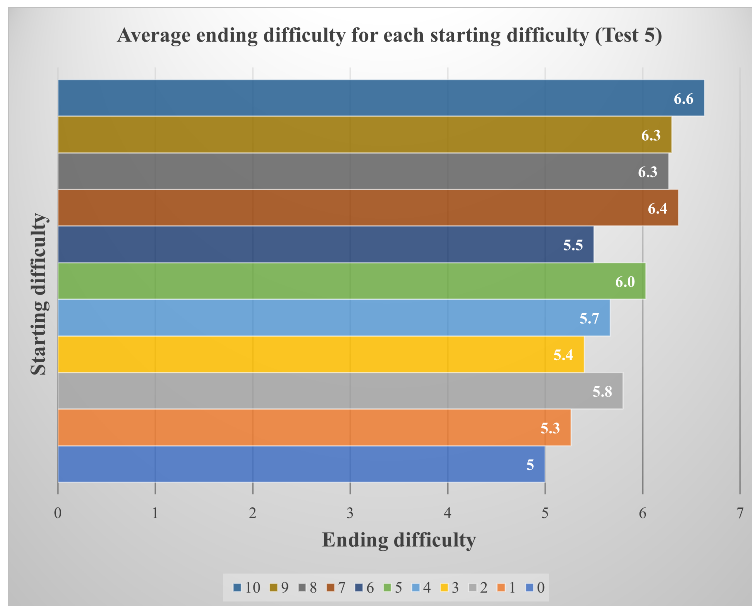


Figure 4.16: Average values of the ending difficulty for each starting difficulties in Test 5.

and some end values significantly differ from their starting point.

In Figure 4.15, the result are considered an improvement over the previous test (Figure 4.10). Even though there is no clear pattern, it is evident that in difficulties 7 to 10 the dispersion of values has increased in relation to the previous test. In this case, this dispersion is considered positive since it indicates that the generator is capable of adapting to the difficulty after multiple attempts. For example, the agent was able to play in difficulty 5 regardless of its starting point.

When it comes to the lower difficulties, the dispersion is still present, however, it can be inferred that the game was significantly harder (the executed changes to the generator were effective) since, in difficulties 0 to 5, the agent never surpassed an end value of 8 except for the outlier in difficulty 2. Therefore, the game got too hard for the agent and thus it cannot reach

a higher difficulty than 8.

These results could be improved (obtaining an even smaller bound value) with a larger sample size, but the conversion of the length and difficulty are still evident with this data. Nevertheless, as mentioned previously, the results are considered positive, since the last goal was for the agent to achieve values that could converge to a small bound value of difficulties. Therefore, with the current version of the generator, it is possible to associate our agent with difficulties 5, 6, and 7.

4.2 Testing With Real Players

When we were finally satisfied with the generator state, the next step was to test the generator with real player. For these tests, we wanted to see if players could notice that levels were being adapted as they were playing, and at the same time, obtain their opinions in regard to the adequacy of the content. Another goal was to understand if the players found this type of gameplay with adaptive difficulty interesting when applied to other platformer games and other genres.

To obtain this data a quiz was conducted. To reach adequate conclusions, the results of this quiz analyzed with the context of the data collected from the game. To perform these tests, 10 individuals were asked to play our game. In terms of age demographic, 8 of these players were in the range of 23 to 25 years of age. The other two players were 52 and 53 years old, one that actively played videogames and the other did not. In terms of gender, the younger group was composed of 4 females and 4 males and the older group consisted of 1 female and 1 male. This way, it is achievable to target two different demographics and also distinguish the people in these groups by the amount of time they played videogames. The players also used their computer's keyboard to play the game (this information is relevant for context).

The test proceeded as follows, each player would play a tutorial level and 3 games types:

- **Tutorial** - first the players played a tutorial level. The goal of this level is to give some freedom to the player so that they can become more comfortable with the controls and the game it self.

- **Game type 1** - after the tutorial, once the players felt ready, they played the game type 1. This game type was played twice and consist of a game run where the player starts in difficulty 0 and plays a level with adaptive difficulty.

- **Game type 2** - after game type 1, game type 2 is played once. It consist of a game run where the difficulty is constant (this value is depended on game 1 performance).

- **Game type 3** - succeeding game type 2, game type 3 is played twice. It consists of a similar game run to game type 1, however the first run had game type 2's starting difficulty, and in the second run the starting difficulty is influenced by the performance score obtained in the first play section of game type 3's.

In each run of each game type, the players had 3 attempts to finish the level, meaning they could lose two times before the run ended. The levels were composed of 10 rhythms for a total of 1000 length and in each rhythm the performance was evaluated. Once the players finish this process they responded to the quiz.

4.2.1 Quiz Results And Data Comparison

The quiz required the players to fill in the information about their age and gender, and contained the following questions:

1. **Personal consideration** - how much time this person plays videogames per day?
2. **Familiarity with platformers** - how many platformers has this person played, if any?
3. **Adaptation of game type 1 and 3** - did the player noticed the difficulty adaptation in games type 1 and 3?
4. **Adaptation of game type 2** - did the player notice that the difficulty was not being adjusted in game type 2?
5. **Quality of the adaptation** - how adequate was the adaptation of difficulty?
6. **Game type preference** - what game type did this person prefer?
7. **Revisiting this concept** - does the player want other games to use procedural content generation and adaptive difficulty?

For each question, with the exception of the last, multiple options were given to choose as an answer. The number of each question in the enumeration specifies what results of each question is being analyzed.

Figure 4.17, illustrates the results of question 1. In the graph, the number of answers relative to the type of answer can be observed. The type of answer from varies based on how much time the players spent playing videogames. The type of answer is described in the following enumeration:

1. **Never played videogames**.
2. **Plays casually** - does not play videogames every day.
3. **Plays less than an hour a day**.
4. **Plays more than an hour a day**.

The results show that the half of players play more than one hour a day, three of them play less than one hour a day and one player never played videogames. Since half of the players play more than one a day our results could be skewed, so this information is relevant when interpreting these answers.

Figure 4.18, shows the results of question 2. In the graph the number of answers relative to the type of answer can be observed. The type of answer varies based on how familiar each player is to platformers:

1. **Never played a platformer**.
2. **Played few platformers**.
3. **Played many of platformers**.

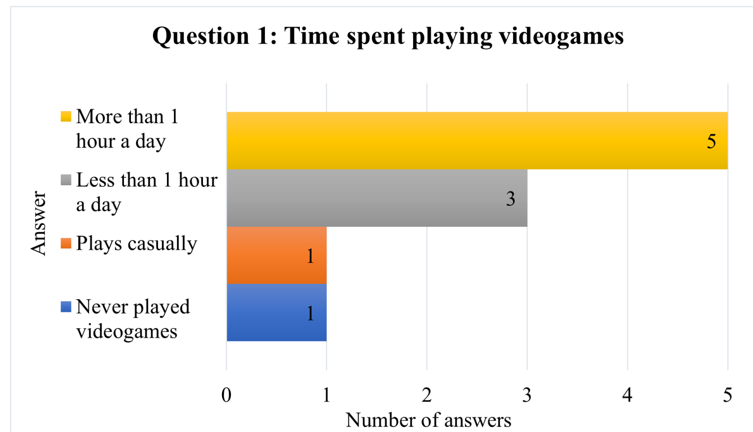


Figure 4.17: Type of answer in relation to the number of answers to question 1.

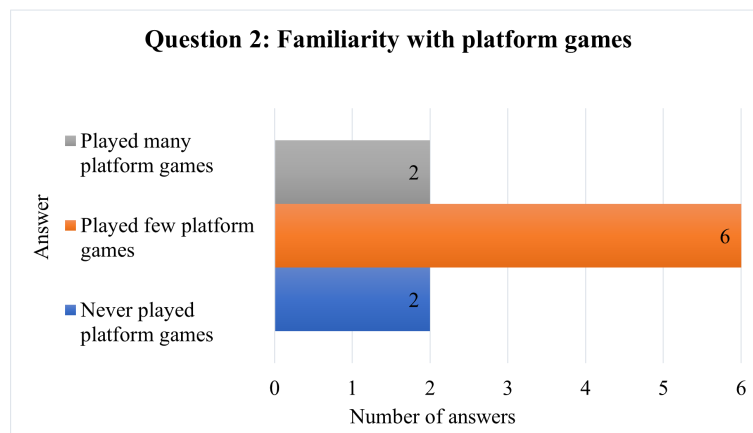


Figure 4.18: Type of answer in relation to the number of answers to question 2.

The results of Figure 4.18 reveal that most players have experienced platform games. This implies that the majority of players have the notion of how platformers function and their type of gameplay. This is a positive note due to the possibility of the players comparing the levels of our generator to the levels other platform games. The data of this graph can also be used as context for the answers given. The players that never played a platformer need to be kept in mind when analysing these answers.

