WITS UNIVERSITY

School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

Literature Review

Using NeuroEvolution of Augmenting Topologies and Novelty Search to Procedurally Generate Diverse Video Game Levels.

Michael Beukman

May 10, 2021

Supervised by Steven James & Prof. Christopher Cleghorn.

Contents

1 Introduction

Procedural generation of video game content refers to "creating game content automatically, through algorithmic means" (Togelius *et al.*, 2011). Many different types of content can be generated, among others platformer levels (Ferreira *et al.*, 2014), racing game tracks (Cardamone *et al.*, 2011), entire games (Cook and Colton, 2011) and weapons in games (Hastings and Stanley, 2009). This review mostly focuses on the generation of levels. There is a wide variety of methods that are commonly used to generate these levels, ranging from exhaustive search (Sturtevant and Ota, 2018), to using optimisation/search based methods (Togelius *et al.*, 2011) to using machine learning (Summerville *et al.*, 2018), and more.

The challenge of procedurally generating video game levels that players find exciting and fun is far from solved. A common problem with generated levels is that they can seem generic and that they do not capture a player's attention and interest them as much as a level that was designed by a developer/designer (Togelius *et al.*, 2013). Having access to high quality level generators would help game developers (especially small scale ones) create entertaining games, with a fraction of the effort and cost (Tutenel *et al.*, 2008; Hendrikx *et al.*, 2013). Procedural content generation (PCG) can also be used as a way to train reinforcement learning agents (Justesen *et al.*, 2018), and being able to generate more diverse levels will positively impact these agents.

The research question considered in this review is on how to combine novelty search and NeuroEvolution of Augmenting Topologies (NEAT) to procedurally generate diverse and high quality levels for video games in real time.

Using the above question as a guiding concept, we discuss the most applicable previous work on the topic of procedural level generation. We attempt to show that the problem of generating diverse levels in real time has not yet been adequately solved. Finally, we also attempt to show that how levels are evaluated and results are reported leaves something to be desired in terms of an objective, robust and game independent metric.

This review is structured based on different aspects of procedurally generating game content. The focus is largely on evolutionary methods, but other approaches are also discussed where appropriate.

Section 2 discusses some background related to genetic algorithms and some specific methods. Section 3 goes over how content is represented and in what space the different learning methods can search. In Section 4 we go on to discuss the different learning methods that are used and finally in Section 5 we discuss how the content is evaluated and how training signals are generated.

2 Background

Many of the methods discussed below use some aspects of evolutionary computing, which is a subclass of optimisation algorithms that attempt to mimic natural selection, as found in the natural world (Goldberg, 1989).

Usually genetic algorithms consist of a population of individuals, each possessing a genotype, which can be thought of as the individual's genes. This genotype impacts the phenotype, which can be thought of as the manifestation of the genotype in the problem domain. For example, when generating platformer levels, the genotype can be a single integer vector representing the height of platforms across the x-axis. The phenotype, then, is the actual level that has been generated using this specific genotype (Goldberg, 1989).

High performing individuals are combined using *crossover*, which is the process of combining two parents to form new individuals for the next generation. This new generation is also mutated slightly to facilitate exploration and prevent stagnation. In general, 'high performing' is quantified by the *fitness function*. For example, when maximising a function

$f(x)$, the function value could simply be the fitness function.
Using an example from Goldberg (1989), if our genotype is a 5 bit binary string and we have parents $A_1 = 01101$, $A_2 = 11000$, we can perform crossover by randomly choosing a number $k \in [1, l-1]$ where l is the length of the strings (5 in this case). We can create two new individuals by swapping the values of A_1 and A_2 from index k until the end of the string.

If $k = 3$, say, then this would result in two new individuals $A_1' = 01100$, $A_2' = 11001$. Mutation would simply flip bits with some small probability, resulting in (for example) $A_1^m = 01000$ (bit 3 flipped), $A_2^m = 11001$ (no change).
Now, say our problem was maximising the function value $f(x) = x^2$ for $x \in \mathbb{Z}, 0 \leq x \leq 31$. Then our representation (i.e. genotype) could be a 5 digit binary string (e.g. 01100) that represents x. The phenotype would then just be the number in decimal (e.g. 12) and the fitness will just be $f(x)$, (which is $12^2 = 144$).

