Gram-Elites: N-Gram Based Quality-Diversity Search

Colan F. Biemer
Northeastern University
biemer.c@husky.neu.edu

Alejandro Hervella
Northeastern University
hervella.a@northeastern.edu

Seth Cooper
Northeastern University
se.cooper@northeastern.edu

ABSTRACT

In the context of procedural content generation via machine learning (PCGML), quality-diversity (QD) algorithms are a powerful tool to generate diverse game content. A branch of QD uses genetic operators to generate content (e.g. MAP-Elites). Problematically, levels generated with these operators have no guarantee of matching the style of a game. This can be addressed by incorporating whether a level is generable by an n-gram into the fitness function. Unfortunately, this leads to wasted computation and poor results. In this work, we introduce n-gram genetic operators, which produce only solutions that are generable by the n-gram model; we call MAP-Elites combined with these operators Gram-Elites. We test on a tile-based side-scrolling platformer, vertical platformer, and roguelike. For all three, n-gram operators outperform standard operators and random n-gram generation, finding more usable (i.e. completable and generable) solutions at a faster rate. By integrating structure into operators, instead of fitness, these genetic operators could be beneficial to QD in PCGML.

CCS CONCEPTS

• **Human-centered computing**;

KEYWORDS

procedural content generation, video games, quality-diversity, n-grams, genetic operators

ACM Reference Format:
Colan F. Biemer, Alejandro Hervella, and Seth Cooper. 2021. Gram-Elites: N-Gram Based Quality-Diversity Search. In *The 16th International Conference on the Foundations of Digital Games (FDG) 2021 (FDG'21), August 3–6, 2021, Montreal, QC, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3472538.3472599

1 INTRODUCTION

Procedural content generation (PCG) is the creation of game content via an algorithm [12]. Its uses can vary from creating levels [3], to attack patterns [7], to entire games [1], and more. Procedural content generation via machine learning (PCGML) [15] uses machine learning algorithms that are trained on existing game content to generate new content in a similar style. A branch of PCGML uses quality-diversity algorithms to generate a variety of new content.

FDG'21, August 3–6, 2021, Montreal, QC, Canada

ACM ISBN 978-1-4503-8422-3/21/08…$15.00
https://doi.org/10.1145/3472538.3472599

One approach to quality-diversity (QD) content generation is the MAP-Elites algorithm [10]. MAP-Elites uses genetic operators to create new solutions that are diverse across multiple behaviors while optimizing a fitness function. The fitness function evaluates properties of solutions, such as how *completable* a generated level is by an agent. In the context of PCGML, we want solutions generated by MAP-Elites to also match the learned style of a target game. N-grams are one approach that has been used to recreate the style of input training levels in 2D tile-based games [3]. To capture this, fitness can incorporate *generability* via n-grams (i.e. whether or not the n-gram could generate a given solution).

In this way, MAP-Elites can generate a variety of *usable* levels—levels that are both *completable* by an agent and *generable* by an n-gram. However, standard genetic operators used by MAP-Elites are not aware of the n-gram model. This can cause wasted computation on levels that are not generable, and fewer solutions are found overall.

In this work, we introduce n-gram genetic operators. For both mutation and crossover, n-grams are used to modify a given solution. As a result, a new solution will always be generable by an n-gram. We call this combined n-gram and MAP-Elites approach *Gram-Elites*.

To test the n-gram operators, we use three tile-based games: *Super Mario Bros.*, *Kid Icarus*, and *DungeonGrams*—a roguelike we developed—and generated short level segments. We compare Gram-Elites to two variants. The first is MAP-Elites with standard operators, crossover and mutation. The second places segments generated by an n-gram into the MAP-Elites grid. For all three games, Gram-Elites finds more usable segments in fewer iterations.

2 BACKGROUND

2.1 N-grams

To generate segments of levels, we use n-grams [3]. As input, a level represented as a matrix of tiles is made into a sequence of slices (e.g. individual columns or rows). An n-gram model can predict an output given a prior. The prior is sequential and is size $n-1$; making an n-gram a $n-1$ order Markov chain. To train the model, it counts the number of times an output occurs for a unique prior. The result is a set of probability tables linked to their priors.

