A Comparative Evaluation of Procedural Level Generators in the Mario AI Framework

Britton Horn
Northeastern University
PLAIT Research Group
Boston, MA, USA
bhorn@ccs.neu.edu

Steve Dahlskog
Malmö University
School of Technology
Malmö, Sweden
steve.dahlskog@mah.se

Noor Shaker
IT University of Copenhagen
Center for Computer Games Research
Copenhagen, Denmark
nosh@itu.dk

Gillian Smith
Northeastern University
PLAIT Research Group
Boston, MA, USA
gillian@ccs.neu.edu

Julian Togelius
IT University of Copenhagen
Center for Computer Games Research
Copenhagen, Denmark
julian@togelius.com

ABSTRACT

Evaluation is an open problem in procedural content generation research. The field is now in a state where there is a glut of content generators, each serving different purposes and using a variety of techniques. It is difficult to understand, quantitatively or qualitatively, what makes one generator different from another in terms of its output. To remedy this, we have conducted a large-scale comparative evaluation of level generators for the Mario AI Benchmark, a research-friendly clone of the classic platform game Super Mario Bros. In all, we compare the output of seven different level generators from the literature, based on different algorithmic methods, plus the levels from the original Super Mario Bros game. To compare them, we have defined six expressivity metrics, of which two are novel contributions in this paper. These metrics are shown to provide interestingly different characterizations of the level generators. The results presented in this paper, and the accompanying source code, is meant to become a benchmark against which to test new level generators and expressivity metrics.

1. INTRODUCTION

Procedural Content Generation (PCG) research is concerned with creating methods for generating game content with limited human involvement, automatically or semi automatically [27, 19]. "Content" is a broad term that involves things such as items, quests, rules and textures, but one of the most commonly generated types of content is levels. Runtime level generation has existed in published games at least since *Rogue* [1], and is important for thriving game genres as different as roguelikes, endless runners (e.g. *Canabalt* [3]) and epic strategy games (e.g. *Civilization* [9]). In recent years, many academic researchers have been inspired to work on the problems of level generation, and the academic literature now contains dozens of papers on the topic. These papers are methodologically very diverse, including approaches using agents, grammars, constraint solving, cellular automata, evolutionary computation, exhaustive search, and answer set programming [30, 10].

A question that naturally comes to mind is how to choose which of these methods to use. Are some methods best for different purposes? Is one generator capable of creating different kinds of content than another? Different games, and even different stages or modes of the same game, pose different content generation problems. For some games, the connectivity and reachability of the levels might be difficult to attain and the most important problem, for others the lifelikeness of certain structures such as walls or vegetation might be most important, or the rhythm of the level, or the fine-tuned challenge of the level. On top of that, PCG solutions have numerous tradeoffs. For example, a common tradeoff is the speed of the solution versus the possibility to guarantee certain properties of the level (such as reachability). The degree to which the character of the generated level can be controlled via parameters might also be in opposition to the diversity of the generated content. In order to obtain meaningful solutions to these questions and problems, a method must first be created to evaluate individual content generators and compare generators to each other.

The current state of evaluation for content generators is largely ad hoc. Some generators are evaluated implicitly, via an evaluation of the game that they are situated in. This form of evaluation is not helpful to understand the qualities of the generator itself, and how to compare it to other generators. Frequently, generators are evaluated via a small sample of their output being shown as representative of the generator's capabilities; this form of evaluation-by-example lacks rigor (how do we know the small sample is representative?) and cannot help with an understanding of the space of potential content that can be created, as well

as any biases in the generator. (A critical reader of such papers tend to suspect that either the shown example is the one good-looking level generated by the generator after many attempts, or that all levels look pretty much like the shown example.) For the PCG research community, it is difficult to make progress without a thorough understanding of the strengths and weaknesses of current approaches, how they compare, and whether a new generator is capable of producing novel results.

In order to make informed decisions about which content generation method would be best suited for a particular type of content generation problem, we need a way to characterize the performance of the content generator in the context of game design concerns. A promising approach to this takes the form of a set of metrics that can be applied to the output of the generator, to characterize the generator's expressive range. This paper is meant to make progress towards a common framework for evaluating content generators through the presentation of several metrics, and evaluating these candidate metrics on a collection of content generators.

We focus on levels for platform games, and in particular we investigate level generators for the Mario AI Benchmark, based on *Infinite Mario Bros* [2], an open-source game inspired by the platform game *Super Mario Bros* [16]. This is done partly because the Mario platforming games are archetypical, their design influencing countless other games. This means that the level generation problems posed by that game are likely to be similar to the level generation problems posed by many other platform games and related games. Another reason is the popularity of the Mario AI Benchmark among academic Game AI and PCG researchers, meaning that this is probably the game for which the largest number of level generators have been made. This paper compares level generators developed by ourselves and by other researchers, in particular the participants in the Level Generation Track of the Mario AI Competition.