Here we show the above example in table form.

$$\begin{tikzcd}
A_1 = 011\mathbf{01} \arrow[r, Rightarrow, "crossover"] \arrow[d, phantom, "\updownarrows"] & A_1' = 011\mathbf{00} \arrow[r, Rightarrow, "mutation"] & A_1^m = 01\mathbf{0}00 \\
A_2 = 110\mathbf{00} \arrow[r, Rightarrow, "crossover"] & A_2' = 110\mathbf{01} \arrow[r, Rightarrow, "mutation"] & A_2^m = 11001
\end{tikzcd}$$

Genotype	Phenotype	Fitness
01100	12	144
11001	25	625

2.1 NeuroEvolution of Augmenting Topologies

NeuroEvolution of Augmenting Topologies, referred to as NEAT, is a method where a genetic algorithm optimises the structure and weights of a neural network (Stanley and Miikkulainen, 2002). This method can perform just like a normal genetic algorithm, except that the genotype is the neural network weights and structure. This method enables complexity to gradually increase as the search process develops, leading to later generations being more complex than previous ones.

2.2 Novelty Search

Novelty search (Lehman and Stanley, 2011) is a different way of designing the fitness function for a genetic algorithm. Instead of pursuing a higher objective function value, novelty search only judges individuals based on how different or novel they are compared to the current generation and an archive of previously novel individuals. Novelty is defined as the average distance between an individual and its k closest neighbours in behaviour space. Distance can be defined in a domain-specific way, like absolute difference in our above example or euclidean distance in a higher dimensional space. Novelty search encourages agents to pursue novel and diverse behaviours. Even though there is no explicit incentive to actually achieve the goal, novelty search can still achieve competitive results, especially in deceptive fitness landscapes when using a traditional objective can lead to agents being stuck in local minima.

3 Level Representation

How a piece of content is represented in a specific method can impact the type of levels that method can generate, its efficiency as well as the ease of crossover and mutation when using a genetic algorithm.
Togelius *et al.* (2011) make a distinction between indirect encodings and direct encodings, where the dividing line is whether the representation size is proportional to the level size. This means that a direct encoding for a level of double the size requires a representation of double the size. Ferreira *et al.* (2014) used a simple, direct vector representation where the size of each vector was the same as the length of the map (each aspect of the map, e.g. enemies, coins, and ground used a different vector). To generate a map from this representation, one simply iterates over the vector and places the appropriate component at that spot. This is a direct encoding, where it is very easy to map between the representation and the level. Liapis *et al.* (2015) used a 2d array of tile types to represent a 2d grid world. This facilitated easy crossover, but it occasionally caused unplayable levels. The authors remedied some of these problems by using a simple, game dependant repair mechanism that removed some tiles if there were too many of a specific type.

Risi *et al.* (2016) used a more indirect representation, where each collectible game item (flower) was represented by a compositional pattern producing network (CPPN) (Stanley, 2007), which is a type of artificial neural networks where each node can use a different activation function. This network could generate a single item by outputting an RGB colour value for each given polar coordinate. Thus, the representation was actually the structure and weights of this network. This method has the potential to scale up, where larger items (more pixels or more layers) can be generated using the same size representation. This can also be viewed as a form of content compression.

One potential downside of indirect encodings is that there might exist some area in the content space that cannot be generated by any genotype (Togelius *et al.*, 2011). This problem is relatively easy to spot when using a direct representation.

3.1 Search Space

The representation determines the search space of the learning method. There are two common approaches.

The first is to directly search in level space. This means that each time the method is run, a new search is carried out for only one level. The encoding here is usually quite direct. Examples of this include Ferreira *et al.* (2014), where the method searched over integer vectors of the same length as the level (200), Liapis *et al.* (2015, 2013) searched over a 2d tilemap representation of the levels and Cardamone *et al.* (2011) generated racing game tracks by using a set of control points for Bezier curves as the representation.