Generation with n-grams is done by inputting a prior and getting a weighted output. Weighted refers to using the counts from training to determine the likelihood of each output. The prior is updated by removing the oldest member and adding the new output. This maintains the order of the sequence. To stop generation, either the desired length is reached or the n-gram reaches an unseen prior [14]. An unseen prior is a prior that was not seen in the training data; the n-gram has no valid output.

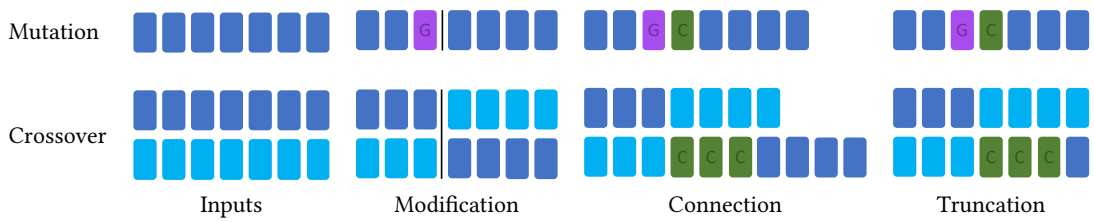

Figure 1: Visualization of n-gram genetic operators. Both operators build a connection (green Cs) to the other side to maintain a full segment with no bad n-grams. The mutation operator removes one slice and uses the n-gram to generate a replacement (purple G) before the connection is built.

2.2 MAP-Elites

MAP-Elites [10] is a QD algorithm that encourages behavioral diversity to find the best solution. MAP-Elites maintains a map of k dimensions where each dimension represents a behavioral characteristic of a solution (e.g. the number of enemies in a level). The map is discretized, to a given resolution, into bins. A bin can contain multiple solutions (called *elites*). In this work, we limit the size of a bin to one.

MAP-Elites starts with initial population generation. A set number of random solutions are placed into bins based on their behavioral characteristics. If the bin associated with the solution is empty, the solution is placed into it as its elite. If the bin is not empty, the fitness of the elite in the bin and the fitness of the new solution are compared. The one with the best fitness is placed into the bin. After the initial population generation, MAP-Elites runs for a set number of iterations. For each iteration, it selects bins and runs genetic operators on the elites. The default selection strategy is random, but there are many other approaches, see work from Gravina et [5]. After the operators run, the new solutions are placed into their associated bin if possible.

3 RELATED WORK

Procedural content generation via machine learning (PCGML) [15] is the generation of content by trained models. An early example comes from Dahlskog et al. [3], which used n-grams for *Mario* level generation. For input, they convert a grid representation of a *Mario* level into a list of columns. They found that generated levels tend to be completable and match the style of the input corpus. $n = 3$ produced results that were both in line with the style of a *Mario* level while still appearing to be novel. When n is too large, generated content will feature long, memorized sequences that do not diverge from the input corpus.

Work from Withington showed that MAP-Elites can generate Mario levels by using a binary representation for whether a block exists or not [17]. The generated segments are completable, but they do not look or feel like a Mario level. Khalifa et al. [6] address the problem of structure by using Constrained MAP-Elites [7]. This builds off of FI-2Pop [8] to have valid and invalid levels inside of every bin. A level is evolved to be completable by an agent in the infeasible population. In the feasible population, levels are evolved to be as simple as possible. This is another way of assessing structure that differs from our approach.

Fontaine et al. [4] show how a quality diversity search can work with a generative adversarial network (GAN). Instead of evolving levels, they evolve vectors as input into the latent space of the GAN. The GAN's objective is to output levels that match the structure of the target game.

In our work, we look at how n-grams can improve MAP-Elites for level generation. Previous work from Lo et al. evolved musical sequences [9] by using a complex fitness function that included n-grams to assess the likelihood of a given sequence. In the context of music, it can be beneficial to evolve a sequence that is not generable by an n-gram.