Questions 3 to 7 allow for a deeper understanding of the adequacy of the generator's content when taking into account each individual player's opinion. Few of these questions gave insight into what type of game do the players prefer and if they believed this level generation and DDA was worth revisiting other games. For these reasons, the answers to the quiz questions are analyzed in conjunction with the test results of each individual (performance scores). The performance scores can give context to the answers given.

Figure 4.19, reveals the results of question 3. In the graph the number of answers relative to the type of answer can be observed. The type of answer varies based on how the players felt in regard to that the change in difficulty:

1. **No DDA noticed** - no difference in difficulty noticed while playing.

2. **Barely noticed DDA** - a small difference in difficulty noticed was while playing.

3. **Noticed DDA** - a considerate difference in difficulty was noticed while playing.
4. **Noticed DDA very well** - a big difference in difficulty was noticed while playing.

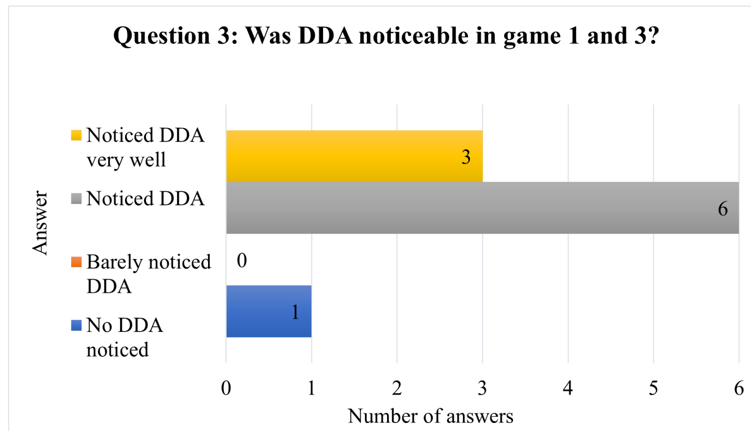


Figure 4.19: Type of answer in relation to the number of answers to question 3.

These results indicate that the majority of players noticed that the difficulty was changing while they were playing. Still, there is one answer of type 1 (No DDA noticed) implying that one player did not notice the difficulty changing. To study this case, it is necessary to analyze the performance scores of this player and compare them to other players.

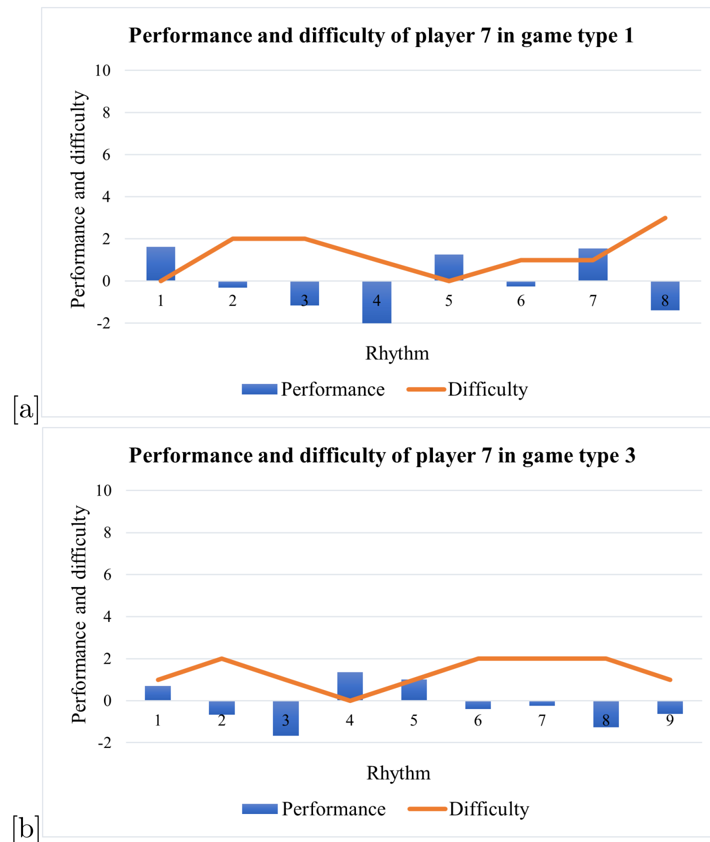


Figure 4.20: Player's 7 performance and difficulty in each rhythm.

In Figure 4.20 the performance scores and difficulty obtained by player 7 in each rhythm can be observed. This player claimed to not notice any change in difficulty, and in previous

questions responded with played less than 1 hour a day and played many platform games. The constructed graphs of Figure 4.20 correspond to the game runs where player 7 achieved its highest number of rhythms and its highest difficulty of 3. Considering that each game type is played twice the graph corresponds to the run where the player went the farthest in the game, thus there is more data to evaluate in these graphs. By observing this data, it is clear that this player remained most of its play time in between difficulties 0 and 2. As a result the player did not notice a significant change in the difficulty in the geometries since the generator was placing this player in an interval of difficulties where the levels had around the same difficulty. In [a] the player begins in difficulty 0 and jumps to 2 in the second rhythm. Afterwards the difficulty value fluctuates within the mentioned interval. The player is only able to reach difficulty 3 one time, and the game ended as seen in rhythm 8 of [a]. The same fluctuation is seen in [b] where the player remains inside the same interval of difficulties. Since the player was not able to complete the easier rhythms, mainly indicated by the drop in performance in rhythm 3 and 4 of [a] and [b], the generator never placed this player in a difficulty higher than 4 due to the player's performance not allowing it to increase. Therefore, this player would not notice the game getting harder, thus the type of answer given is justified.

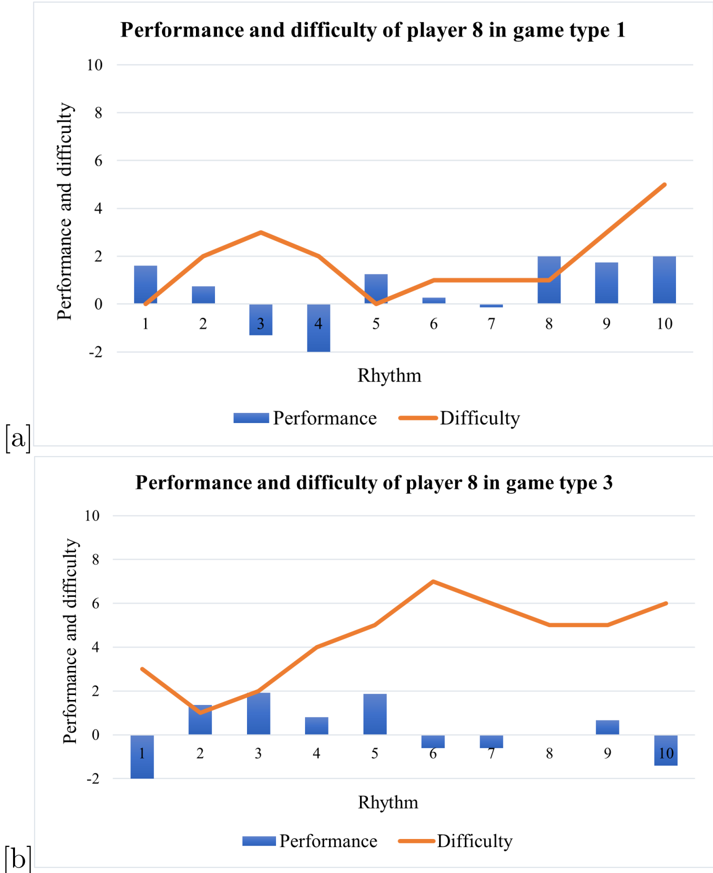


Figure 4.21: Player's 8 performance and difficulty in each rhythm.

In Figure 4.21 the data of the same game types of another player can be observed (player 8). Player 8 noticed that the difficulty was being modified, thus answered question 3 with Noticed DDA. In [a] and [b], player 8 was able to obtain an larger interval of difficulties those being [0; 5] and [1; 7] respectively. When comparing the range of difficulties obtained in Figures 4.20 and 4.21, it is seems more likely that player 7 was not able to discern if the difficulty changing while

the player 8 was. By observing these Figures it can be inferred that the generator is often able to associate a player with a given interval of difficulties. Player 7, this player was not able to reliably complete level in difficulty 2, thus the generator kept lowering the difficulty, while in player 8's run game type 3 the generator required around 6 rhythm before the difficulty was to stabilized. In Figure 4.21 [b] it is evident that after rhythm 5 this player consistently remains in difficulties 5 to 7, implying that if the player only played in these difficulties while having the same performance scores, then presumably it did not notice the difficulty changing.