The second is to search in generator space. This involves searching for or learning a generator of levels. This generator can be queried to generate a large number of levels by varying its input parameters. The representation used is usually more abstract, for example, the policy of a reinforcement learning agent (Khalifa *et al.*, 2020) or the weights of a neural network. Kerssemakers *et al.* (2012) used a genetic algorithm to search for parameter vectors that determine the behaviour of a non-deterministic (and thus reusable) generator.

Risi *et al.* (2016) use a method that is somewhere in between. The method evolves a single neural network for each game item (which is an image of a flower). Different parts (layers) of the item are generated using a different layer input to the network and these are then stacked on top of each other to form the final item. This is actually still searching in 'level' space, however, since each representation generates only one single item, even though the representation is indirect. Potentially, though the authors didn't explore this avenue, each of these networks can generate multiple items, by using sets of disjoint layer numbers (e.g. item 1 uses layers 1-5, item 2 uses layer numbers 6-10, etc.).

The main advantage of the first representation is that it is quite simple to design a genotype and genotype to phenotype mapping (in our example, we simply converted the binary string to a decimal number). This conversion usually does not require much computation, e.g. Ferreira *et al.* (2014) simply iterate over the vector representation of the level and place the item at that index at the corresponding position in the level. This method can suffer performance problems, however, because each new level necessitates performing a whole new search, which does take time. These methods are therefore unsuited to real time, online level generation, although there are some exceptions, like Galactic Arms Race (Hastings and Stanley, 2009) which evolves weapons in real time. Togelius *et al.* (2011) describe another potential issue when using search based methods, that of "Catastrophic Failure" (Compton, 2010), which can result in either unplayable or low quality content or long search times to make up for a bad random initialisation. This risk might be unacceptable for certain commercial games.

Searching in generator space is more challenging, as one has to design some mapping that the generator can make between a parameter vector (or random seed) and a game level. But, since a generator is learned, one can move the computational burden to an offline training stage, and still achieve real time level generation by simply querying a policy, which makes it suitable for generating levels on the fly in games. Khalifa *et al.* (2020) frame the level generation task as a sequential process, where at each step the agent receives some observation (i.e. the whole map and some extra information) and performs a single action, which could be to change the current tile or a specific tile of its choosing. This approach is very fast and does not require prior training data. It does have a possible limitation, though, in that it assumes that the level generation process is sequential which might not be appropriate for some games levels, where a more global design sense is desired.

Kerssemakers *et al.* (2012) use a two phased approach to create a level generator. The inner generator consists of a collection of non-deterministic agents that move through the level and modify it as they go. These agents are parametrised by a parameter vector, which is evolved using the outer generator. Since the agents are not deterministic, each inner generator can generate a large amount of levels. The authors showed that this method can generate levels in real time and that, depending on the specific generator used, it can also generate diverse levels. Since the agents cannot communicate with each other and some agents are only active in certain parts of the levels, the variation between different parts of a level can be quite large, which might lead to some perceived dissonance on the player's part.

In Table 1 we summarise some of the advantages and disadvantages of each method.

Generator Space	Level Space
Fast generation	Slow generation
Difficult to design	Simpler to design
Easier to generalise	Game specific
Long training time	No training time
Relatively consistent quality	Can have large variance in quality

Table 1: A comparison of searching in generator space vs searching in level space.

4 Learning Methods

Historically, evolutionary methods have been one of the most popular approaches to PCG (Summerville *et al.*, 2018; Togelius *et al.*, 2011). These methods are usually simple to implement and one does not need a labelled set of training data to be able to generate levels.

For example, Ferreira *et al.* (2014) used a simple genetic algorithm, whereas Liapis *et al.* (2015) used a 2 population genetic algorithm with the novelty metric as the fitness function. Using two populations - one consisting of feasible

individuals and the other of infeasible individuals - was used to ensure a high number of playable levels.
Liapis *et al.* (2015) discussed two different methods that can be used depending on if one wants to generate diverse levels or generate playable levels relatively quickly. Due to the high computational costs involved, however, these methods are not always suitable for real time level generation.

Recently, however, more focus has been put on applying machine learning/deep learning to this task (Summerville *et al.*, 2018; Liu *et al.*, 2020). Khalifa *et al.* (2020) used reinforcement learning, by formulating the task of level generation as a Markov Decision Process (MDP) and using existing methods. The agent generated the levels sequentially by performing a specific action (which involved changing a single tile of a 2D map) at each time step.

