Evolving Levels for Super Mario Bros Using Grammatical Evolution

Noor Shaker, Miguel Nicolau, Georgios N. Yannakakis, *Member, IEEE*,
Julian Togelius, *Member, IEEE*, and Michael O'Neill

***Abstract*— This paper presents the use of design grammars to evolve playable 2D platform levels through grammatical evolution (GE). Representing levels using design grammars allows simple encoding of important level design constraints, and allows remarkably compact descriptions of large spaces of levels. The expressive range of the GE-based level generator is analyzed and quantitatively compared to other feature-based and the original level generators by means of aesthetic and similarity based measures. The analysis reveals strengths and shortcomings of each generator and provides a general framework for comparing content generated by different generators. The approach presented can be used as an assistive tool by game designers to compare and analyze generators' capabilities within the same game genre.**

I. Introduction

The design of game content is a creative activity that consumes a lot of resources in terms of time and money. Consequently, there has been increasing interest recently in automatic generation of game content with or without human designer interaction. Using these computational techniques, it is not only possible to reduce development cost, but also to generate an endless variation of content that provides a unique experience with every replay. This content could even be adapted to the preferences and skills of individual players. Exploring vast spaces of content can support creativity in several ways, including finding artifacts that would not have been designed by humans due to biases in human creativity and by allowing a designer to swiftly visualize the results of a design idea. It is important, however, to evaluate the content generated by each of these techniques and compare it against content generated by other techniques. Because of the large amount of content that can be generated, it is not feasible to humanly judge the results, and automatic evaluation becomes a necessity.

This paper explores and adopts an approach to Genetic Programming (GP) [1] called Grammatical Evolution (GE) [2] to evolve levels for the platform game *Super Mario Bros*. GE, to the authors' knowledge, has not been exploited for game content creation previously. In addition to the advantages GP provides in terms of producing competitive solutions to those developed by human designers, GE incorporates domain knowledge through its underlying grammatical representation. This allows level designers to maintain greater control of the output and makes it possible to easily generalize to different types of games. The expressivity range of the GE-based levels generator is then analyzed and quantitatively compared to other feature-based level generators that have been used in our previous work [3] and the original level generator for the game. The purpose of the work presented is to provide a framework for analyzing and comparing the expressivity ranges of different content generators.

II. Background

A. Procedural Content Generation

Procedural Content Generation (PCG) is a field that has recently emerged and proven its potential for automatically generating different aspects of game content such as game rulesets [4], [5], maps [6], [7], levels [8], [9], [10], racing tracks [11], [12] or even whole games [13], [14]. PCG can be used both offline, in order to make the game development process more efficient, and online, to allow the generation of endless variations of a game, make it infinitely replayable and adapting its content to the player [15], [16]. An overview of the state of the art can be found in [17], [18].

B. Grammatical Evolution

One of the techniques used to automatically generate content is Evolutionary Computation (EC). Evolutionary Design is one of the areas where EC has demonstrated promising results that are competitive to those created by human experts [19], [20].

GE is the result of combining an evolutionary algorithm with a grammatical representation [2]. GE has been used intensively recently for automatic design [21], [22], [23], a domain where it has been shown to have a number of strengths over more traditional optimization methods.

GE has been adopted in the paper to generate content for *Super Mario Bros* because of the advantages it provides; it maintains a simple way of describing the structure of the levels and it enables the design of aesthetically pleasing levels by exploring a wide space of possibilities.

C. Analyzing Content Generators

In most published papers on PCG, the focus is on the system design and implementation, and little if any emphasis is given to analyzing the space of possible content the generators can produce. While samples of the systems' output are sometimes presented, few studies include meaningful statistical measures of the systems' performance.

NS, GNY and JT are with the Center for Computer Games Research at the IT University of Copenhagen (nosh@itu.dk, yannakakis@itu.dk, juto@itu.dk). MN and MO are with the Natural Computing Research and Applications Group, University College Dublin, Ireland (Miguel.Nicolau@ucd.ie, m.oneill@ucd.ie).

978-1-4673-1194-6/12/$31.00 ©2012 IEEE

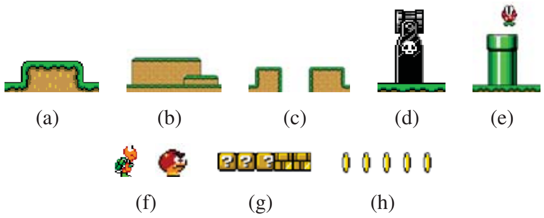


Fig. 1. The geometric representation of the chunks used; a flat platform (a), hills (b), a gap (c), a cannon (d), a tube (e), enemies (f), boxes (g) and coins (h).

Smith et al. [24] suggested a framework for analyzing the expressivity range of a level generator by defining a set of description metrics, collecting a large number of representative samples of the generator's capabilities, visualizing the generative space, and finally analyzing the impact of the generator's parameters on the generator's expressivity.

The work presented in this paper adopts this framework for analyzing the expressivity range of the developed generator and extends it through: (1) defining more informative aesthetic measures of the generators' expressivity and (2) applying these measures to analyze and compare the expressivity ranges of three level generators of the same game.

