Procedural Generation via Machine Learning

Philip Blaskowski
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
blask017@morris.umn.edu

ABSTRACT

The automatic generation of content can useful for video game developers creating a game. It can provide developers with the capabilities to automatically generate interesting content for gamers to play through. The field that allows developers to do this is called Procedural Content Generation. While this field has been around for a very long time (and it might not even have to relate to gaming), progress in technology has allowed developers to think up of new algorithm that allows them to generate content better and more suited to the gamer population they're catering to. In this paper we will talk about two Procedural Content Generation methods, both of which employ Machine Learning.

Keywords

Machine Learning, Procedural Content Generation, Games

1. INTRODUCTION

Procedural Content Generation is the use of algorithms to create data with little to no human interactions [6]. It has uses outside of gaming, but for the purposes of this paper procedural Content Generation is going to be in the context of video games. Video games have seen the most interesting use of this.

Probably the most famous example of this is Minecraft, a game where a massive open world is randomly generated. This world is vast and meant to be explored by the player. Underground caves are generated filled with materials the player can use to build to tools to explore even more. Furthermore the world is filled with monsters and animals, and together with the things a player can find, they can change the world as they see fit. As evident by the amount of things randomly generated in the game, it's no wonder that the game employs Procedural Content Generation.

Procedural Content Generation is important to game developers because it's cheaper than humans manually creating the content by hand. Large amounts of content can be created without much human input. With that said, this also helps smaller studios develop games, these studios can concentrate on making a good game with good mechanics instead of worrying about if they have the manpower to create assets for their game [2]. Not that developers should only ever use Procedural Content Generation however, the games that are critically acclaimed are usually designed around the fact that the game does generate things randomly. This is especially true with smaller developers, bigger studios usually supplement the things already in the game with this. Further development of this field, at least in gaming, can lead to many possibilities: with the use of Machine Learning, Procedural Content Generation can be used seamlessly, making a more cohesive world and content can even be generated tailor fit for the person playing the game.

In this paper we'll explore the possibilities brought upon by combining Procedural Content Generation with Machine Learning

UMM CSci Senior Seminar Conference, April 2019 Morris, MN.

2. BACKGROUND

There are certain concepts that we need to cover before diving into specific methods for Procedural Content Generation. All of the research materials used in this paper employ Machine Learning as a main feature in their methods, so it's a given that we need to go over certain Machine Learning concepts and models used in the field. Also my paper references multiple video games, so we'll be going over a brief overview of what those games are about.

2.1 Machine Learning

Machine learning is a scientific study that employs algorithms and statistical models to perform tasks without the need for specific instructions, relying on patterns and inference instead. Machine Learning algorithms use training data to build a model to perform a task. These algorithms can be used in many different kinds of applications like image recognition and filtering one's email.

2.2 Training

Machines are much better at processing information and storing said information than human. But usually they are bound by their programs. Training is a way to leverage a machines ability to process information while also making machines able to be more intelligent. By feeding machines with data relevant to a certain task, they can look for patterns, and relationships in the data. The end goal of this is to get a machine that "understands" the given problem enough to finish tasks autonomously. Of course the success of this is dependent on the data given, if the data given is bad(i.e. has nothing to do with the problem), you're not going to get the desired results from training.

2.3 Encoding data

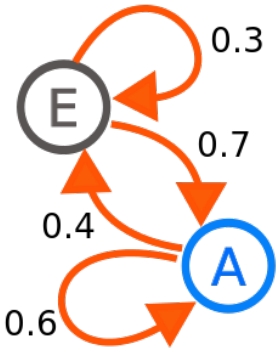

Figure 1: An example of a Markov chain. Transitioning from E to A has a 70% of occurring while staying in E is 30%. Going from A to E has a 40% chance while staying in A has a 60% chance. As we can see all these probabilities equal 100%, to account for everything. Taken from [1]

Encoding data is vital for training. Machines can't actually understand the human language, a machine learning algorithm isn't actually reading through your emails or recognizing your face, it's just looking for patterns that allow it to fulfill it's tasks.

