Generating Automatic Mario Levels With a Genetic Algorithm

by

© *Shawn Douglas Sabraw*

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

Master of *Science*

Supervisor: *Dr. David Churchill*

Department of *Computer Science*

Memorial University of Newfoundland

November 2024

St. John's Newfoundland

Abstract

Procedural Content Generation (PCG) is one of the key features of modern entertainment, showcased in a vast set of applications such as, animation, film, and of course many diverse genres of games. Using the fundamental elements of levels such as the environment, enemies, and player, procedural content generation has the capability to construct levels, maps, and even entire games. This thesis concentrates on the application of a Genetic Algorithm (GA) to autonomously create levels for 2-D platformer games, exemplified by games like Super Mario. These generated auto-levels are compared through randomization, parameter tweaks and level space restrictions. We successfully uncovered insights in PCG indicating that the generation of complex levels is intrinsically linked to the appropriate investment of generation time and a balanced utilization of essential level tiles. Notably, our findings highlight the importance of these factors in achieving optimal level design. To enhance the complexity of the generated levels, we have introduced a "least-block" fitness function. This novel approach not only sheds light on positive aspects but also identifies areas for improvement, distinguishing between cluttered and sparse generated levels.

In loving memory to my poppy, Douglas Gerald Summers, who believed in me and encouraged me to go further in my academic career. I will never stop missing you and I know you would have been so proud to see this finished.

Acknowledgements

This journey started in a meeting with Dr. David Churchill where multiple paths going forward were laid out, the substantial decision of which path to take came down solely to me. I would like to give my appreciation and the highest thanks to my supervisor and esteemed professor, Dr. David Churchill, my mother, Natalie Sabraw, and of course my father, Dion Sabraw for their guidance and continuing support over the following years from that meeting. My sister, Chantal Brokenshire, my brother-in-law, Burke Brokenshire, and my niece and nephew, Hazel and Frederick, along with all of my family have helped encourage and motivate each step on this path and I appreciate them immensely. Along this new chapter in the journey, I'd gain colleagues and friends walking a similar path to mine, who offered me support in the most crucial of times. I'd like to thank Caroline Strickland, Rory Campbell, Rick Kelly, Lucas Critch, and Rob Bishop. Thank you, everyone, for all of your support, and thank you, the reader, for taking some time to examine a glimpse into the path I've chosen.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The demand for Artificial Intelligence (AI) has reverberated within all industries and continues to increase. We see various types of AI systems put in place to learn and generate data used in crucial decisions from linguistics to medicine, and of course games [35]. These systems are highlighted in implementations such as ChatGPT [31] for natural language processing, where an AI analyzes language, infers meaning and context to then respond to the user and exhibit intelligent behaviour. Within medicine, we see AI systems used in chemotherapy regimens, risk prediction, and early detection of diseases, including cancers [9]. In contemporary gaming, AI plays a crucial role in shaping various aspects, such as character movement and behaviour, music composition, art generation, and even level design. Its integration significantly contributes to enhancing and creating engaging game content [29]. Level design is crucial for a video game's success as a level is the environment in which all of the entities in the game interact.

1.1 Platformer Games

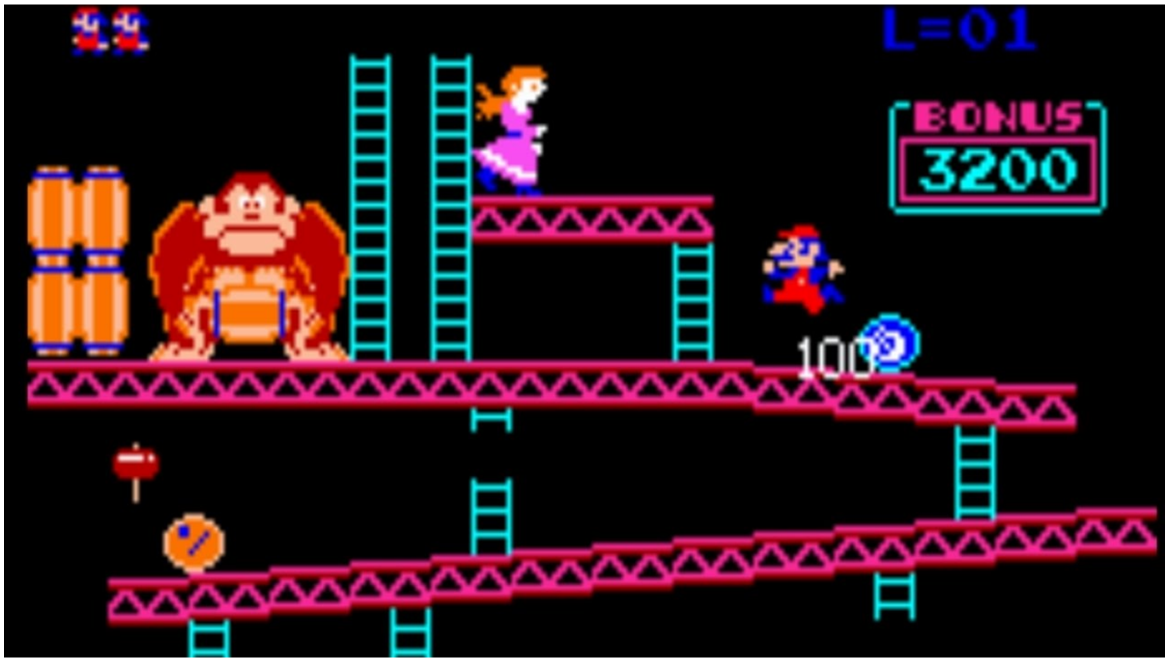


Figure 1.1: A screenshot of Donkey Kong (1981).[1] Click this figure to play the video that this screenshot references.[2]

Platforming games, or platformers, are defined by a specific set of gameplay features and mechanics. These encompass precise character movement and jumping, granting players control over navigating a two-dimensional or three-dimensional environment while avoiding hazards, surmounting obstacles, or reaching elevated platforms. The intricately designed levels incorporate platforms and obstacles at diverse heights and distances, demanding players to showcase agility and timing. The inclusion of power-ups and collectibles further enhances gameplay, requiring players to

[1] `https://www.youtube.com/watch?v=rYNMatF5hcU`

[2] All following screenshots similar to figure 1.1, can be clicked in order to view their respective video.

gather items for scoring, advancing through levels, or enhancing their character's abilities. Enemies and bosses, the hostile entities populating the levels, introduce elements of combat or evasion. Lastly, puzzles contribute an additional layer of challenge, and the overall game structure includes a sense of progression, whether it follows a linear or non-linear path.

Before the term "Platformer games" emerged, there existed a game genre referred to as "Maze games", reminiscent of titles like Pac-Man. This genre underwent continuous evolution, reaching a milestone with the release of Space Panic in 1980 for arcades. Space Panic introduced elements like ladders, platforms, and notably, gravity [33]. While some argue that Space Panic marked the true beginning of Platformer games, the title officially belongs to Donkey Kong, launched in 1981. Donkey Kong, as displayed in Figure 1.1, introduced a crucial element for platformers — jumping. The immense popularity of Donkey Kong played a pivotal role in establishing and popularizing the new genre of Platformer games [4]. Donkey Kong's triumph paved the way for the inception and sustained prosperity of the Mario franchise [4]. This journey began with the release of Super Mario Bros. in 1985 and continues to thrive, evident in the recent launch of a new Nintendo Switch game, Super Mario Bros. Wonder. Notably, this latest instalment has achieved the title of the fastest-selling Super Mario game in history [18].

Super Mario Maker and Super Mario Maker 2, as referenced in Figure 1.2; released in 2015 and 2019 respectively, are games that allow the player to not only play new Mario levels but create and then share them with people online. To create these levels the player is shown a library filled with all the different assets throughout Mario games including, blocks, mechanics, music, enemies, and more. The player takes the assets

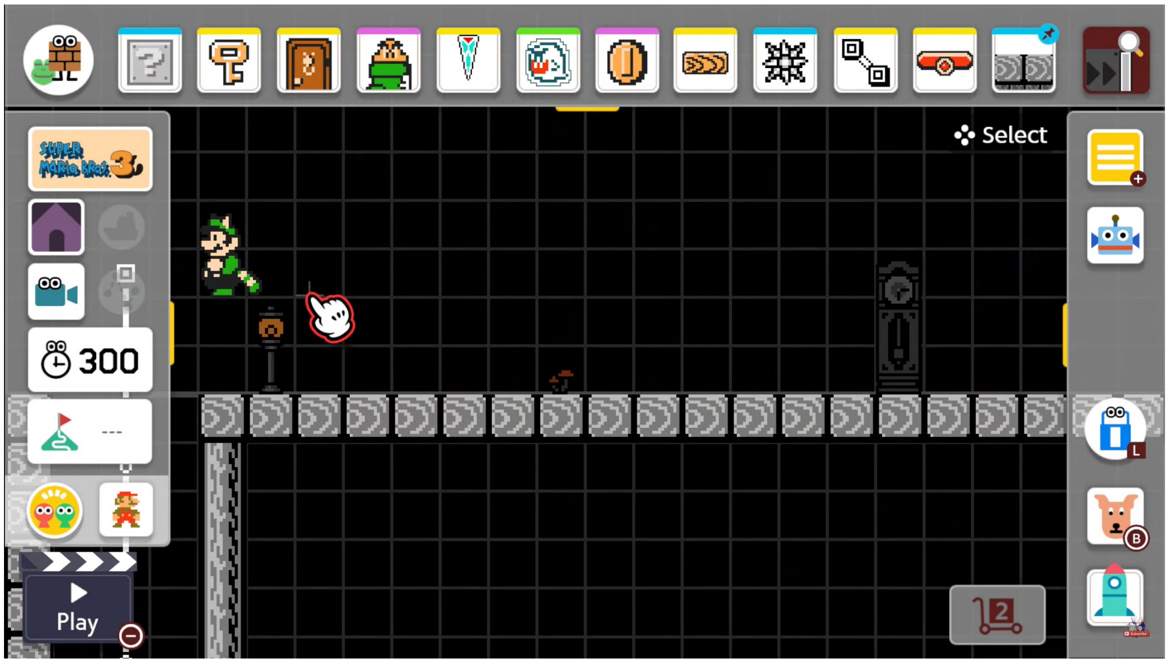


Figure 1.2: A screenshot of Mario Maker 2 (2019). Click this figure to play the video that this screenshot references.[3]

and can place them down on a 2D-grid space in any way they can think up to create a level, they can then test their level at any time and finally publish the level to be downloaded and accessed through its Course ID. Over 2 years from its release in 2019 to April 13, 2021, there have been over 26 million user-made stages created, and in this seemingly boundless library of levels, the users have created their own genres [30].

With the continuous rise in the number of created levels, there is also a rise in difficulty and complexity within most of these levels. The level designers like to create tough levels and challenges for players to complete. As players continue to conquer the newest challenges, the challenges continue to get more complex. These difficult

[3]`https://www.youtube.com/watch?v=5WkaKoQRwjg`

stages are called Kaizo or Troll levels [17]. To be deemed Kaizo, the level must be extremely difficult with a less than 1% win rate, but still have a linear sense of progression where there are multiple puzzles, usually with a single clear solution. A troll level, is extremely difficult with a non-linear sense of progression, less of a puzzle and more random and chaotic mechanics.

1.1.1 AutoPlay Levels

On the other side of the genre spectrum, there are Automatic levels; Figure 1.3 as an example, which are usually the complete opposite of Kaizo or Troll levels. AutoPlay(Automatic) levels are stages where the level guides the player's character through itself by the use of in-game mechanics and physics. We refer to these automatic levels as AutoPlay levels to highlight that there is usually no player interaction with the controls required at all. These levels typically aim to evoke a wow-factor by propelling the character swiftly through narrow gaps, perilously challenging environments, or by orchestrating a song generated as the character bounces and interacts with various blocks throughout the level.

