Guiding Generative Graph Grammars of Dungeon Mission Graphs via Examples

Abdelrahman Madkour,[1] Stacy Marsella, [1] Casper Harteveld, [1] Magy Seif El-Nasr [2] Jan-Willem van de Meent [1]

[1] Northeastern University
[2] UC Santa Cruz
madkour.a@northeastern.edu, s.marsella@northeastern.edu, c.harteveld@northeastern.edu, mseifeln@ucsc.edu, j.vandermeent@northeastern.edu

Abstract

Generative Graph grammars are an established technique for procedural content generation (PCG) of the mission space of dungeon levels, as they allow for explicit specification of design intention. However, they are difficult to control beyond the initial specification. Defining effective grammars requires both design expertise and familiarity with how grammars work, and even then the resulting grammar is not guaranteed to generate missions that fit a designer's intention. In this paper, we propose a system that attempts to allow designers to affect an existing grammar's generative space by allowing designers to control the probability distribution induced by the grammar. The system does this by learning the parameters of a probabilistic graph grammar from examples. Designers can control the generation of these examples via specification of assessment criteria, and a threshold above or below which the output is generated. We conclude with a demonstration of the efficacy of the system in shifting the distribution of the generative space.

Introduction

Procedural content generation (PCG) in games is a growing field in game AI that has received increasing interest in recent years (Liapis 2020). It promises to reduce the burden of game development by providing tools that assist with generating game artifacts such as levels. Designing levels is a creatively involved process that is difficult to do well, even for specialized experts. Therefore, despite this growing body of work, automatically creating levels that are of similar quality to those made by human designers remains a challenge (Rodriguez Torrado et al. 2020).

Because of this challenge, many techniques opt to have their generative process be as transparent and controllable as possible, such that game developers can adjust it if needed. One approach to make the generation process interpretable by level designers is graph grammars. Graph grammars are an established framework for visual programming (Rozenberg 1997), and were popularized for use in the context of dungeon level generation by Dormans and Bakkes (2011).

Graphs are a common representation for content generation in other contexts as well (Li and Riedl 2015; Londoño and Missura 2015), making graph grammars potentially widely applicable. However, coming up with a graph grammar that captures the intent of the designer is a difficult and time-consuming task. Predicting the effects of rules on the generative space of the grammar is hard and error-prone. Oftentimes, unintended consequences of rules result in undesirable artifacts, such as unplayable levels. To avoid this, the design of grammars focuses on guaranteeing playability above all else.

However, there are other design considerations that are taken into account when coming up with a dungeon level. For example, Adams et al. (2002) outlines that low-quality levels suffer from the 'pointless area' problem, where entire sections of the level do not have any reward to the player and are not part of the main path to complete the level. Or a level may be completable but trivially so, requiring the player to traverse one or two rooms to reach the end. Such considerations are difficult to capture in grammar rules that must also guarantee playability.

Therefore, some approaches opt to use a more general grammar and let designers control which rules are applied either manually step-by-step (Dormans and Bakkes 2011) or through a list of which rules to apply and in what order, which Dormans (2011) refers to as recipes. These recipes serve as the means by which designers restrict the generative space of the grammar to areas that fit their design intent. In other words, recipes adjust the probability distribution induced by the grammar such that the grammar is less likely to generate undesirable graphs, and more likely to generate desirable ones. Lavender (2016) demonstrates how this could be done by using the grammar given in Dormans and Bakkes (2011); by utilizing different recipes, they were able to generate different kinds of dungeons without altering the original grammar. These recipes, however, operate in the space of the grammar, and therefore requires designers to be familiar with how grammars work.

Ensuring design criteria are met is a common concern for many level PCG techniques, especially those that rely on machine learning algorithms (Summerville et al. 2018). There have been attempts to address this by reducing the likelihood of a generator to produce undesirable levels (Summerville et al. 2016; Di Liello et al. 2020). Such efforts have yet to be extended to graph grammars. In fact, to the best of our knowledge, no previous attempt has been made to adjust an existing graph grammar towards captur-

Copyright © 2021 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-3217/paper8.pdf

ing design considerations via learning from designer made examples or examples generated by the grammar.

