Generating 2D Platforming Auto-Play Levels with Genetic Algorithms

Shawn Sabraw
Department of Computer Science
Memorial University of Newfoundland
St. John's NL, Canada
sds660@mun.ca

David Churchill
Department of Computer Science
Memorial University of Newfoundland
St. John's NL, Canada
dave.churchill@gmail.com

***Abstract*—Procedural Content Generation (PCG) is not only the key feature of rogue-like games, but it is also used in a vast set of applications, including animation, film, and of course many diverse genres of games. By taking the building blocks of games such as ground, enemies, and the player, PCG can build games, maps, and levels. This paper focuses on an explorative Genetic Algorithm (GA) that is used to generate Mario auto-levels in a custom 2-D game engine. The generated auto-levels are compared through randomization, parameter tweaks and level space restrictions. The results lead to an understanding that each level; either restricted or unrestricted, has a median of required blocks along with the necessary generation time to produce plausible levels. These results are tweaked using a "least-blocks" fitness function in the GA in hopes of cleaning up excess blocks, which provides some positive and negative insight into cluttered versus scarce level generation.**

***Index Terms*—genetic algorithm, procedural content generation, video games, level generation**

I. 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, more so in the game industry. Specifically, character movement/behaviour, music, art, and level design utilize AI to enhance or create content. Level design is crucial for a video game's success as it contains all the mechanics and interactions for the players.

A. Procedural Content Generation

Map or content generation is extremely important in the gaming industry and surrounding fields. There are even competitions to create Procedural Content Generation (PCG) AIs and research into general content generation for levels [1]. The undertaking of creating a AAA (A grading given to high-budget, conspicuous video game titles, usually created by larger game studios.) game takes a considerable amount of time. AI is a promising way of speeding up that development time and also increasing efficiency overall [2].

In this paper, we will generate content, specifically *auto-play* levels for a generic 2D platformer video game engine. An *auto-play* level is an environment where the player does not move the character at all, but with the help of specific game physics and carefully placed level geometry, propels the character through the level usually in some interesting fashion. Videos[1][2] of our automatic level gameplay can be seen by clicking either image in Figure 1. Details of our experiments and 2D game engine will be explained in Section III.

B. Algorithmic Approaches

When creating game levels, it has been found that designers find a lot of value in utilizing AI as a tool [3]. An AI could be programmed to conform to a set of rules or constraints to produce similar levels as a developer would require, or the AI could be allowed to explore, to create never imagined levels or games. A Genetic Algorithm (GA) is an example of an AI system which replicates the process of natural selection to solve the problem of optimizing a problem's solution space. By creating the correct fitness functions for our Genetic Algorithm, we can combine GAs and PCG, yielding a system which automatically generates complex and interesting content.

In this paper, we will first give a brief background of previous work using GAs for PCG, followed by a description of our 2D game engine and its game mechanics. We will then give the details of our methodology, followed by a description of our experiments and their results. Finally, we conclude with some ideas for future research in the area.

II. Background

Several researchers have used GAs for PCG, specifically with a goal of level generation in mind, however, it is typically for a broad set of game levels where the end goal is for a human to play [4]–[6]. It has also been identified that performance and processing complexity are always the most common issues we face in this area [7], [8]. In this section, we will explore some related work to better understand other researchers' objectives and the issues they faced.

A. Algorithms

All through this field 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.

[1]Auto-Play Level Video 1: https://youtu.be/tVn_qHCmImk
[2]Auto-Play Level Video 2: https://youtu.be/-RcKtYsHwVA

Fig. 1. Example GA-generated auto-play levels in our custom 2D game engine. On the left is a relatively cluttered level, while on the right is a more sparse generation. Different levels of sparsity can be adjusted with the GA fitness parameters. In these automatic levels, the player (shown as the MegaMan sprite in the middle) is automatically propelled forward by conveyor blocks, slides on ice blocks, and jumps when it hits a bounce (music note) block. Click on either image to see a YouTube video of the automatic level play in action, demonstrating each of the various block physics.

A very common genetic algorithm type seen throughout the related area of experiments is the Feasible-Infeasible Two-Population (FI-2Pop) [9]. 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). Grammatical Evolution (GE) is another approach used by some researchers in this area [10], [11]. 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. 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.

B. Generating Content

In this field, PCG has been explored with a variety of different approaches. An interesting and popular idea with PCG is the ability to generate content based on difficulty [8], [14], [15]. The question of what defines difficulty in games was explored in these experiments and to generate content around it, had to be simplified. 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.

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 [5], [10], [16]. 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.

In our literature survey, we have yet to find any research pertaining to the topic of generating auto-play levels, which comes with its own set of constraints, such as requiring specific configurations of the environment that automatically propel the player forward.