The main contribution of this paper is a thorough evaluation and comparative study of several existing level generators that have never been compared before. Several of these metrics have been used before (though some have been newly ported to the Mario AI framework); however, we also provide some new metrics, in particular the pattern-based metrics, and levels which have not been analysed before, in particular the original SMB levels. The result of this work is a baseline against which researchers can compare both new metrics and new generators, including an ability to easily visualize the results. The source code for the metrics and the evaluated levels has been publicly released, to make this evaluation framework available for all interested parties.[1]

2. RELATED WORK

There is a relatively large body of work aimed at understanding game design in general and level design in particular. For example, Koster's *Theory of Fun* [13] focuses on the progression of challenges and learnability of games, while others prefer to understand games as systems of interlocking feedback loops [4]. Recently there has been a push towards understanding and describing games using a pattern language. Björk and Holopainen [5] catalog recurring patterns that can be found across many games, while others have examined level design-specific patterns in domains such as first-person shooters [11], 2D platforming games [24, 8], and role-playing games [15, 23]. While some of these patterns are largely intended as qualitative descriptions of game properties, others take the view that design patterns can be solutions to specific design problems.

Most of the aforementioned work in understanding game and level design is based in qualitative analysis and theories. Several of the metrics in this paper form a step towards building upon such theories, including the new design pattern metrics described in Section 3.2. Further operationalization of these theories to develop more sophisticated metrics is an interesting potential area for future work.

Recently, there has been some work in trying to automatically and quantitatively measure aspects of game quality. Within search-based procedural content generation there is a need for evaluation functions, and for this reason several researchers have tried to quantitatively capture what they deem to be crucial aspects of game quality. This includes Browne's various metrics for board games, such as drawishness, length, drama and outcome uncertainty [6], and Togelius and Schmidhuber's learning-based metric [28]. There has also been some work on trying to measure the quality of platform game metrics specifically. For example, Smith and Whitehead defined two key metrics—linearity and leniency—as well as introducing a method for visualizing the expressive range of a generator [25]. Shaker et al. followed up this work by introducing further metrics, some of them based on theories of player experience and others based on data mining [18].

Finally, more broadly than PCG for games, there is some work in evaluating computationally creative systems; Jordanous provides a survey of current evaluation methods [12]. It is important to note that these evaluation criteria are being used to answer a different, though related, question: computational creativity evaluation asks the extent to which a system is creative, while PCG evaluation asks how expressive and controllable the system is. For example, Pease et al. incorporate an evaluation of the *process* that the generative system follows as well as rating the *product* produced by the system [17]. Similar evaluations have been performed on platform game level generators [7]. While we recognize the importance of process in understanding creativity, and feel that such discussions would be of great value to PCG researchers, it lies outside the scope of this paper.

3. EXPERIMENTAL TESTBED

For our experiments, we use the Mario AI Benchmark [29], built on top of *Infinite Mario Bros.* The world representation in this framework is not 100% faithful to the original *Super Mario Bros.*, and some of the graphical elements resemble elements from later games in the series. In particular, not all of the items and creatures found in *Super Mario Bros*, can be found in *Infinite Mario Bros*, but the missing features tend to be infrequently used in the original game.

3.1 Generators

In this section we describe the different generators that are compared in this paper. The generators were chosen because they have been the subject of academic papers or have taken part in the level generator track of the Mario AI Championship, and so as to maximize the number of different approaches to level generation represented.

The **Notch** generator is the default level generator that

[1] `http://sokath.com/fdg2014_pcg_evaluation/`

comes with *Infinite Mario Bros.* It writes levels from left to right, adding components according to probabilities. Basic checks are performed to make sure the levels are playable.

The **Parameterized Notch** generator is a version of the Notch generator that takes parameters, which bias how levels are generated. These parameters are the number of gaps, width of gaps, number of enemies, enemy placement, number of powerups and number of boxes. The test explores all possible combinations of high and low values for these parameters. See [21] for more information.

Hopper was written for the Level Generation track of the 2010 Mario AI Championship. Like *Notch* and *Parameterized Notch*, it generates levels through writing them from left to right and placing features with specific probabilities. It was built with adaptability in mind, so that the probabilities could easily be altered depending on the player's prior performance. The generated parts are alternated with pre-designed parts. See [20] for more information.

Launchpad is a rhythm-based level generator that uses design grammars for creating levels that obey rhythmical constraints. The original version of Launchpad incorporated several level elements that are not present in the framework (e.g. springs); this ported version has attempted to remain as faithful as possible to the original grammar-based implementation, substituting level components as needed. See [25] for information on the original Launchpad.

The **Occupancy-Regulated Extension (ORE)** generator was also an entry for the Level Generation track of the 2010 Mario AI championship [20]. It works by piecing together small, hand-authored chunks of levels. Each chunk has an "anchor point" used to determine how the chunks can be pieced together. It can create quite complex levels that are stylistically quite different from the original Mario levels.