In this paper, we propose a system for adjusting the probability distribution over the mission graphs of dungeons induced by a graph grammar via examples generated by the grammar. To facilitate these adjustments, we propose an extension to the existing formalization of graph grammars in Dormans and Bakkes (2011). We alter their definition to include explicit probabilities at different levels of specificity. This provides the system with parameters to adjust, and thus provides a knob by which designers can control the output of the grammar after its specification. These parameters are learned from examples, the generation of which is controlled by designers. We evaluate our approach, demonstrating its ability to shape a grammar's distribution towards that of the given examples, thereby making the grammar generate graphs that are similar to the given examples.

Related Work

Previous work on dungeon level generation has largely focused on search-based techniques (van der Linden, Lopes, and Bidarra 2014). These approaches require specification of a fitness function that summarizes the quality of the level. This function is often difficult to come up with, even for domain experts. In most of these techniques, it is also the primary means of control designers have over the generation process. Alvarez et al. (2018) address this by providing more control to the designer via a co-creative system. This system provides suggestions as the designer is creating a dungeon. Dungeons in this system are in a tile-based representation. The system then identifies two levels of patterns, micro and meso. Micro patterns are concerned with base tiles and how they relate to the tiles surrounding it. Meso-patterns on the other hand, consider the relation of micro-patterns and meso-patterns. The suggestions the system proposes use these tile-based patterns for chunks of the dungeon, whilst our work focuses on a high level depiction of the flow of the level that is specified by graphs.

Generation of this more abstract notion of a level via graph grammars was proposed by Dormans and Bakkes (2011). They split the generation of dungeon levels into two distinct processes; 'mission' generation and 'space' generation. The mission, represented as a graph, encodes the desired trajectory of the player through the level, whilst the space specifies where that experience takes place. Dormans and Bakkes (2011) use shape grammars to turn a mission graph into level geometry, whilst other work has explored using GANs Gutierrez and Schrum (2020). We focus in this paper on the generation of mission graphs.

The graph grammars proposed by Dormans and Bakkes (2011) constrains the generative space of the grammar with what is referred to as recipes (Dormans 2011). The recipes dictate what rules should be applied and in what order. These recipes are hand-coded by the designers and as such they increase the specification requirements of this approach. In addition, these recipes are difficult to come up such that the resulting graphs are playable and meet specific design criteria. We extend this work by allowing the grammar to be guided by examples, which allows designers to think in terms of the design of the mission graphs they wish to generate and not grammar rules.

Outside of dungeon generation, graph grammars have found other uses in the PCG literature. Londoño and Missura (2015) proposed a graph representation of *Super Mario Levels* and a system to learn a probabilistic graph grammar from existing levels. Hauck and Aranha (2020) extend this work and propose a system for generating new Mario levels that utilizes the learned structure of the graph representation. This work focuses on context-free graph grammars (CFGGs). These are grammars that constrain the left-hand side of a production rule to be a single non-terminal vertex, whilst context-sensitive graph grammars (CSGGs) are more general and allow for the left-hand side to be a graph comprised of both non-terminal and terminal vertices. Adams et al. (2002) showed that unlike string based grammars, CFGGs are significantly less expressive than CSGGs. There are many rules that can be expressed in a CSGG that cannot be reduced to rules in a CFGG. This severely limits the capability of context-free grammars as a generative method. Due to this limitation, our work and that of Dormans and Bakkes (2011) focuses on context-sensitive graph grammars.

Another system that uses context-sensitive graph grammars is that proposed by Valls-Vargas, Zhu, and Ontañón (2017). They utilize multiple stochastic context-sensitive graph grammars to generate puzzle levels for an educational game. Each rule in the grammar has a weight associated with it and a discount factor. When a rule is used, its weight is decreased by the discount factor. Despite parameterizing the rules of the grammars with weights, they did not explore learning these parameters from examples.