4 N-GRAM OPERATORS

While standard operators can be used in MAP-Elites, they are not aware of the n-gram model and can create sequences impossible for the n-gram to generate. To address this, we developed n-gram operators for mutation and crossover; see Figure 1. The n-gram operators are designed such that given input sequence(s) that are generable by the n-gram, the output sequence(s) will also be generable.

Both operators rely on a concept we call *connection*. To combine two sequences, we build a connection between the two. This is built using the last $n-1$ slices of the starting sequence (start prior) and the first $n-1$ slices of the ending sequence (end prior). The connection is built using a breadth-first search on the n-gram. It starts from the start prior and searches for a path to the end prior. Note that for the approach to work, such paths must exist between priors in the n-gram, and thus relies on using an n-gram that has at least one path between the start and end prior. We can guarantee that a path exists if the n-gram is *strongly-connected*. An n-gram is strongly-connected if and only if every prior has a path to every other prior in the n-gram. The breadth-first search creates a, potentitally empty, sequence of slices that connects the two original sequences. With *connection*, we can create the operators n-gram mutation and n-gram crossover.

N-gram mutation chooses a random point in a sequence. The random point is bounded by the requirement that there must be $n-1$ slices on either side of it. The slice at this point is removed. A new output is generated using the last $n-1$ slices of the sequence to the left of the point. The new output updates the start prior. The start prior is used to connect the two sequences around the random point. See Figure 1 (top).

N-gram crossover receives two parent segments and finds a crossover point. This point has the same requirements as mutation. The point is selected randomly but there must be $n-1$ slices on