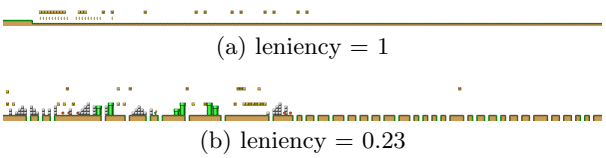
The **Pattern-based** generator uses evolutionary computation to generate levels. Levels are represented as sequences of "slices", or "micro-patterns" which are taken from the original Mario. Each micro-pattern is one block wide and has the same height as the level. The fitness function counts the number of occurrences of specified sections of slices, or "meso-patterns". The objective is to find levels with as many meso-patterns as possible. See [8] for more information.

The **Grammatical Evolution (GE)** generator uses evolutionary computation together with design grammars. Levels are represented as instructions for expanding design patterns, and the fitness function measures the number of items in the level and the number of conflicts between the placement of these items. See [18] for more information.

Finally, the **original levels** from *Super Mario Bros 1* [16] are included. They have been reproduced as faithfully as possible by manual translation from the ROM code of the original game. The exceptions are those elements which are not part of the design vocabulary of Infinite Mario Bros (and thus of the Mario AI benchmark), and the water-based levels which cannot be simulated in the current version of the framework and for which completely different design principles are likely to hold. Levels were between 148 and 377 blocks in length, with an average length of 200 blocks.

3.2 Metrics

To compare the levels produced by our various levels, we have used a number of metrics, most of which come from previous literature but the two pattern-based metrics are introduced in this paper. The metrics are meant to capture relevant aspects of the levels including player experience (e.g. leniency) and level composition, but as there are many potentially relevant aspects, the current set of metrics should not be seen as exhaustive.



(a) leniency = 1

(b) leniency = 0.23

Figure 1: Example levels from (a) the parameterized notch randomized and (b) the pattern-based weighted count generators with very low and high leniency values.

3.2.1 Individual level metrics

Most of our metrics work on a single level, and return a single real number as its evaluation of that level. All of our metrics are normalized by level length as appropriate, and are further normalized by the total of output of that metric.

The **leniency** metric is an attempt to capture how difficult a level is for a player. Leniency was calculated by finding all points in the level where an action by the player is needed, e.g. the edge of a platform or the end of a string of blocks, and then determining how lenient that particular challenge would be to the player. Gaps where a player dies are given a 0 weight for leniency and other enemies and jump lengths are weighted based on the challenge and death possibility given to a player. When a jump is detected, it looks ahead to see if any other obstacles would be in the way while landing. These other obstacles lower the leniency of the first challenge by a factor equal to their assumed harm level. Areas with no threat of harm are given a score of 1. Once all obstacle weights were calculated, they were then normalized based on the length of the level and how many possible paths were available at each point in the level. Two example levels from the parameterized notch randomized and the pattern-based weighted count generators with very low and high leniency values are presented in Figure 1. Note that this description of leniency is different from those used in previous work by Smith et al. [25] and Shaker et al. [18].

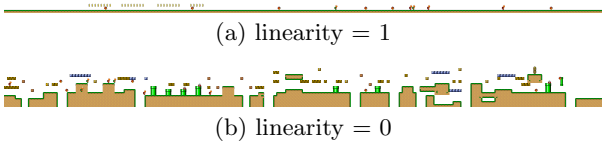
The **linearity** metric is calculated by finding the R2 goodness-of-fit measure for a line that has been fit to the end points for each platform in the level. This means that levels with many height differences will have low linearity, while levels that follow a straight line (flat or otherwise) will have maximum linearity. Linearity is originally defined in [25]. Figure 2 presents two examples from different generators having extreme linearity values.

Density is a measure of how many platforms are stacked on top of each other. The density calculator assigns a density value to each position depending on how many different heights Mario could possibly stand on. The density value for a level is simply the average density value for all positions on the level. Density is defined in [18] and two example figures for levels from two different generators with comparable density score are presented in 3.

Pattern density measures how many meso-patterns from the original Super Mario Bros game can be found in the level. This metric is the same calculation as the evaluation func-