Though little work has explored adjusting a graph grammar based on examples, there has been work on altering other types of grammars. Shaker et al. (2012) proposed a system for evolving grammar rules for a context-free grammar that generated Mario levels. This system operated on string-based grammars and adjusted the rules based on hand-written fitness functions and not training data.

Dang et al. (2015) do propose a framework that adjusts context-free shape grammars based on examples. The type of shape grammars used in this work are established grammars for automatically generating 3D objects such as buildings, trees and tables. They propose a human-in-the-loop framework that allows designers to designate the distribution over the generated 3D models of these grammars via scoring of examples. This framework is similar to our system but operates on context-free shape grammars, we focus on a more expressive type of grammars; context-sensitive graph grammars. Thus, we parameterize our grammar formulation differently.

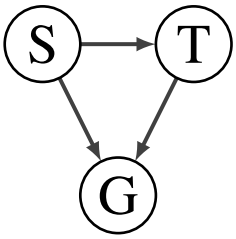
Methodology

The Graph Grammar System

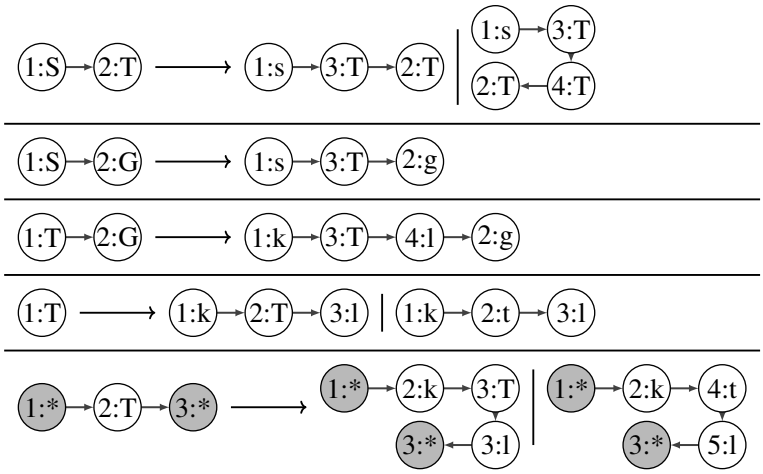
There are many types and formalizations of graph grammars (Rozenberg 1997). For this work, we consider the graph grammar system proposed by Dormans and Bakkes (2011). These grammars operate by applying rules to rewrite a graph

Label	Meaning
S/s	Start
T/t	Task
G/g	Goal
k	key
l	lock
*	Wildcard

(a) Node Labels



(b) Initial graph



(c) The set of production rules

Figure 1: An example of a simple graph grammar for generating dungeon mission graphs;a) shows the node labels of graphs in the grammar b) shows the starting graph of the grammar whilst c) shows the grammar's production rules. The dark grey nodes in the production rules are wildcard nodes that can match with any node. Adapted from Shaker, Togelius, and Nelson (2016, Chapter 5)

until no rule is applicable or a maximum number of applications is met. Figure 1 is a toy example of such a grammar. They are defined by an initial graph that gets altered according to the production rules. Rewriting a graph works by first finding a subgraph in the original graph that matches the left-hand side graph of a production rule. For two graphs to match, they must have the same graph topology and the same node labels. Figure 1a shows the node labels of the toy grammar. Note that graphs in production rules have nodes that can be labeled as 'wild-card' which means they can match with a node that has any label. In the toy grammar, this is represented by the '*' character. Next, we mark that subgraph by copying the markers of the nodes from the left-hand side. These markers indicate which nodes will be replaced by which nodes in the right-hand side of the rule. Then we check if there are nodes or edges in the subgraph that exist in the right-hand side of the rule, if not then we delete them. Then we add any nodes and edges in the right-hand side that do not exist in the left-hand side. Finally, we remove all the markers from the subgraph.