III. Custom 2D Game Engine

Our experimental environment is a custom 2-D game engine written entirely in C++ and built from scratch, which contains standard 2D platformer physics such as gravity, walking on platforms, and various tiles that interact with the player in interesting ways. The architecture used is an Entity-Component-System (ECS) design, with a modular and efficient design which allows for the easy creation of new levels based on a simple input integer format. The rendering system for the game uses the Simple and Fast Multimedia Library (SFML) library, which is a very fast and efficient rendering library built on top of OpenGL. This combination of a custom game engine and lightweight rendering library yields a very fast custom game engine capable of simulating thousands of game frames per second, a necessary speed feature when using game simulation as a measure of fitness for a GA.

Sample screenshots from the game engine can be seen in Figure 1. The tile entities (blocks) used in generating levels are one of several types (explained later) and given a fixed size (64x64 pixels) and texture. Each block's texture representation is rendered on a grid, constrained to the given size parameters and loaded within a single playable scene. While our game engine is capable of implementing many different game mechanics from many genres of games, for this paper we will limit its behaviour to that of a typical 2D platforming game.

The main entities inside this game scene are called *blocks*. Blocks are simple 64x64 pixels entities which interact with the player when the player collides with them via movement. These blocks and the only geometry used to build levels in this iteration of our game engine. There are five main types of blocks used in the creation of a level: Bricks, Conveyors, Ice, Bounce, and None. When the player collides with the sides or bottom of any block, it will simply fall downward, however, each block has a unique game mechanic when the player lands on it from the top. Bricks have a high amount of friction and are immovable and impassable, the player will typically come

to a stop when they land on top of them. Ice blocks have very little friction and the player will slide along them until they reach the end of the block. Conveyors impart a large amount of speed to the player horizontally to the right, providing the main mode of horizontal travel. Bounce blocks will bounce the player in the opposite direction of the collision, resulting in a large vertical speed if fallen on from above, similar to a trampoline. Finally, wherever there is blank space in a map, this is denoted by a None block and is completely passable and invisible. At all times, the player is affected by gravity and is pulled downward toward the bottom of the screen.

A. Game Physics Simulation

Our custom game engine is capable of running in two main modes, render mode and headless mode. In the render mode, we can observe the game engine simulate the physics of the game world and the player interacting with the player in real-time at any fixed rendering frame rate. While in headless mode, the rendering of the world is disabled and the game physics simulator can run as fast as it is capable, typically in the thousands of simulation frames per second, depending on the number of blocks in the level. This fast simulation speed allows us to feasibly use the game engine as a fitness evaluation function for our GA, which will be described in the next section.

IV. Methodology: Genetic Algorithm

Genetic Algorithms are a type of evolutionary algorithm, a class of algorithms that attempt to solve optimization problems by simulating the natural process of evolution. Starting with an initially random population of candidate solutions (in this case, our levels to be generated), a GA will mimic the process of evolution by assigning fitness values to each individual (level), selecting high-fitness individuals for reproduction to create new offspring levels, and applying random mutations in an attempt to introduce desirable level traits. Typically this process results in new populations, which over time contain higher and higher fitness individuals, leading to better and better candidate solutions. In this paper, our auto-play level fitness function will evaluate how far the player has travelled to the right within a given level. GAs contain many other parameters such as a genetic representation of individuals, population size, reproduction methods, and mutation rates. Each of these will be discussed in the following sections.

A. Individual Level Representation

In our game engine a single screen size of a level is 20 blocks wide and 10 blocks high, yielding a grid of 200 possible places per screen to place blocks to form a level. With 5 possible block types to choose from, this yields 5^{200} or approximately 10^{139} possible permutations of blocks per screen, which is far too many to exhaustively search. Thankfully this is the exact type of optimization problem that GAs are traditionally successful in trying to solve.

Our game engine stores each level as a 2-D array of integer values, with each of the previously described block placements occupying an integer from 0-4 in that array describing which of the 5 types of block to be placed in that location. Using this simple representation, we can apply a trivial transformation to obtain a 1-D array of integers which is to be used as the genotype in our genetic algorithm representation. The transformation from 1-D genotype to 2-D level representation (phenotype) for fitness evaluation is extremely efficient and allows for very quick population evolution in our game engine and GA.