An AutoPlay Level could be thought of as a Rube Goldberg machine, but in the context of a 2D platformer game. A Rube Goldberg machine is a series of linked systems which interact with each other to solve a simple problem in a very elaborate or humorous way. These machines are derived from an American cartoonist, Rube Goldberg's cartoons in which he depicts hilariously complicated machines to solve very simple everyday problems, such as opening a can, wiping your face with a napkin, and much more [16]. Other research done in level generation have also taken inspiration

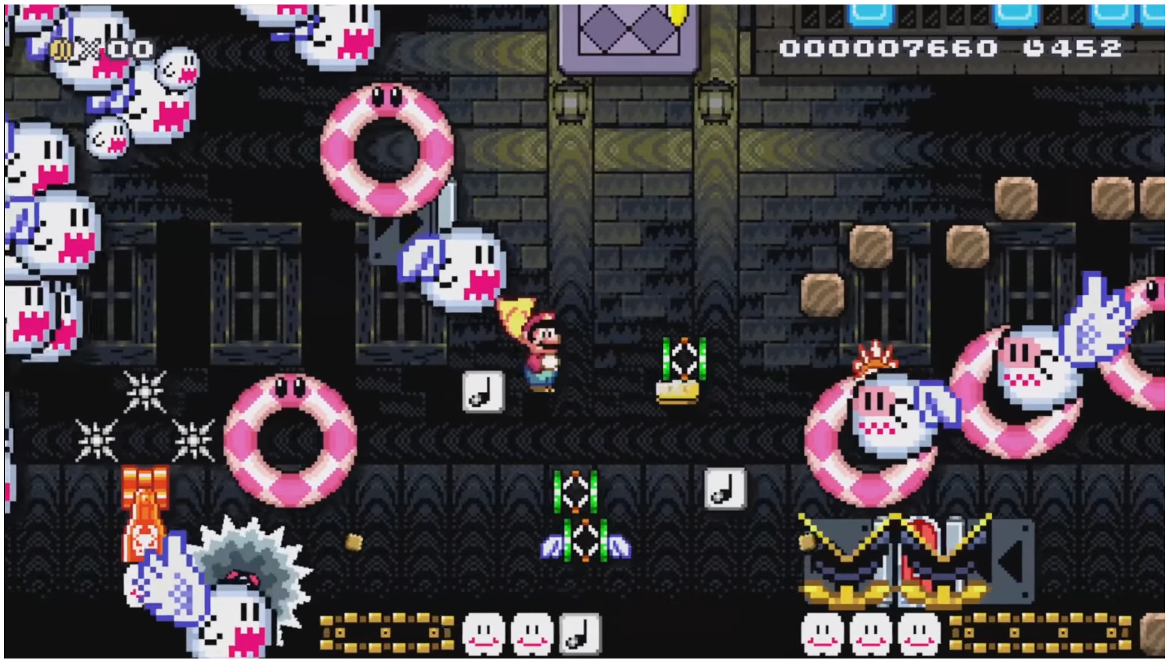


Figure 1.3: A screenshot of an AutoPlay level. Click this figure to play the video that this screenshot references.[4]

from Rube Goldberg machines seen in Figure 1.4, by utilizing a “domino effect” created by stringing together systems and structures based in the game Angry Birds. Abdullah et al. [1] has been able to generate levels that act similar to a Rube Goldberg machine, starting a chain reaction from a singular starting point all the way to completing the level.

Aside from letting a chain reaction solve a very challenging and complicated level, there are other motivations for creating these AutoPlay levels. Some level creators look to push the bounds of what is possible in the game’s physics engine, others want to make a very enjoyable or comedic adventure, and some creators will even want to showcase music in their levels by using the blocks to emulate the music, exhibited in

[4]https://www.youtube.com/watch?v=08i09c3Kwmc

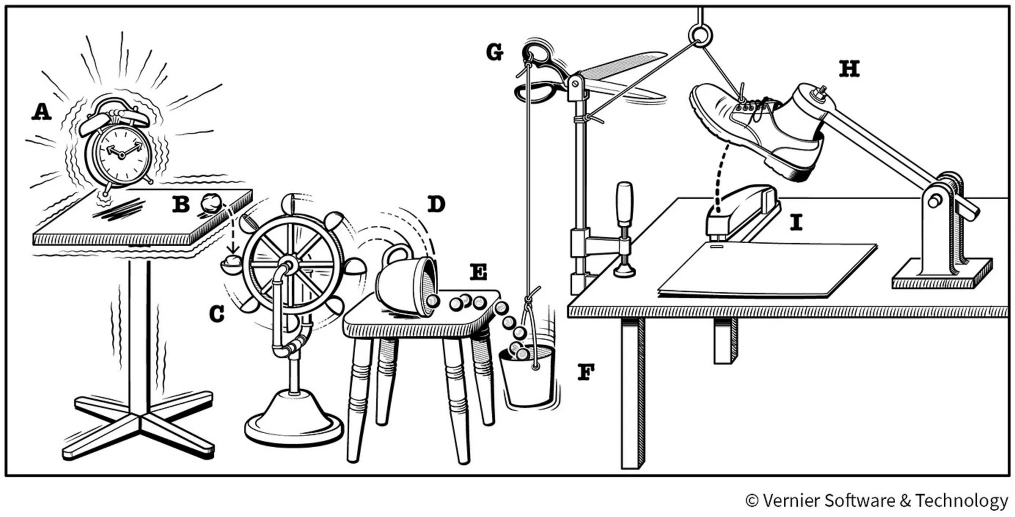


Figure 1.4: A diagram of a Rube Goldberg machine project. Click this figure to see the project that this screenshot references.[5]

Figure 1.5.

There are a few sub-types of AutoPlay levels such as: no press, press forward, and RNG (Random Number Generator). No press levels are the ones we will be using in this thesis, where the entire level controls the flow and carries the player to the finish line. Press Forward or Press Button levels are where the player must hold a single button down either to make the character jump, or move in a singular direction constantly and that interaction acts as the catalyst to eventually get the player to the end of the level.

Finally, an RNG or troll-AutoPlay level is a level that works only sometimes, these are usually the chaotic type of maps that will throw the character around and

[5]`https://www.vernier.com/experiment/pep-16_rube-goldberg-machine/`

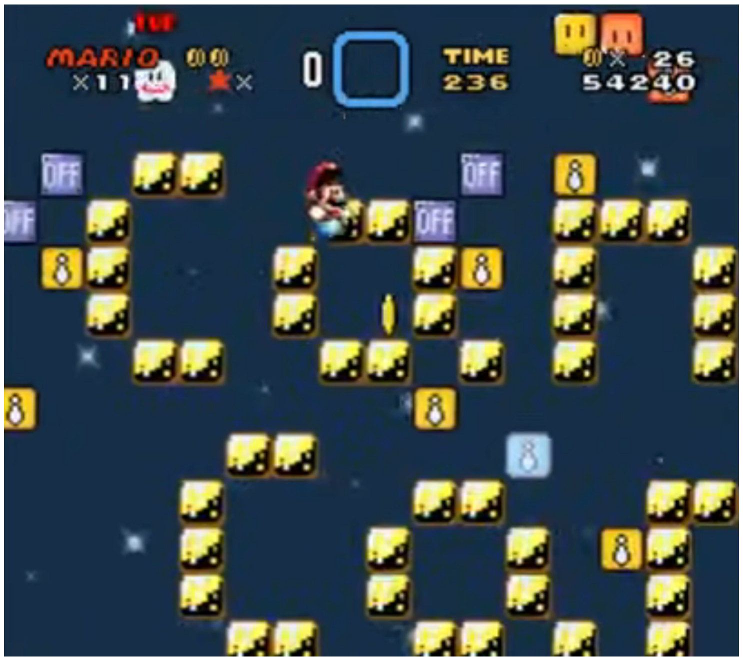


Figure 1.5: A screenshot of a Mario AutoPlay Level Synced to the song “Can Can”. Click this figure to play the video that this screenshot references.[6]

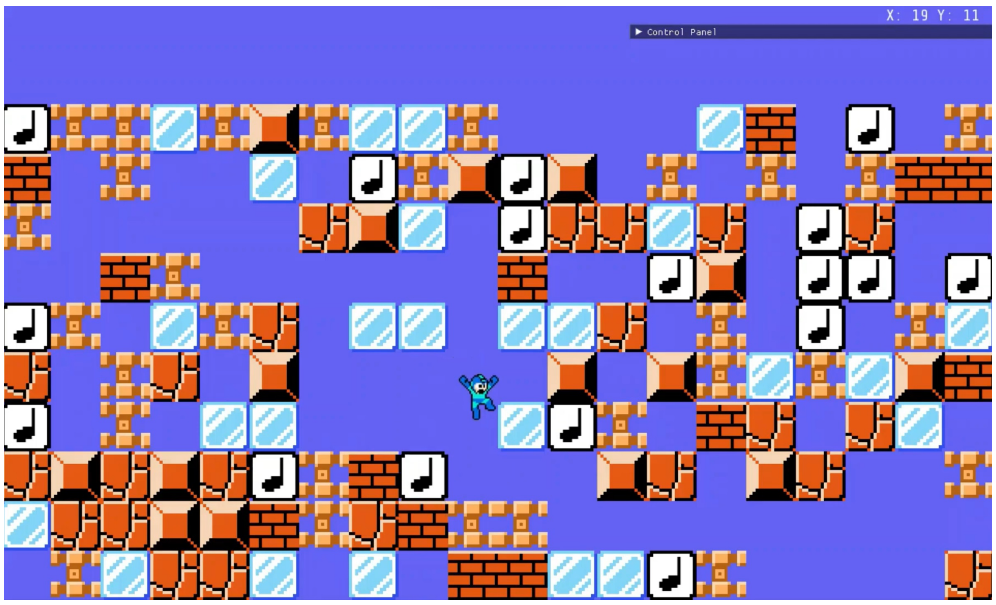


Figure 1.6: A screenshot of a Mario AutoPlay level generated by the algorithms presented in this thesis. Click this figure to play the video that this screenshot references.[7]

potentially they get to the end via the chaotic interactions and chance.

Throughout these types of Mario levels you will see the same type of mechanics and interactions because of the specific block types used. In order to propel, move, or bounce Mario, level blocks must be able to interact with Mario in some way. Some interactions that could bounce Mario upward would be a shell, Note block, or enemy once Mario collides with the top of that entity. Another type of interaction for lateral movement could be a conveyor, cloud, ice, or moving platform where the character is pushed or advanced in the direction the block is directing as shown in one of our levels in Figure 1.6.

1.2 Procedural Content Generation

Procedural Content Generation (PCG) is the technique of generating content algorithmically and is a highly sought-after system for game designers in the gaming industry [14]. In games, PCG abstracts decisions for game designers by letting the program adjust and fine-tune parameters to create a plethora of content, reducing the burden on game developers, and is commonly used for procedural dungeons, levels, and maps. The strategy of combining PCG with AI is to create an AI tool that assists a developer in their work.

Map and content generation play a pivotal role in the gaming industry, with continuous advancements and innovations. Notable examples that showcase the diverse applications of Procedural Content Generation (PCG) include Minecraft, No Man's Sky, Diablo 4, and Spelunky.

[6] `https://www.youtube.com/watch?v=Xey9yqwXECE`
[7] `https://www.youtube.com/watch?v=m95ur6hykKU`

Spelunky, a roguelike[8] platformer, stands out for its unique gameplay, art style, and world design. In this game, players dig downward, navigating traps and enemies to accumulate wealth and progress. PCG is integral to Spelunky's success, delivering a challenging and endlessly replayable experience.

Diablo 4, an action/adventure massively multiplayer online role-playing, employs PCG to dynamically generate dungeons, altering layouts, monster placements, and treasure drops. This use of PCG enhances the game's adaptability and replay value, providing players with a fresh and unpredictable dungeon-crawling experience.

No Man's Sky, an open-world survival game, presents an expansive and virtually infinite universe. Each solar system, planet, and its elements, including weather, flora, and fauna, are procedurally generated. This showcases the versatility of PCG, enabling the creation of a vast and diverse gaming environment that encourages exploration and discovery.

Minecraft, another open-world survival game, integrates PCG into nearly every aspect of gameplay. This includes world generation, where the game creates terrain, biomes, and biome-specific structures. Resource distribution also relies on PCG, ensuring ores, minerals, flora, fauna, chests, and loot items are appropriately and accurately dispersed based on the generated world location. Additionally, structures like villages, mineshafts, and dungeons are incorporated into intricate cavern systems with multi-level terrain, influenced by the surrounding biomes. This underscores the critical role of PCG in almost every facet of the game.