generator	leniency	linearity	density	pattern density	pattern variation	compression distance
GE	0.84 (0.06)	0.02 (0.03)	0.47 (0.16)	0.1 (0.03)	0.27 (0.06)	0.56 (0.04)
hopper	0.72 (0.04)	0.15 (0.16)	0.6 (0.15)	0.1 (0.02)	0.29 (0.05)	0.65 (0.05)
launchpad	0.7 (0.05)	0.66 (0.31)	0.24 (0.04)	0.11 (0.03)	0.17 (0.05)	0.8 (0.07)
launchpad-rhythm	0.74 (0.07)	0.49 (0.32)	0.11 (0.04)	0.09 (0.03)	0.13 (0.06)	0.81 (0.09)
notch	0.67 (0.06)	0.1 (0.11)	0.4 (0.16)	0.13 (0.02)	0.27 (0.08)	0.53 (0.03)
notch param	0.85 (0.06)	0.04 (0.05)	0.81 (0.08)	0.08 (0.03)	0.24 (0.07)	0.36 (0.08)
notch param rand	0.86 (0.08)	0.08 (0.06)	0.8 (0.1)	0.08 (0.03)	0.17 (0.09)	0.47 (0.08)
ORE	0.51 (0.08)	0.05 (0.06)	0.43 (0.15)	0.16 (0.03)	0.35 (0.05)	0.73 (0.04)
original	0.61 (0.18)	0.02 (0.02)	0.35 (0.37)	0.14 (0.06)	0.3 (0.1)	0.76 (0.11)
pb count	0.63 (0.1)	0.07 (0.09)	0.08 (0.05)	0.39 (0.17)	0.41 (0.07)	0.85 (0.04)
pb occurence	0.6 (0.08)	0.04 (0.06)	0.06 (0.09)	0.08 (0.02)	0.64 (0.11)	0.79 (0.08)
pb weighted count	0.61 (0.12)	0.06 (0.08)	0.09 (0.08)	0.08 (0.03)	0.24 (0.07)	0.86 (0.05)

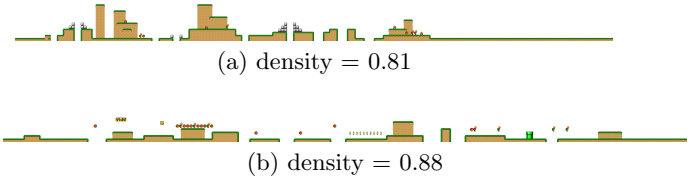
Table 1: Overview comparison of level generators: mean value (standard deviation) of each metric on the output of each generator.



(a) linearity = 1

(b) linearity = 0

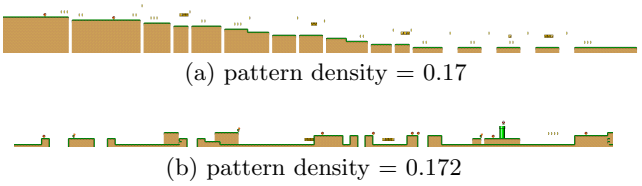
Figure 2: Example levels from (a) the parameterized notch randomized and (b) the ORE generators with very low and high linearity values.



(a) density = 0.81

(b) density = 0.88

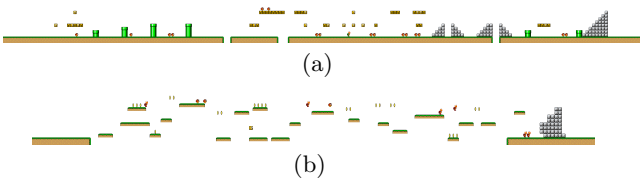
Figure 3: Example levels from (a) the notch and (b) the hopper generators with comparable density values.



(a) pattern density = 0.17

(b) pattern density = 0.172

Figure 4: Example levels from (a) the launchpad and (b) the GE generators with comparable pattern density values.



(a)

(b)

Figure 5: Examples from the original levels that are dissimilar according to the compression distance, ncd = 0.9.

tion for the evolutionary algorithm in the pattern-based level generator, and is described in [8]. The metric is normalized according to level length. Figure 4 presents two illustrative levels for this measure.

Pattern variation, on the other hand, measures only unique occurrences of patterns and gives higher values to levels with diverse meso-patterns instead of many reoccurring meso-patterns. The metric is also normalized according to level length.

3.2.2 *Level distance metrics*

The other category of metrics are those that do not work on individual levels, but on pairs of levels by measuring how different they are, or in other words their *distance* in some space.

In the comparison performed for this paper we only have one level distance metric: **compression distance** is a domain-general metric based on the principle that if two strings are similar, you save more space when compressing them together. In this implementation, we use the standard gzip algorithm and compare the length of the resulting string when compressing each of two levels individually and when compressing them together. Compression distance is described in [14] and applied to platform game levels in [18]. Figure 5 presents two example levels transcribed from the original game that are found to be very dissimilar to each other according to this measure.

Another example of a metric that would fit this category is the edit distance metric used in clustering *Launchpad*'s rhythm groups [25].

4. GENERATOR COMPARISON

All level generators were instructed to output levels of approximately 200 blocks in length, which on average would be about a minute of playing time for a proficient player, and which is close to the median length of the original SMB levels. All level generators were used to produce 1000 unique levels; from the original game, we have 22 unique levels (omitting 10 levels from the original, since 2 levels are "under water" and 8 levels are "boss-fight levels") We never analyzed the bonus areas since they primarily are "warp zones"

or filled with coins.