III. Testbed Platform Game

The testbed platform game used for our study is a modified version of Markus "Notch" Persson's *Infinite Mario Bros* (IMB). Several features make Super Mario Bros particularly interesting from PCG perspective. The most important of these is the potentially very rich environment representation.

For more details about the game and its use as a benchmark for research, the reader may refer to [25].

IV. Level Representation

The internal representation of the levels in Infinite Mario Bros is a two-dimensional array of objects, such as brick blocks, coins and enemies. In "small" state, Mario is one block wide and one blocks high. In the work presented in this paper we construct short levels, only 100 blocks wide, which take roughly 30 seconds to play. This is in order to be able to compare the generated levels with the ones generated in our previous work [26].

The levels can also be represented as a set of chunks. The list of chunks that has been considered in this work includes platforms, gaps, tubes, cannons, boxes, coins, and enemies. Each of these chunks has a distinguishable geometry and properties. Fig. 1 presents the different chunks that collectively constitute a level. In order to allow more variations in the design, we distinguish between two types of platforms; *obstruct-platforms* which block the path and enforce the player to perform a jump action (Fig. 1.(a)), and *hills* that give the player the option to either pass through or jump over them (Fig. 1.(b)).

We assume that the level initially contains a flat platform that spans the whole x-axis. This assumption ensures that all chunks in the resulted design will be connected and explains the need of defining gaps as one of the chunks.

V. GE-based Level Generator

GE is a grammar-based form of GP that specifies the syntax of possible solutions through a context-free grammar, which is then used to map integer strings to syntactically correct solutions. Those integer strings can therefore be created by any search algorithm.

GE employs a genotype-to-phenotype mapping process: the population of the evolutionary algorithm consists of variable-length integer vectors. Each vector is used to choose production rules from a grammar, which creates a phenotypic program, syntactically correct for the problem domain. Finally, this program is evaluated, and its fitness returned to the evolutionary algorithm.

A. Design Grammar

The process in which the level is constructed is represented in the input grammar that GE uses in the construction of a solution (in this case a level design). Several methods for specifying the design grammar have been discussed during the development process, however, due to the context-free nature of the grammar used by GE and since we wanted to keep the grammar as simple as possible to ease designer interaction with the system; the solution proposed, inspired by the work of Morel et al. [27], is to add a chunk to the 2D level array regardless of the positioning of the other chunks. With this solution, however, arises a number of conflicts in level design that should be resolved. Section V-B discusses this issue in details.

The early version of the grammar that has been designed is presented in Fig. 2. The level is constructed by placing a number of chunks each assigned with two or more properties, the x and y parameters specify the coordinates of the chunk starting point position in the 2D level array and are limited to the ranges [5..95] and [3..5], respectively. The first and last five blocks in the x dimension are reserved for the starting platform and the ending gate, while the y values have been constrained in a way that insures playability (the existence of a path from the start to the end position) and that all items are placed in areas reachable by *Mario* by performing jumps. The w_g parameter specifies the width of gaps, w_b defines the number of boxes, w_e determines the number of enemies, w_c defines the number of coins, and h indicates the height of the flower tubes and cannons.

An example phenotype that results from the grammar in Fig. 2 can be $hill(10,4,4)platform(74,3,4)tube(62,4,3)$. Note that the genotype to phenotype mapping is a deterministic process guided by the grammar specified. This also includes the assignment of the parameters for each chunk since the parameters are also part of the grammar. Note also that because of the context-free nature of the grammar, the chunks generated in the phenotype are not necessarily ordered in x or y dimensions.

An example of a resulting level is depicted in Fig. 3. Visualizing samples of the outputs and thoroughly examining the design grammar reveal limitations in the design exposed by the grammar. The definition of gaps, tubes, and cannons

```
<chunks> ::= <chunk> |<chunk> <chunks>
<chunk> ::= gap(<x>,<y>,<w_g>)
  | platform(<x>,<y>,<w>)
  | hill(<x>,<y>,<w>)
  | cannon_hill(<x>,<y>,<h>)
  | tube_hill(<x>,<y>,<h>)
  | coin(<x>,<y>,<w_c>)
  | cannon(<x>,<y>,<h>)
  | tube(<x>,<y>,<h>)
  | boxes(<x>,<y>,<w_b>)
  | enemy(<x>,<y>,<w_e>)
<x> :: = [5..95] <y> ::= [3..5]
<w_g> ::= [2..5]  <w> ::= [3..15]
<h> ::= [2..3]    <w_b> ::= [2..7]
<w_e> ::= [1..7]
```

Fig. 2. The first version of the grammar employed to specify the design of the level.

in the grammar only specifies the width of the gaps and the height of the tubes and cannons. As a result of this definition, each one of these elements will be generated with equal-width platform surrounding it (Fig. 3). According to game designers, the width of the platform before and after these elements plays an important role in the gameplay experience. For example, the width of platform before a gap affect the difficulty of the game since speeding up is sometimes required to launch a wide jump to overcome a wide gap. Therefore, two parameters have been introduced specifying the width of the platform before w_{before} and after w_{after} each of these chunks. The addition of these parameters also accommodates for more control and variation in the design.