B. Block Placement Parameters

Even though GAs are well suited for this type of problem, in our experiments we often found that they created very dense level geometries with many more blocks than would be typically seen in a human-made game level. This also resulted in rather dull-looking gameplay, typically with the GA just inserting several conveyor blocks that would push the player along in a single horizontal row. To encourage a sparser level design, we introduced numbered parameters that allow for the control of how many blocks appear in our final generated levels. This numbering scheme takes the form A-B, where A is the number of forced None blocks (blanks) in each column, and B is the number of forced non-None blocks in each column. So for example, with each of our columns containing 10 blocks, 9-1 would indicate that there must be 9 None blocks in each column, and exactly 1 non-None block in each column, creating a very sparse level representation with one block in each column. In this case, the GA would essentially just be deciding where in each column this single block would be located, and which type it would be. We can see an example of a 9-1 level on the right-hand side of Figure 1. 0-10 would be a level which is full of non-None blocks. 0-0 on the other hand would indicate no restrictions for each column in the level, allowing the GA to place each type of block.

A final parameter on block placement is an extra number introduced at the end of this numbering scheme, giving us A-B-C, where C is the extra chance that a mutation in the GA will produce a blank instead of a visible block type. Since we have 5 types of blocks, there is only a 20% chance that a blank will be generated at random. We, therefore, introduce this parameter as a way of allowing the GA to produce more or less sparse levels without specifically indicating how many blocks should be in each column. We can see an example of a level produced by 0-0-0.7 on the left-hand side of Figure 1, which has no forced blocks, but a blank rate of 70%. This C parameter can also supersede the A-B restriction, for example, a 9-1-0.5 column would essentially have a 50% chance of being a blank column since the one block contained within would have a 50% chance of being removed by each mutation. We hypothesize that by using parameters which force a more sparse level geometry, we will facilitate higher fitnesses over time by making room for the player to move more freely.

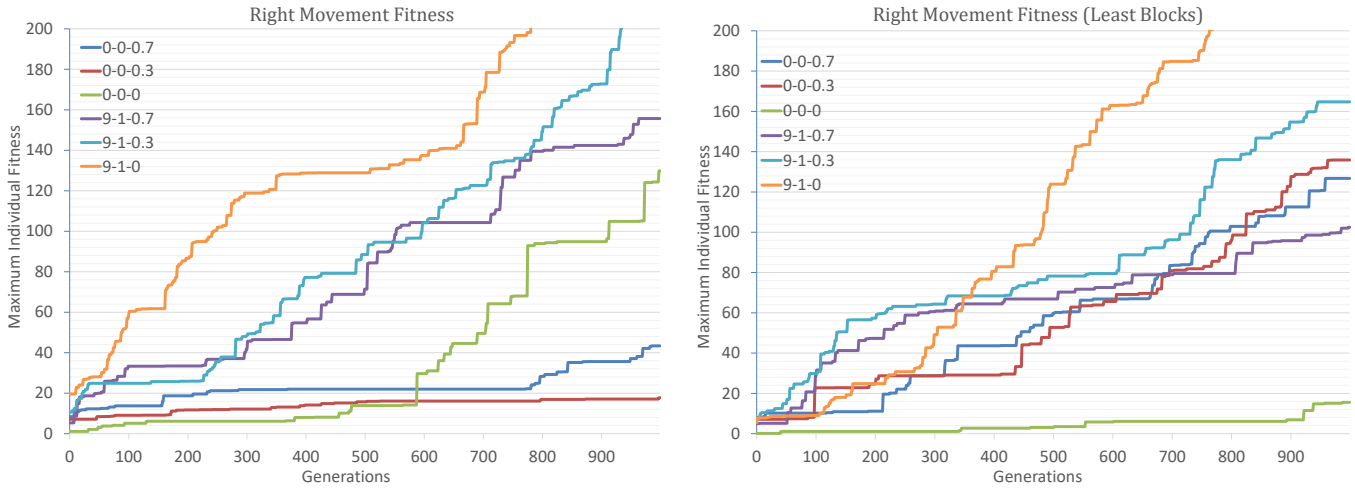

Fig. 2. Experimental results for various level restriction parameters, with GA generations increasing along the x-axis, and population max individual fitness on the y-axis. Shown on the left is for Right Movement Fitness with the Total Movement tie-breaker, and on the right is the Least Blocks fitness tie-breaker.

C. Fitness Evaluation

Fitness functions and tie-breakers are how the GA evaluates each generated individual and selects them for reproduction, making it arguably the most important part of any GA. The main fitness functions guide the algorithm to produce levels with a desired quality, which in our case focuses on moving the player automatically to the right using the game's incorporated blocks and physics. The parameters for each experiment allowed for adjustments of the allotted physical game space, memory, time, and randomness in each generation. The fitness function for this genetic algorithm is simply how far the player makes it to the right, measured in blocks, meaning that a fitness of 200 means that the player was moved 200 columns to the right, or approximately 10 screens of total movement. If two individuals are tied in movement in the right direction, the tie-breaker is total movement including moving left, up, down, and right. So overall, the further right the player makes it, the more viable the level is. In another experiment, we introduce a different tie-breaker: "least-blocks", to help rid the level of clutter and more unnecessary or cluttered blocks.