The human error also needs to be accounted. Players can make mistakes even in lower difficulties, thus increasing the number of rhythms required to converge the difficulty.

Players can also improve while playing and achieve difficulties that are higher than their previous. In Figure 4.20 [a] player 7 was able to eventually reach difficulty 3 and in Figure 4.21 [a] player 8 was able to reach difficulty 5 after a few mistakes.

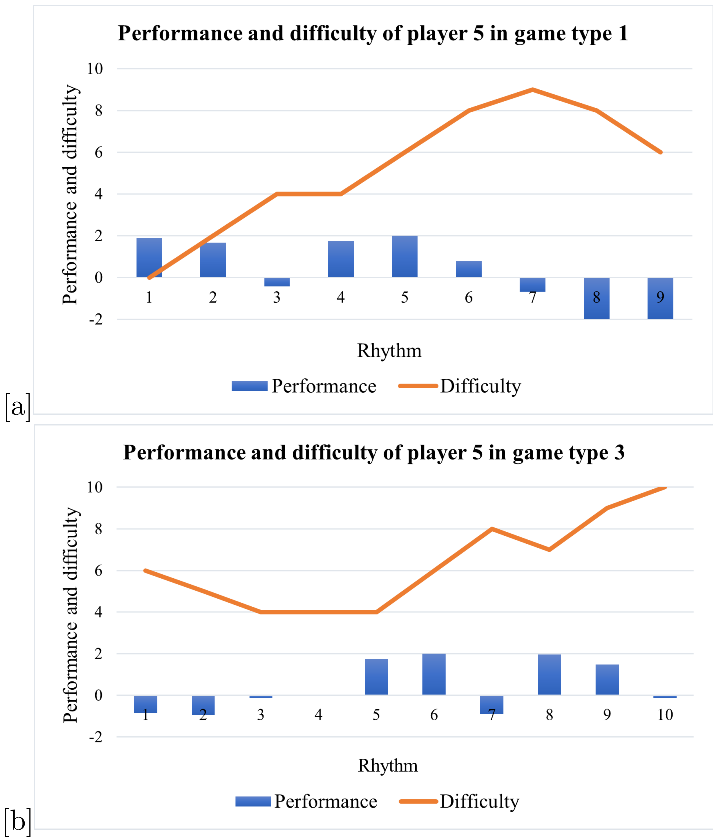


Figure 4.22: Player's 5 performance and difficulty in each rhythm.

Figure 4.22 represents the data of a player (player 5) that considered the change of difficulty to very apparent. In [a] the difficulty increases with each rhythm, until in rhythms 7 and 8 the difficulty drops the player obtaining worse performance scores. In [b] player 5 began in difficulty 6 and was able to reach difficulty 10, implying that the change in difficulty was noticeable.

To sum up, question 3 seems to dictate positive results implying that most players were able to discern that the difficulty was changing, and those who were not were already set to a certain difficulty interval.

In Figure 4.23, the results of question 4 can be observed. In this graph, the number of answers relative to the type of answer can be visualized. The type of answer are identical to question 3 (Figure 4.19).

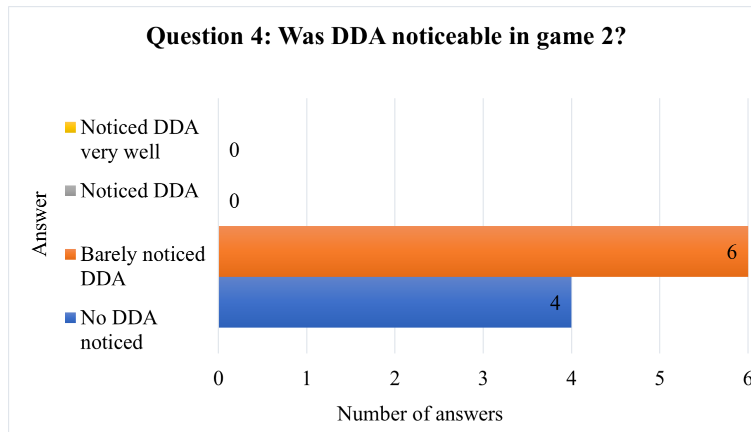


Figure 4.23: Type of answer in relation to the number of answers to question 4.

The graph in Figure 4.23 reveals that the slight majority of players felt a small change in difficulty which could be attributed the randomness of action generation. For example, if a level contains one or two generated rhythms that are considered easier than the rest due to the randomness of the actions, then it is logical that players feel that the difficulty could change slightly during the gameplay. Another possible reason for these answers could be the difficulty at which each player was placed for this test. A graph was constructed to illustrate the plausibility of this last point.

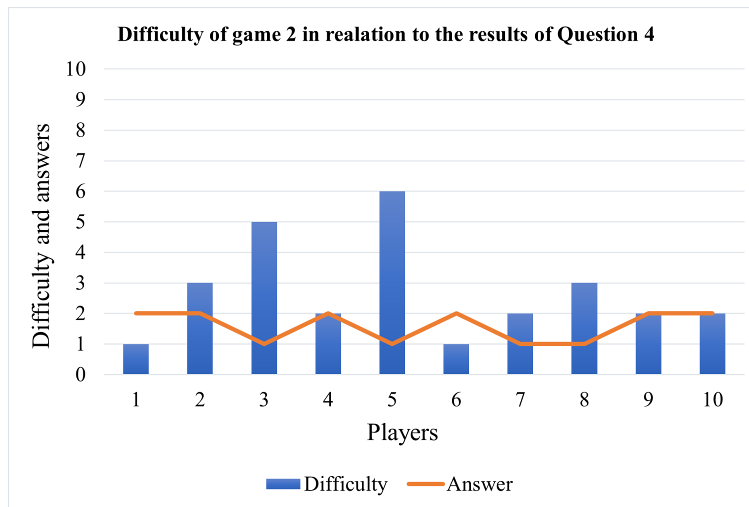


Figure 4.24: A combination graph shows the answers to question 4 in relation to the difficulty that the players were placed in game type 2.

In Figure, 4.24 the players that were able to achieve higher difficulties answered that the difficulty did not change (players 3, 5 and 8, player 7 being the exception). While players attributed with lower difficulties noticed a small change in difficulty (player 2 being the exception). This presumably due to the action generation in easier difficulties, where it is less likely that an harder action is generated often, and if this phenomenon happens at least once, the players might feel a change in difficulty. Nevertheless, these results are considered positive since some players were able to notice that the difficulty did not change and those who felt like the difficulty was changing only indicated to be by a small amount.

In Figure 4.25, the results of question 5 can be observed. In the graph the number of answers

relative to the type of answer can be observed. Type of answer varies based on how adequate the change in difficulty was:

1. **Not adequate** - an unsuitable change in difficulty for the type of game.
2. **Less adequate** - a not so suitable change in difficulty for the type of game.
3. **Adequate** - a suitable change in difficulty for the type of game.
4. **Very adequate** - a very suitable change in difficulty for the type of game.

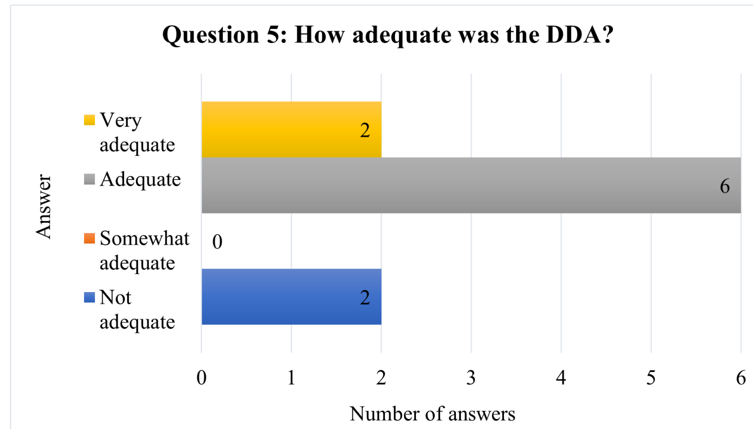


Figure 4.25: Type of answer in relation to the number of answers to question 5.

These results show that most players were satisfied with the change in difficulty. The players that responded that the adaptation was not adequate were the player that never played videogames (player 6) and the player 7 (associated with Figure 4.20). Upon analysing the results of player 6 and the answers given to the previous questions it is expected that this player found the lower difficulties harder due to the lack of experience (player 6 answered question 1 with never played videogames). Player 7 also found the game too difficult. To further comprehend these answers, the results obtained by these player were verified.