These examples underscore the significance of PCG across different genres, demon-

[8]A subgenre of role-playing games, usually containing elements such as permadeath and randomized levels.

strating its ability to enhance difficulty, replayability, and the overall gaming experience. From the challenging depths of Spelunky to the dynamically evolving dungeons in Diablo 4 and the boundless universe of No Man's Sky, PCG continues to drive innovation and creativity in game development.

There are even competitions to create PCG AIs and research into general content generation for levels [32]. The undertaking of creating a AAA[9] game takes a considerable amount of time. AI is a probable way of speeding up that development time and also increasing efficiency overall [8].

Our system will generate content, specifically automatic levels for a platformer game with the likeness of Mario. A platformer game is a video game in which the player controls a character and navigates them through an environment with multiple interactions such as, running, jumping, climbing, or attacking. An automatic Mario level is an environment where the player does not move the character at all, but the level; with the help of physics and other mechanics, moves the character to the end of the level usually in some interesting fashion. The full extent of the experiment and platformer will be explained in section 3.

1.3 Thesis Outline

In Chapter 2, we delve into the background and previous research in the AI for games domain, specifically focusing on Procedural Content Generation (PCG) and Genetic Algorithms, alongside other notable algorithms used in PCG. Chapter 3 will thoroughly examine our Custom 2-D Game Engine, detailing its significance and usage

[9]A grading given to high-budget, conspicuous video game titles, usually created by larger game studios.

in our experiments concerning level generation. Moving on to Chapter 4, we deconstruct the Genetic Algorithm, elucidating its efficacy as a PCG tool and elucidate on the algorithmic representation of our generated levels. Chapter 5 will detail the conducted experiments, elaborating on the level naming convention's rationale and its significance, along with the evaluation metrics employed and the resultant findings. Finally, Chapter 6 will summarize the conclusions drawn from our research and explore potential future directions.

Chapter 2

Background

In analogous research within the realm of Procedural Content Generation (PCG), the primary objective commonly revolves around utilizing artificial intelligence (AI) for the purpose of level generation. We see some researchers try a more open and generic approach to include more types of games and levels [36, 37, 19]. However, performance and processing complexity are always the most common issues we face in this area [39, 38]. In this section, we will explore some related work to better understand other researchers' objectives and the issues they faced.

2.1 Algorithmic Approaches

When designing levels, it has been found that designers find a lot of value in utilizing AI as a tool [12]. An AI could be programmed to adhere to a predefined set of rules or constraints, producing levels in a manner consistent with a developer's specifications. Alternatively, the AI could be granted the freedom to explore, generating levels or games that were never envisioned before.

A Genetic Algorithm (GA) is an example of an AI system which replicates the idea of evolution by reworking rules and constraints to achieve a goal, which will be explained in detail in Chapter 4. With level design as a goal, a GA would be able to evolve multiple different levels based on a designer's discretions. Utilizing this AI as a tool would allow designers to improve the GA's decision by intermittently giving human input into generating levels. The genetic algorithm allows for some constraints to be followed, but also for exploration, which is perfect for the goal of the experiment we conducted for this thesis, further explained in Chapter 4 and 5.

By combining AI and PCG, developers can create a toolset which generates complex content. Due to the vast genres of games, currently, it is implausible to make a single AI tool that can produce all content necessary for compiling multiple game content of different genres. It is better suited for developers to create an AI for a single goal and complete it efficiently.

2.2 Genetic Algorithms for PCG

All through the field of PCG and surrounding area, many algorithms are being used in an attempt to generate levels, games, and even player-AI interaction. The genetic algorithm is a popular choice for generating levels, but researchers have taken many different approaches with their restrictions and classifications.

A very common genetic algorithm type seen throughout the related area of experiments is the Feasible-Infeasible Two-Population (FI-2Pop) [21]. This GA is the implementation of two types of populations. One keeps track of the worst evaluated individuals (infeasible) and the other, the best individuals (feasible), this is

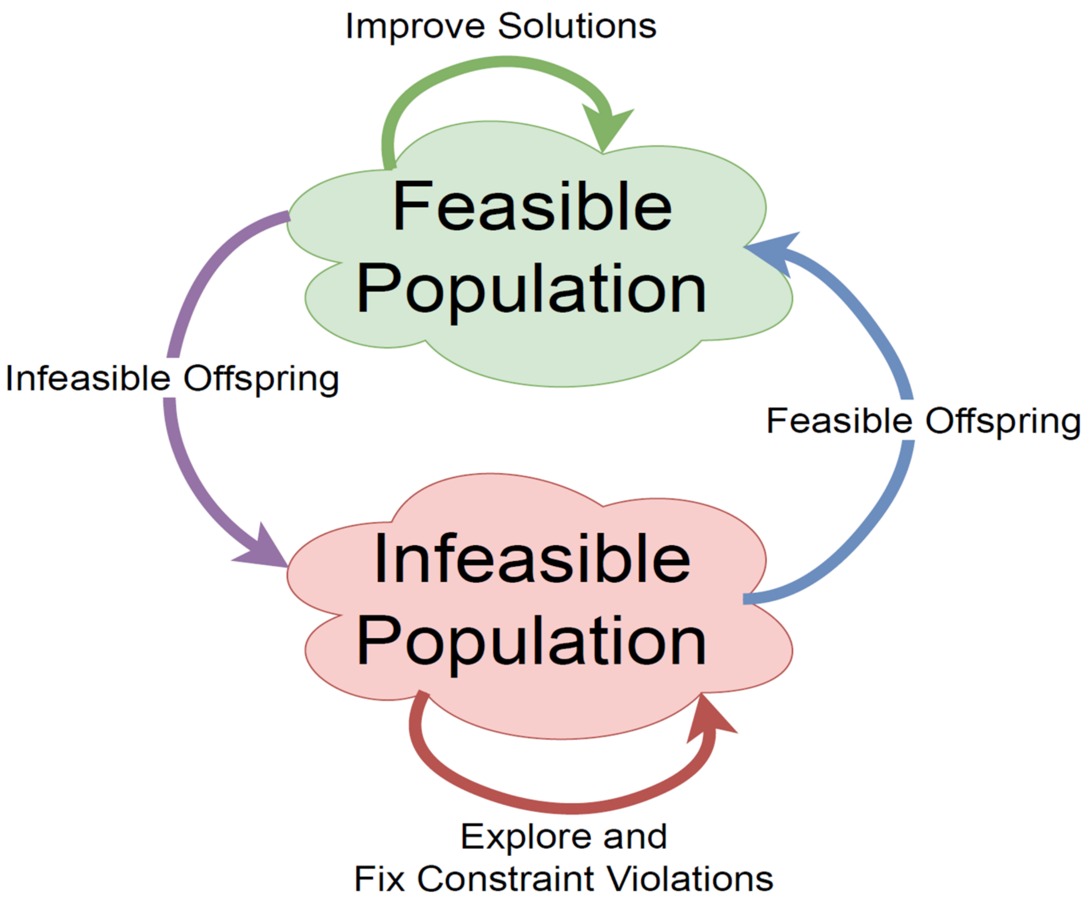


Figure 2.1: A diagram representing the implementation of the FI-2Pop algorithm.

highlighted in Figure 2.1. This proves advantageous as it enables the infeasible population to explore boundary regions without undergoing evaluation by the objective function, potentially revealing the location of the global maximum.

Grammatical Evolution (GE) is another approach used by some researchers in this area [25, 34]. A grammar is a way to programmatically link a set of rules or instructions to a context-free medium such as an integer or integer string as shown in Figure 2.2. This idea allows very complex levels to be simplified enough to viably be used in search-based algorithms without having a very hefty processing time.

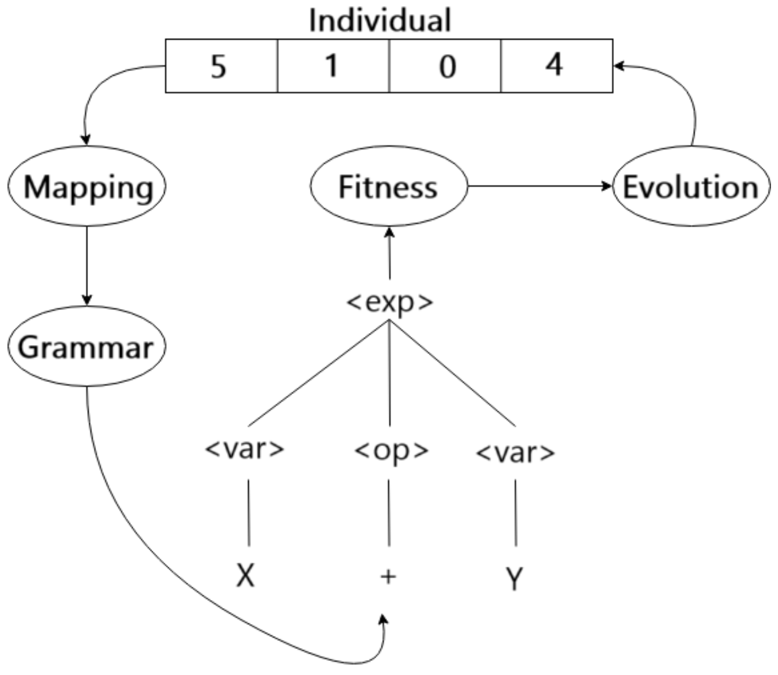


Figure 2.2: A simple diagram representing a grammatical evolution scheme.

The challenge in the context of video game rule generation involves creating algorithms that, given a level, can generate rules to ensure the level's viability. [20]. In [20], they explore the inverse of that problem to create a game based on a set of rules. Their genetic algorithm searches for feasible games/levels that do not break the constraints of the rules.

Similar to that of inverting the rule generation problem, [15] decided to use a genetic algorithm and a state machine in tandem with a Mario level to create a perfect bot for each level given to it. The genetic algorithm will continue to explore different possible transitions in the state machine to find the best possible individual for the level.

2.3 Generating Content

Procedural Content Generation is widely employed across various methods to computationally generate content for games. These methods encompass applications in reinforcement learning, machine learning, and even deep learning, leveraging neural networks [24]. These types of methods are used for generating music/sounds, textures/art, text, and particularly video game levels, which constitute the primary focus of this thesis. Many PCG methods reliant on search algorithms employ evolutionary algorithms [39]. In this thesis, we have chosen Genetic Algorithms for their ability to produce superior solutions to optimization and search challenges. While it's conceivable to apply a simple search algorithm to this problem, the main obstacle lies in the immense size of the problem space. Using a simple search algorithm would necessitate iterating through every game tile against every possible placement on the screen, across every screen included for generation. The search space is overwhelmingly vast, which is where a genetic algorithm excels. Unaided, a GA can navigate through the staggeringly large search space and deliver an optimal solution for generating level content.

PCG has been investigated using diverse approaches in the surrounding research related to this thesis. An interesting and popular idea with PCG is the ability to generate content based on difficulty [5, 3, 38] or a Difficulty Curve shown in Figure 2.3. These papers delved into the inquiry of defining difficulty in games requiring simplification to generate content around it. Usually, the difficulty would be determined by ideas such as: how many floor tiles there are, the number of obstacles, or enemies. A formula would have to be derived from such ideas and then used as an evaluation for the produced content.

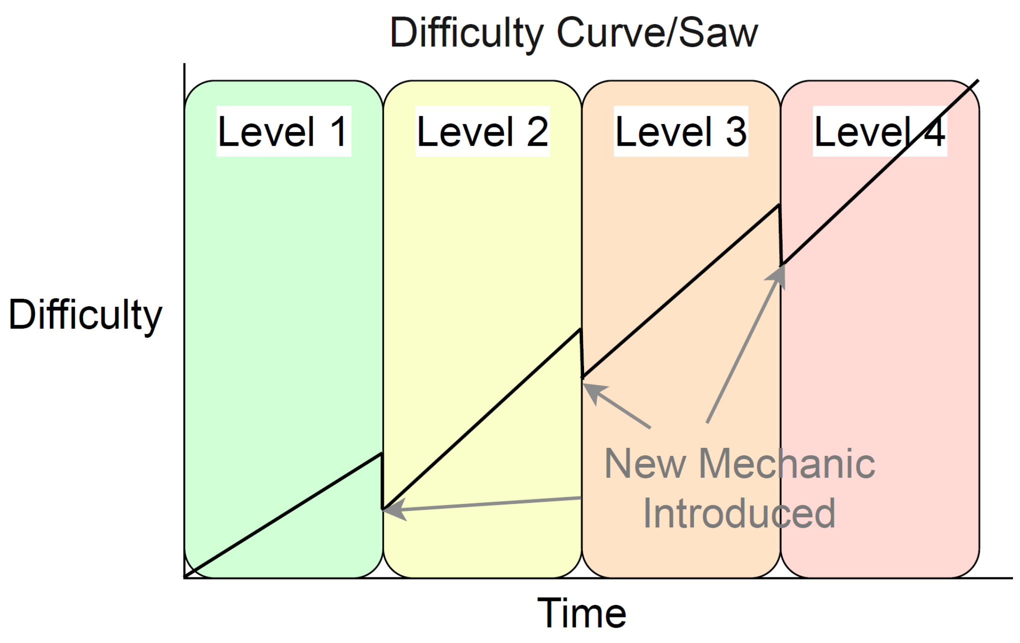


Figure 2.3: A diagram highlighting the difficulty curve of a game.