V. Experiments and Results

The experiments we conducted aim to create auto-play levels that are automatically completed by the level's blocks moving the player through the level as far as possible. Each of the parameters for the computing of the GA are kept the same for each computation such as generations, population, mutation rate, etc. Each experiment then tests different settings for the level restriction parameters and fitness functions, which can be seen in the legend in Figure 2. We spent several days experimenting with various parameters for the GA and finally settled on a good balance between computation time and the quality of results. Each of our experiments ran for 1000 generations with a population of 300 individual levels. Roulette wheel selection was used as the parent selection mechanism with a single midpoint crossover for recombination, with a mutation rate of 70%.

We can see from the results in Figure 2 that there is a clear early distinction in the max fitness over time curves between 9-1 and 0-0 levels. The 9-1 levels quickly ascend in fitness values and finish with the highest overall fitness, whereas in the same amount of time the 0-0 levels are slow to improve, if at all. Our intuition for this significant distinction is twofold: first, forcing sparser levels allows for more free space for the player to be moved more freely by the blocks, and second, having fewer blocks allows the GA to more quickly increase the fitness over time since there is less geometry to optimize. We noticed that in the cases where 0-0 levels achieved a high fitness score, the produced levels were typically quite flat, with the player simply sliding along the top row with several conveyor belts and ice blocks with very little vertical movement, leaving the underlying geometry completely 'wasted' from an aesthetic point of view (not touched by the player). This problem of 'wasted' blocks was partially solved by introducing the Least Blocks fitness tie-breaker, but still contained more wasted blocks overall than the 9-1 forced sparse geometries.

Unexpected results are sometimes undesirable and hopefully avoided when testing a hypothesis. However, they can be very helpful and even crucial in catching misunderstandings or the unpredictable. 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. With this tie-breaker, most 9-1 levels finished at a lower fitness value overall and had a much slower start than our previous experiments. One of the major inconsistencies that we can see is 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 negative results is

for the 9-1 levels, are now too scarce 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 gets stuck in the clutter of blocks. An observation we can draw from these results is that there should be some base number of blocks which is required for each screen in order for the fitness function to improve at a decent rate. If there are too many blocks then the player gets stuck too often, but too many blank columns means that the player cannot traverse the level. For our particular block physics, it appears 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.

VI. Conclusions and Future Work

In this paper, we have shown that Genetic Algorithms can be an effective tool for generating levels for automatic play. We have also shown that by using block placement restriction parameters, we can force the levels to be more or less sparse, which can help the GA to produce better fitness individuals more quickly. If a game level becomes too saturated and cluttered with unnecessary objects, or on the other extreme too scarce and bland, then overall fitness can suffer. Finding the right number of entities in each section of the level is not only crucial for the GA but the level design and game overall. A potential area for further research in this field would include experimenting with the effects of different crossover recombination methods that take level geometry into account. The simple single-point crossover used in our experiments could have probably been improved by instead performing crossover on various rows or columns of the level geometry. A similar approach may be to recombine levels with the player's path in mind, such as where their paths collide or come close to overlapping, allowing for overall more fluid movement in levels to be generated more efficiently.

References

[1] S. PSnodgrass, "Probabilistic foundations for procedural level generation," 2014.

[2] 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.

[3] 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.

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

[5] 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*, vol. 3, no. 3, pp. 229–244, 2011.

[6] 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.

[7] 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*, vol. 3, pp. 172 – 186, 10 2011.

[8] 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*. Semarang, Indonesia: IEEE, 2020, pp. 554–559.

[9] 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*, vol. 190, no. 2, pp. 310–327, oct 2008.

[10] 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*. Kerman, Iran: IEEE, 2017, pp. 141–146.

[11] 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 Conference on Computational Intelligence and Games (CIG) IEEE*. Granada, Spain: IEEE, 2012, pp. 304–311.

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

[13] 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)*, 2011, pp. 85–89.

[14] 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.

[15] 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)*. Niagara Falls, ON, Canada: IEEE, 2013, pp. 1–8.

[16] 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*, 2018, pp. 59 – 69.

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

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

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

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

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

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

[23] T. Härkönen, "Advantages and implementation of entity-component-systems," 2019.

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

[25] R. Hagström, "Frames that matter: The importance of frames per second in games," 2015.

[26] L. Valente, A. Conci, and B. Feijó, "Real time game loop models for single-player computer games," in *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, vol. 89. Citeseer, 2005, p. 99.

[27] F. Mourato, M. P. dos Santos, and F. Birra, "Automatic level generation for platform videogames using genetic algorithms," in *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology - ACE 11*. ACM Press, 2011.

[28] 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.