In Figure 4.26 the results of performance and difficulty of player 6 can be observed. In these graphs, it is possible to visualize that in both instances where this player reaches difficulty 2, the performance value decreases significantly due to the player losing afterwards. In [a] player 6 was only able to complete a rhythm of difficulty 1 once (rhythm 4), while in [b] the player manages to succeed in this difficulty more often. This is presumably due to this player having more experience with the game when playing game type 3. Even so, in [b] and in the rhythms that have a difficulty value of 1, this player has a performance score close to 0, implying that the player is committing mistakes when the difficulty is stabilized. Meaning that, for this player, the game is considered too hard even in the lowest difficulties, thus it is likely that the player felt that the game does not adequately change the difficulty for its skill level. However, there was another player that never played platformer games and felt like the adaptation was adequate, yet this player plays videogames casually. To make the game easier more sets of geometries that are considered easier could be constructed and generated in the lower difficulties to ensure that the generator creates adequate content for players that never played videogames.

Since the players were monitored while playing, it was simple to identify the instances where player 6 found the game to be hard. Player 6 struggled more with geometries that were associated

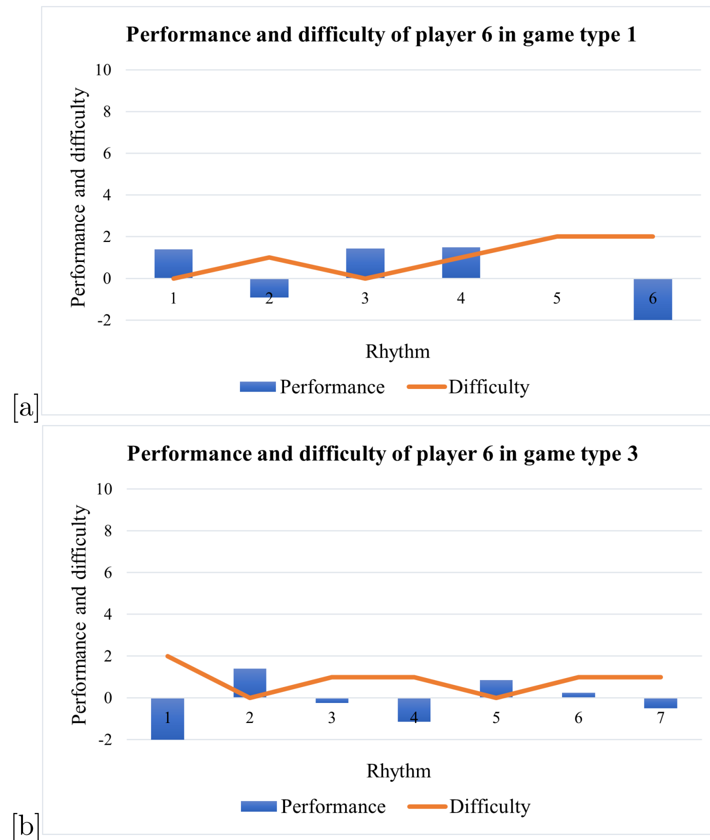


Figure 4.26: Player's 6 performance and difficulty in each rhythm.

with the run jump action. This action, in the lower difficulties, requires the most coordination of inputs (requiring three inputs) to perform the action correctly (further explained in subsection 2.2.1). Thus, since this player possessed less coordination than the others (presumably due to the lack of experience), the run jump action felt arduous and not adequate for that difficulty. When comparing player 6 to the player that plays causally but never played platformers (which answered question 5 with very adequate), it is reasonable to assume that the difference in game genre did not affect this player's perception of the adequacy of the difficulty. Thus, it is likely that player 6 found the game to be more difficult due to the lack of coordination.

Player 7 expressed the same concerns about the difficulty and the present geometry, having a harder time in the run jump and the duck actions that required the player to run. Based on the criteria in subsection 2.2.1 both of these actions have similar levels of difficulty, since they both require the player to perform multiple inputs simultaneously while having good timing to jump or duck. However, these types of actions are considered harder when compared to the other actions that can be generated in the lower difficulties. Player 7 claimed to have no experience using a keyboard to play videogames, making it arduous to complete more complex geometries. This statement complements the point of the players finding these actions more difficulty due to their lack of experience and coordination. However, it also reveals that we were able to make a good distinction when it comes to the difficulty of geometries in section 2.2.1.

These types of actions are able to be generated in these difficulties to increased the number of geometries that can appear in the game in the lower difficulties. Nevertheless, this implies that these actions should only be generated in slightly higher difficulties with the goal of making the lower difficulties easier for less experienced players. This way, players with experience are

not negatively influenced by these changes (the game becomes too easy).

Nonetheless, the results of question 5 help identify the flaws of our generator. These flaws were not noticeable when using the AI agent for validation due to the agent only making mistakes in the specific geometries, making it complicated to evaluate the difficulty of some geometries. Even so, excluding players with less experienced, players found the adaptation of difficulty to be adequate.

In Figure 4.27 there is an illustration of the results of question 6. The graph shows the number of answers relative to the type of answer. The possible answers varied between the game types 1, 2 and 3 and represent the favoured method of playing:

1. **Game type 1** - a game with adaptive difficulty starting at the lowest possible difficulty.
2. **Game type 2** - a game with no adaptive difficulty starting at a difficulty determined by game 1.
3. **Game type 3** - a game with adaptive difficulty starting at a difficulty determined by game 1.

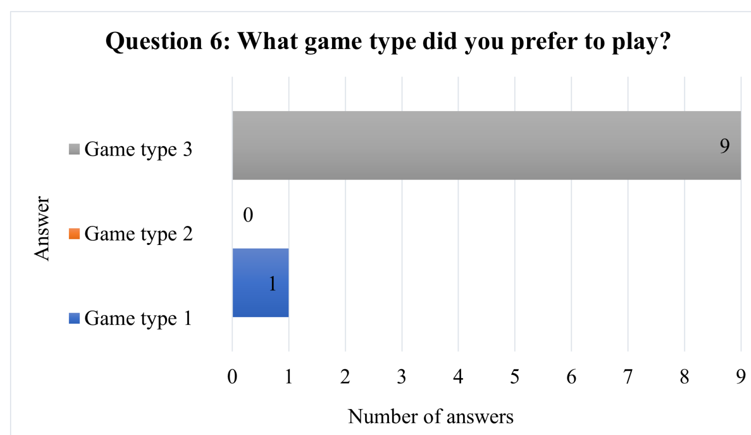


Figure 4.27: Type of answer in relation to the number of answers to question 6.

Figure 4.27, shows that the majority of players prefer game 3, thus by using the data of game type 1 to determine the starting difficulty player experience was improved. By beginning in a difficulty more suited for the player, the game can adapt faster requiring less rhythms. However, this method requires previous data to associate the player with a level of difficulty. Reaching this interval sooner implies that the players faces their preferred challenges faster (the difficulty is not too easy or too hard thus improving its experience).

These results of question 6 (Figure 4.27) confirm that the generator is not ideal at converging players that never played videogames into an interval of difficulties, seeing that player 6 preferred game type 1. It is logical that this player would prefer game type 1 since it is the most adequate for this particular player due to it beginning in difficulty 0. If easier geometries were implemented and generated more often in the lower difficulties then this player could presumably prefer game type 3.

In Figure 4.28, the results of question 7 can be observed. Players were asked the concepts of PCG and DDA implemented were worth revisiting in other games, and the majority responded with yes. The results are in tune with the question 6, however it means that players do not just

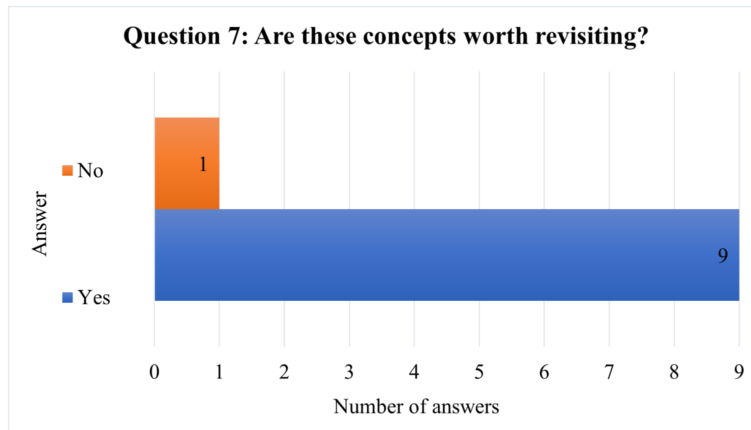


Figure 4.28: Type of answer in relation to the number of answers to question 7.

prefer these concepts together but actually enjoyed them enough to want to see them in other products.

With a larger sample size different results for each question could be obtain. It is plausible that if there were more players that never played videogames played in our data pool, then more people could probably answer question 6 (Figure 4.27) by preferring game type 1. However, for players that already play videogames, the generator is capable of creating levels that suit the player's skill which overall means that the results are considered positive.