There are seven generators included in the analysis, as described in Section 3.1. Of these, there are two level sets produced by the *Parameterized Notch* generator, and two level sets produced by the *Launchpad* generator, with different parameter settings. The *Parameterized Notch Randomized* level set comes from the *Parameterized Notch* generator, with the values of the controllable parameters chosen at random. The *Launchpad-Rhythm* levels come from the *Launchpad* generator, by varying only the rhythm parameters while holding the length of rhythm groups and component probabilities constant. The *pattern-based* generator has three major variants, arising from three different fitness functions being used: *pattern occurrence* (counting each meso-pattern only once), **pattern count** (counting each occurence of a meso-pattern) and *weighted pattern count*, where the patterns are weighted by their rarity. Remaining generators each had a single set of levels generated for them. The result is 12 different sets of levels to be analyzed and compared.

The remainder of this section describes the results of these experiments, including providing visualizations of expressive range for several generator/metric combinations, and a brief description of the controllability for each generator.

4.1 All Metrics

Table 1 presents a high-level comparison of all generators. For each generator, we present the average value of its levels on all metrics, and the standard deviation of the that value. A number of observations can be made based on this table. To start with, there is a lot more variance between generators (as compared to levels generated by the same generator) on some metrics than others. For pattern density, the variation between generators is comparable to the variance "within" generators (on levels generated by the same generator), whereas for lenience, linearity and density it is much higher. Therefore, it seems that the latter three metrics are better for telling level generators apart; a complementary interpretation is that all generators are bad at providing variance in those three dimensions.

Studying each metric in detail, we can see a number of patterns. The lowest leniency value can be found for occupancy-regulated extension. Looking at the levels, it is clear that they feature more gaps than other levels. This doesn't necessarily make them more difficult, as there are often many different jumping-off point for each gap. For the linearity metric, the outlier is instead the two versions of the Launchpad generator, which have much higher linearity (and larger variety with respect to linearity) than the other generators. It is here very clear that Launchpad was originally designed with another kind of platform game in mind, more akin to rhythm-based games such as *Sonic the Hedgehog* [26] which feature more or less constant forward motion.

For the density metric, the real outlier is the pattern-based level generator. As described above, the density metric counts the number of platforms at different heights at each tile. The original levels have a relatively low density as well, but not at all as low as the levels generated by the pattern-based generator. This points to that level segments with multiple level platforms are for some reason not reproduced very well by the pattern-based generator (according to its design criteria, an ideal pattern-based generator should have values similar to those of the original levels on all metrics). Several other generators, including the Notch generators, generate far too many overlapping platforms of different height as compared to the original levels.

The two pattern-based metrics are dominated by two different versions of the pattern-based generator, as would be expected given that these metrics are fitness functions for these two versions of the generator. The highest value on pattern density is thus scored by the pattern count generator, and the highest score on pattern variation by the pattern occurence generator. Both ORE and the original levels have very high scores on both pattern variation and pattern density, which is logical given that the pattern that the metric looks for are based on the original levels, and on that the ORE generator manages to cram a lot of structure into a short level space. The two versions of Launchpad scores low on pattern variation, pointing to the relatively sparse character of their levels; the Notch levels also scores low on this metric. Interestingly, pattern density and pattern variation appear highly correlated except when it comes to the pattern-based generator, where they diverge sharply.

The compression distance metric, being fundamentally different in that it is a between-level metric, must be discussed separately. One thing that stands out here is that the various versions of the Notch generator have the lowest compression distance. This indicates that the levels, from an information-theoretic viewpoint, are all very similar. Interestingly, this corresponds very well with qualitative observations that the levels in Infinite Mario Bros appear quite similar to each other. On the other end of the scale, the pattern-based levels have the highest compression distance, meaning that very little is gained by compressing two such levels together. This could be explained by the way that generator assembles levels out single slices, micro-patterns with length 1 block. As there are no fixed orders of slices (a given meso-pattern could be implemented through many different slice combinations), this means that a compression algorithm based on finding commonly occurring substrings would not find much to build on.

Figure 6 shows a visualization of every metric and generator using a boxplot graph. Each of the six metrics is clustered together per generator. This visualization shows that each generator has its own unique profile for the sets of metrics, not only in mean and standard deviation (as shown in Table 1), but also in the overall range of each metric.

4.2 Expressive Range Visualization

The evaluation framework thus far has viewed each metric largely independently from the others. While this provides a good high-level view on the properties of a particular generator, it does not indicate any relationships between metrics or show the shape of a generator's expressive range. For example, viewing all metrics at a high level can show that a particular generator might be pre-disposed towards creating a medium range of levels in terms of both linearity and density, but it would not be able to show any correlation between levels according to those metrics. Visualizing the expressive range of generators allows such biases to be easily seen [25]. This visualization involves plotting a 2D histogram as a heatmap, where each axis on the plot is one of the metrics, and each bucket in the histogram is assigned a color based on how many levels are in the bucket.