Proposed Extension

To effectively control the generative space of the grammars, we need parameters that we can adjust. To this end, we extend the graph grammar system described earlier to incorporate explicit probabilities for production rules. There are two types of these probabilities: the probability of which of the possible left-hand-side graphs is chosen and the probability of which right-hand-side graph of a rule is being chosen. The probability of which right-hand-side gets chosen is what previous work on probabilistic grammars has considered (Dang et al. 2015). However, this alone does not fully parameterize the generation process, and a level of uncontrolled stochasticity remains.

To illustrate this, consider the toy grammar in Figure 1. We see that there are multiple left-hand side graphs that we can match to the starting graph. Initially all rules are applicable, if we select the first rule to apply on the initial graph then the second is no longer applicable, and if we select the second rule then the first would no longer be applicable. The probability of which left-hand-side we pick controls which of these scenarios plays out in the generation process. Recipes as defined in (Dormans 2011) let the designer specify this directly by stating what rules ought to be selected and in what order. In contrast, our approach attempts to learn this from examples.

When defining a grammar, it is tractable to specify the probabilities of which right-hand-side gets chosen in a rule, the same cannot be said of the probabilities of which left-hand-side gets picked. These probabilities are defined based on which rules are applicable, and thus changes from one step of the generation process to the next. Therefore, we set the distribution of these probabilities to be initially uniform and learn it from examples. To be able to do that, we need a way of keeping track of which left-hand-side in the applicable rules is selected and the set of other left-hand-side graphs that could have been picked. Thus, for each graph generated by the grammar, we keep track of the production rules that were applied and what production rules could have been applied in what we call a *generation chain*. This generation chain keeps track of what the applicable production rules were at every step of the generation process, and which production rule in that set was selected.

Updating Grammar Parameters Once we have a training set, we can observe the production rules used to generate each graph and use that information to update the production rule probabilities. First, we update the probabilities of the right-hand side graphs according to what right-hand side was selected in the training data. This involves counting how

many times a given right-hand-side graph was chosen and dividing it by how many times its corresponding rule was applied in the training set. Formally, given a set of graphs $G_X = \{G_1 \ldots, G_n\}$, we can update the i-th right-hand side probability of production rule r, $p(r : G^{\text{left}} \rightarrow G_i^{\text{right}} \mid G_X)$ by setting it equal to:

$$\frac{C(r : G^{\text{left}} \rightarrow G_i^{\text{right}} \mid G_X)}{\sum_{G_a^{\text{right}}} C(r : G^{\text{left}} \rightarrow G_a^{\text{right}} \mid G_X)} \tag{1}$$

where $C(r : G^{\text{left}} \rightarrow G_a^{\text{right}})$ is the count of how many times G_l has been replaced with G_a in the training set G_X.

We then update the probabilities of the left-hand side graphs. Recall that we defined a generation chain that kept track of which production rules were chosen and which were available at the time a rule was picked. We go through the generation chains of all the graphs in our data-set and keep track of which sets of options were available. We then count how many times a given left-hand side graph was chosen in a given set of applicable rules and divide it by how many times that set of applicable rules shows up in the training set.

Formally define P_d to be the set of production rules that are available at a given depth d of the generation chain, $C(P_d \mid G_X)$ to be the count of how many times this set appears at *any* depth in the chains that generate the training data-set, G_X and pr $\in P_d$ to be a production rule in that set. We update the probability of this production rule by

$$p(\text{pr} \mid P_d) = \frac{C(\text{pr} \mid G_X, P_d)}{C(P_d \mid G_X)} \tag{2}$$

where $C(\text{pr} \mid G_X, P_d)$ is the count of how many times pr was chosen in G_X when the options for rule applications were P_d.

Generating Examples

Our system attempts to help designers inform grammars of their design considerations by providing examples from which a grammar can learn. These examples ought to capture the desirable qualities that a designer wants the grammar to replicate. The best means by which designers can generate those examples is a research question in its own right. For the purposes of this paper, we assess the grammar's output via a scoring function and then specify a threshold, whereby any graph that exceeds or falls short of the threshold is added to the training set. For example, a designer may give a function that measures how easy a mission graph is to play through and use a certain threshold to indicate the desired difficulty of graphs to be generated by the grammar. We take this approach as it clearly demonstrates how the generative space shifts towards the provided examples.