4.3 Summary Of The Results Obtained

In this chapter, the level generator was tested with an AI agent and with real players. The goals of the agent's tests were to converge the agent's ending difficulty to a bound value, to distinguish the difficulties, to ensure stochastic generation, and to guarantee an adequate adaptation. Multiple metrics were analyzed but only a few showed some clear patterns that helped us reach these goals, namely the performance score and the length traveled. Initially, a test with reused rhythms was conducted which revealed that the results have a bias toward the agent's behavior and the rhythms. Even so, the same rhythm was able to generate different levels which were evident in the dispersion of the results and in the outliers obtained. In Test 3, the difficulties were able to be discerned with the length traveled since this value on average decreased with the increase of the difficulty. However, to accomplish the other goals, some changes to the parameters of the DDA method were implemented (based on the agent's behavior) to improve the adaptation of the difficulty. In test 5, the ending difficulty of the agent converged on average to a bound value between [5; 6.6] and, when the agent was given more attempts to make mistakes, the length traveled also converged. Thus, in this state, we considered that the generator was constructing adequate content and was ready for the test with real players.

To complete the validation of the generator, players were asked to play three types of games. These three games were distinguished by the quality or absence of DDA. Therefore, game types 1, 2, and 3 respectively consisted of a game with adaptive difficulty that starts at difficulty 0, a game with no adaptive difficulty but it was set in a difficulty that was considered adequate for the player based on the first game, and a game with adaptive difficulty that began in the same difficulty as game 2. After answering the quiz, most players preferred game type 3 and

considered the adaptation to be noticeable and adequate. However, the players that disagreed with the majority consistently achieved the lowest values of difficulty and stated that certain geometries were too difficult. This was expected since the type of geometries mentioned are more difficult than the others (in these difficulties) when considering the parameters stated to affect the difficulty in subsection 2.2.1. Therefore, our discernibility was a success, seeing as the players with less experience found these geometries to be harder (these geometries were included to increase the diversity of the inputs), while the other players found the adaptation to be adequate. Finally, when asked if this type of PCG and DDA were worth revisiting, the majority of the players responded with yes.

Chapter 5

Conclusion

In this thesis, the main goals set for our project were achieved. We created a rhythm-based level generator for a 2-D platformer that was able to adapt the difficulty of the game based on the player's performance as it is being played. The generator was then tested with a validation tool (an AI agent) and, after multiple adjustments, it was able to converge the ending difficulty of the agent to what was considered an adequate bound value. The final phase of the project consisted of another test but with real players, where the objective was to obtain their opinion regarding the quality of the generated content. The results of these tests revealed that most players found that the generated content was adequately adapting to their skill level and that the concepts of PCG and DDA were interesting to revisit in other platform games and other genres.

The project was facilitated due to the methodology used to generate actions. These actions were redefined to be an input or a combination of inputs that are associated with a group of geometries and that the player must perform to progress in the game. This definition simplified the process of action generation since there were fewer restrictions that needed to be considered, and it became easier to correlate these inputs with a set of specific geometries. The only issue with this approach is that it requires a decent number of geometries to not make the game stale, similar to the combining pre-made parts method (subsection 1.4.3).

The method used to discern the difficulty appears to be one of the best qualities of the project. The difficulty of an action is discerned by some parameters which include the number of inputs, duration of inputs, the timing of the inputs, number of simultaneous inputs, and the variation of inputs with time (subsection 2.2.1). By defining and understanding each parameter it was possible to discern which actions/geometries were easier or harder, which was proven successful in the test with real players. Very inexperienced players did not seem to find the easier difficulties adequate and mentioned that specific geometries were too difficult. These statements were in tune with the parameters used to discern difficulty seeing as the geometries that these players considered hard were placed in the lowest difficulties to increase input variety, even though these were considered slightly harder than others by our parameters.

The validation method chosen for this project seemed to have some issues. By using an AI agent, a bias is added to the generated content. This is due to the agent being used to extract benchmark values for the DDA method and used to ensure the generator is able to discern the difficulty of the game. However, because the agent serves both these functions, the DDA method and action generation odds were directly influenced by the agent's behavior. Thus, the generator changes the difficulty based on what the agent finds easier or harder. Therefore, by determining

how the agent behaves, where it succeeds and fails, we consequently dictate what is considered difficult in the game. Thus, when modifying the DDA method and action generation odds, the agent's behavior was always put into consideration. The necessity of obtaining benchmark values for the DDA method meant the agent needed to be able to complete most geometries, therefore, when it came to the tests, the agent always achieved times that were very close to the benchmark times. In consequence, to facilitate the discernibility of the difficulties, the time metrics required a reduction in weight. To ensure the difficulties were able to be distinguished during the tests, when constructing the agent's behavior it was settled that the agent should fail in the geometries that were considered the hardest. In our case, these were actions that generated multiple enemies, which contained all the parameters stated to make an action difficult (subsection 2.2.1).

When it comes to player experience, this issue only impacts the diversity of playstyles because, as mentioned previously, the weights of metrics such as the time taken to finish a rhythm and time moving backward had to be reduced due to the agent's behavior. Therefore, if a player is fast at completing a rhythm but takes damage it will be less rewarded than a player who was slow and completed the game without being damaged. Nonetheless, most players enjoyed the generated levels and found that the adaptation was adequate (section 4.2).

Even though this type of procedural content generation and DDA method was implemented in a platformer, this methodology can be easily replicated and applied to other game genres. The definition set for actions in subsection 2.2.1, allows for various types of geometries to be generated even for the same types of inputs. The number of possible geometries is directly correlated to the complexity of the game's mechanics (in general more inputs more combinations). To generate content, it is necessary to set the objective of the game, define combinations of inputs that contribute to the player's progress toward that goal, set properties for each action to ensure the generated geometry is possible to complete, and discern the difficulty of each action. For example, considering an FPS, if the goal is to defeat every enemy in an area without being damaged, some possible inputs could be inputs to crouch, move, aim, and fire. When combining these inputs, it is feasible to associate them with geometries where the player needs to move from cover to cover, in order to defeat enemies, while not being punished for receiving damage. To replicate the DDA method of section 2.3, it is mandatory to determine and set metrics that can be used to recognize how well the player is performing in the game. These parameters reflect the performance of the player when completing an action and should determine how well the player is reaching the main goal. For example, in our generator, the goal is to reach the end of the level, however, the player is also evaluated for picking up coins and defeating enemies. These geometries force the player to perform specific inputs, and if these are not executed correctly, the player is punished even if it reached the goal. These generalizations imply that this methodology can be applied to other game genres and produce similar outcomes. However, the definition of difficulty explained in subsection 2.2.1 cannot be applied equally to any game genre. For example, in turn-based RPGs, the difficulty can for example stem from the knowledge necessary to understand the game's mechanics in order to achieve the intended goal. Thus, our definition of might not be ideal since the parameters stated in subsection 2.2.1 may not be coherent with the goal of the game.

To sum up, the qualities of this thesis are the following:

- **Action redefinition** - the redefinition of actions allowed for an simpler method of gen-

eration geometry.

- **Difficulty** - the definition of difficulty and the properties used to discern it proved effective when considering the results of the test with real players.
- **DDA** - the level generator was able to converge and associate a bound value of difficulty to our agent and our players, even when considering the bias introduced by the agent's behavior.
- **Understanding player opinions and online generation** - the test with real players revealed that the majority of players would like to play more games that use PCG and DDA. Based on player feedback, modifying the difficulty and level geometry as the game was being played improved player experience.
- **Validation tools** - the results of this project reveal the main issues of using AI agent as validation tool and the bias that it brings to the studies.
- **Replicating the PCG and DDA methods** - the methodology used can be replicated for other game genres and produce similar outcomes.

However, some aspects of this project could be improved in the future:

- **Improve the method of validation** - the project showed that if an AI agent is used to determine benchmark values for the DDA method and used to discern the difficulty of the game, a bias is introduced. To solve this issue, a different tool should be used to achieve both goals.
- **Improve the DDA method** - if the DDA method is used in other games then it is imperative to explore and analyze more ways of determining player performance.
- **Action difficulty quantification** - even though we were able to identify some parameters that affect the difficulty of a geometry, more research could be conducted to create an equation that quantifies the difficulty of performing the required inputs of an action. This would further improve the quality of the DDA method and the adequacy of the content to any demographic of players.
- **Less dispersion in the results** - to facilitate the interpretation of the results it is necessary to find a way to minimize the dispersion.
- **Using a different game** - these methods of PCG and DDA could be implemented in a game that possesses more complex mechanics, allowing for more combination of actions to be made and more geometry to be created. With a more complex game, the issue a difficulty not possessing diverse geometries should not be present.
- **Different types of levels** - the levels created by the generator all have the same objective (to reach the end goal at the most right part of the level). Since player experience was improved with the generation of different content, then by changing the position of the end goal different levels can be generated adding more content.