The other limitations concern the generation of boxes and enemies. The definition proposed in the grammar results in generation of groups of only rocks or only blocks. In IMB boxes are usually presented as groups of rocks and blocks collectively, each of which may contain a coin, a powerup or it can be empty. For this to be allowed a refinement of the grammar has been made as can be seen in the second version of the grammar (Fig. 4). The same argument holds for enemies and a similar solution has been adopted to allow for different types of enemies to be introduced.

The final limitation relates to the placement of enemies; in the first version of the grammar, enemies are spawned in groups, and to make sure enemies are always placed on a platform, whenever an enemy is generated, an associate platform on which the enemy is placed is created. This produced groups of enemies of the same type to be always placed on a separate platform (Fig. 3). To support more variabilities, the grammar has been improved to allow enemies of different types to be placed on any generated platform (around gaps, tubes, etc.). This has been accomplished by (1) constructing the physical structure of the level, (2) calculating the possible positions on which an enemy can be placed (this includes all positions where a platform has been generated) and (3) placing each generated enemy in one of the possible positions. The place on which the enemy is placed has also been defined as a parameter in the grammar to maintain the deterministic genotype to phenotype mapping. The final version of the grammar can be seen in Fig. 4.

```
<level> ::= <chunks>  <enemy>
<chunks> ::= <chunk> |<chunk> <chunks>
<chunk> ::= gap(<x>,<y>, <w_g>,<w_before>,<w_after>)
  | platform(<x>,<y>,<w>)
  | hill(<x>,<y>,<w>)
  | cannon_hill(<x>,<y>,<h>,<w_before>,<w_after>)
  | tube_hill(<x>,<y>,<h>,<w_before>,<w_after>)
  | coin(<x>,<y>,<w_c>)
  | cannon(<x>,<y>,<h>,<w_before>,<w_after>)
  | tube(<x>,<y>,<h>,<w_before>,<w_after>)
  | <boxes>

<boxes> ::=  <box_type> (<x>,<y>)^2 | ...
  | <box_type> (<x>,<y>)^6

<box_type> ::= block_coin | block_powerup
  | rock_coin  | rock_empty

<enemy> ::= (koopa | goompa)(<x>) ^2 | ...
  | (koopa | goompa)(<x>) ^10
<x> :: = [5..95] <y> ::= [3..5]
```

Fig. 4. A simplified version of the final grammar employed to specify the design of the level. The superscripts (2, 6 and 10) are shortcuts for the number of repetition.

B. Conflict Resolution

There are a number of inherent conflicts in the design approach followed. According to the design approach, each chunk generated can be assigned any x and y values from the ranges [5..95] and [3..5], respectively, depending on the genotype. This means that it is very likely that there will be an overlap between the coordinates of the generated chunks. For example: $hill(65,4,5)$ $hill(25,4,4)$ $cannon\_hill(67,4,4,4,3)$ $coin(22,4,6)$ $platform(61,4,4)$ is a phenotype that has been generated by the grammar and contains a number of conflicts; for example, $hill(65,4,5)$ and $cannon\_hill(67,4,4,4,3)$ have been assigned the same y value, and an overlapping x values.

To resolve these conflicts, a priority value has been defined and assigned to each of the chunks. Whenever two chunks overlap, the one with the higher priority value is maintained and the other is removed. Nevertheless, to allow more diversity, some of the chunks are allowed to overlap such as hills of different height (Fig. 1. 1(b)), and coins or boxes with hills (hills here refer to all types of hills; cannon-hills, tube-hills and flat hills).

C. Implementation and Experimental Setup

The existing GEVA software [28] has been used as a core to implement the needed functionalities. The experimental parameters used are the following: 1000 runs each ran for 10 generations with a population size of 100 individuals, the ramped half-and-half initialization method. The maximum derivation tree depth was set at 100, tournament selection of size 2, int-flip mutation with probability 0.1, one-point crossover with probability 0.7, and 3 maximum wraps were allowed. Since this is a preliminary experiment on level

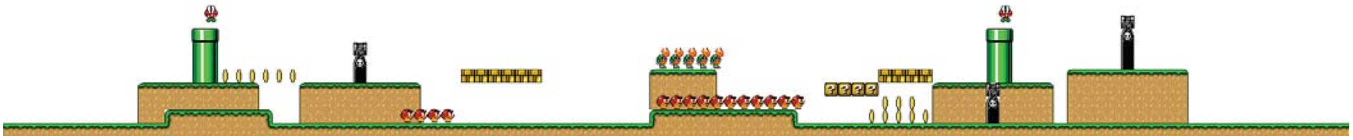


Fig. 3. An example level generated by the first version of the grammar. The design illustrates a number of limitations in the grammar such as the placement of enemies and the generation of boxes.

construction using GE, the main objective of the fitness function is to create levels with an acceptable number of chunks. Thus, the fitness function used is a weighted sum of two normalized measures; the first one, f_p, is the difference between the number of chunks placed in the level and a predefined threshold that specifies the maximum number of chunks that can be placed. The second, f_c, is the number of different conflicting chunks found in the design. Apparently, the two fitness functions partially conflict since optimizing f_p by placing more chunks implicitly increases the chance of creating overlapping chunks (f_c).