There are many scoring functions one can define, in this paper we consider metrics used in previous work to analyze dungeon level generators. There does not exist an established set of metrics for dungeon levels like there is for platformer levels (Horn et al. 2014). Therefore, we follow the example set by Smith, Padget, and Vidler (2018) and adapted the metrics put forth by Lavender (2016), which were in turn inspired by Smith and Whitehead (2010):

- **Mission Linearity:** which captures the linearity of the mission structure. We calculate it as follows:

$$\frac{\#\text{of nodes on shortest direct path to the end}}{\text{Total nodes in graph}}$$

- **Map Linearity:** which captures the linearity of the map layouts. In the graphs this is seen by how many outgoing edges a node has. We calculate it as follows:

$$\frac{\#\text{of single exit rooms} + (0.5 \times \#\text{of double exit rooms})}{\text{Total rooms with exits}}$$

- **Leniency:** which captures how easy the level is to play based on the amount of rooms where the player is likely to encounter danger. We calculate it as follows:

$$\frac{\#\text{of safe rooms}}{\text{Total rooms}}$$

- **Path Redundancy:** which captures how many useless rooms there are in the graph. Useless rooms in our context are defined as nodes that do not have any out-going edges and do not provide the player with any reward in themselves. We calculate it as follows:

$$\frac{\#\text{of useless rooms}}{\text{Total rooms}}$$

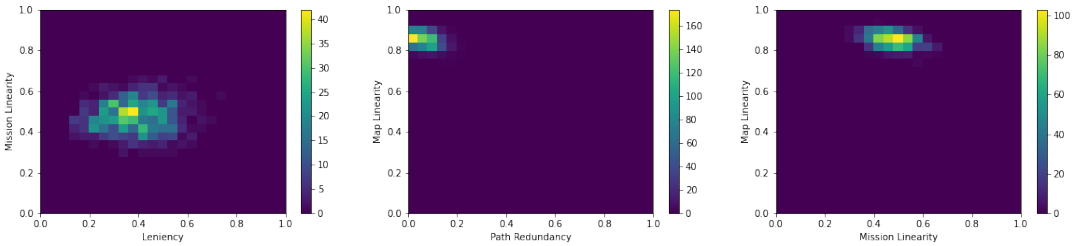
Results

For our experimental evaluations, we utilized the grammar given in Dormans and Bakkes (2011). Code of our system and the experiments we ran can be found on Github [1]. Following the approach taken by Smith, Padget, and Vidler (2018) and Lavender (2016), we generate 1000 mission graphs of the unaltered grammar and calculated their scores according to the four metrics we described earlier. Figure 2a shows the expressive range heat-map of the unaltered grammar, without the use of recipes. Next we picked a threshold for Leniency, Path Redundancy and Mission Linearity. We then generate our training set by choosing graphs that exceed the threshold, train the grammar on that training set and visualize the heat-map of the resulting grammar. Figure 2b shows the expressive range heat-map of the altered grammar. We see that the generative space has moved towards the values above the threshold.

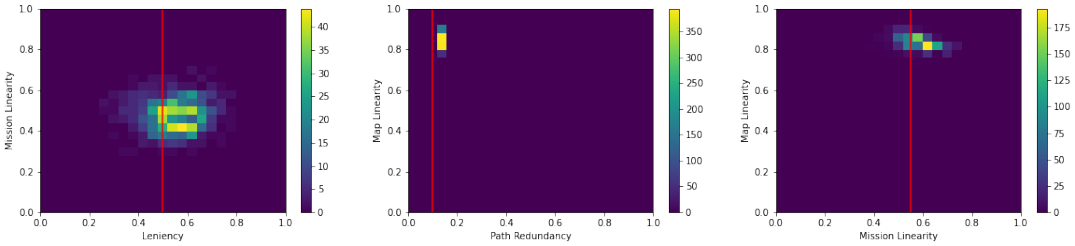
We repeated the same process but changed the threshold value and picked examples that were below the threshold. Figure 2c shows the expressive range heat-map of the resulting grammar. Once again, we see that the generative space has shifted towards the specified value.