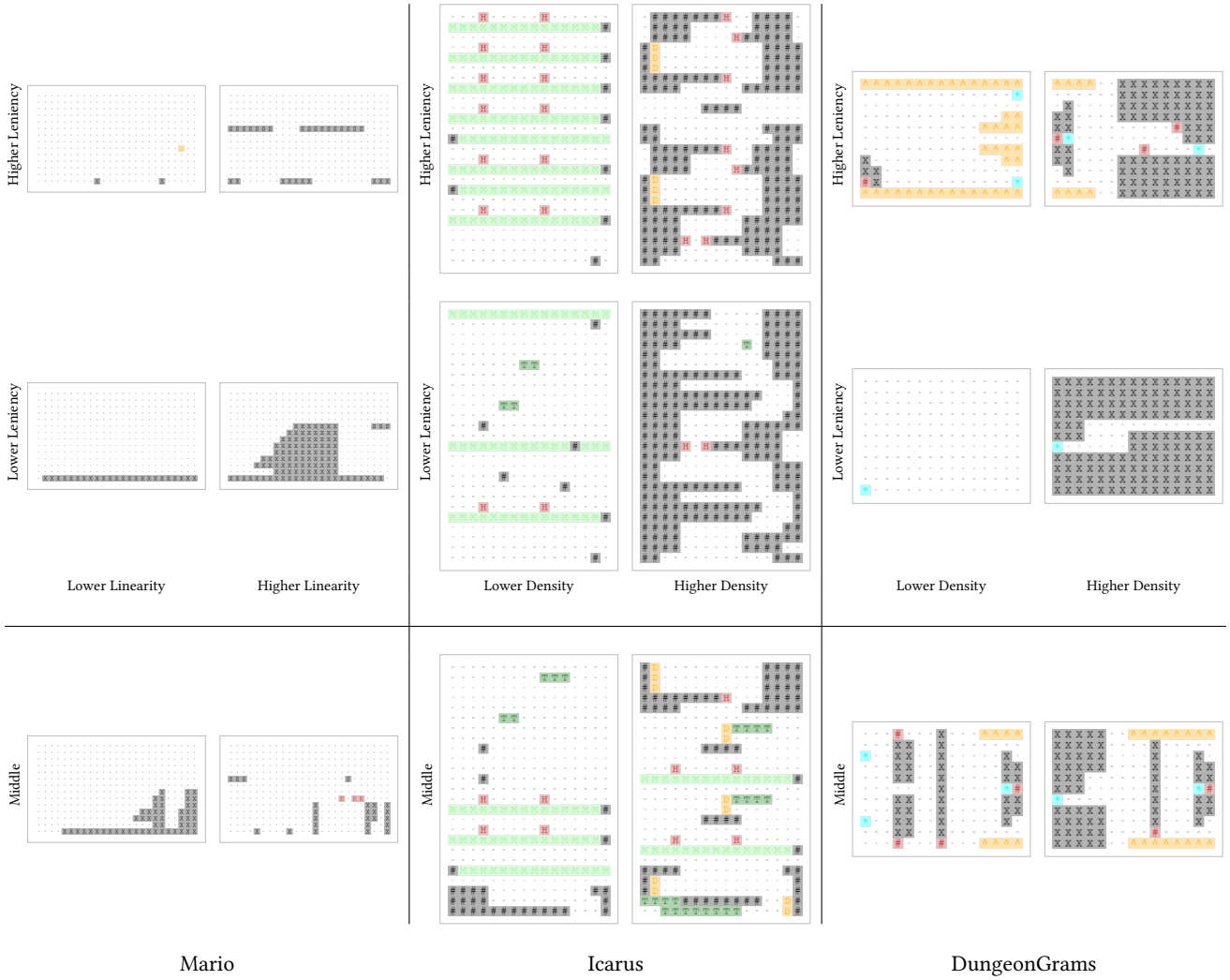

Figure 2: Example segments generated with Gram-Elites for all three games. The top shows segments from around the four corners of the MAP-Elites grid to the extent found for each game. The bottom left of each is a simple level, based on behavioral characteristics, and top right is more complex. The bottom shows segments from near the middle of those found in the grid.

either side of the point for both parent segments. Crossover is run with both segments with only one difference: we build an n-gram connection between the two segments. See Figure 1 (bottom).

Both of the n-gram operators can increase the size of a sequence. The case for increasing the size of a sequence is shown in Figure 1. To enforce a fixed sequence length, slices past the maximum length are truncated for both operators.

5 GAMES

In this work, we use three different games: *Mario* and *Icarus*, platformer games, and *DungeonGrams*, a roguelike dungeon game we developed. All three games use the same two-part fitness function $f(s)$. In the first part, an agent plays the game, and returns a completability score $C(s) \in [0, 1]$ for any level segment s. This represents the percent that an agent can complete segment s. In all games, the segment is padded with slices on each side when testing completability, as a proxy for the segment being completable as part of a larger level. The second part is the number of bad n-grams in the level, $B(s) \in \mathbb{N}_0$. A bad n-gram is the result of an unseen prior or an unknown output given a valid prior. The final fitness calculation of a segment is $f(s) = B(s) + 1 - C(s)$. In this work, MAP-Elites aims to minimize $f(s)$; a segment s is *usable* if $f(s) = 0$.

5.1 Mario

N-gram. The n-gram for *Mario* uses the *Super Mario Bros.* levels in the Video Game Level Corpus [16] that are not underground, do not have moving platforms (the grid representation does not capture the behavior well), and do not have springs (they are not present in the processed map files). The length of all generated segments is 25. Based on the work from Dahlskog et al. [3], we use 3-grams.

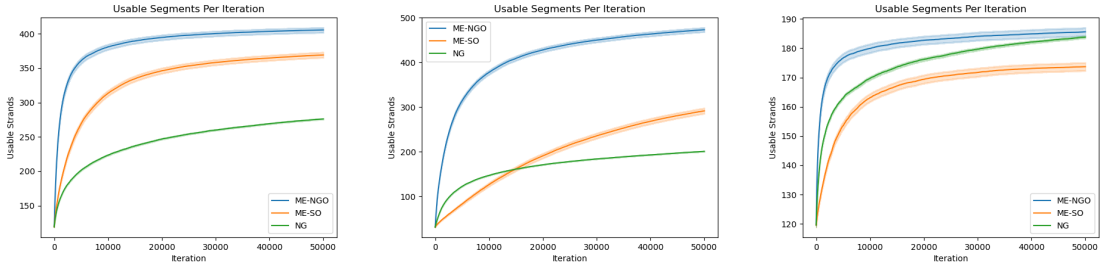

Figure 3: Plots of usable segments per iteration for each game. Plots show the average and 95% confidence interval for 100 runs.