VI. Other Generators

In order to test the generator's capabilities and expressive range, we investigate two other generators for the same game and compare the content generated by the GE-generator with those generated by the other generators.

A. Notch Level Generator

The Notch level generator is the one that comes originally with the game. It constructs levels by incrementally placing different chunks according to certain heuristics. The level generation can be parameterized by defining the level of difficulty which affects the number of generated gaps, enemies and the type of enemies. For the experiments presented in the paper and for comparison purposes, the difficulty of all generated levels has been set to 2.

B. Parameterized Level Generator

In our previous studies [3], [26] we conducted experiments based on a heavily modified version of the Notch level generator. The level generator of the game has been modified to generate content according to the six content features; the number of gaps in the level, G; the average width of gaps, $\bar{G}_w$; the number of enemies, E; enemies placement, $E_p$which has been determined by three probabilities which sum to one: on or under a set of horizontal blocks, P_x; within a close distance to the edge of a gap, P_g and randomly placed on a flat space on the ground, P_r; the number of powerups, N_w; and the number of boxes, B, which specify the number of the different types of boxes that exist.

The generator is allowed to randomly generate the other aspects of game content such as the number of cannon and flower tubes, the number of coins, the differences in platform height, and the number of hills.

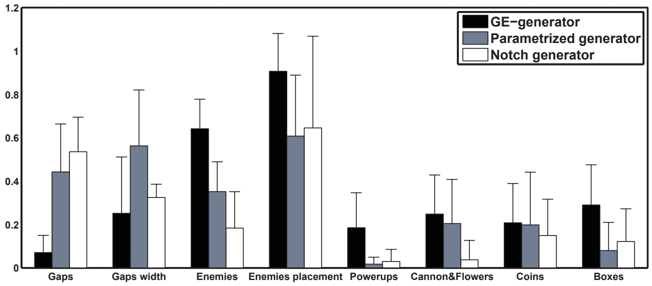


Fig. 5. Average and standard deviation values of eight statistical features that have been extracted from all generated levels across all generators.

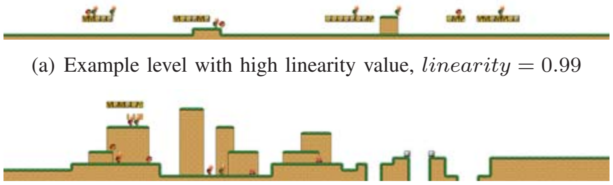
VII. Expressivity Analysis

To analyze the design and the expressivity of the generators, several statistics have been extracted from the 1000 levels generated by the different generators. Fig. 5 presents a comparison between the average values of eight key statistical features that have been extracted from the data of all levels across each generator: numbers of coins, boxes, powerups, enemies and gaps, the average gap width, as well as the enemy placements which measure how enemies are placed in the level; around gaps, around boxes or randomly scattered. All feature values are normalized to the range [0,1] using max-min normalization.

As can be seen from Fig. 5, the GE generator appears to generate more features for all aspects of game content except for the number and width of gaps. This might be the result of defining a rather high threshold for the total number of chunks that can be placed in the level when designing the fitness function. The generator appears to be biased towards generating a low number of gaps, a large number of enemies and boxes and placing enemies around boxes. The standard deviations are roughly comparable, though the GE generator appears to have less variation in enemy numbers and placement.

The Notch generator and the parameterized generator, on the other hand, appear to generate around the same number of boxes, coins, powerups, and gaps. The main differences between these two generators are in the number of enemies created and the width of gaps. A larger number of enemies (including flower-tubes and cannons) and wider gaps have been generated in the parameterized levels compared to the ones generated by the random generator.

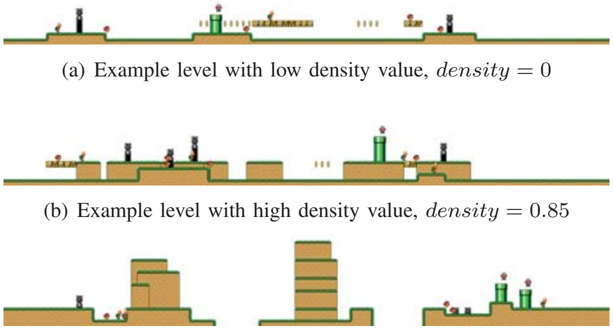
The statistical analysis draws a picture of the generators'



(a) Example level with high linearity value, $linearity = 0.99$

(b) Example level with low linearity value, $linearity = 0$

Fig. 6. Two example levels with different linearity values.



(a) Example level with low density value, $density = 0$

(b) Example level with high density value, $density = 0.85$

(c) Example level with high density value, $density = 1$

Fig. 7. Three example levels with different density values.