For more than two metrics, there is no clear way to produce a multi-dimensional histogram. While there are several potential uniform visualizations that show metric values for

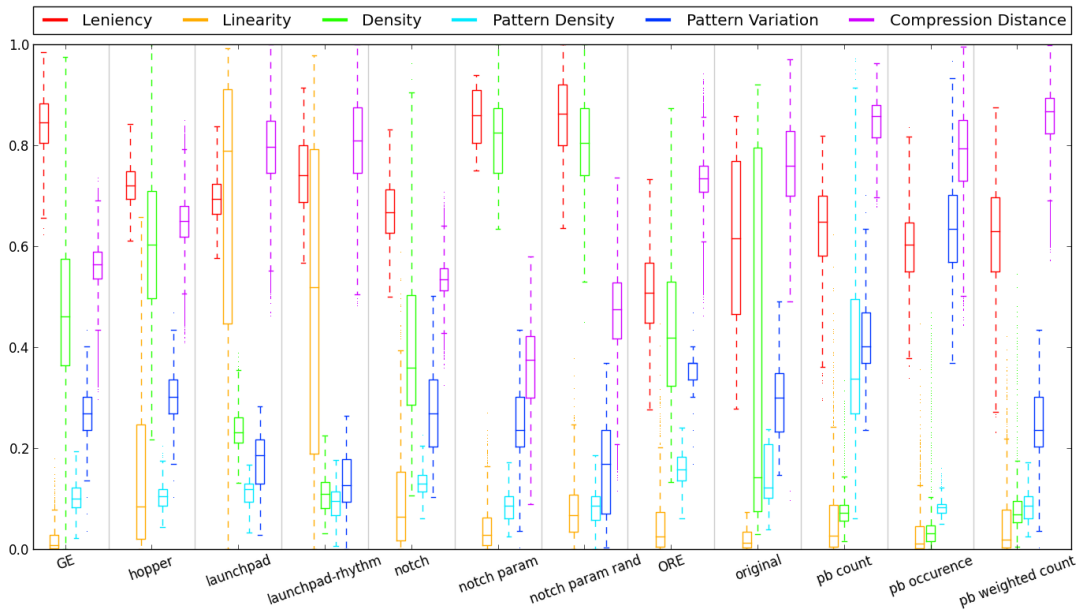


Figure 6: A visual comparison of all generators included in this analysis using all of the metrics. Each generator is evaluated using six metrics, denoted in different colors. The boxplot for each generator-metric pair shows the median, and upper and lower quartiles. The whiskers extend to data points that fall within 1.9 IQR of the upper and lower quartile, and outliers from this range are depicted as small dots.

each individual level (e.g. a circular heatmap, or a stacked bar chart), the crucial insight gained from expressive range evaluation is identifying dense areas in a plot resulting from many levels sharing similar metric values. Thus, in this paper, we have chosen to compare expressive range by generating several graphs per generator for pairwise combinations of metrics. For space reasons, not all expressive range graphs are shown here; we have selected one of these sets of graphs with particularly interesting features to show in this paper.

In each of the graphs corresponding to metric pair shown in Figure 7, the warmest area of the heatmap is red (corresponding to 40 or more levels in that bin), while the coolest area of the heatmap is dark blue. Expressive range for the original SMB levels uses a different scale, however, as there are considerably fewer levels available to measure; the warmest area of the Original SMB level heatmaps corresponds to 5 levels being in that bin.

Figure 7 shows several subtleties of the generative space for each generator formed by the density and leniency metrics. The rough shapes of the expressive range correspond to what would be expected, given the range of each metric shown in Figure 6. However, the version of *Launchpad* that varies its rhythm parameters (row 1, column 4) shows an unexpected correlation between density and leniency, which is not mirrored in the fixed-rhythm version. The *Parameterized Notch* generator (row 1, column 6) shows that it biases two distinct clusters of levels, again with a slight correlation between the two metrics. From all of the graphs side by side, it is clear that the expressive ranges of each of these generators are overall quite different, though there are some similarities. Overall character and shape of the generators is somewhat easier to interpret in these graphs than using a boxes and whiskers diagram.

4.3 Controllability

As discussed in the introduction, there is a potential and sometimes perceived partial conflict between expressivity and controllability in procedural level generation. While this paper is chiefly about the expressive range of the various generators involved, an important aspect of evaluating content generators is evaluating how they can be controlled. Here we discuss the ways in which each level generator can be controlled, to help us in gaining an initial understanding of the relationship between controllability and expressivity in this domain. Table 2 summarizes how each generator can be controlled by a designer; note that the table contains only the main generators, and does not list different configurations of the same generator.