Bibliography

[1] S. Gillian, "Launchpad: A rhythm-based level generator for 2-d platformers," 2011.

[2] M. Sharif, A. Zafar, and U. Muhammad, "Design patterns and general video game level generation international journal of advanced computer science and applications," vol. 8, p. 393–398, 2017.

[3] N. Shaker, J. Togelius, G. N. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi, G. Smith, and R. Baumgarten, "The 2010 mario ai championship: Level generation track. ieee transactions on computational intelligence and ai in games," vol. 3, no. 4, p. 332–347, 2011.

[4] T. Michael, W. Glenn, A. Ken, and L. Jon, "Rogue: Exploring the dungeons of doom." Epyx, 1980.

[5] M. Persson, "Minecraft." Mojang, 2011.

[6] S. Meier, "Civilization." MicroProse, 1991.

[7] G. Software, "Borderlands." 2K Games, 2009.

[8] M. P. Silva, V. D. N. Silva, and L. Chaimowicz, "Dynamic difficulty adjustment through an adaptive ai. brazilian symposium on games and digital entertainment, sbgames," p. 172–186, 2012.

[9] C.Pedersen and G. N. Yannakakis, "Modeling player experience for content creation. ieee transactions on computational intelligence and ai in games," 2010.

[10] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey. ieee transactions on computational intelligence and ai in games," vol. 3, no. 3, p. 172–186, 2011.

[11] N. Shaker and G. N. Yannakakis, "Towards automatic personalized content generation for platform games. proceedings of the aaai conference on artificial intelligence and interactive digital entertainment," 2010.

[12] G. Yannakakis and J. Togelius, "Experience-driven procedural content generation (extended abstract). 2015 international conference on affective computing and intelligent interaction, acii 2015," p. 519–525, 2015.

[13] J. Roberts and K. Chen, "Learning-based procedural content generation. ieee transactions on computational intelligence and ai in games," vol. 7, no. 1, p. 88–101, 2015.

[14] J. Fisher, "How to make insane, procedural platformer levels," 2012.

[15] T. Adams and T. Short., *Procedural generation in game design.* 2017.

[16] G. K. Sepulveda, F. Besoain, and N. A. Barrig, "Exploring dynamic difficulty adjustment in videogames," 2020.

[17] M. Zohaib, "Dynamic difficulty adjustment (dda) in computer games: A review. advances in human-computer interaction," 2018.

[18] S. Xue, M. Wu, J. Kolen, N. Aghdaie, and K. A. Zaman, "Dynamic difficulty adjustment for maximized engagement in digital games," p. 465–471, 2017.

[19] C. V. Segundo, K. Emerson, A. Calixto, and R. P. Gusmao, "Dynamic difficulty adjustment through parameter manipulation for space shooter game," 2016.

[20] P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma, "Online adaptation of game opponent ai with dynamic scripting," vol. 3, no. 1, p. 45–53, 2004.

[21] R. Hunicke and V. Chapman, "Ai for dynamic difficulty adjustment in games," p. 91–96, 2004.

[22] V. Corporation, "Half-life." Sierra Entertainment, 1988.

[23] L. Rantala, "Video game difficulty and the intended player experience," 2021.

[24] J. Knorr, "Dynamic difficulty adjustment in first-person shooters," 2021.

[25] P. Andrade, "Procedural generation of 2d games," 2020.

[26] S. Miyamoto, "Super mario." Nintendo, 1985.

[27] J. Togelius, N. Shaker, S. Karakovskiy, and G. N. Yannakakis, "The mario ai championship 2009–2012," 2013.

[28] J. Togelius and S. Karakovskiy, "Mario ai benchmark," 2009-2010.

[29] M. Ian, *AI for games.* 2020.

Appendix A

Algorithms

Algorithm 1 GenerateGeometry

```
ListOfActions ← GenerateActions(Rhythm, Lengthtraveled)
SumOfLength ← Lengthtraveled
for Action in ListOfActions do
    switch ActionType do
        case Begin
            assert(GenerateInitialGeometry(Action, SumOfLength))
            break
        case Move
            assert(GenerateMoveGeometry(Action, SumOfLength))
            break
        case Jump
            assert(GenerateJumpGeometry(Action, SumOfLength))
            break
        case RunJump
            assert(GenerateRunJumpGeometry(Action, SumOfLength))
            break
        case Up
            assert(GenerateUpGeometry(Action, SumOfLength))
            break
        case Duck
            assert(GenerateDuckGeometry(Action, SumOfLength))
            break
        case MoveLeft
            assert(GenerateMoveLeftGeometry(Action, SumOfLength))
            break
        case WallJump
            assert(GenerateWallJumpGeometry(Action, SumOfLength))
            break
        case Wait
            assert(GenerateWaitGeometry(Action, SumOfLength))
            break
        case End
            assert(GenerateEndGeometry(Action, SumOfLength))
            break
    SumOfLength ← SumOfLength + ActionLength
    PreviousAction ← Action
end for
```

Appendix B

Relevant Figures

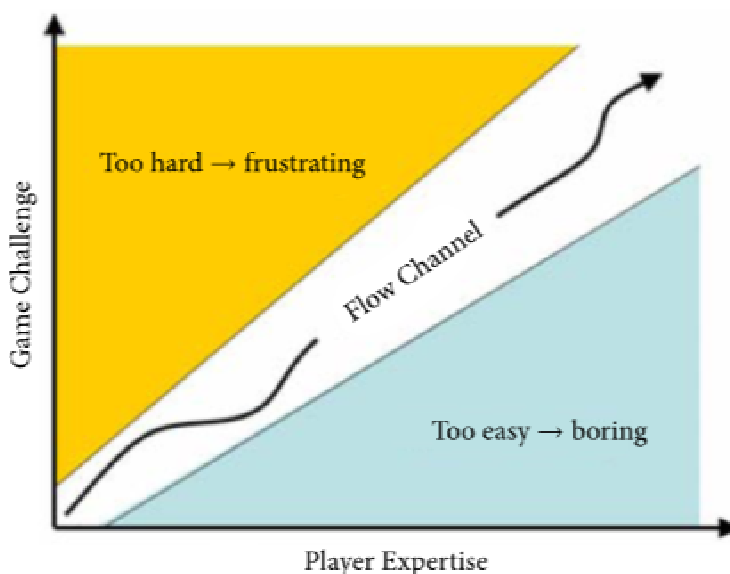


Figure B.1: A representation of the flow channel of a player [17].

Phase	Platform	Default	DDA	Delta	Gain
I	iOS	1,118,237	1,167,616	+49,379	+4.4%
	Android	789,640	825,182	+35,543	+4.5%
II	iOS	855,267	915,334	+60,067	+7.0%
	Android	1,137,479	1,228,473	+90,995	+7.9%
III	iOS	711,749	763,508	+51,759	+7.2%
	Android	1,285,448	1,366,820	+81,373	+6.3%

Figure B.2: Comparison of the number of rounds played with and without DDA [18].

Phase	Platform	Default	DDA	Delta	Gain
I	iOS	3,684,082	3,847,516	+163,435	+4.4%
	Android	2,686,781	2,814,953	+128,172	+4.8%
II	iOS	2,916,570	3,148,722	+232,152	+7.9%
	Android	3,787,414	4,129,762	+342,348	+9.0%
III	iOS	2,582,809	2,788,690	+205,881	+8.0%
	Android	4,619,907	4,956,680	+336,773	+7.3%

Figure B.3: Comparison of the time played with and without DDA [18].

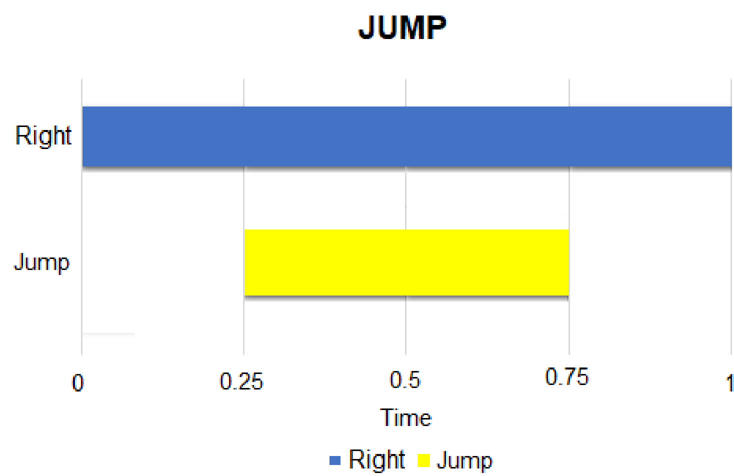


Figure B.4: A Gantt chart for a Jump action.