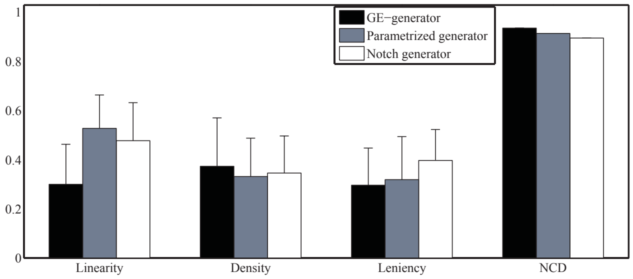


Fig. 8. The average and standard deviation values for the expressivity measures for all generators.

capabilities but a more in-depth analysis is required, if we are to examine the space of possibilities the generators' output cover and the density of the levels generated along different aspects of expressivity measures. For this reason, we have defined several more complex level design metrics and employed them to evaluate the generated levels. In the following sections, we describe these measures and the results of applying them to examine the qualities of the generators' output. Two of these measures are similar to the ones proposed by Smith et al. [24]. Since the expressivity of some of the generators have constrained along some aspect of content generation (such as the parameterized generator), we tried to define expressivity measures that allow us to compare the generators' outputs along dimensions orthogonal to the ones directly controlled by the parameters.

A. Linearity

Linearity in IMB is affected by the existence of different types of hills along the level, as well as the differences in the platform height. A highly non-linear level is the one with frequent changes in the platform height or the one containing hills scattered around. A level with such characteristics requires the player to perform more jumps, gives him the possibility to reach higher places and/or presents more than one possible path to reach the end of the level. Two levels of very high and low linearity values are depicted in Fig. 6.

We follow the approach proposed in [24] to measure linearity by calculating the linear regression for each level. This has been calculated by traversing the level from left to right and accumulating the values of the absolute differences between the center-point of the highest platform or hill and the corresponding point on a predefined line. The results are then uniformly normalized to [0,1].

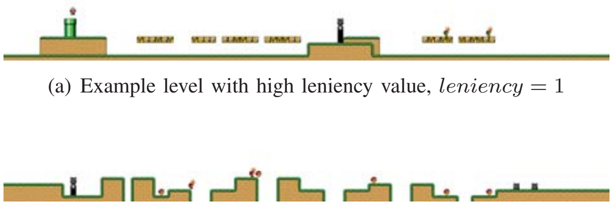
Fig. 8 presents the average values of the linearity measure obtained from ranking the levels generated by all generators. The results show that the levels generated by the GE generator are, on average, less linear than the ones generated by Notch generator, which are in turn less linear than the ones generated by the parametrized generator.

B. Density

In IMB, hills of different height can be stacked on top of each other allowing *Mario* to reach higher places and introducing new patterns in the level design. We defined a density measure that ranks the levels according to the number of density chunks occurrences. The density is calculated by assigning a density value to each point along the width of the level according to the number of platform stacked at that point. The density of the level is the normalization of the sum of these values. Fig. 7 presents three levels having extreme density values. Note that since normalization has been performed based on the density values obtained from all the levels generated, Fig. 7.(c) is assigned a density value equals to 1 because it has the maximum density value of all the levels generated.

The density measure taken together with the linearity measure give an indication of the distribution of hills along the level. A level with a high density value can either contain hills scattered along the level or they can be stacked in one or more segments. Fig. 7.(b) and Fig. 7.(c) present two example levels with high density, yet having a very different distribution of hills. The linearity values assigned for these two levels, however, are 0.4 and 0.9 for the former and latter level, respectively, indicating a wide range of differences in the structure of the levels. The level with hills compressed in a small segment is assigned with a higher linearity value than the one with hills spread along the level since linearity takes into account only the highest platform at each position.

As can be seen from Fig. 8, the GE-generator constructs levels with higher density than the parameterized and Notch generator. It's also worth noting that all generators construct levels with low average density (less than 0.5).



(a) Example level with high leniency value, $leniency = 1$

(b) Example level with low leniency value, $leniency = 0$

Fig. 9. Two example levels of different leniency values.

C. Leniency

We adopt a leniency measure, similar to the one proposed in [24], to account for how tolerant the level is in terms of how easy it is for the player to complete the level. We assign a lenience value for different chunks as follows:

- Gaps: -0.5
- Average gap width: -1
- enemies (goompas and koopas): -1
- Cannon and flower tubes: -0.5
- Powerups (mushrooms and flowers which make Mario grow Big or turn him turn into Fire mode): +1

Different types of enemies are given different lenience values according to their characteristics. The leniency of the level is the weighted sum of the leniency of each of the chunks presented in the level. The leniency values for all generated levels are normalized to [0,1]. Two levels with different leniency values are presented in Fig. 9. Note that despite the fact that the level presented in Fig 9.(a) contains four enemies, this level have been assigned a very high leniency value because 85% of the boxes presented in that level hide powerups.

The average leniency values obtained for the generators are presented in Fig. 8. Notch generator constructs the most lenient levels followed by the parametrized generator, while the levels generated by the GE-generator are the least lenient.