A similar topic to PCG based on difficulty was that of rhythm. Rhythm is the idea of how much interactivity a player has with the game over a specific period. Researchers use this idea to generate content based on a rhythm set or group [25, 22, 37]. A rhythm group is a set of actions the player must complete to beat the level. A typical rhythm would be fewer interactions at the beginning, and progressively ramping up as the level continues, usually dipping down throughout the level.

A closely related topic to PCG rhythm is patterns. This is where content for a level is created in pre-defined formations and generated together [6, 7, 26]. The usage of patterns works well in tandem with rhythms since specific jumps or interactions are included in the patterns.

In [11], they discuss the idea of scene stitching. This is where similar parts of a level containing the same type of mechanics are "stitched" together in a way to make the resulting level feasible. They use a genetic algorithm in comparison to a greedy algorithm to compare the system with a given set of mechanics.

To generate complex and lengthy mazes, [2] combined a genetic algorithm with cellular automata. Their genetic algorithm would evolve the rules that control the cellular automata that end up generating a playable maze. To get mazes with interesting and lengthy paths, the genetic algorithm's evaluation step fine-tunes the cellular automata with more complex rules.

2.4 Purpose

This thesis aims to examine the interaction between Procedural Content Generation (PCG) and Genetic Algorithms (GA) in creating auto-play Mario levels. In previous studies, an external agent was often required to test generated levels. Such agents could include human players controlling the character or algorithms like A* search, designed to find the optimal path through the generated level. However, in our research on auto-play levels, no external agent is necessary because the level itself functions as the agent. This approach reduces the processing and testing time required for evaluating the generated levels.

Chapter 3

Custom 2-D Game Engine

One of the initial and significant challenges in designing experiments for this thesis was determining how to effectively implement algorithms into the game. Somehow obtaining the source code of Mario, along with all associated assets, and integrating an algorithm directly within that environment proved impractical. Consequently, a more viable solution was pursued. The decision was made to address this challenge by creating a custom 2-D game engine; referenced in Figure 3.1, capable of playing and simulating a 2-D platformer game similar to Mario. This engine provided the flexibility to seamlessly apply algorithms to various aspects of the game, facilitating the experimentation process.

The custom 2-D game engine is written entirely in C++ and built from scratch. The architecture is designed following the Entity-Component-System (ECS) model, where each aspect of the design is modular and controlled independently. The engine utilizes an external library, the Simple and Fast Multimedia Library (SFML), which manages the loading and rendering of textures as well as user input. SFML is one of

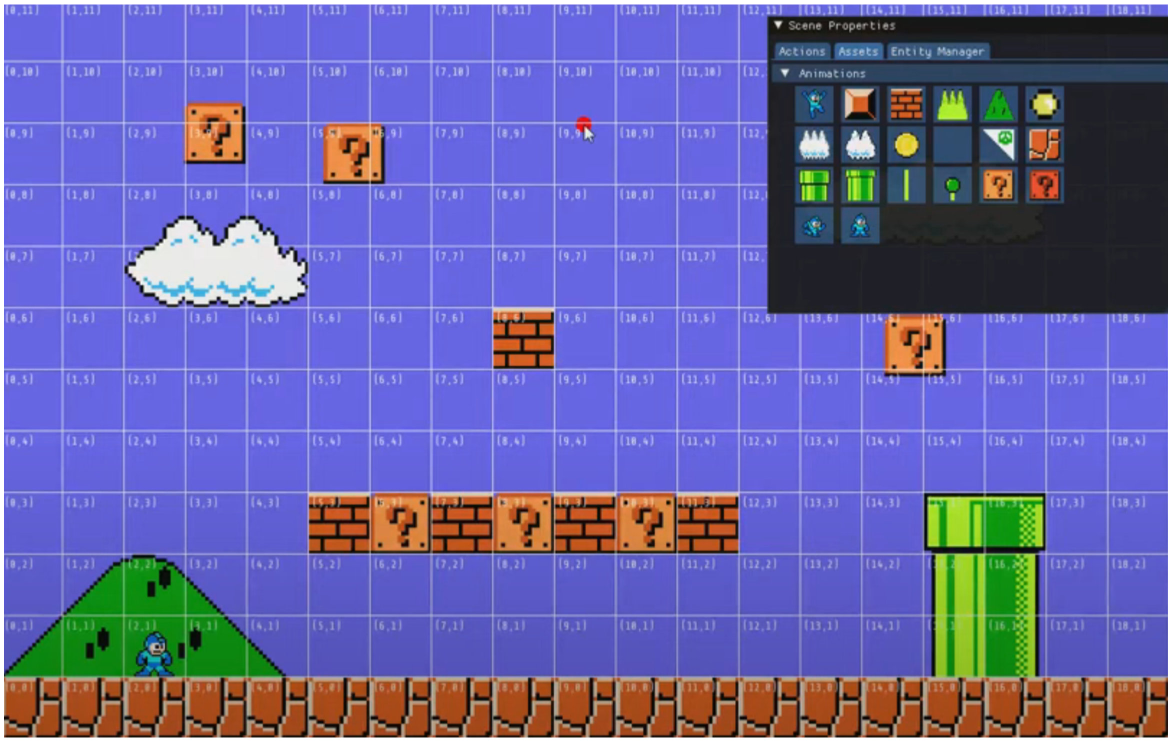


Figure 3.1: A screenshot of the game engine.

the best graphical libraries when targeting a wide variety of different machines [23]. SFML combined with the ECS design allows for the creation of many types of 2-D games.

The entities (blocks) used in generating levels are comprised of a 64x64pixels texture. Each block's texture representation is rendered on a grid, constrained to the given size parameters and loaded inside a single scene. The game engine contains many different scenes for different experiments, levels, or even entirely different games. The subsequent section will delve further into the specifics of each component within the custom 2-D game engine.

3.0.1 ECS

Entity-Component-System is a very popular architecture design pattern used in many recent interactive systems, specifically games. It offers solutions to many issues with other programming designs that are centralized around Object-Orientated Programming (OOP) [13]. Modularity is the main selling point for ECS. As the name implies, ECS is broken down into three sections:

1. Entity: This is the representation of an object within the game, which encapsulates a collection of Components.

2. Component: A module containing a trait or aspect of an entity, such as size, health, colour, etc.

3. System: These contain the overall game logic on how entities interact and their components altered.

Since the components are modular, they can be switched, removed, and updated at any time by one of the systems. With this powerful approach, ECS can address and improve on the main issues with similar architecture: computer code modularity and the overall engine performance [28]. This is why ECS was chosen to be the primary architecture design of the custom 2-D game engine.

With this design, many new game objects and components to go along with them can be created without affecting any underlying code. This also allows for the creation of new experiments through the use of different systems. The main systems in the game engine include:

1. Update: The main update loop, deals with updating all systems and maintains the flow of the game.

2. Lifespan: Each entity has a lifespan which when finished, starts the clean-up process.

3. Movement: Controls all entities' movement and even player movement with input from the keyboard.

4. Collision: Deals with entity collision logic and handles the physics of what happens to each entity.

5. Animation/Render: Deals with animating and drawing each entity to the screen depending on its position and status.

3.0.2 Scenes

The custom 2-D game engine can have many different scenes and each one being a different game or another level. A scene is simply a container to keep specific entities and game logic in. This GA experiment will be contained in a singular scene modelled after the game Mario. In this scene, multiple levels can be loaded in or even created through the use of the GA, which will be explained further in section 4.

The main entities inside this scene are known as blocks. Blocks are simple 64x64 pixels entities which interact with the player and are used to build the entire game level. There are five main types of blocks, used in the creation of a level: Bricks, Conveyors, Ice, Bounce, and None. Bricks have a normal amount of friction and are immovable and impassable. Conveyors move the player in a straight vector and

Blocks	Name	Mechanic
	Bricks	High friction
	Conveyors	Add speed
	Ice	Low friction
	Bounce(Note)	Bounce the player up
🟦	None	No collision

Table 3.1: Level Blocks used in the creation of our platformer levels.

increase that player's velocity. Ice has very little friction. Bounce blocks will attempt to bounce the player up, increasing their upwards velocity on contact. None blocks are designed to act as air, completely passable and invisible. The blocks we used to create our levels for example in Table 3.1, focus on interacting with Mario and applying vertical and horizontal velocities that can be strung together to progress Mario from the left to the right side of the current scene. Most of all of these interactions take place within the Collision and Movement systems, both dealing with the physics system. Incorporating a modular physics system allowed for the use of not only gravity, but also to give blocks specific properties, such as friction, movement, and triggers. Since the player is also an entity, its physics such as velocity and air friction is mutable as well.

3.0.3 Benefits

The game engine can load autoplay Mario levels and play them in a loop while recording the results however, one of the strong capabilities of the game engine is headless mode and simulation. The game engine can forget about framerate and instead of loading every single system and displaying it all in real-time, a simulation can skip all unnecessary systems and run the levels as fast as the computer's processor will let it. The necessary systems include movement and collision while ignoring the rendering or animations.

With headless mode, the ability to simulate all the levels makes it now possible to test hundreds or thousands of levels all at once, which creates a perfect environment to use a genetic algorithm. Speed is one of the most important hurdles to overcome due to the search space of a game level being so unfathomably big.

Another benefit of using a custom engine with the ECS design is the ability to add a GA as just another modular system. This enables the GA to access all required information to evaluate each level's viability and execute different simulations without going through the rendering segment.

Chapter 4

Genetic Algorithm

GAs are a type of evolutionary algorithm that was created to mimic the process of evolution and apply it to numerical optimization problems. It is based on Darwin's theory of Natural Selection in which positive traits are reinforced and replicated as time goes on [27]. Apart from initialization, the three main parts that make up a genetic algorithm are:

1. Selection: The selection process involves choosing individuals to serve as parents for the next generation. Elitism ensures that a certain number of percentage of the best individuals (those with the highest fitness) are selected.

2. Crossover: The "elite" individuals are then combined by exchanging data to produce new offspring.

3. Mutation: Here is where the individuals' data is altered slightly. This step is very important as it causes the algorithm to explore different possibilities in search of the global optima instead of getting stuck in local optima.

Algorithm 1 Genetic Algorithm Pseudocode

```
procedure GeneticAlgorithm
    $Population \leftarrow$ RandomIndividuals;                          ▷ INITIALIZE
    while $CurrentEvaluation \neq Goal$ do                 ▷ REPEAT UNTIL TERMINATION
        Evaluate(Population);                                            ▷ EVALUATE
        $Parents \leftarrow$ SelectElite($Population$);                    ▷ SELECT
        $Offspring \leftarrow$ Combine($Parents$)                          ▷ COMBINE
        $Offspring \leftarrow$ Mutate($Offspring$)                          ▷ MUTATE
        $NewPopulation \leftarrow Offspring$                        ▷ NEXT POPULATION
```

4. Evaluation: A quantitative function is used to evaluate each individual, these are known as fitness functions. This step allows the selection process to choose the best individuals.