There has also been work on using a combination of the above methods, for example, Risi *et al.* (2016) used a neural network to represent collectible items, but evolved the structure and weights using NEAT (Stanley and Miikkulainen, 2002). This method is still largely evolutionary, however, since there is no other learning mechanism in play.

Another notable work is the one by Volz *et al.* (2018) where the authors used unsupervised learning to train a generative adversarial network (GAN) on a collection of existing game levels. They then used evolutionary methods to search for a latent input vector to this GAN, which generates the level. Building up from this, Schrum *et al.* (2020) went one step further, by instead evolving a CPPN (Stanley, 2007) that takes in the coordinates of a level segment and outputs the latent vector that the GAN uses to generate that segment. Many of these segments are then combined to form the complete level. This method has the advantage of being able to scale up to larger levels with the same CPPN and GAN, as one would just use different coordinates for later segments.

The main disadvantage with the more traditional machine learning based approaches is that existing training data is required, which limits its usefulness for creating levels for new games. Another subtle effect of this need for data is that it can influence the model to generate levels that are close in character to the training data. This might be desirable, but there also could be cases where a user would like to create new and innovative levels using a PCG system.

5 Evaluation

The training/reward/fitness signals used in these methods to evaluate a piece of content are very important. Using the wrong one can result in the learning or searching process not resulting in desired levels.

One way to evaluate content is to let users pick those pieces of content that they prefer, which is referred to as interactive evaluation (Togelius *et al.*, 2011).
This means that a human rating/score is the fitness function, and there is not an explicitly defined objective (as we had in the simple example). Cardamone *et al.* (2011) generate racing game tracks and let users rate the generated levels, which serves as the fitness function. Risi *et al.* (2016) use the same structure, where users can breed the specific collectible items that they like, thus guiding the evolutionary process. This method does not have an explicit measure of fitness though, as users get to manually choose which parents to use for crossover (as opposed to ranking individuals and letting the algorithm choose the specific combinations). Kerssemakers *et al.* (2012) used a similar, interactive, method to evolve its level generators. To facilitate finding payable levels early on, the evolution was seeded with a set of highly playable levels that were generated offline and without interactivity. The authors do state that if a user wants to generate levels with a very specific feature, it might still take a long time for the population to evolve towards that goal. Another potential downside of this method of evaluation is that the population size one can use is severely limited, which might impact how much of the space the population can search and how effective the searching is (Vrajitoru, 2000).

This type of fitness function is useful in cases where a synthetic one is difficult to design, or where it is desirable to have humans guiding the process. The downside is that it is asynchronous and slow, as one has to wait for humans to evaluate content before continuing with the search. It can also interrupt the player's experience. This is fine in a game like *Petalz* (Risi *et al.*, 2016), where the whole game revolves around this mechanic, but might not be appropriate when fast and automatic level generation is desired. One can also use implicit data collection, where the game tracks player action information in the background, and uses that as the metric by which to evaluate the level. The problem with this is that the data collected can often be noisy, infeasible and not of much use (Togelius *et al.*, 2011).

Other methods, called direct evaluation, are largely similar in that they involve the designer creating some notion of fitness that can easily be computed. Ferreira *et al.* (2014) used simple rules to evaluate Super Mario Bros levels, like using the sparsity of the enemies and the entropy of the ground. This does require some domain knowledge and it can be difficult to design a metric that guides learning successfully towards desired goals.
Khalifa *et al.* (2020), which used reinforcement learning to incrementally change a level, created game specific modules that assessed the difference in quality each incremental change results in and used that as the reward signal.

Another approach in a similar vein is to use a more high level fitness function that measures the challenge and expected enjoyment of a level and does not focus on the low level features (Shaker *et al.*, 2011). This type of method can be a useful way to generalise fitness functions across games, but might be liable to guiding the learning method away from the intended result and towards undesirable results that maximise this more indirect metric.