This is why it's necessary to take the real world data you've gathered and turn into something a machine can understand. Things like ones and zeros, graphs, and matrices. Encoding your data is key to getting good training data for use in training.

2.4 Markov Chains and Multi Dimensional Markov Chains

Markov chains are a stochastic model(a collection of random variables) [1], which describes a sequence of events such that the current state is dependent on the previous states. Figure 1 shows this.

A multi dimensional Markov chain on the other hand is an extension of Markov chains, where in a multi dimensional representation of a Markov chain (such as a graph or matrix) every state in that representation can be dependent on any of the other states [4].

2.5 Super Mario Brothers

Super Mario Brothers is a classic video game series published by Nintendo in 1983. You follow the titular character Mario as he runs and jumps his way through a level to save Princess Peach. This game is a 2-d platformer, which means it's in two dimensions and you use your skills to jump on platforms to get through a level. These tasks can be of varying difficulty depending on the level.

2.6 Quake

Quake is a classic game published by Id Software in 1996. You play as Ranger as he is sent by humanity to eliminate the enemy known as Quake who is sending armies through portals to test humanities martial prowess. This games is a first person shooter, which naturally means it's in 3-d. First person shooters mean that you'll be shooting things like in real life. This is in contrast to a third person shooter where you're looking over the shoulder of the player character(like an out of body experience).

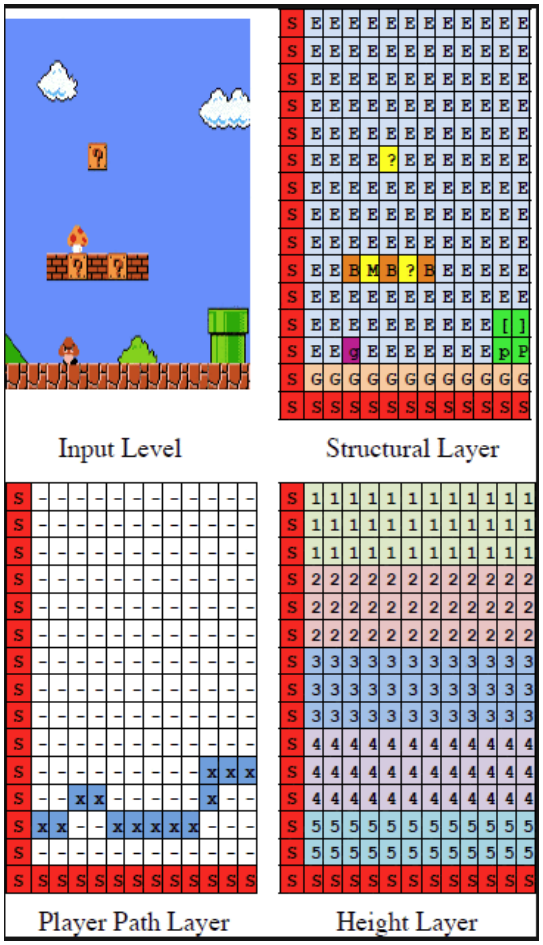

Figure 2: Structural layer (top right), player path layer (bottom left), and height layer (bottom right). Taken from [5]

3. REPRESENTING LEVELS VIA LAYERS

This method, like all the other method described in this paper, makes use of Machine Learning. The defining part of this method is that makes use of layers to represent a level in a game, these layers can be represented as: $L = \{L_1, L_2, , L_n\}$, where L_n is a two dimensional matrix with dimension $h * w(height * width)$. Each of these layers has a sets of tiles, t_i, which varies between these layers [5]. This makes it so that each layer can represent the different parts of level. As an example the a layer can have tiles representing the structure of a level, another layer representing the paths a player can take through a level, and one more layer which represents the height of a level. Figure 1 shows a section of a Super Mario Bros. level represented as the described layers, a structural layer (top right), a player path layer (bottom left), and a height layer (bottom right).