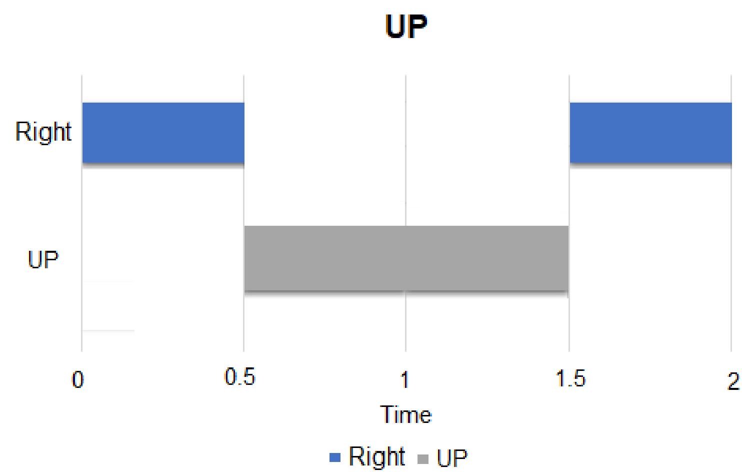


Figure B.5: A Gantt chart for a Up action.

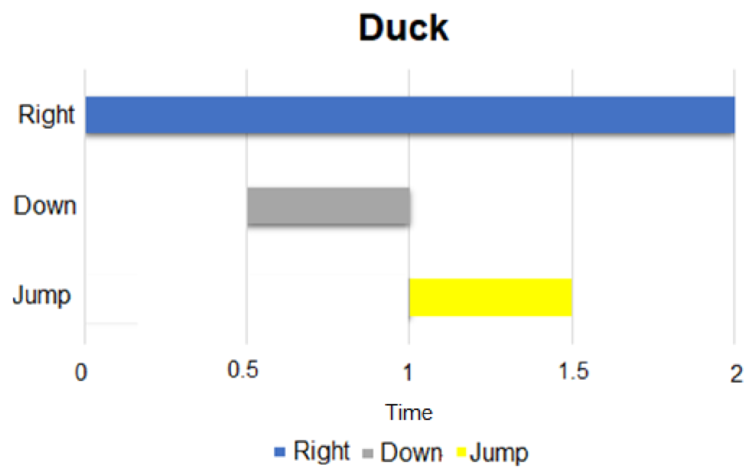


Figure B.6: A Gantt chart for a Duck action.

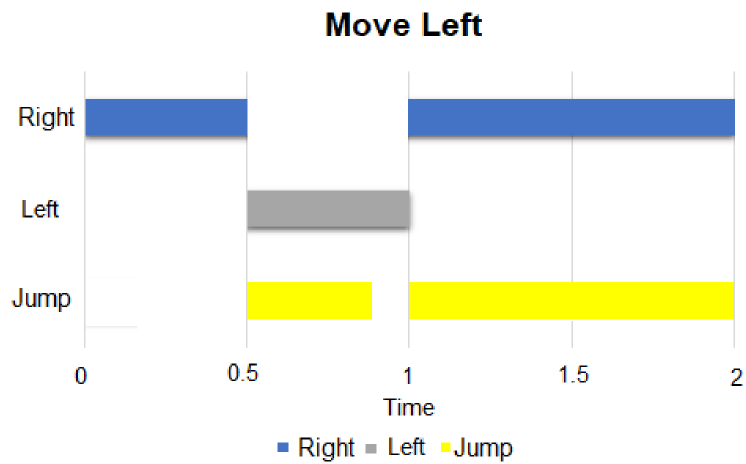


Figure B.7: A Gantt chart for a Move Left action.

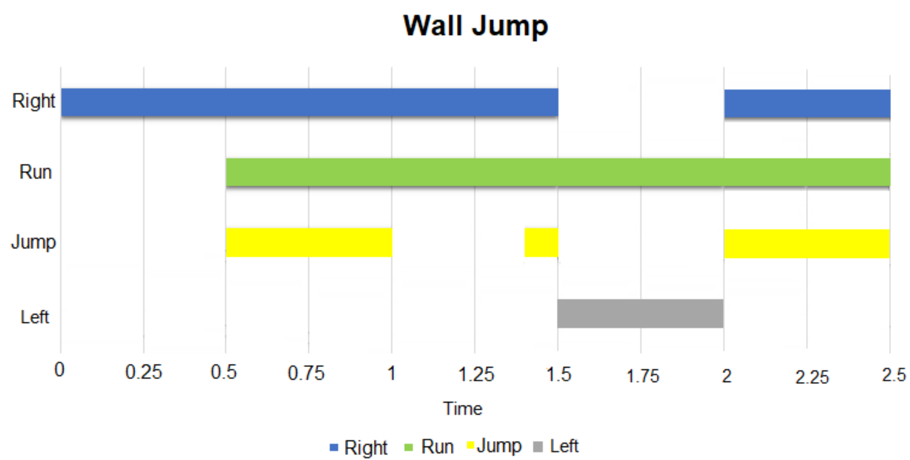


Figure B.8: A Gantt chart for a Wall Jump action.

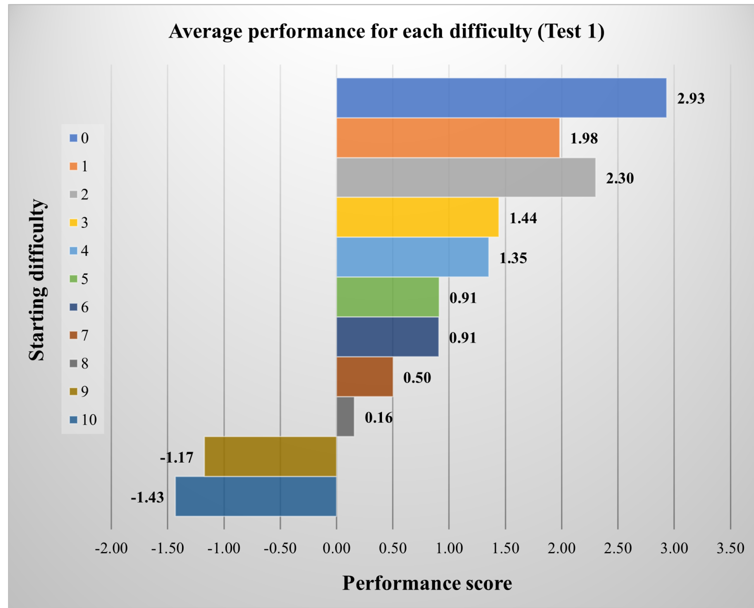


Figure B.9: Average of Test 1's performance scores in each difficulty with the outliers.

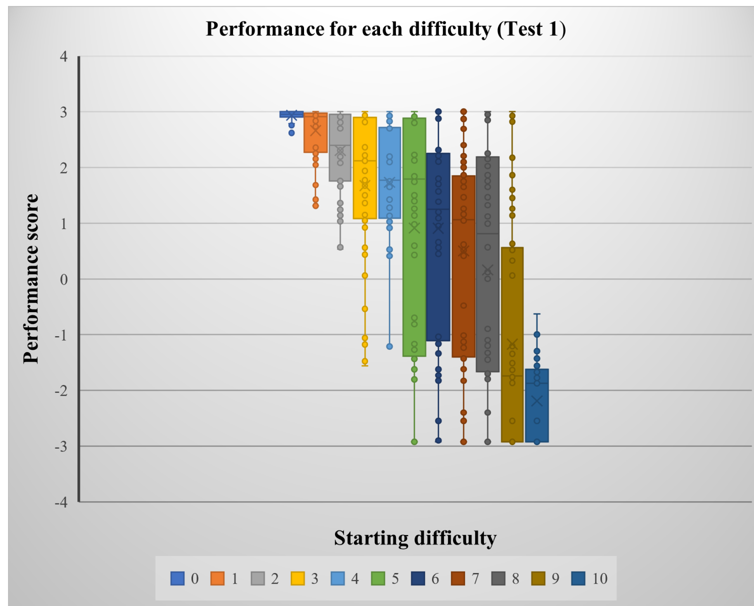


Figure B.10: Box plots of Test 1's performance scores in each difficulty with the outliers removed.