4.0.1 Representation

The phenotype of a game level in our game engine is a singular map consisting of virtually limitless screens of open space. The space can be filled by blocks or level entities and is 10 blocks high and 20 blocks long per screen. To encode these levels to a genotype, we convert this genetic algorithm's individuals to an integer array, highlighted in Figure 4.1 and 4.2. Each different number of the array corresponds to an entity type defined in the algorithm for each block. It is difficult to apply crossover to a level because Mario levels can be very different without a proper place to splice them together. For this, we splice one half of a level to the opposite half of the other. Mutation simply alters a block type from one to another, depending on the mutation

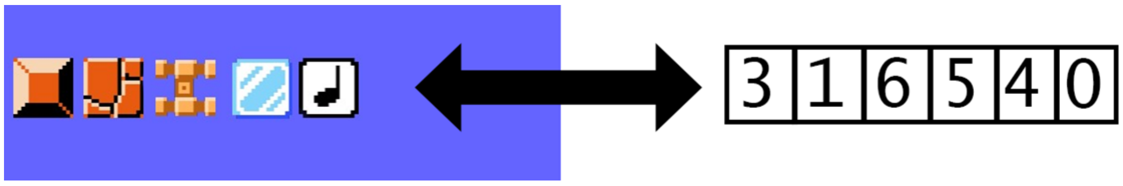


Figure 4.1: A model showing how level entities are represented as a genotype.

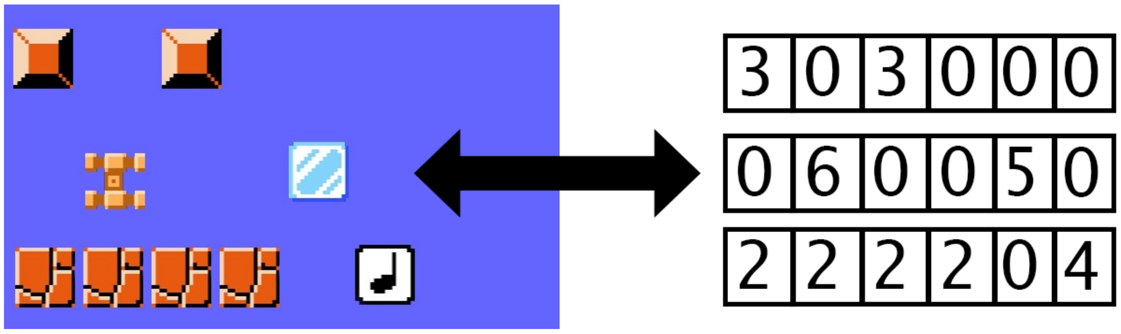


Figure 4.2: A model showcasing a simple level design and as a genotype.

rate and some other parameters. The evaluation of each level will depend on the fitness function and tie-breakers and is discussed further in section 4.0.5.

4.0.2 Parameters

The main parameters of a genetic algorithm; highlighted in Table 4.1, are: Generation Limit, Population Size, Elitism Percentage, Crossover Rate, and Mutation Rate. The Generation Limit serves as a strict constraint on the number of iterations the genetic algorithm undergoes. This not only ensures that the algorithm runs for a sufficient duration but also helps eliminate any unnecessary runtime beyond the specified limit which could lead to over generation of a level. A Phenotype is the actual representation of the population and Genotype is the computational representation of the population. Knowing the lengths of these can help in designing which

Parameter	Description
Generation Limit	The amount of total generations through the lifetime of a genetic algorithm.
Phenotype Length	The length of the actual representation of the population.
Population Size	The total amount of Genotype representations of the Phenotype.
Elitism Percentage	Amount of highest evaluated selectees from the population.
Crossover Rate	Percentage of parent elite selectees to be melded together and produce child pairs.
Mutation Rate	Percentage of random mutation applied to the new population chromosomes.

Table 4.1: Parameters used in the lifetime of a Genetic Algorithm.

data type to use when designing the genetic algorithm. Ideally, it would be nice to reduce the Genotype representation to the most simplest data type possible for an increase in processing speed. Population Size controls how many individual Genotype representations we run through the entire process. As Crossover and Mutation takes place, at the beginning of each generation, the Population Size should usually be the same. Elitism Percentage represents the amount of the population whose traits that we would like to keep, crossover and then mutate for the next generation. This parameter makes sure that we are always exploiting our population to improve it, by only taking the best individuals. Crossover Rate denotes the percentage of the elite population which is then combined in a way to produce hopefully better individuals for the next generation. Finally, Mutation Rate acts as the percent chance for a singular individual in the population to be changed just slightly. This parameter helps introduce some randomness and exploration to our algorithm. Through trial and error, we tweak these parameters to help us find and navigate through local maxima in hopes of eventually finding the global maximum of our goal.

4.0.3 Domain-Specific Parameters

A single screen size of a Mario level is approximately 25 blocks wide and 19 blocks high. This means a permutation of 475 blocks out of all available blocks per screen is possible to create. This is a huge search space for all types of possible Mario levels. This is also the main reason why there are so many local optima in this space. Exploration by mutation is the best way to combat the overwhelming local maxima. The mutation step is the main system for introducing exploration, where the Mario level blocks are altered randomly. For this huge search space, the GA needs a high mutation rate, but there are a few other restrictions and parameters to help search through this space.

By setting a requirement for row and column sizes, the genetic algorithm does not have to worry about unnecessary blocks. Instead of generating a lot of unnecessary blocks that the player would never touch, this focuses the route the player would take to a single path.

4.0.4 Exploration

In the scope of a Mario level, exploration allows for the discovery of wonderful level designs which may have never been used [5]. Exploration for this GA utilizes a few parameters which modify the random chance of generating blocks and even change the landscape of levels by introducing harsher fitness functions.

The blank rate adjusts how often a block does not appear. Through mutation, each block is altered to another type of block and possibly a "none" block, which is invisible and intangible. However, the blank rate adds another layer of randomness to the mutation, by having the possibility of not even selecting from the block pool and instantly being assigned as a "none" block. This is introduced to create more space for movement within the generated map.

4.0.5 Evaluation

Fitness functions and tie-breakers are how the GA evaluates each generated individual. The main fitness functions tune the algorithm to produce levels, which focus on moving the player automatically using the game's incorporated blocks and physics. The parameters for each experiment permitted adjustments to the allocated physical game space, memory, time, and randomness within each generation. The fitness function for this genetic algorithm is simply how far the player makes it to the right. The tie-breaker is total distance including moving left, up, down, and right. So overall, the further right the player makes it, the more viable the level is. In another set of experiments another tie-breaker, "least-blocks" is introduced, to help rid the level of clutter and more unnecessary blocks.

The fitness functions and tie-breakers in the algorithm are completely modular. This allows us to create a list of fitness functions where the first one has the most important value to evaluation and the following lesser. Each fitness function adds more complexity to the level, but it also increases the number of generations for the genetic algorithm for the tie-breakers to have any real effect.

The evaluation takes place after all individual levels in the population have been simulated. Some researchers have rid their experiment of simulation altogether, by instead using a more direct fitness function that evaluates generated levels based on their criteria [10]. This would be ideal to cut out simulation altogether and only focus on processing evaluations however, this would not work for evaluating an automatic Mario level without adding another layer of complexity and searching for a path with A* or something equivalent.

Chapter 5

Experiments and Results

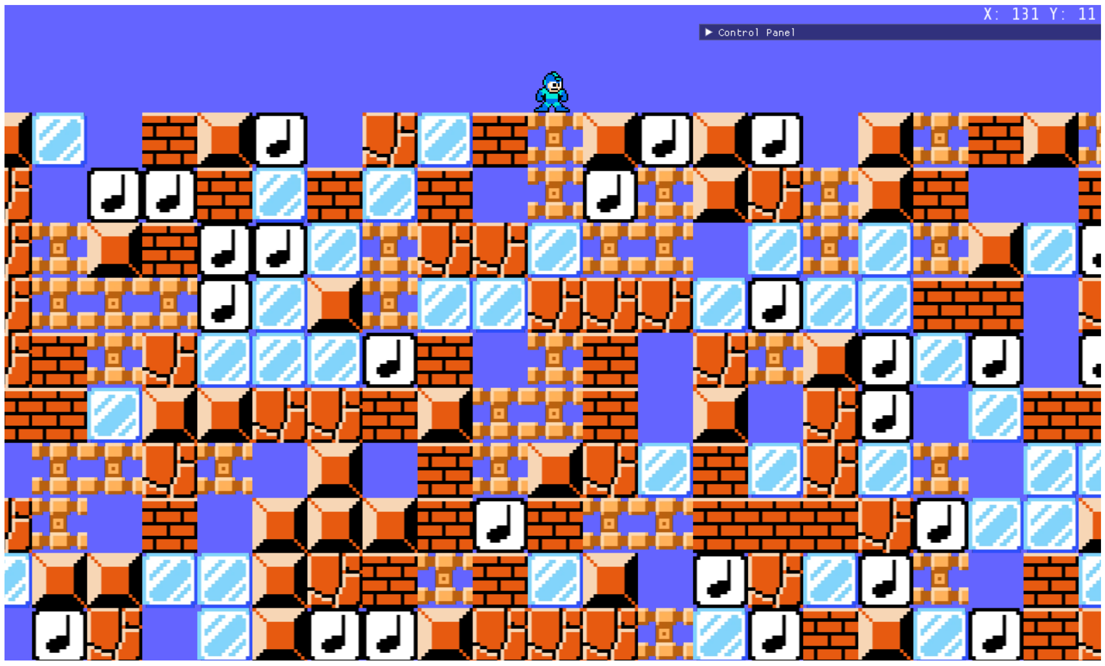


Figure 5.1: A snapshot of "0-0-0" a cluttered level with 0% extra blanks.

Our experiments aim to employ a genetic algorithm for generating auto-play levels in platformer games. We seek to assess the impact of varying parameters and fitness functions within an expansive game environment. To compare the levels, most of the controlling parameters are kept the same such as the mutation rate, except the number of required rows and columns, which separates the levels into two major groups, cluttered and sparse. Figure 5.1 exhibits the base of our experiments, which is a cluttered level that is expensive to generate resource-wise. The other variations which separate the experiments further are the blocks' percentage of mutating into blanks or Blank Rate, as well as an important fitness function tie-breaker called "Least Blocks", which ends up having some negative connotations clarified later.

5.1 Level Representation

Levels for platforming games have traditionally been represented as a grid of tiles, whith each (x,y) location in this grid being given an integer value associated with a type of tile that can be placed in that grid cell (brick, ice, etc). Our initial experiments showed that simply attempting to create a level by letting a Genetic Algorithm modify this large set of tiles resulting in processing times that were far too long, and the levels it generated looked dense and cluttered (Fig 5.1), nothing like standard human levels, which are much more sparse.

To alleviate this problem, one of the contributions of this thesis is the introduction of a numbering scheme for level representation that allows us to specify how dense or sparse our levels should be on average, in order to constrain the search space of the genetic algorithm and allow processing times to be much more feasible. In order to define our experiments, we need to introduce a numbering scheme which promptly and accurately describes each level. We propose a numbering scheme in the format

Table 5.1: Numbering Scheme and Parameters for the GA Levels.

Level #	**Req. Rows**	**Req. Columns**	**Mut. Rate**	**Blank Rate**
1	**0**	0	0.7	0
2	0	0	0.7	0.3
3	0	0	0.7	0.7
4	9	1	0.7	0
5	9	1	0.7	0.3
6	9	1	0.7	0.7

R-C-B where R denotes the number of required rows per screen space of the level. C represents the amount of required column blocks per column. The two main level types are cluttered, represented as "0-0", which means 0 rows and 0 columns are required; basically it is unrestricted as shown in Figure 5.2, and sparse, represented as "9-1", which means for each row number (i.e. row 0, row 1, to row 9) per column, only 1 block may be filled in as shown in Figure 5.3. Finally, B then explains how many blank blocks are applied to the level, this is symbolized by "0.0", "0.3", and "0.7" for "0%", "30%", and "70%" respectively. This blank rate creates the mutation rate of blocks to be influenced more toward blank blocks by increasing the chance of the mutation to blanks instead of other block types.