Completability agent. Completability of *Mario* levels is evaluated using a modified version of the Summerville A* agent [16] which now finds the furthest point that an agent can reach. $C(s)$ is the maximum x-coordinate the agent reached divided by the number of columns in the level.

Behaviors. The behavioral characteristics are linearity and leniency [13]. Linearity is calculated by finding the minimum height for every column and finding the line of best fit. For every minimum height, the absolute value of the difference between it and the line of best fit are calculated. This is divided by the maximum linearity for a level segment to give a percentage. Leniency is calculated based on whether a column contains an enemy (+1/2) or contains a gap (+1/2) for a max score of 1; for a score of 1, a column must contain an enemy and a gap, not two enemies. This value is divided by the number of columns in the level to get a percentage. We use a resolution of 40 for each behavior.

5.2 Icarus

N-gram. The n-gram for *Icarus* uses the *Kid Icarus* levels in the Video Game Level Corpus [16]. *Icarus* uses rows for slices and $n = 2$ for more variety based on past work [2]. The height of all generated segments is 25.

Completability agent. Completability is evaluated using a modified version of the Summerville A* agent [16] which now finds the highest point and can wrap around in the x-direction. $C(s)$ is the highest y-coordinate the agent reached divided by the number of rows in the level.

Behaviors. The behavioral characteristics are density (percent of solid blocks) and a modified leniency. Every row is evaluated for containing a door (+1/3), a moving platform (+1/3), and an enemy (+1/3) for a max score of 1. The final leniency is divided by the number of rows in the level to give a percentage. We use a resolution of 35 for each behavior, with a maximum of 0.5 for each dimension.

5.3 DungeonGrams

DungeonGrams is a roguelike built for this project. The player traverses a top-down dungeon to reach the exit. Movement is turn-based on a discrete grid. For the player to win, they must hit every switch to unlock a portal and then go through the unlocked portal. Along the way, the player will have to correctly navigate, avoid spikes, and avoid enemies. The enemies can move every other turn. Each enemy stays near its spawn point and will move toward the player if the player is within a few tiles of that point; otherwise, the enemy will move back to their spawn point. If the player collides with a spike or enemy, they lose.

Training levels. We manually designed a set of 44 small levels for *DungeonGrams* with 11 rows each. We intentionally designed these levels to have repeating patterns that an n-gram model could make use of. All generated segments are length 15. *DungeonGrams* used 3-grams.

Completability agent. The *DungeonGrams* agent uses an A*-based pathfinding algorithm to hit the switches, avoid enemies and spikes, and reach the exit. The state space is large enough that a full search becomes too time-consuming. For quicker evaluation, we made the agent greedy and put in a search space limit based on the size of the level being solved. Completability is 1 if the agent can beat the level. Otherwise, completability is $0.9 * ((ag_{sw} + (ag_{mx}/lv_w))/(lv_{sw} + 1))$, where ag_{sw} is the number of switches the agent hit, ag_{mx} is the maximum x-coordinate the agent reached, lv_w is the width of the level, and lv_{sw} is the number of switches in the level. There are cases where the agent can get all the way to the right of a level and hit all the switches but still not beat a level, and scaling by 0.9 distinguishes these from when the agent does beat the level.

Behaviors. The behavioral characteristics are density (percent of solid blocks) and a modified leniency. Every column is evaluated for containing an enemy (+1/3), a spike (+1/3), and a switch (+1/3) for a max score of 1. Note that the existence of two enemies in the same column would only add 1/3 and not 2/3 to the score. The final leniency is divided by the number of columns in the level to give a percentage. We use a resolution of 20 for each behavior, with the one modification that the max leniency is 0.5 instead of 1 which tightens the search in that dimension. A smaller resolution is used since a smaller segment size is required for speed.