We additionally ran 100 trials, where we repeated the process described above and kept track of how many graphs were above or below the threshold before and after training. Table 1 summarizes our findings. Despite not guaranteeing that the grammar will only generate graphs that meet the desired threshold, we greatly reduce the number of graphs that fall outside it.

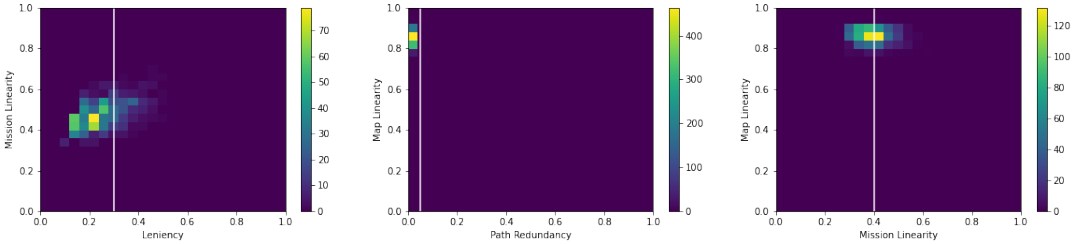
[1] https://github.com/a3madkour/pgg



(a) The expressive range across the metrics proposed in Lavender (2016) and used in Smith, Padget, and Vidler (2018) of the grammar before alteration.



(b) The expressive range of the grammar after being trained on examples that are above a threshold. The threshold is indicated by the red line.



(c) The expressive range of the grammar after being trained on examples that are below a threshold. The threshold is indicated by the white line.

Figure 2: The expressive range of grammars; 2a is for the unaltered grammar, whilst 2b-c for each of the design scenarios we considered.

Scenario	Before	After
Le above 0.5	123 (± 10.3)	684.4 (± 25.7)
PR above 0.1	182 (± 13.6)	612.5 (± 24.3)
ML above 0.55	122 (± 10.9)	719 (± 23.2)
Le below 0.3	240 (± 14.9)	762.3 (± 16.4)
PR below 0.04	325 (± 14.3)	700.8 (± 18.1)
ML below 0.4	112 (± 9.5)	591.3 (± 22.8)

Table 1: Average number of graphs (std) that were above or below a given threshold before and after grammar adjustment. Le is leniency, PR is path redundancy and ML is mission linearity.

Scenario	Le	PR	ML
Figure 2b	232	130	105
Figure 2c	245	340	101

Table 2: The size of the training set for each scenario used to train the grammars for Figure 2.Le is leniency, PR is path redundancy and ML is mission linearity.

Discussion

Our experimental analysis demonstrates that learning grammar parameters from examples successfully adjusted the distribution induced by the grammar. Thus, we demonstrated the feasibility of controlling generative graph grammars through example generation; which gives designers a way of adjusting a generative grammar towards their design intention without needing to think about grammar rules.

A limitation of the system is that examples that are used for training must be generated by the grammar. A promising avenue of future work is exploring the possibility of learning them from designer generated examples. Another would be looking into learning grammar rules from the examples. Only altering the probabilities of the system does not change the original generative space. It alters in which areas in the generative space the grammar is more likely to generate.

This work also calls for a user study that assesses what

mechanism designers prefer using to control distribution of the grammar. In the context of our system, this would entail figuring out what means of generating training examples designers prefer using, and which most facilitates thinking in terms of design intention as opposed to grammar rules. Ideally, designers only have to think in terms of the artifacts they wish to generate, not the logic or parameters of their generator. Designing well-formed and sufficiently generative grammars requires a different skill set than designing dungeons. Thus, providing designers with the tools that allows them to operate in the space of what they know well, and use it to adjust an existing grammar will improve the usability of graph grammars as a generative tool.