Table 5.1 showcases this scheme for each level type. The reason why we need the main definition between level types of cluttered and sparse is to highlight that decluttering the level of unnecessary blocks will allow for more movement within the level and a faster generation time. One of our hypotheses for our experiments later in this thesis is that cluttered levels will just take too long to declutter a level by itself

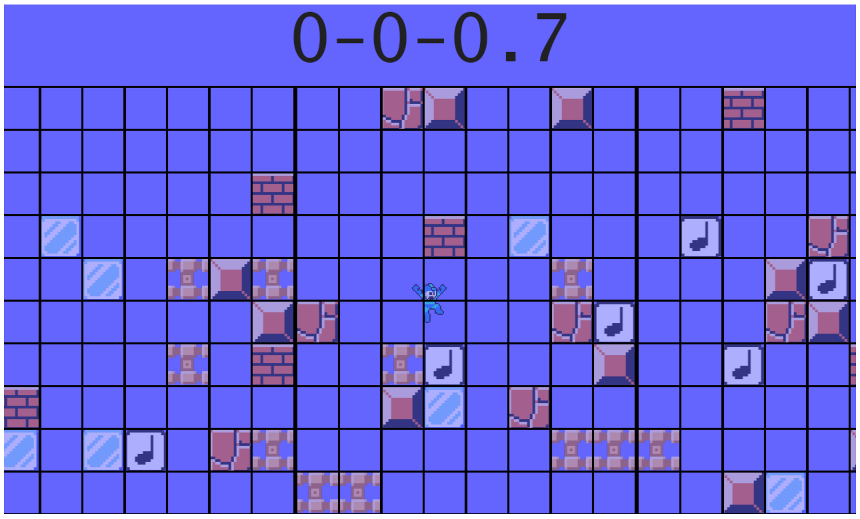


Figure 5.2: A “0-0-0.7” level with a grid overlay. As you can see, multiple blocks are allowed vertically in a single column, for all row numbers. It has a 70% blank rate, introducing many more blanks than its counterparts. It is completely unrestricted.

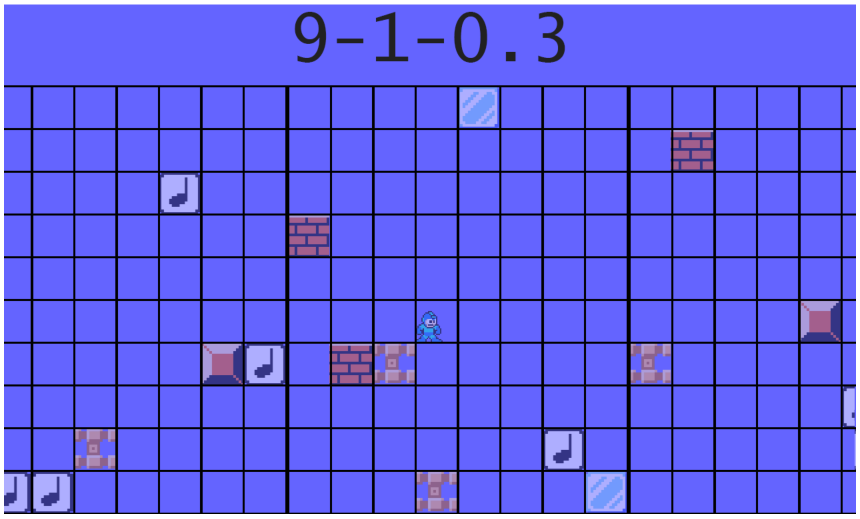


Figure 5.3: A “9-1-0.3” level with a grid overlay. As you can see, only a single block is allowed vertically in a single column, utilizing rows from number 0 to 9. It has 30% blank rate, including less blanks than its counterparts. It is restricted to a single block per column.

and therefore lead to potentially unwanted results.

5.2 Evaluation Metrics

Fitness function evaluations are the central metric used by GAs in determining the next generation of individuals for their population. Therefore, modulating and manipulating the different fitness functions are perfect for steering the GA to our required goal. There are three main fitness functions used in our experiments, Right Movement, Total Distance, and Least Blocks with their priority in the same order. Right Movement and Total Distance are functions that keep track of how far right the player moves and how much the player moves in all directions respectively by increasing the final evaluation for each positive movement.

The most important part of the fitness function heuristic is how much the player has moved to the right, because that is where we want the level to grow. In order for the level to start on the left side of the screen and grow towards the right, the player must be moved in that exact same motion and end up to the furthest right position possible. By adding in the total distance, it incentivizes more verticality as we build the level moving to the right. This improves the evaluation by firstly checking which level made it the furthest horizontally and tie-breaking those scores by the most vertical fluctuations.

After multiple generations, the Least Blocks function will filter out unused blocks from each level by applying a negative value on each block that exists in the map. Each level is loaded into a simulation after mutation and crossover. When simulating the levels, they do not need to be rendered and can therefore be processed at much faster speeds than normal. The levels are evaluated with the fitness functions, the priority initially being how far right the player has moved. In our results, we can examine that in most cases, just moving to the right is easily accomplished, but very dull. We wanted to see more lateral movement in the generated levels, which is when the Total Distance tie-breaker was introduced. This incentivized lateral movement for the GA and helped guide it closer to our overall goal.

5.3 Experiment Levels

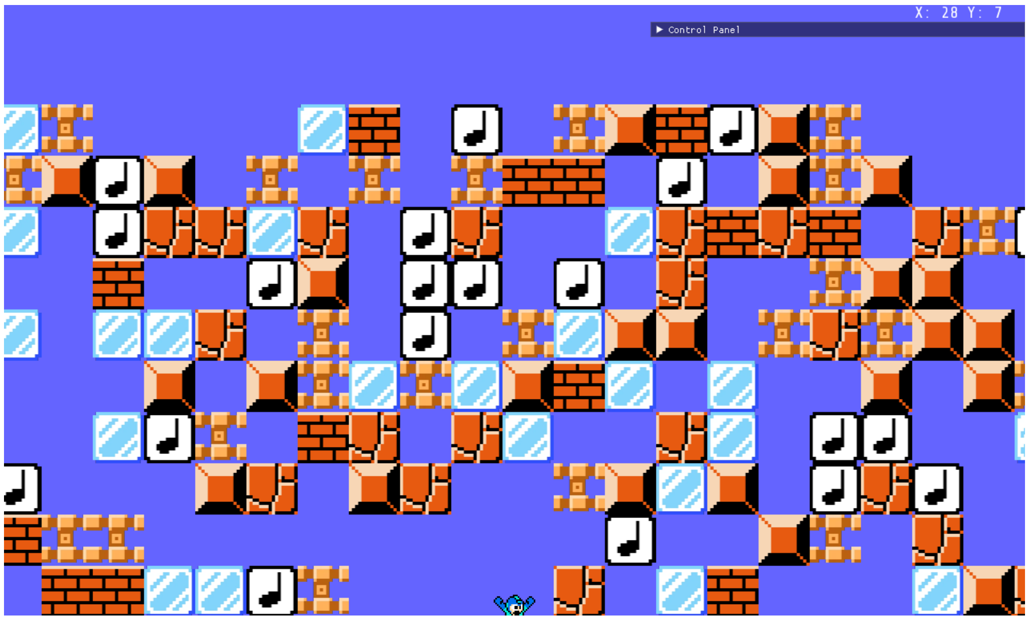


Figure 5.4: A snapshot of "0-0-0.3" a cluttered level with 30% extra blanks. A path was carved through the clutter of blocks in a downward curve which created a hole.

After the evaluation of each level in the population, they must be compared for crossover and elitism. The levels are compared based on their fitness values in priority

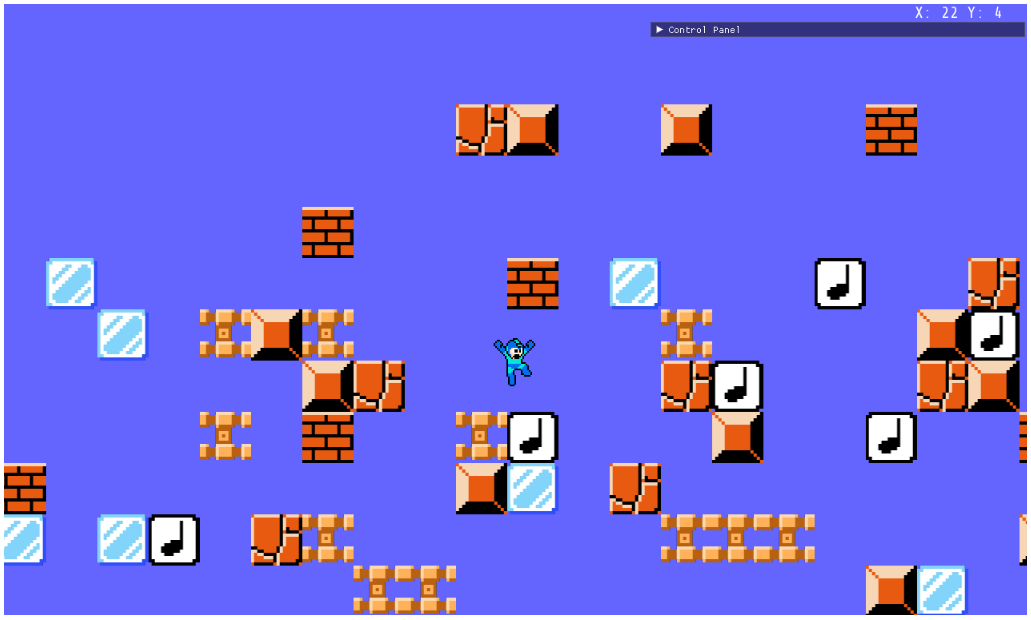


Figure 5.5: A snapshot of “0-0-0.7” a cluttered level with 70% extra blanks. Many new paths open up, which push our character through the level.

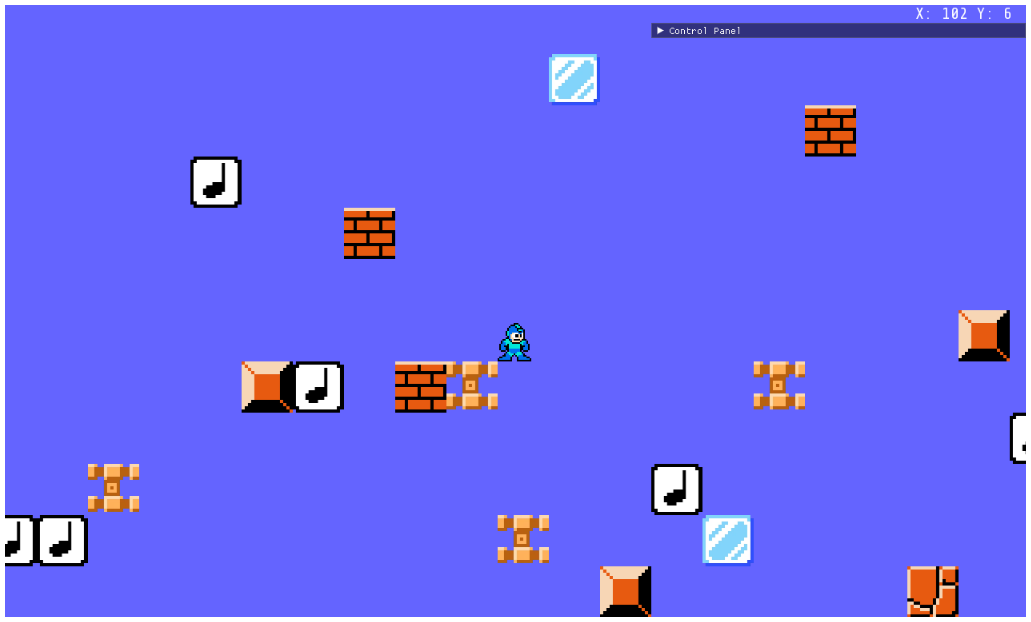


Figure 5.6: A snapshot of “9-1-0” a sparse level with 0% extra blanks. Notice how each column has only 1 block and opens the level up much more.