D. Compression Distance

In order to measure the overall structural similarity between the outputs of each generator, we converted all levels into sequences of numbers representing the existence of different types of content items as well as changes in the level geometry (see [3] for more details and examples). The following content events as well as their possible combination have been considered when converting the levels into sequences:

- Increase/decrease in platform height
- The existence/non-existence of enemies and items (coins or boxes)
- The beginning/ending of a gap

The diversity of the resulting levels sequences for each generator is measured using the normalized compression distance (NCD) measure [29]. The results of applying this measure on each pair of the content sequences for each generator showed a high dissimilarity between the sequences; NCD was found to be higher than 0.6 in 93%, 91% and 89% of the cases for the levels generated by the GE-generator, the parameterized generator and the Notch generator, respectively (Fig. 8).

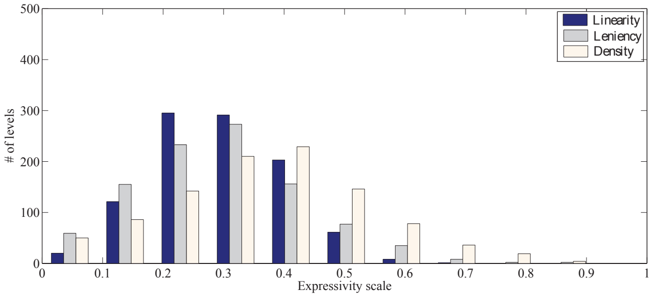


Fig. 10. The histograms of the linearity, leniency and density measures for the 1000 levels generated by the GE-generator.

E. Histogram comparison

The expressive range of a generator can be analyzed by plotting the histogram that illustrates the distribution of the generated levels along the expressivity measure. The 1000 levels generated by each generator have been processed and ranked by the linearity, leniency and density measures. Fig. 10, 11 and 12 present the expressivity ranges obtained for the GE-generator, the parameterized generator and Notch generator, respectively.

Different distributions have been obtained for each measure across the generators. The GE-generator, as can be seen from Fig. 10 and the parameterized generators (Fig. 11) appear to be slightly biased according to the linearity measure; while the GE-generator constructs levels that are slightly non-linear, the parametrized generator appears to be generating more linear levels. On the other hand, both generators appear to be biased towards generating non lenient levels. It is interesting to note, however, that the Notch generator (Fig. 12) constructs levels with a distribution for the linearity that approximates the normal distribution around 0.5. This generator appears to be very biased towards generating averagely lenient levels (more than 80% of the levels have a lenience value between 0.3 and 0.5). Very small percentage of the levels generated by all generators fall in the extreme ranges of the expressivity measures.

We anticipated the bias towards generating linear levels by the parametrized generator since in IMB levels, the flat platform is the basic element when designing the levels and the addition of hills and the changes in the height are supplementary requirements in order to allow richer design diversity and gameplay experience. Also, this generator has been designed to generate levels according to a predefined set of features that resulted in highly condensed levels, leaving a few number of segments where a hill can be generated.

Unsurprisingly, the parametrized and Notch generators appear to generate similar levels according to linearity compared to the levels generated by the GE-generator. This was anticipated since the parametrized generator is a modified

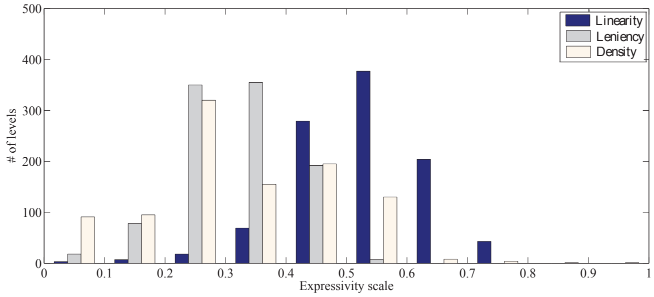


Fig. 11. The histograms of the linearity, leniency and density measures for the 1000 levels generated by the parameterized generator.

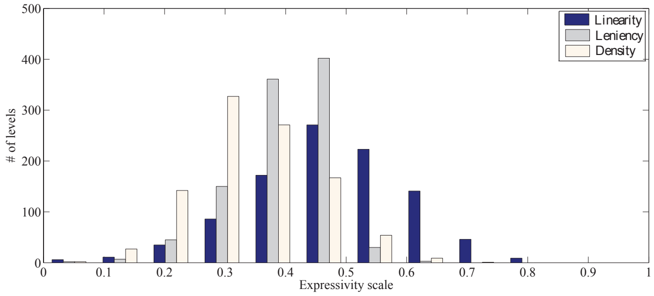


Fig. 12. The histograms of the linearity, leniency and density measures for the 1000 levels generated by Notch generator.

version of Notch generator. However, the shift in the center of the distribution of the levels generated by the GE-generator along the linearity dimension, compared to the ones obtained from the parameterized and Notch generator, can be explained by the different methodology used by this generator when constructing the levels.