These layers aren't bound to what is being described in this section however. Another such layer can be a difficulty layer, which, as the name suggests would represent the difficulty through a level. This method seems very straight

forward, but it is very flexible; allowing developers to create more complicated, and consistent levels.

3.1 Training

For this method, Multi-dimensional Markov chains (MdMC) are used to train our data. But we must first describe how we estimate a Conditional Probability Distribution(CPD) with a single layer MdMC and then describe how we achieve the same thing with a multi-layered MdMC.

To train a single layer MdMC we'd need a network structure and training levels. The network structure tells us which states the current state is dependent on. A set of these states is called the Previous Tile Configuration. Using the network structure and training levels, the CPD can be calculated, based on the frequencies observed in the training data.

To train a multi-layered MdMC, we would still need a network structure and training levels, but the training levels are instead represented in multiple layers. The way to calculated the CPD is similar to the way it was calculated in the single layer MdMC. The difference between the two methods is that in the multi-layered approach, is that the CPD can hold information from multiple layers. Thus allowing the CPD to learn dependencies across different layers [5].

3.2 Sampling

This section we will talk about sampling(generating) new levels levels via the methods described above. First we sample levels utilizing a single layer MdMC and then a multi layer MdMC. After that we sample one last time using a constrained sample extension.

Before we start our sampling process using a single layer MdMC, we'd need our desired levels dimensions, and the Conditional Probability distribution as we trained it. To begin sampling we first must pick a starting point Snodgrass et. al. picked the bottom left corner. We then move from our starting point to completing the current row before we move on the next one. This process is repeated until an entire level is sampled. To sample a tile, we look towards our Conditional Probability Distribution and the previous configuration.

While sampling we employ two procedures: A look ahead and fallback procedure. To avoid errors during sampling we employ two procedures: A look ahead and fallback procedure. The look ahead procedure allows us to avoid any unseen states, a state resulting in a combination of tiles that we didn't come across in the training data. Think of the pipe in figure 2 it would be weird if the top of the pipe was dirt or incomplete. This procedure works by sampling a given tile and then generating a number of tiles ahead of the sampled tile. If an unseen state is observed a different tile is sampled.

The fallback procedure is used when an unseen state cannot be avoided [5]. During our training process we trained multiple MdMC models, each of which are trained with increasingly simple network structures. When we sample a tile we start of by using the most complex model, if we see an unavoidable unseen state we fall back to simpler ones until we generate a tile that satisfies our look ahead procedure.

Sampling a level using a multi-layered MdMC works very similarly to the single layered approach. The difference is that the trained distribution P_M models the probability of tiles in the main layer, and the previous configurations contain states from the other layers. This means that the model can see dependencies from other layers.

To ensure we actually have playable levels, we add constraints to our sampling approach. This forces our sampling algorithm to enforce playability, which is done through a resampling process. Snodgrass et. al. apply their constraints through an algorithm.

3.3 Experiment Overview

To test the performance of their method and compare their single layer MdMC model against their multi-layer MdMC model, Snodgrass et al ran their method on Super Mario Brothers. They were especially looking at whether their multi-layered approach was able to recreate levels more accurately and create more interesting situations than their single-layered approach.

To create their training levels they used the layers described above (structural, player path, and height layers). For their structural layer, Snodgrass et al represented it from a set of 34 tile types to signify the type of object being encountered (the ground, platforms, pipes, enemies etc.). Note that the tile set used in this layer for these experiments is expanded from the norm. This is because the tiles sets used by other researchers for their experiments weren't expressive enough; often taking away important details from the level.

For their height layer, they split the level up by grouping together multiple rows. This essentially allows for more focused training within a section. For this layer, the researchers used a set of 6 tiles to represent the layer. Four tiles for the three consecutive rows, one for the final two rows, and the last to represent the boundaries of the layer.

Lastly their player path layer only uses three types of tiles: one tile (x in this case) is used to signify a part on the path, another (-) to signify parts not on the path, and like the other two a tile used to represent the boundaries of the layer. Figure 1 illustrates this set-up if it still seems a bit confusing.