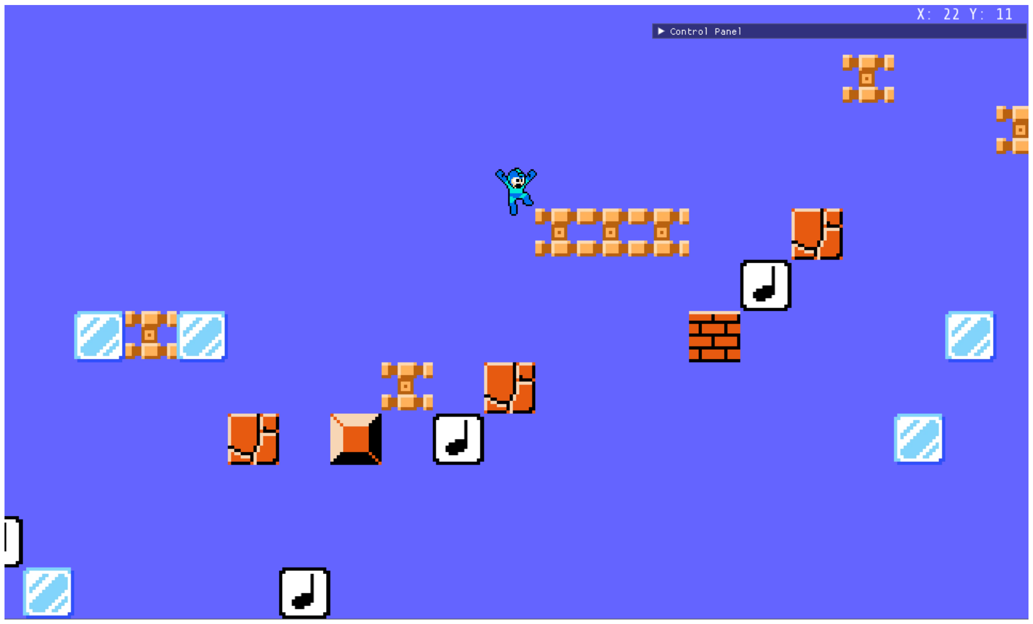


Figure 5.7: A snapshot of “9-1-0.3” a sparse level with 30% extra blanks. The level is opened up much more and has a linear path of progression.

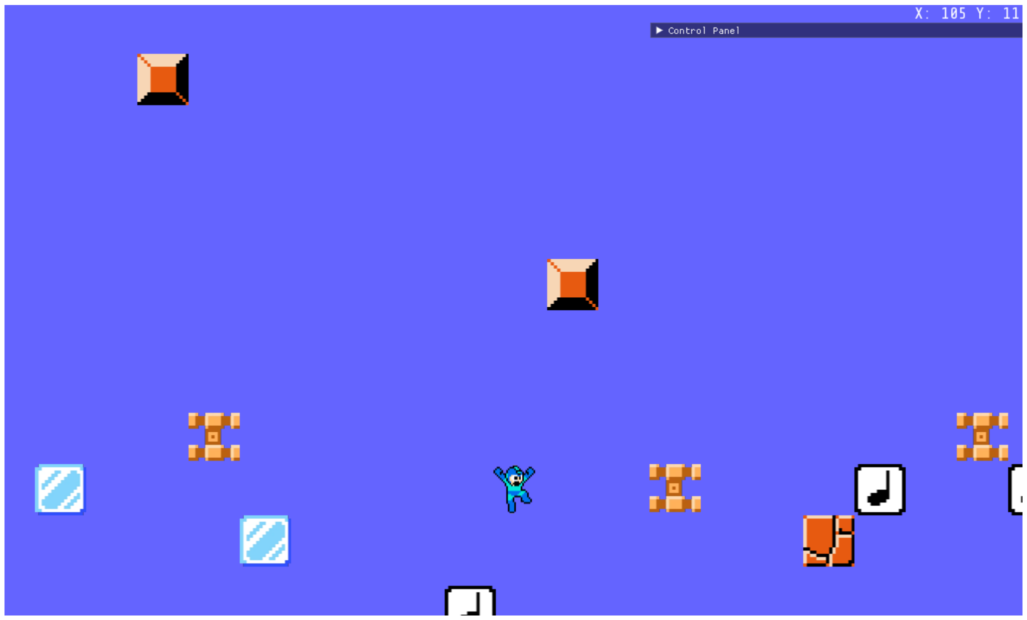


Figure 5.8: A snapshot of “9-1-0.7” a sparse level with 70% extra blanks. The sparse level is even further so with many more blank spaces.

of the order of fitness functions. In evaluating Right Movement, a cluttered level (0-0) consisting unrestricted rows and columns might underperform a sparse level (9-1) which consists of a single block per column. However, in evaluating Total Distance, it is more likely a 0-0 level would get the player stuck in a repetitive movement loop and therefore outperform a 9-1 level.

5.4 Results

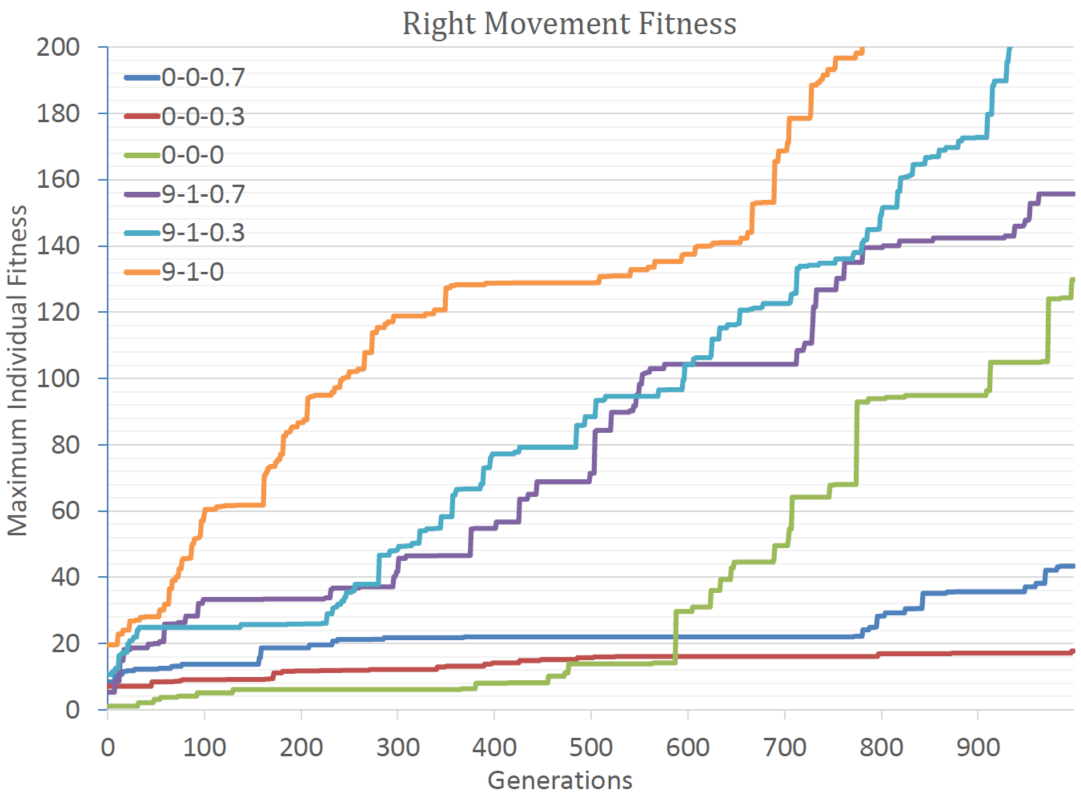


Figure 5.9: Maximum Individual Right Movement Fitness over 1000 generations.

The experiments ran for 1000 generations with a population of 300 individual levels. There is a clear early distinction between 9-1 and 0-0 levels. The 9-1 levels quickly ascend in fitness values and finish as the highest, preforming significantly better than the 0-0 levels. The justification for the significant distinction is the number of blocks that needed to be mutated and influenced. While creating a path through the level, most of the blocks above and below the player are completely untouched and unnecessary, so to change them through mutation or other means is a waste of processing time. This was solved by restricting the level generation to one block per column (the 9-1 levels). Due to the density of the level, the one outlier 0-0-0 (0% extra blanks) was able to get so far was by generating an essentially solid level where the player just slid across the top row, resulting in very uninteresting behavior, which is the antithesis of autoplay levels. These results are highlighted in Figure 5.9, where the 9-1 levels are trending up at a high acceleration towards their maximum fitness and except for the outlier, the 0-0 levels are very slow on the incline.

5.5 Least Blocks

For our experiments, we ran another category using the “least-blocks” tie-breaker. The tie-breaker got rid of much of the excess blocks throughout all of the levels and appears to help the majority of the levels reach a higher fitness value much faster. As seen in Figure 5.10 in comparison to Figure 5.9, the 9-1-0.3 level completed it’s generations at a 20% lower fitness and level 9-1-0.7 finished at a 30% lower fitness. The change in slope, which would be how fast the level was generated, which was at approximately 0.372 Fitness points per Generation, now is at approximately 0.2 Fitness points per Generation. One of the major inconsistencies that we can see is

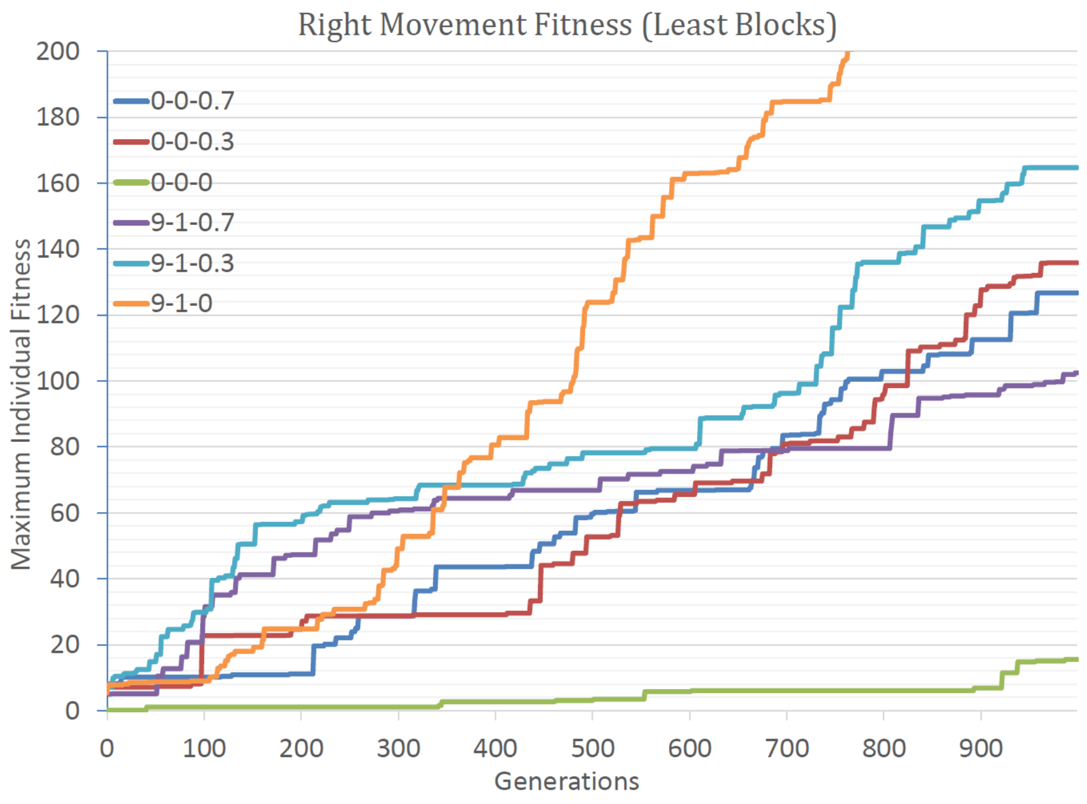
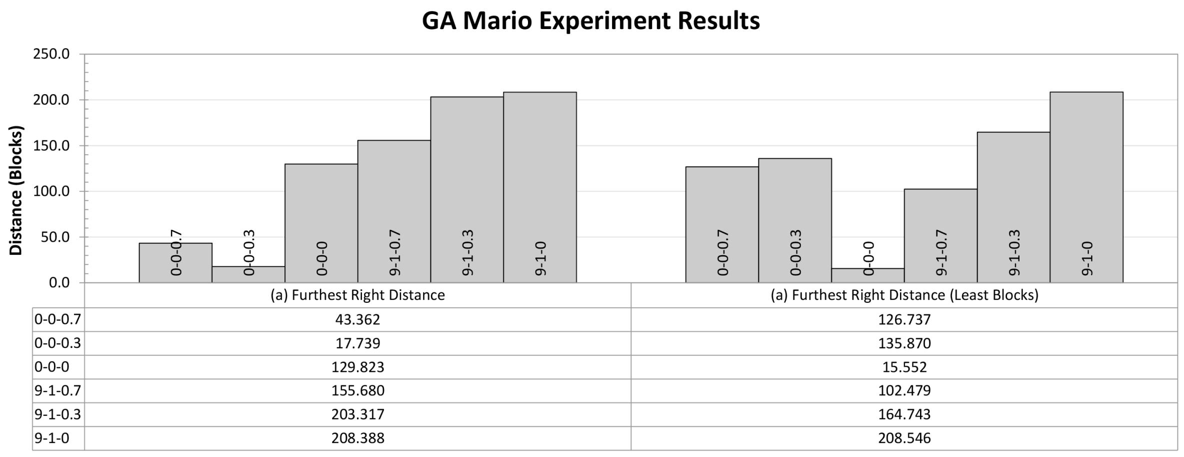