Domain independent methods that don't require as much game specific designing can also be used.
Liapis *et al.* (2015) used the notion of the *novelty metric* (Lehman and Stanley, 2011), which was described in the background section, to evaluate feasible levels and used either the novelty metric or the distance to feasibility for infeasible levels. The distance function was defined as the visual difference between two tilemaps. This approach leads to the individuals in the population having behaviours that are diverse and novel. It does forsake a general objective though, which can cause the algorithm to focus on exploring the vast infeasible space. This is why the authors used a 2 population (feasible and infeasible levels) algorithm to protect the feasible individuals from being killed off for not being novel enough. This can result in a high level of diversity between the levels, as well as levels that have desirable qualities. This level of diversity is usually difficult to obtain when using normal evolutionary methods.

Another way to evaluate a generated level is using an AI agent that plays the level (Togelius *et al.*, 2011) and the score is derived from how the agent played (e.g. how quickly the agents solved the level, its total distance travelled, etc.). One can even let the agent learn when playing the level and somehow incorporate that into the evaluation function.

5.1 Constraints

Another aspect of evaluation is how to deal with infeasible individuals that don't satisfy some of the many constraints (e.g. there can only be one player, the level must be solvable, etc.). There are different methods to address these concerns, for example, Liapis *et al.* (2015, 2013) use two different populations, where one consists of the infeasible individuals and the other has the feasible ones. Offspring from either population can migrate to the other one if they satisfy the feasibility/infeasibility condition. They independently evolve these two populations, which results in a larger number of feasible levels because feasible individuals are not immediately killed off for not being novel enough compared to the infeasible individuals.

A constraint satisfaction method can also be used to repair unplayable levels that resulted from crossover or mutation (Shaker *et al.*, 2011). In general, however, to design such a system can require a lot of domain knowledge and might even be as difficult as creating an explicit level designing algorithm (Liapis *et al.*, 2015).

Other methods just kill off the individual by setting the fitness/reward to 0 when encountering an infeasible individual (Lehman and Stanley, 2010). This can result in a low amount of diversity, or low quality levels, because the best levels are often close to the border between feasibility and infeasibility (Michalewicz and Schoenauer, 1996; Liapis *et al.*, 2013).

5.2 Final Evaluation

How one chooses to evaluate the generated levels for comparison between different methods is also important. For example, Khalifa *et al.* (2020) used some preset 'goal' criteria to evaluate the level. These criteria stipulated that the level has to be solvable in more than X steps, it should have only 1 player, 1 door and 1 key, etc. Liapis *et al.* (2015) used 4 different metrics, namely how many runs resulted in a feasible individual being found, the final number of feasible individuals and their average diversity, which is measured as the average number of non matching tiles in a tilemap. Ferreira *et al.* (2014) simply state that their method can generate levels that are similar to the original Super Mario Bros levels.

It is quite difficult to compare these different results, as they usually measure different aspects, and often don't use the same units or methods of measuring.

It would be useful to have a consistent way of evaluating a level that potentially includes metrics such as diversity, difficulty, and quality. Horn *et al.* (2014) echo this sentiment and attempt to create a valid range of metrics that can be used to evaluate levels, but these have not been widely adopted. Most of the metrics are defined for a single level, measuring aspects like linearity, density, and leniency (how easy a level is). The one metric that compares the diversity of levels is the compression distance, which measures the relative similarity between two levels from the same generator. The authors add that promising future work could be to extend these metrics by adding in a simulation based evaluation score, where an artificial agent plays the generated level, and the score is derived from the playing style of this agent.

6 Conclusion

Even though there is a lot of work in the PCG field, there are still some unsolved problems. The main one is being able to generate (1) high quality, diverse levels that (2) can be generated in real time by (3) a learned generator (4) without existing training data.

A second shortcoming is the evaluation metrics used when reporting results. These methods often prohibit comparison between different works and it can be very game specific, which makes it difficult to compare methods across different games. These metrics are also not very robust to what humans see as novel or diverse. A simulation based, game independent metric will enable sensibly comparing different generators.

Therefore, the proposed research questions have not been adequately addressed as a whole in the current literature.

References

Cardamone, L., Loiacono, D., and Lanzi, P. L. Interactive evolution for the procedural generation of tracks in a high-end racing game. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, page 395–402, New York, NY, USA, 2011. Association for Computing Machinery.