The *compression distance* metric can be useful for understanding the impact of parameters in a parameterized generator. By illustrating the compression distance as a 2D matrix with a heatmap applied to it (with cooler colors representing low distance), it is possible to see patterns with sets of levels that have low distance between each other. For example, Figure 8(a) shows the compression distance matrix for the *Parameterized Notch* generator. The checkerboard pattern corresponds to common combinations of parameters: those that share the same parameters are more similar to each other than those that do not. When examining the compression distance matrix for *Launchpad* with varied rhythm parameters (Figure 8(b)), a different effect is seen; variety is overall higher (as reflected by higher compression distance scores), with particularly high variety when the rhythm beat type is regular.

5. FUTURE WORK

There are many possible metrics that we have not in-

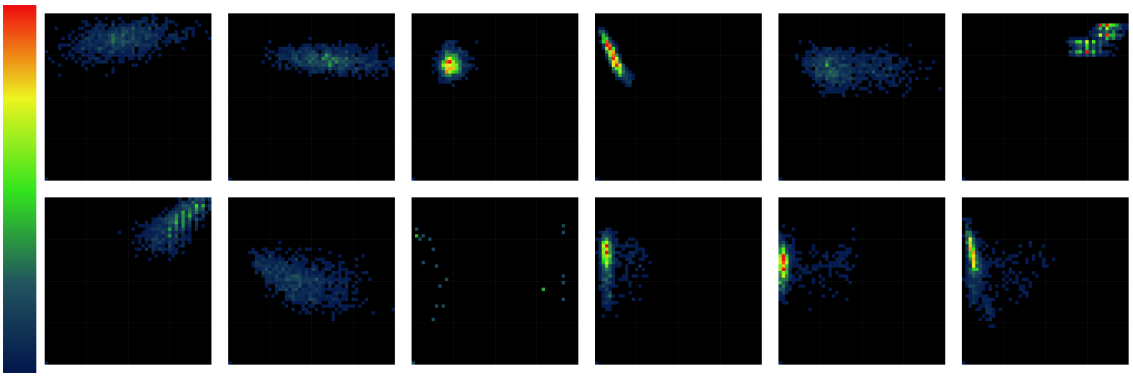
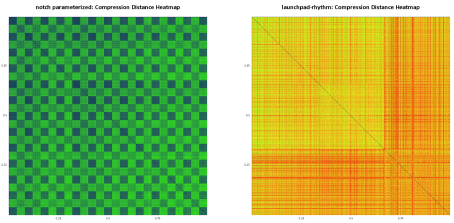


Figure 7: Heatmaps visualizing the expressive range of each generator according to the Density (x-axis) and Leniency (y-axis) metrics. The order of generators (left to right, top to bottom) is: GE, hopper, launchpad, launchpad-rhythm, notch, parameterized notch, parameterized notch-randomized, ORE, original levels, pattern-based-count, pattern-based-occurrence, pattern-based-weighted-count.

generator	control type
GE	*indirect*, via changing evolution parameters
hopper	*parameterized*, for implicitly defined difficulty levels
launchpad	*parameterized*, for component appearance and rhythm
notch	*none*
notch (param.)	*parameterized*, for component appearance
ORE	*knowledge representation*, can change input chunks
pattern-based	*indirect*, via changing evolution parameters; and *knowledge representation*, can change input patterns

Table 2: Controllability of the main generators tested in this paper, using vocabulary from [22].



(a) (b)

Figure 8: Heatmaps visualizing the compression distance matrix, showing the impact of varying parameters. (a) Parameterized Notch generator. (b) Launchpad with varied rhythm parameters.

cluded in this study. These include metrics that measure macro-scale progression and repetition in the level. They also include simulation-based metrics, which would use an artifical agent to play the level and analyse its playing style. Further, we could use metrics that try to judge the shape of the level, for example through computer vision methods. Or we could associate individual level patterns and situations with player experience through machine learning, and build level metrics on top of the output of such models. Lacking any previous comparative PCG evaluation, we focused primarily on existing research metrics.

A question that becomes more pressing the more metrics we accumulate is how to choose between them, or perhaps combine them. One way would be to use principal component analysis, or some similar dimensionality reduction technique. This could give us a smaller number of joint metrics that still capture the essential variance between levels. Or simpler, we could cross-correlate the various metrics and only keep the least correlated ones. However, we also need to weigh the importance of having human-interpretable metrics and results; it is important for designers and AI researchers to understand how generators differ from each other in a design-relevant context.

This assumes all metrics are somehow equally important. Clearly, that is not true for most specific intendend usages, e.g. to design an intriguing, fun or challenging level. We would therefore need to complement our computational investigation with user studies, where we associate metrics with their effects on player experience. The level distance metrics could also be validated by investigating how different to each other various levels are perceived to be.

Finally, the comparison of generators performed here is only possible because each generator shares a common context and framework. Evaluating within a common framework is helpful; however, it also obscures the importance of creating a content generator to meet a specific game's context. Clearly, some metrics can be easily applied to multiple level generation contexts (such as compression distance) while others may need to be fine-tuned for a new context.