6 EVALUATION

To evaluate n-gram operators, we tested MAP-Elites with n-gram operators (ME-NGO, or Gram-Elites), MAP-Elites with standard operators (ME-SO), and MAP-Elites binning with only n-gram generation (NG) on *Mario*, *Icarus*, and *DungeonGrams*. Each ran a total of 50,000 iterations with an initial population, generated with an n-gram, of 500. To get an average, we ran each a hundred times.

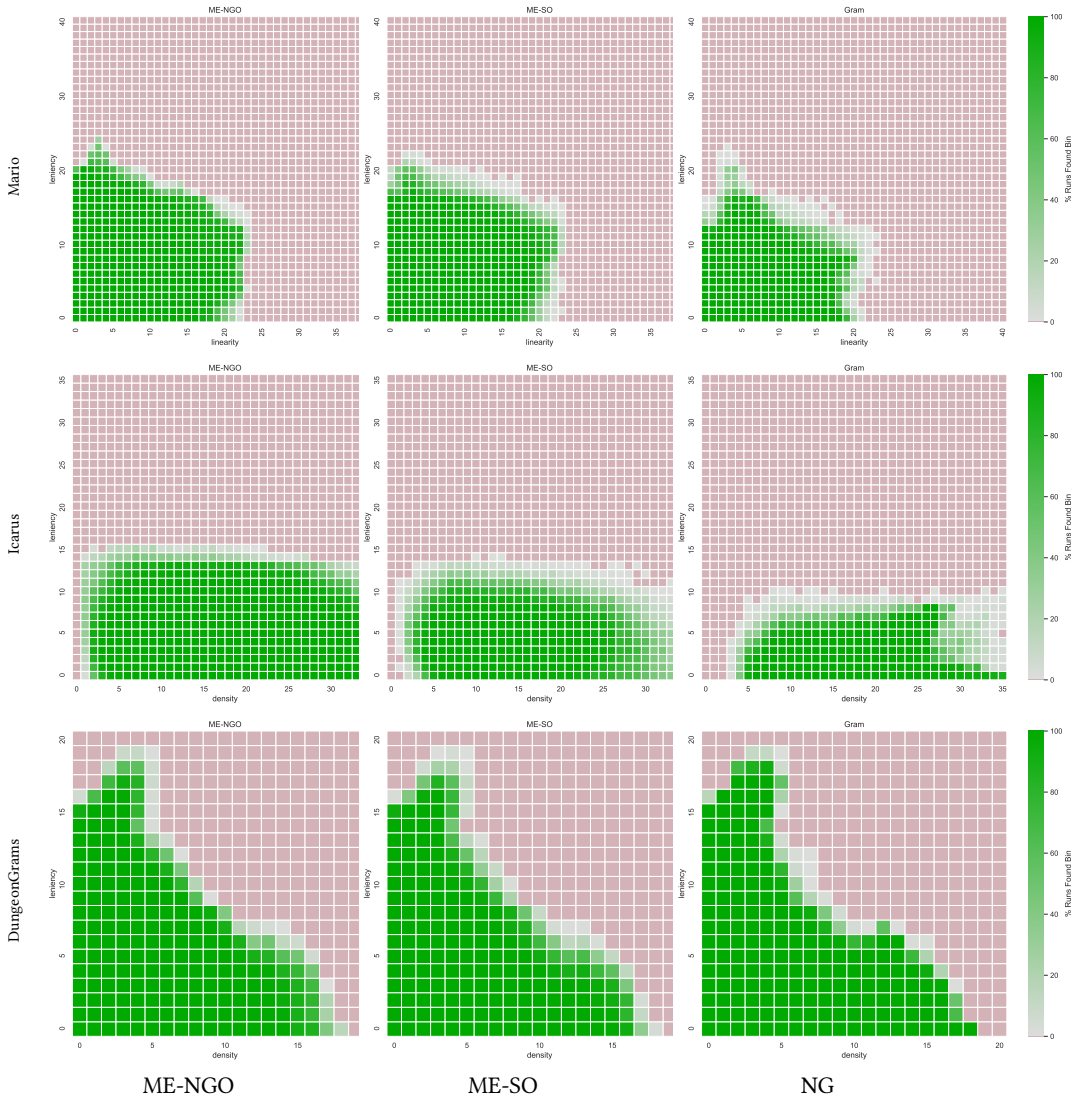

Figure 4: MAP-Elites grid for all three algorithms and games over 100 runs. The value is the percentage of runs that found the bin.