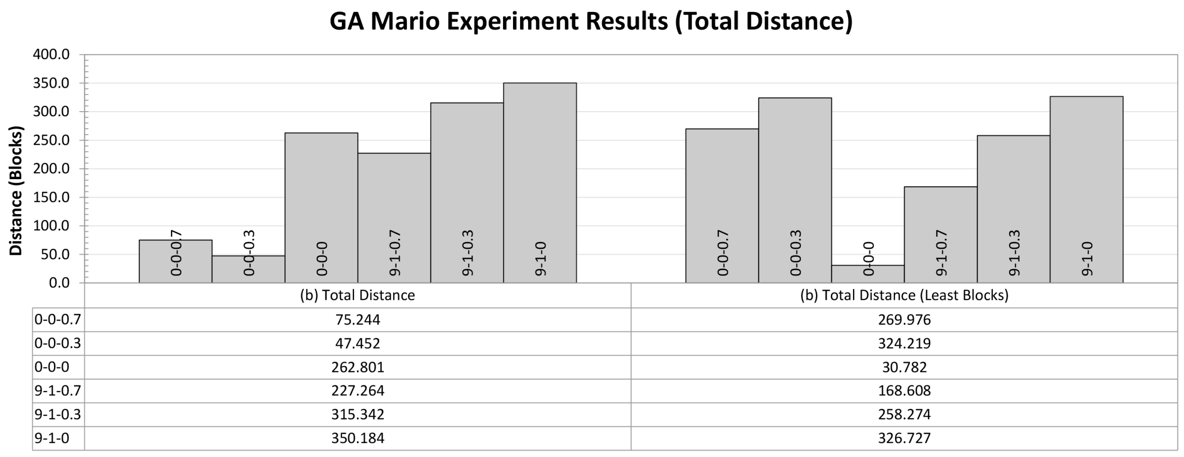
Figure 5.10: Maximum Individual Right Movement Fitness with Least Blocks tie-breaker over 1000 generations.

our outlier from before “0-0-0” is now much slower in the refinement process and it doesn’t even finish anywhere near the other 0-0’s. The other 0-0 levels are improving rapidly and finish as high as the 9-1 levels. The rationale behind these results is for the 9-1 levels, are now too sparse and unable to make sufficient mutations to ramp up their fitness values. The “0-0-0” level should be helped, however, the “least-blocks” tie-breaker creates enough room for the player to be moved but still stuck in the clutter of blocks. The conclusion we can draw from these results, helps us understand that there has to be a median number of blocks which is required for each screen, relying on the overall level and game. For this platformer, it appears



	(a) Furthest Right Distance	(a) Furthest Right Distance (Least Blocks)
0-0-0.7	43.362	126.737
0-0-0.3	17.739	135.870
0-0-0	129.823	15.552
9-1-0.7	155.680	102.479
9-1-0.3	203.317	164.743
9-1-0	208.388	208.546

Figure 5.11: A distance comparison between the Furthest Right fitness and with least blocks enabled for all levels.



	(b) Total Distance	(b) Total Distance (Least Blocks)
0-0-0.7	75.244	269.976
0-0-0.3	47.452	324.219
0-0-0	262.801	30.782
9-1-0.7	227.264	168.608
9-1-0.3	315.342	258.274
9-1-0	350.184	326.727

Figure 5.12: A distance comparison between the Total Distance fitness tiebreaker and with least blocks enabled for all levels.

the 9-1 levels with an extra blank value between 0 and 0.3 perform the best with or without the “least-blocks” tie-breaker. Figure 5.11 and 5.12 show the final results of the experiments from the Furthest Right Distance fitness function and the Total Distance tie-breaker respectively. We can see from these that in any case the 9-1 levels out preform the 0-0 levels with or without the “least-blocks” tie-breaker.

Chapter 6

Conclusions and Future Work

In this thesis we have shown that genetic algorithms are a tool that can be used to successfully generate plausible autoplay levels, that without the player controlling the character, the level can navigate the character through itself in an complex way. By implementing higher mutation rates for the levels, the GAs resulted in higher evaluated levels. Also, by introducing the R-C-B scheme, it allowed us to effectively reduce the search space for the GAs, resulting in much faster generation of higher-evaluation levels. Finally, the blank rate parameter did not appear to help the more sparse 9-1 levels, but it did help the 0-0 level when the Least Blocks tie-breaker was used.

A premise for further research in this field would include experimenting with crossover for different level types. Unlike other GA experiments, levels are much harder to crossover with significant impact. Splicing two levels together horizontally or vertically does not make much sense if they have different paths. An idea would be to splice levels together where their paths collide or at least come very close to overlapping. However, this would have to include tracking the path or all possible paths a player could take through the level. In the future, it would be beneficial to generate something like a multi-screen level in sections. This would allow the GA to focus on the successive screen's blocks without having to mutate and crossover the blocks before it, and could even create better and more complex pathways to lead to new types of autoplay levels.

Bibliography

[1] F. Abdullah, P. Paliyawan, R. Thawonmas, T. Harada, and F. A. Bachtiar. An angry birds level generator with rube goldberg machine mechanisms. In *2019 IEEE Conference on Games (CoG)*, pages 1–8, 2019.

[2] C. Adams and S. Louis. Procedural maze level generation with evolutionary cellular automata. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, 2017.

[3] D.-F. H. Adrian and S.-G. C. A. Luisa. An approach to level design using procedural content generation and difficulty curves. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8, Niagara Falls, ON, Canada, 2013. IEEE.

[4] J. Bycer. *Game Design Deep Dive: Platformers*. CRC press, 2019.

[5] J. Classon and V. Andersson. Procedural generation of levels with controllable difficulty for a platform game using a genetic algorithm. Master's thesis, Linköping University, 2016.

[6] K. Compton and M. Mateas. Procedural level design for platform games. In *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 109–111, 01 2006.

[7] S. Dahlskog and J. Togelius. Patterns as objectives for level generation. In *Second Workshop on Design Patterns in Games*, 05 2013.

[8] R. G. de Pontes and H. M. Gomes. Evolutionary procedural content generation for an endless platform game. In *2020 19th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE, nov 2020.

[9] Z. Dlamini, F. Z. Francies, R. Hull, and R. Marima. Artificial intelligence (ai) and big data in cancer and precision oncology. *Computational and Structural Biotechnology Journal*, 18:2300–2311, 2020.

[10] L. Ferreira, L. Pereira, and C. Toledo. A multi-population genetic algorithm for procedural generation of levels for platform games. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, jul 2014.

[11] M. Green, L. Mugrai, A. Khalifa, and J. Togelius. Mario level generation from mechanics using scene stitching. pages 49–56, 08 2020.

[12] M. Guzdial, N. Liao, J. Chen, S.-Y. Chen, S. Shah, V. Shah, J. Reno, G. Smith, and M. O. Riedl. Friend, collaborator, student, manager: How designof an ai-driven game level editor affects creators. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, may 2019.

[13] T. Härkönen. Advantages and implementation of entity-component-systems. 2019.

[14] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1), feb 2013.

[15] N. C. Hou, N. S. Hong, C. K. On, and J. Teo. Infinite mario bross ai using genetic algorithm. In *2011 IEEE Conference on Sustainable Utilization and Development in Engineering and Technology (STUDENT)*, pages 85–89, 2011.

[16] R. G. Inc. All about rube: A cultural icon. `https://www.rubegoldberg.org/all-about-rube/a-cultural-icon/`, 2023. Accessed: 2023-07-27.

[17] M. R. Johnson. Playful work and laborious play in super mario maker. *Digital Culture & Society*, 5(2):103–120, 2019.

[18] C. Kerr. Super mario bros. wonder hits 4.3 million sales in two weeks. `https://www.gamedeveloper.com/business/super-mario-bros-wonder-hits-4-3-million-sales-in-two-weeks`, Nov. 2023. Accessed: 2023-11-13.

[19] A. Khalifa and M. B. E. Fayek. Automatic puzzle level generation: A general approach using a description language. In *Sixth International Conference on Computational Creativity*, 2015.

[20] A. Khalifa, M. Green, D. Perez Liebana, and J. Togelius. General video game rule generation. pages 170–177, 08 2017.

[21] S. O. Kimbrough, G. J. Koehler, M. Lu, and D. H. Wood. On a feasible–infeasible two-population (FI-2pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch. *European Journal of Operational Research*, 190(2):310–327, oct 2008.

[22] A. Kołodziejek and D. Szajerman. Procedural generation of game levels based on a genetic algorithm and taking the player's experience into account. In *Computer Game Innovations*, pages 59 – 69, 2018.

[23] A. le Clercq and K. Almroth. Comparison of rendering performance between multimedia libraries allegro, sdl and sfml, 2019.

[24] J. Liu, S. Snodgrass, A. Khalifa, S. Risi, G. N. Yannakakis, and J. Togelius. Deep learning for procedural content generation. *Neural Computing and Applications*, 33(1):19–37, Oct. 2020.

[25] A. B. Moghadam and M. K. Rafsanjani. A genetic approach in procedural content generation for platformer games level creation. In *2nd Conference on Swarm Intelligence and Evolutionary Computation*, pages 141–146, Kerman, Iran, 2017. IEEE.

[26] F. Mourato, F. Birra, and M. Próspero dos Santos. The challenge of automatic level generation for platform videogames based on stories and quests. In D. Reidsma, H. Katayose, and A. Nijholt, editors, *Advances in Computer Entertainment*, pages 332–343, Cham, 2013. Springer International Publishing.

[27] F. Mourato, M. P. dos Santos, and F. Birra. Automatic level generation for platform videogames using genetic algorithms. In *Proceedings of the 8th Inter-*

national Conference on Advances in Computer Entertainment Technology - ACE 11. ACM Press, 2011.

[28] M. Muratet and D. Garbarini. Accessibility and serious games: What about entity-component-system software architecture? In *International Conference on Games and Learning Alliance*, pages 3–12. Springer, 2020.

[29] M. J. Nelson and M. Mateas. Towards automated game design. In R. Basili and M. T. Pazienza, editors, *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, pages 626–637, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[30] Nintendo. Hack'n time attack 20th "final battle! guru guru kuppa castle" delivered! `https://www.nintendo.co.jp/switch/baaqa/pc/information/index.html`, 2021. Accessed: 2023-05-19.

[31] OpenAI. Chatgpt. `https://openai.com/chatgpt`, 2023. Accessed: 2023-11-13.

[32] S. PSnodgrass. Probabilistic foundations for procedural level generation. 2014.

[33] retrogamedeconstructionzone. What was the first platformer, space panic or donkey kong? `https://www.retrogamedeconstructionzone.com/2020/07/what-was-first-platformer-space-panic.html`, July 2020. Accessed: 2023-11-13.

[34] N. Shaker, M. Nicolau, G. N. Yannakakis, J. Togelius, and M. O'Neill. Evolving levels for super mario bros using grammatical evolution. In *2012 IEEE Con-*

ference on Computational Intelligence and Games (CIG) IEEE, pages 304–311, Granada, Spain, 2012. IEEE.

[35] S. B. Sirisha and G. Sharma. Towards the era of intelligent machines: Artificial intelligence. 2019.

[36] N. Sorenson and P. Pasquier. Towards a generic framework for automated video game level creation. In *Applications of Evolutionary Computation*, pages 131–140. Springer Berlin Heidelberg, 2010.

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

[38] E. K. Susanto, R. Fachruddin, M. I. Diputra, D. Herumurti, and A. A. Yunanto. Maze generation based on difficulty using genetic algorithm with gene pool. In *2020 International Seminar on Application for Technology of Information and Communication*, pages 554–559, Semarang, Indonesia, 2020. IEEE.

[39] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3:172 – 186, 10 2011.