6. CONCLUSIONS

We have defined a framework for evaluating and comparing the expressivity of level generators, and quantita-

tively compared seven different platform game level generators (and several variations of them), along with the original Super Mario Bros levels, using six different metrics. We have also discussed the role of controllability in level generation and its relation to expressivity. Our results constitute the first quantitative comparison of multiple level generators, and contain both expected and unexpected outcomes. Among the expected outcomes are that the differences between generators on most metrical dimensions correspond to the qualitatively observed differences between levels generated by them. Among the unexpected outcomes is that parameterization plays a very large role in changing the nature of generated levels by some generators (e.g. Notch, Launchpad) but not others (e.g. pattern-based). Metrics that correlate for one generator might not correlate for another version of the same generator. We believe the information contained in this paper provides a good baseline against which to characterize new generators and metrics, and have made freely available level samples and source code.

7. REFERENCES

[1] Rogue (PC game), 1980.

[2] Infinite Mario Bros! (PC Game), 2008.

[3] Canabalt (PC game), 2009.

[4] E. Adams and J. Dormans. *Game mechanics: advanced game design*. New Riders, 2012.

[5] S. Björk and J. Holopainen. *Patterns in game design*. Cengage Learning, 2005.

[6] C. Browne and F. Maire. Evolutionary game design. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(1):1–16, 2010.

[7] M. Cook, S. Colton, and A. Pease. Aesthetic Considerations for Automated Platformer Design. In *AIIDE*, 2012.

[8] S. Dahlskog and J. Togelius. Patterns as Objectives for Level Generation. In *Proceedings of the Design Patterns in Games Workshop*, 2013.

[9] Firaxis Games. Civilization IV (PC game), 2005.

[10] M. Hendrikx, S. Meijer, J. Van der Velden, and A. Iosup. Procedural Content Generation for Games: A Survey. *ACM Transactions on Multimedia Computing, Communications and Applications*, 2011.

[11] K. Hullett and J. Whitehead. Design patterns in FPS levels. In *proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 78–85. ACM, 2010.

[12] A. Jordanous. Evaluating evaluation: Assessing progress in computational creativity research. In *Proceedings of the second international conference on computational creativity (ICCC-11). Mexico City, Mexico*, pages 102–107, 2011.

[13] R. Koster. *Theory of fun for game design*. O'Reilly Media, Inc., 2004.

[14] M. Li, X. Chen, X. Li, B. Ma, and P. M. Vitányi. The similarity metric. *Information Theory, IEEE Transactions on*, 50(12):3250–3264, 2004.

[15] M. McNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford, and D. Parker. ScriptEase: Generative design patterns for computer role-playing games. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 88–99. IEEE Computer Society, 2004.

[16] Nintendo Creative Department. Super Mario Bros. (NES), Sept. 1985.

[17] A. Pease, D. Winterstein, and S. Colton. Evaluating Machine Creativity. In *Workshop on Creative Systems, 4th Internation Conference on Case Based Reasoning*, pages 129–137, 2001.

[18] N. Shaker, M. Nicolau, G. N. Yannakakis, J. Togelius, and M. O'Neill. Evolving levels for Super Mario Bros using grammatical evolution. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 304–311. IEEE, 2012.

[19] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2014.

[20] N. Shaker, J. Togelius, G. N. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi, G. Smith, and R. Baumgarten. The 2010 Mario AI championship: Level generation track. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(4):332–347, 2011.

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

[22] G. Smith. *Expressive Design Tools: Procedural Content Generation for Game Design*. PhD thesis, UC Santa Cruz, Santa Cruz, CA, June 2012.

[23] G. Smith, R. Anderson, B. Kopleck, Z. Lindblad, L. Scott, A. Wardell, J. Whitehead, and M. Mateas. Situating quests: Design patterns for quest and level design in role-playing games. In *Interactive Storytelling*, pages 326–329. Springer, 2011.

[24] 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*, pages 209–216. ACM, 2010.

[25] G. Smith, J. Whitehead, M. Mateas, M. Treanor, J. March, and M. Cha. Launchpad: A rhythm-based level generator for 2-d platformers. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(1):1–16, 2011.

[26] Sonic Team. Sonic the Hedgehog (Genesis), 1991.

[27] J. Togelius, A. J. Champandard, P. L. Lanzi, M. Mateas, A. Paiva, M. Preuss, K. O. Stanley, S. M. Lucas, M. Mateas, and M. Preuss. Procedural content generation: Goals, challenges and actionable steps. *Artificial and Computational Intelligence in Games*, 6:61–75, 2013.

[28] J. Togelius and J. Schmidhuber. An experiment in automatic game design. In *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*, pages 111–118. IEEE, 2008.

[29] J. Togelius, N. Shaker, S. Karakovskiy, and G. N. Yannakakis. The Mario AI Championship 2009–2012. *AI Magazine*, 34(3):89–92, 2013.

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