For ME-SO and ME-NGO, two parents were selected randomly and cross-over was always performed. Each child sequence had a 2% chance of mutation. NG randomly generated a new segment using the n-grams each iteration. For fitness calculation, $\forall s : B(s) = 0$ for NG and ME-NGO.

Example segments generated with Gram-Elites can be seen in Figure 2. For *Mario*, in the segment with higher linearity and higher leniency, the player's path appears fairly linear. This occurs because the linearity calculation is not based on the path but the lowest block in each column. For *Icarus*, higher leniency and lower density shows that repeating patterns are sometimes used to reach a target behavioral characteristic. For *DungeonGrams*, we can see that spikes can be used to get higher leniency.

Figure 3 shows the average number of usable bins found after each iteration for the hundred runs. For *Mario*, NG is the worst performing model. It does not appear to converge at the max iteration count. ME-SO quickly finds levels and has diminishing returns around 20, 000 iterations. ME-NGO is the fastest to find the most levels and also has diminishing returns around 20, 000 iterations. In *Icarus*, ME-SO is slow to find levels and does not appear to converge by 50, 000 iterations. However, ME-NGO finds the most levels in the fewest iterations. NG is faster to find levels than ME-SO initially but is overtaken after 20, 000 iterations. Lastly, for *DungeonGrams*, ME-SO performed the worst. ME-NGO converged at about 20, 000 iterations. NG did not appear to converge after 50, 000 iterations and caught up to ME-NGO.

Figure 4 shows the MAP-Elites plots for the hundred runs. Each bin shows the percent of times it was filled with a usable elite. For *Mario*, ME-NGO and ME-SO fill a similar space. ME-NGO, though, does fill in a larger space than ME-SO and finds more levels on average. Both clearly outperform NG. For *Icarus*, the difference between the algorithms is most clear. ME-NGO fills in the largest portion of the grid, and is close to filling in the horizontal axis. ME-SO fills in less. It is also the most variable in terms of whether or not a level is found. NG fills in less space than both ME-SO and ME-NGO. *DungeonGrams* has the least variance between the three methods. ME-NGO fills in slightly more space than the other two but not by much. This may partly be due to the use of a smaller resolution.

A common metric used to assess MAP-Elites is the QD-Score [11]. This score is the sum of the highest fitness values found in every bin in the grid. Its goal is quantify both quality and diversity found after a MAP-Elites run. We do not use it for two reasons. First, we are minimizing, not maximizing, which means a large QD would be bad in our case. Second, quantifying an empty bin as the max number of bad n-grams plus one could be overly harsh. The combination of these yields the fact that a QD-score would be misleading in our use case, and thus we did not use it.

7 CONCLUSION

In this work, we present Gram-Elites: an extension to MAP-Elites that uses n-gram based population generation and n-gram operators for mutation and crossover. We compare Gram-Elites to two baselines: MAP-Elites with standard operators and n-gram generation with placement into a MAP-Elites grid. We test with three games and find that Gram-Elites finds more usable segments, in fewer iterations.

One property of the current approach is that the n-gram operators do not take into account the likelihood of a particular sequence according to the n-gram, just that it can be generated. This may allow it to generate more unlikely "extreme" sequences in those areas of the bin, but also not reflect the distribution of slices in the training data. Incorporating the likelihood of a sequence being generated as a MAP-Elites behavior might help address this.