Recent work on guidelines for human-AI interaction suggest that for AI systems to better fit with human intention, more direct involvement is required (Amershi et al. 2019). Though some work has explored how to better explain the output of PCG systems (Cook et al. 2021), there is little on how to make them more directly specifiable and controllable by nontechnical designers. As Cook et al. (2021) describe, the parameters of generators often do not map directly to output metrics that are of relevance to designers. By allowing designers to operate in the space of the outcome of the generator, we position our work as a step towards reducing this mismatch of desired outcome and specification of PCG systems, and thus achieve more accessibility in controlling them.

Conclusion

We presented a system for adjusting the generative space of a graph grammar for dungeon generation based on examples. To do so, we proposed an extension of existing graph grammars to allow for the use of probabilities that can then be learned. Our experiments demonstrate that our approach is successful in shifting the generative space towards that of the generated examples. Finally, we discuss how generating a set of high quality examples is potentially an accessible method of providing designer control over generative grammars beyond initial specification.

References

Adams, D.; et al. 2002. Automatic generation of dungeons for computer games. *Bachelor thesis, University of Sheffield, UK. DOI= http://www. dcs. shef. ac. uk/intranet/teaching/projects/archive/ug2002/pdf/u9da. pdf* .

Alvarez, A.; Dahlskog, S.; Font, J.; Holmberg, J.; Nolasco, C.; and Österman, A. 2018. Fostering creativity in the mixed-initiative evolutionary dungeon designer. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, 1–8.

Amershi, S.; Weld, D.; Vorvoreanu, M.; Fourney, A.; Nushi, B.; Collisson, P.; Suh, J.; Iqbal, S.; Bennett, P. N.; Inkpen, K.; Teevan, J.; Kikin-Gil, R.; and Horvitz, E. 2019. *Guidelines for Human-AI Interaction*, 1–13. New York, NY, USA: Association for Computing Machinery. ISBN 9781450359702. URL https://doi.org/10.1145/3290605.3300233.

Cook, M.; Gow, J.; Smith, G.; and Colton, S. 2021. Danesh: Interactive Tools For Understanding Procedural Content Generators. *IEEE Transactions on Games* 1–1. doi:10.1109/TG.2021.3078323.

Dang, M.; Lienhard, S.; Ceylan, D.; Neubert, B.; Wonka, P.; and Pauly, M. 2015. Interactive design of probability density functions for shape grammars. *ACM Transactions on Graphics* 34(6): 1–13. ISSN 07300301. doi:10.1145/2816795.2818069. URL http://dl.acm.org/citation.cfm?doid=2816795.2818069.

Di Liello, L.; Ardino, P.; Gobbi, J.; Morettin, P.; Teso, S.; and Passerini, A. 2020. Efficient Generation of Structured Objects with Constrained Adversarial Networks. In Larochelle, H.; Ranzato, M.; Hadsell, R.; Balcan, M. F.; and Lin, H., eds., *Advances in Neural Information Processing Systems*, volume 33, 14663–14674. Curran Associates, Inc.

Dormans, J. 2011. Level Design as Model Transformation: A Strategy for Automated Content Generation. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, PCGames '11. New York, NY, USA: Association for Computing Machinery. ISBN 9781450308724. doi:10.1145/2000919.2000921. URL https://doi.org/10.1145/2000919.2000921.

Dormans, J.; and Bakkes, S. 2011. Generating Missions and Spaces for Adaptable Play Experiences. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3): 216–228. ISSN 1943-0698. doi:10.1109/TCIAIG.2011.2149523.

Gutierrez, J.; and Schrum, J. 2020. Generative adversarial network rooms in generative graph grammar dungeons for the legend of zelda. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, 1–8. IEEE.