3.4 Experiment set-up

For this experiment Snodgrass et. al. used 25 training levels to train their single and multi layered MdMCs. After training they sampled 1000 levels per each MdMC type. Since the multi layered MdMC employs the player path layer they decided to sample with 4 different player path layers of differing complexities(250 samples per player path layer); the significance of this will be detailed later on. These player paths are based on different levels in Super Mario Bros. One of the paths involve a spring board, which will be used to evaluate the model's ability to make interesting interactions.

To evaluate the approach's capabilities in generating interesting level designs, the spring boards mentioned above come into play; spring boards are an infrequent tile type that allows the player controlled character to jump much higher than normal. To measure the approach's capabilities, we calculate the ratio of the amount springs in the sampled level against the amount of springs required to complete the sampled level. This allows us to see if a spring is just placed there by chance or if the springs are actually being utilized to complete a level. Another interesting point Snodgrass et. al. is interested in is their approaches ability to allow for the paths used in the player path layer. To achieve this they calculate the discrete Fréchet distance between the provided player path and the actual path taken through the level. Finally, knowing how well the approach follows the training data is important, the linearity and leniency of sampled lev-

Linearity	This measures how well the platforms in the level can be approximated with a best fit line. It returns the sum of distances of each solid tile type from the best-fit line, normalized by the level length.
Leniency	This approximates the difficulty of the level by summing the gaps (weighted by length) and enemies (weighted by 0.5), and normalizing by the level length.
Fréchet	This measures the distance between two paths. Intuitively it can be thought of as the minimum length of a rope needed to connect two people walking on two separate paths over the course of the paths.

Table 1: This table goes into more detail into the metrics used to compare generated levels and training levels. Taken from [5]

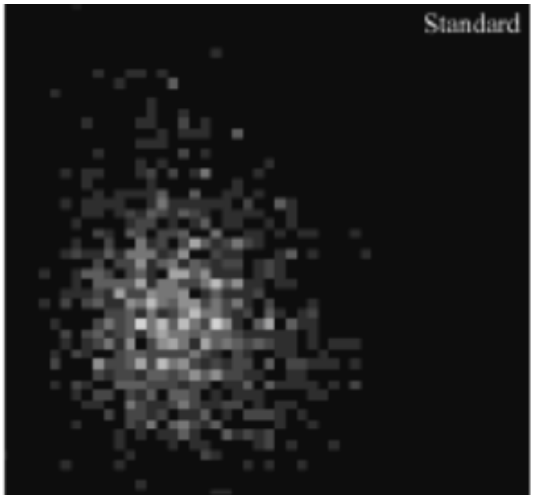

Figure 3: Shows the expressive range of the different models. The y-xis is the leniency and the x-axis is the linearity. Taken from [5]

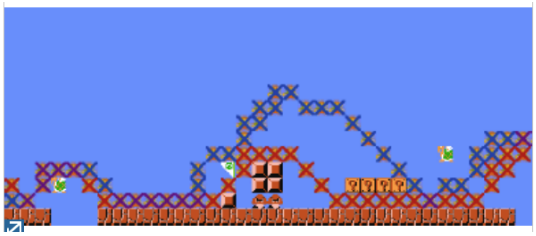

Figure 4: Shows the similarities between the path generated by the model vs the given path. Blue is the generated path, red is the intended path and purple is when the paths cross. Taken from [5]

els are compared to those of the training data. Table 1 gives more information on these metrics.

3.5 Results

Both single layer and multi layer MdMCs had linearity and leniency values similar to the ones calculated from the training levels. We can see this in Figure 3 because the points in the graph are clustered together. This means that both models were able to mimic the structural aspects of the training levels. This does not mean that they are exact 1:1 copies though, it just means that the levels look alike. This is where the similarities end however. The single layer MdMC had a hard time placing springboards with intent. In contrast the multi layer model was able to place springboards with intent more reliably. This shows that the multi layer MdMC was wable to capture nuances better than the single layer MdMC, which is good.