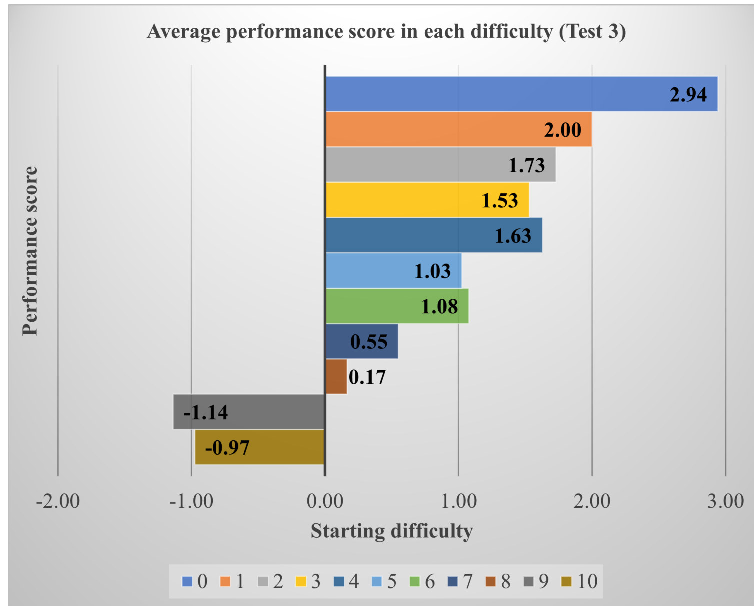


Figure B.11: Average performance score obtained in the first rhythm of Test 3.

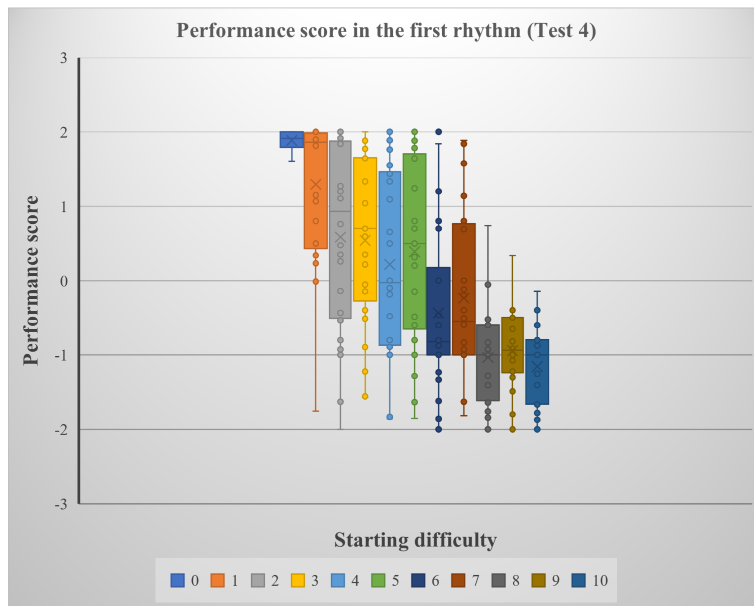


Figure B.12: Box plots of Test 4's performance scores in each difficulty.

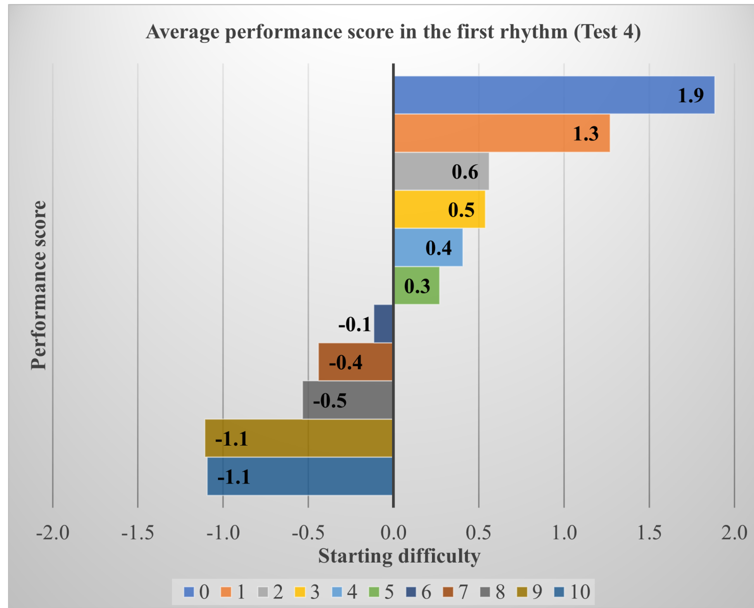


Figure B.13: Average performance score obtained in the first rhythm of Test 4.

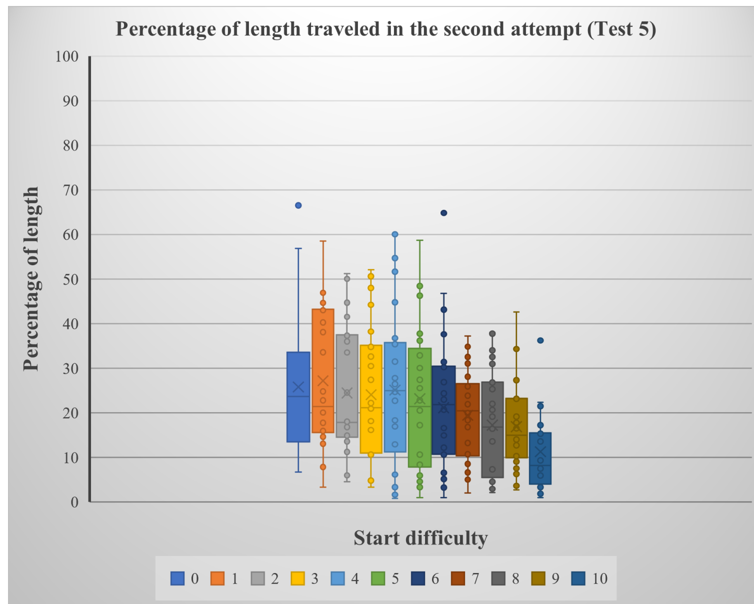


Figure B.14: Box plots of the percentage of length travelled in the second attempt of Test 5.

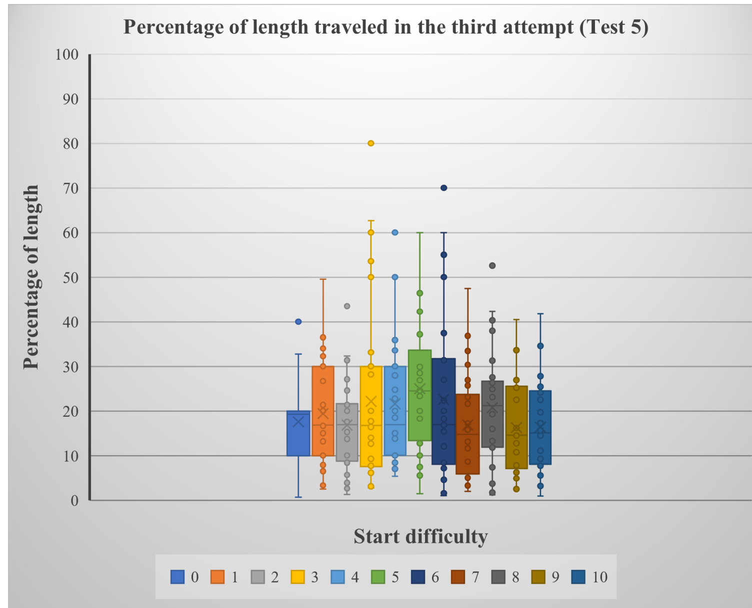


Figure B.15: Box plots of the percentage of length travelled in the third attempt of Test 5.

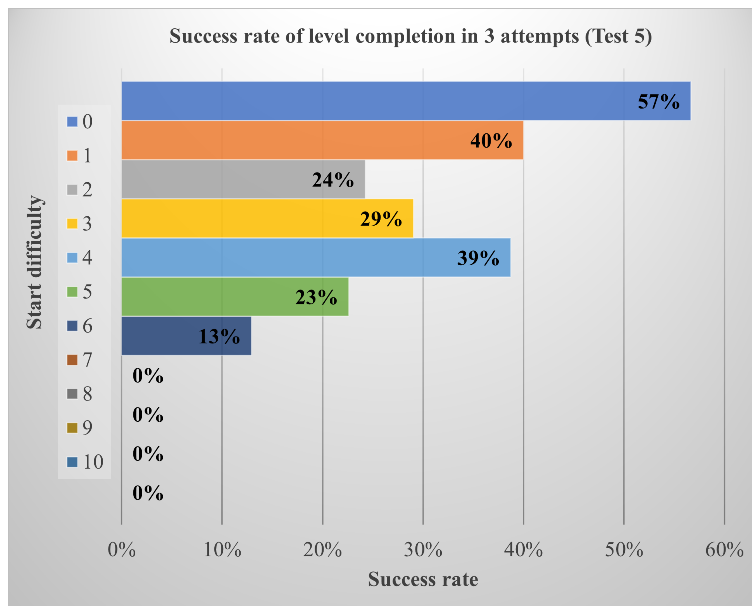


Figure B.16: Percentage of games won by the agent in Test 5.