Hauck, E.; and Aranha, C. 2020. Automatic Generation of Super Mario Levels via Graph Grammars. In *2020 IEEE Conference on Games (CoG)*, 297–304. doi:10.1109/CoG47356.2020.9231526.

Horn, B.; Dahlskog, S.; Shaker, N.; Smith, G.; and Togelius, J. 2014. A comparative evaluation of procedural level generators in the mario ai framework. In *Foundations of Digital Games 2014, Ft. Lauderdale, Florida, USA (2014)*, 1–8. Society for the Advancement of the Science of Digital Games.

Lavender, R. 2016. The Zelda Dungeon Generator: Adopting Generative Grammars to Create Levels for Action-Adventure Games.

Li, B.; and Riedl, M. 2015. Scheherazade: Crowd-powered interactive narrative generation. *Proceedings of the AAAI Conference on Artificial Intelligence* 29(1). URL https://ojs.aaai.org/index.php/AAAI/article/view/9782.

Liapis, A. 2020. 10 Years of the PCG Workshop: Past and Future Trends. In *International Conference on the Foundations of Digital Games*, FDG '20. New York, NY, USA: Association for Computing Machinery. ISBN 9781450388078. doi:10.1145/3402942.3409598. URL https://doi.org/10.1145/3402942.3409598.

Londoño, S.; and Missura, O. 2015. Graph Grammars for Super Mario Bros Levels. In *FDG*.

Rodriguez Torrado, R.; Khalifa, A.; Cerny Green, M.; Justesen, N.; Risi, S.; and Togelius, J. 2020. Bootstrapping Conditional GANs for Video Game Level Generation. In *2020 IEEE Conference on Games (CoG)*, 41–48. doi: 10.1109/CoG47356.2020.9231576.

Rozenberg, G., ed. 1997. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. USA: World Scientific Publishing Co., Inc. ISBN 9810228848.

Shaker, N.; Nicolau, M.; Yannakakis, G. N.; Togelius, J.; and O'neill, M. 2012. Evolving levels for super mario bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 304–311. IEEE.

Shaker, N.; Togelius, J.; and Nelson, M. J. 2016. *Procedural Content Generation in Games*. Springer Publishing Company, Incorporated, 1st edition. ISBN 3319427148.

Smith, G.; and Whitehead, J. 2010. Analyzing the Expressive Range of a Level Generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10. New York, NY, USA: Association for Computing Machinery. ISBN 9781450300230. doi: 10.1145/1814256.1814260. URL https://doi.org/10.1145/1814256.1814260.

Smith, T.; Padget, J.; and Vidler, A. 2018. Graph-based generation of action-adventure dungeon levels using answer set programming. In *Proceedings of the 13th International Conference on the Foundations of Digital Games - FDG '18*, 1–10. ACM Press. ISBN 978-1-4503-6571-0. doi: 10.1145/3235765.3235817. URL http://dl.acm.org/citation.cfm?doid=3235765.3235817.

Summerville, A.; Guzdial, M.; Mateas, M.; and Riedl, M. 2016. Learning player tailored content from observation: Platformer level generation from video traces using lstms. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 12.

Summerville, A.; Snodgrass, S.; Guzdial, M.; Holmgård, C.; Hoover, A. K.; Isaksen, A.; Nealen, A.; and Togelius, J. 2018. Procedural content generation via machine learning (PCGML). *IEEE Transactions on Games* 10(3): 257–270.

Valls-Vargas, J.; Zhu, J.; and Ontañón, S. 2017. Graph Grammar-Based Controllable Generation of Puzzles for a Learning Game about Parallel Programming. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17. New York, NY, USA: Association for Computing Machinery. ISBN 9781450353199. doi:10.1145/3102071.3102079. URL https://doi.org/10.1145/3102071.3102079.

van der Linden, R.; Lopes, R.; and Bidarra, R. 2014. Procedural Generation of Dungeons. *IEEE Transactions on Computational Intelligence and AI in Games* 6(1): 78–89. ISSN 1943-0698. doi:10.1109/TCIAIG.2013.2290371.