The level distribution along the density dimension varies among the three generators with all of them generating low to average density levels. The shift in the density values obtained from the levels generated by the GE-generator can be explained by a design choice which is implicitly imposed by the design grammar; the range of possible height for each chunk generated has been constrained in a way that the chunk will be reachable by *Mario*.

The Notch generator appears to cover a narrower expressivity range for all measures than the other generators. None of the generators was able to express a uniform distribution of levels along the expressivity measures defined. Nevertheless, it is not clear whether this is desirable and necessitates covering a wider range of player preferences.

The statistical analysis of these measures across all levels generated by each generator (Table I) showed strong positive correlations between linearity and leniency for the levels generated by all generators, while strong negative correlations have been obtained between linearity and density and leniency and density.

The positive correlation between linearity and leniency can be explained by the interconnection between the content elements involved when measuring these scores. The presence of gaps and enemies (cannon and flower tubes) which mostly implies changes in the platform height lead to generating levels with low linear and lenient score. The negative correlation between linearity and density, on the other hand, points out a bias in the generators towards generating levels with hills spread along them rather than stacked on top of each other.

VIII. Conclusions and Future Work

This paper presents the use of Grammatical Evolution to evolve the design of *Super Mario Bros* levels. The structure of the levels has been defined in a grammar that GE uses to evolve levels. The paper demonstrates the process followed to implement the GE level generator. A number of expressivity measures have been defined to test the generator's capabilities and the space of content the generator's output cover. A framework for comparing content generated by different generators has been presented by employing two other level generators for the same game with different generation methods. The expressivity range of each generator has been analyzed and quantitatively compared to the other generators by plotting the histograms of 1000 levels generated by each generator across the expressivity scales defined. The results obtained showed different characteristics of each generator and a wide variety in the space of content each generator covers. The approach proposed can be potentially used by game designers to test and compare different generators within the same game genre.

Future work on automatic level design using GE includes incorporating player experience in the design process in a closed loop manner. A model of player experience can be used as a fitness function in the evolutionary process to rank the generated content. The content evolution can be guided by the fitness function towards generating content that maximizes specific player experience according to player playing style. Personalizing the design grammar is another interesting approach towards tailoring content generation to specific player needs and characteristics.

The expressivity analysis highlights limitations in the expressivity of each generator. For example, the design grammar in the GE-generator is unable to generate levels with high density due to the height constraint defined in the grammar forcing the generated chunks to be placed within a predefined height limit to ensure playability. One possible solution is to define a constraint-free grammar and play-test the generated levels to check for the playability. This can be done automatically by exploiting the use of AI agents that pass through the levels and check for possible path from the start to the end, and/or check whether all chunks generated are reachable. Another solution is to adopt context-sensitive grammar such as attribute grammars to control the parameter values of the solutions as they are being generated during the mapping process [30].

Another future direction includes defining more in-depth expressivity measure along which content quality can be analyzed and compared. The measures presented in the paper provide a mean to compare content but covering a

TABLE I

TESTING FOR CORRELATION BETWEEN THE OBTAINED SCORES FOR EACH MEASURE ACCROSS THE THREE GENERATORS. THE SIGNIFICANT DIFFERENCES ($p - value < 0.01$) ARE PRESENTED IN BOLD. THE SIGN OF THE CORRELATION IS PRESENTED IN PARENTHESES.

	GE-generator		Parametrized generator		Notch generator	
	$Leniency$	$Density$	$Leniency$	$Density$	$Leniency$	$Density$
$Linearity$	$\mathbf{2.86 * 10^{-51}}$	$(-)\mathbf{2.29 * 10^{-155}}$	$\mathbf{19.85 * 10^{-6}}$	$(-)\mathbf{2.99 * 10^{-21}}$	$\mathbf{1.52 * 10^{-29}}$	$(-)\mathbf{3.61 * 10^{-20}}$
$Leniency$		$(-)\mathbf{6.15 * 10^{-23}}$		$(-)\mathbf{4.88 * 10^{-11}}$		$(-)\mathbf{4.32 * 10^{-51}}$

wider range along these measures doesn't necessarily mean better content quality. Designers knowledge or the player experience models constructed in [3] (that map game content to players reported affect) can be used as content quality measures to rank the content generated according to the gameplay experience it provides. Another direction would be to ask players to rank different levels generated by different generators.

The generator's parameters highly influence its expressivity range. For example, the fitness function and the design grammar used by the GE-generator can bias the search towards different kinds of maps. Analyzing this effect constitutes a future direction. The framework presented for analyzing the expressivity range of a generator can potentially be used by game designers or players to generate content with user defined expressivity parameters. This could be done by biasing the content generated according to these parameters.

ACKNOWLEDGMENTS

The research was supported in part by the Danish Research Agency, Ministry of Science, Technology and Innovation; project "AGameComIn" (274-09-0083).

REFERENCES

[1] J. Koza and R. Poli, "Genetic programming," *Search Methodologies*, pp. 127–164, 2005.

[2] M. O'Neill and C. Ryan, "Grammatical evolution," *Evolutionary Computation, IEEE Transactions on*, vol. 5, no. 4, pp. 349–358, 2001.