Compton, K. *Remarks during a panel session at the FDG workshop on PCG*, 2010.

Cook, M. and Colton, S. Multi-faceted evolution of simple arcade games. In *2011 IEEE Conference on Computational Intelligence and Games, CIG 2011*, pages 289–296, October 2011.

Ferreira, L., Pereira, L., and Toledo, C. A multi-population genetic algorithm for procedural generation of levels for platform games. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, page 45–46, New York, NY, USA, 2014. Association for Computing Machinery.

Goldberg, D. E. *Genetic algorithms in search*, chapter 1. Addison Wesley Publishing Co. Inc., 1989.

Hastings, E. and Stanley, K. Evolving content in the Galactic Arms Race video game. In *CIG2009 - 2009 IEEE Symposium on Computational Intelligence and Games*, pages 241–248, 09 2009.

Hendrikx, M., Meijer, S., Van Der Velden, J., and Iosup, A. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 9(1):1–22, 2013.

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

Justesen, N., Torrado, R. R., Bontrager, P., Khalifa, A., Togelius, J., and Risi, S. Illuminating generalization in deep reinforcement learning through procedural level generation. *arXiv preprint arXiv:1806.10729*, 2018.

Kerssemakers, M., Tuxen, J., Togelius, J., and Yannakakis, G. A procedural procedural level generator generator. In *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012*, pages 335–341, 09 2012.

Khalifa, A., Bontrager, P., Earle, S., and Togelius, J. PCGRL: Procedural content generation via reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, pages 95–101, 2020.

Lehman, J. and Stanley, K. O. Revising the evolutionary computation abstraction: minimal criteria novelty search. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 103–110, 2010.

Lehman, J. and Stanley, K. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19:189–223, 06 2011.

Liapis, A., Yannakakis, G., and Togelius, J. Enhancements to constrained novelty search: two-population novelty search for generating game content. In *GECCO 2013 - Proceedings of the 2013 Genetic and Evolutionary Computation Conference*, pages 343–350, 07 2013.

Liapis, A., Yannakakis, G. N., and Togelius, J. Constrained novelty search: A study on game content generation. *Evolutionary computation*, 23(1):101–129, March 2015.

Liu, J., Snodgrass, S., Khalifa, A., Risi, S., Yannakakis, G. N., and Togelius, J. Deep learning for procedural content generation. *Neural Computing and Applications*, pages 1–19, 2020.

Michalewicz, Z. and Schoenauer, M. Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary Computation*, 4:1–32, 03 1996.

Risi, S., Lehman, J., D'Ambrosio, D. B., Hall, R., and Stanley, K. O. Petalz: Search-based procedural content generation for the casual gamer. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3):244–255, 2016.

Schrum, J., Volz, V., and Risi, S. CPPN2GAN: Combining compositional pattern producing networks and gans for large-scale pattern generation. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, page 139–147, New York, NY, USA, 2020. Association for Computing Machinery.

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

Stanley, K. O. and Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, June 2002.

Stanley, K. O. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8(2):131–162, June 2007.

Sturtevant, N. and Ota, M. Exhaustive and semi-exhaustive procedural content generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 14, 2018.

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

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

Togelius, J., Champandard, A. J., Lanzi, P. L., Mateas, M., Paiva, A., Preuss, M., and Stanley, K. O. Procedural content generation: Goals, challenges and actionable steps. In Lucas, S. M., Mateas, M., Preuss, M., Spronck, P., and Togelius, J., editors, *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 61–75. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

Tutenel, T., Bidarra, R., Smelik, R. M., and Kraker, K. J. D. The role of semantics in games and simulations. *Computers in Entertainment (CIE)*, 6(4):1–35, 2008.

Volz, V., Schrum, J., Liu, J., Lucas, S. M., Smith, A., and Risi, S. Evolving mario levels in the latent space of a deep convolutional generative adversarial network. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 221–228, 2018.

Vrajitoru, D. Large population or many generations for genetic algorithms? Implications in information retrieval. In *Soft Computing in Information Retrieval*, pages 199–222. Springer, 2000.