For Gram-Elites to work, there must be a path from the start prior to the end prior. In the case of *Mario* and *DungeonGrams*, both n-grams are strongly connected; there exists a path from every prior to every prior. In the case of *Icarus*, though, the n-gram used has "dead starts" or priors with no incoming edges. Because the current implementation chooses mutation and crossover points from the middle, these dead starts can only appear at the start of any segment. This, though, is problematic since no segment will ever have a different start prior. In future work we plan to address this by modifying crossover and mutation. When this change is made, the n-gram built for *Icarus* will, in theory, yield segments that cannot be connected. To address this we will prune the n-gram to be strongly connected.

To extend this work, we want to build a method to analyze an input corpus for different sizes of n-grams. The analysis will find the best n to produce recognizable structures without memorization. It may also be interesting to characterize the performance of ME-NGO relative to ME-SO based on structural properties of the n-gram, such as the number of outputs for each prior. More immediately, we intend to use Gram-Elites to build larger levels by connecting segments in the MAP-Elites grid, and improve the linearity metric for *Mario* to use naive path-finding.

ACKNOWLEDGMENTS

Parts of this work were supported by the La Comunidad Latina En Acción Scholarship from Northeastern University's Latinx Student Cultural Center.

REFERENCES

[1] Michael Cook, Simon Colton, and Jeremy Gow. 2016. The ANGELINA videogame design system—part I. *IEEE Transactions on Computational Intelligence and AI in Games* 9, 2 (2016), 192–203.
[2] Seth Cooper and Anurag Sarkar. 2020. Pathfinding agents for platformer level repair. In *Proceedings of the Experimental AI in Games Workshop*.
[3] Steve Dahlskog, Julian Togelius, and Mark J. Nelson. 2014. Linear levels through n-grams. In *Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services*. 200–206.
[4] Matthew C. Fontaine, Ruilin Liu, Ahmed Khalifa, Jignesh Modi, Julian Togelius, Amy K. Hoover, and Stefanos Nikolaidis. 2020. Illuminating Mario scenes in the latent space of a Generative Adversarial Network. *arXiv:2007.05674 [cs]* (Dec. 2020).
[5] Daniele Gravina, Ahmed Khalifa, Antonios Liapis, Julian Togelius, and Georgios N Yannakakis. 2019. Procedural Content Generation through Quality Diversity. *arXiv:1907.04053* (2019).
[6] Ahmed Khalifa, Michael Cerny Green, Gabriella Barros, and Julian Togelius. 2019. Intentional computational level design. In *Proceedings of The Genetic and Evolutionary Computation Conference*. 796–803.
[7] Ahmed Khalifa, Scott Lee, Andy Nealen, and Julian Togelius. 2018. Talakat: Bullet hell generation through constrained map-elites. In *Proceedings of The Genetic and Evolutionary Computation Conference*. 1047–1054.
[8] Steven Orla Kimbrough, Gary J Koehler, Ming Lu, and David Harlan Wood. 2005. Introducing a feasible-infeasible two-population (FI-2Pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch. *European Journal of Operational Research* (2005).
[9] ManYat Lo and Simon M Lucas. 2006. Evolving musical sequences with n-gram based trainable fitness functions. In *2006 IEEE International Conference on Evolutionary Computation*. IEEE, 601–608.
[10] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. *arXiv:1504.04909 [cs, q-bio]* (April 2015).
[11] Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. 2016. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI* 3 (2016).
[12] Noor Shaker, Julian Togelius, and Mark J Nelson. 2016. *Procedural content generation in games*. Springer.
[13] Gillian Smith and Jim Whitehead. 2010. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. 1–7.
[14] Sam Snodgrass, Adam Summerville, and Santiago Ontañón. 2017. Studying the effects of training data on machine learning-based procedural content generation. In *Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.
[15] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. 2018. Procedural content generation via machine learning (PCGML). *IEEE Transactions on Games* 10, 3 (2018), 257–270.
[16] Adam James Summerville, Sam Snodgrass, Michael Mateas, and Santiago Ontanón. 2016. The VGLC: The video game level corpus. *arXiv:1606.07487* (2016).
[17] Oliver Withington. 2020. Illuminating Super Mario Bros: quality-diversity within platformer level generation. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*. 223–224.