[3] N. Shaker, G. Yannakakis, and J. Togelius, "Feature analysis for modeling game content quality," in *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*. IEEE, 2011, pp. 126–133.

[4] A. M. Smith and M. Mateas, "Variations Forever: Flexibly Generating Rulesets from a Sculptable Design Space of Mini-Games," *IEEE Transactions on Computational Intelligence and AI in Games*, 2010.

[5] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," pp. 111–118, 2009.

[6] L. Cardamone, G. Yannakakis, J. Togelius, and P. Lanzi, "Evolving interesting maps for a first person shooter," *Applications of Evolutionary Computation*, pp. 63–72, 2011.

[7] J. Togelius, M. Preuss, and G. Yannakakis, "Towards multiobjective procedural map generation," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010, p. 3.

[8] G. Smith, J. Whitehead, and M. Mateas, "Tanagra: A mixed-initiative level design tool," in *Proceedings of the Fifth International Conference on the Foundations of Digital Games*. ACM, 2010, pp. 209–216.

[9] C. Pedersen, J. Togelius, and G. N. Yannakakis, "Modeling player experience for content creation," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 1, pp. 54–67, 2010.

[10] N. Sorenson, P. Pasquier, and S. DiPaola, "A generic approach to challenge modeling for the procedural creation of video game levels," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, no. 3, pp. 229–244, 2011.

[11] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Interactive evolution for the procedural generation of tracks in a high-end racing game," *Interface*, pp. 395–402, 2011.

[12] J. Togelius, R. De Nardi, and S. Lucas, "Towards automatic personalised content creation for racing games," in *IEEE Symposium on Computational Intelligence and Games, 2007.* IEEE, 2007, pp. 252–259.

[13] C. Browne and F. Maire, "Evolutionary game design," *IEEE Transactions on Computational Intelligence and AI in Games.*, vol. 2, no. 1, pp. 1–16, 2010.

[14] M. Cook and S. Colton, "Multi-faceted evolution of simple arcade games," in *IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2011, pp. 289–296.

[15] E. J. Hastings, R. K. Guha, and K. O. Stanley, "Evolving content in the galactic arms race video game," in *Proceedings of the 5th international conference on Computational Intelligence and Games*, ser. CIG'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 241–248.

[16] N. Shaker, G. N. Yannakakis, and J. Togelius, "Towards Automatic Personalized Content Generation for Platform Games," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. AAAI Press, October 2010.

[17] G. Yannakakis and J. Togelius, "Experience-driven procedural content generation," *Affective Computing, IEEE Transactions on*, vol. 2, no. 3, pp. 147–161, 2011.

[18] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne, "Search-based procedural content generation," *Applications of Evolutionary Computation*, pp. 141–150, 2010.

[19] J. Koza, M. Keane, M. Streeter, W. Mydlowec, J. Yu, and G. Lanza, *Genetic programming IV*. Kluwer Academic Publishers, 2003.

[20] P. Bentley, *Evolutionary design by computers*. Morgan Kaufmann, 1999, vol. 1.

[21] G. Hornby and J. Pollack, "The advantages of generative grammatical encodings for physical design," in *Proceedings of the 2001 Congress on Evolutionary Computation, 2001.*, vol. 1. IEEE, 2001, pp. 600–607.

[22] J. Byrne, M. Fenton, E. Hemberg, J. McDermott, M. ONeill, E. Shotton, and C. Nally, "Combining structural analysis and multi-objective criteria for evolutionary architectural design," *Applications of Evolutionary Computation*, pp. 204–213, 2011.

[23] M. O'Neill, J. Swafford, J. McDermott, J. Byrne, A. Brabazon, E. Shotton, C. McNally, and M. Hemberg, "Shape grammars and grammatical evolution for evolutionary design," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. ACM, 2009, pp. 1035–1042.

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

[25] J. Togelius, S. Karakovskiy, J. Koutník, and J. Schmidhuber, "Super mario evolution," in *Proceedings of the 5th international conference on Computational Intelligence and Games*, ser. CIG'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 156–161.

[26] N. Shaker, G. Yannakakis, and J. Togelius, "Digging deeper into platform game level design: session size and sequential features," *Applications of Evolutionary Computation*, pp. 275–284, 2012.

[27] P. Morel, H. Hamda, and M. Schoenauer, "Computational chair design using genetic algorithms," *Concept*, vol. 71, no. 3, pp. 95–99, 2005.

[28] M. O'Neill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, and A. Brabazon, "Geva: grammatical evolution in java," *ACM SIGEVOlution*, vol. 3, no. 2, pp. 17–22, 2008.

[29] M. Li, X. Chen, X. Li, B. Ma, and P. Vitányi, "The similarity metric," *IEEE Transactions on Information Theory*, vol. 50, no. 12, pp. 3250–3264, 2004.

[30] M. ONeill, R. Cleary, and N. Nikolov, "Solving knapsack problems with attribute grammars," in *Proceedings of the Third Grammatical Evolution Workshop (GEWS04)*. Citeseer, 2004.