Another difference in the two models is the Fréchet distances, the multi layer MdMC was able to generate levels with lower Fréchet distances than the single layer model. This means that the levels generated by the multi layer representation was able to generate levels that accommodated for the given play paths better. Figure 4 shows this, the intended path and the generated paths cross over quite a bit through out the figure and even when they don't cross, they are close to each other and are shaped similarly.

In summary the multi layer MdMC model was able to create levels with more nuance while also allowing for preset player paths throughout a level. The single layer MdMC was able to only really mimic structural similarities in the training levels, but lose out on everything else.

4. LEARNING BASED GENERATION

A challenge in procedural content generation is the chance of unplayable content being generated [3]. Situations where payers are put into impossible situations, like being placed in a room with enemies that kill the player in one hit. Existing methods employ the use of player feedback to determine predefined player types, and personalities to encapsulate the game's general player population. While these methods see success, the used categories might not fit all players and can lead to difficulty in inferring player types/styles from a computational perspective. Also, subjective feedback can be quite inaccurate and noisy, which can make things more difficult. There are ways to avoid this problem however; through the use of public play testers, a generic play style can be found to design around. But this method is expensive and time consuming.

A solution Jonathan Roberts et. al. found is through the use of their framework: LBPCG(Learning Based Procedural Content Generation). This framework attempts to mimic commercial game development, therefore the generation process is split into three stages. The development stage(involves game developers), public test stage(involves public test phase, and adaptive stage(involves target players) [3]. They go about this by encoding the knowledge of game developers(development stage), and model the experience of test players(public test phase). All this should result in a framework that is able to generate appealing content(adaptive stage).

4.1 The Three Stages

In this section we will discuss the different models Roberts et. al. use to model the different stages of development.

We will start with the Development Stage which consists of two stages: the Initial Content Quality(ICQ) and Content Categorization models. Both models serve to encode developer knowledge. The ICQ model is used to filter out awful/unacceptable content and the CC model is used to partition the acceptable content to meaningful sub spaces that elicit good cognitive experiences from the players [3]. This process allows developers to be more flexible in categorizing content and then associate said content to certain design interests and player populations. Furthermore, these models allow developers to limit the search space for personalized content generation. This stage is essential for enabling models described in the other stages.

Public user testing is proven to serve an essential role in modern game development by allowing developers to enhance the end product further before the game is released [3]. To model this Jonathan Roberts et. al. proposes two models again: the Generic Player Experience model(GPE) and the Play-log Driven Categorization model(PDC). The GPE model is used to capture the public players' feedback by playing games that represent the categories found by the CC model above. While the feedback received from these testers are subjective the GPE model attempts to find a consensus on each game. The PDC model attempts to model the experience of the test players and the category of the game they were playing.

The Adaptive Stage only employs one model: the Individual Preference (IP) model, which controls the content generator with the four models used in the other two stages. The IP model shoul be able to deal with the four main issues: finding the preferences of the target audience and the category of the games that the developers chose(PDC and GPE), making sure that the quality of the generated content is consistent(ICQ, CC and GPE), detecting when the content is diverging from the category(PDC), and automatically detecting and tackling crisis situations [3].

4.2 Learning

In this section, we will go over our learning-based enabling techniques. Jonathan Roberts et. al. formulate the problems in implementing their models and then propose a solution and how to apply those solutions to their models.

Jonathan Roberts et. al. confines their work to a class of representations characterized by a game parameter vector D, which is denoted by $g = (g_1, g_2, ...g_D)$, which defines a content space G where $g \in G$. They also assume that they can record gameplay using a play-log l of L event attributes represented by $l = (l_1, l_2, ...l_L$. Developers then choose F where $F \geq 1$ content features; these features are decided upon by the developers themselves. These features are denoted by $c = (c_1, c_2, ...c_F$ for content categorization. As an example, we look at a game Jonathan Roberts et. al. will use for their experiments: Quake. They categorize the game with $D = 9$ parameters, the content space contains $|G| = 116,640$ core games, the play-log consists of $L = 122$ attributes and the developers chose $F = 1$ features with $c_1 = "difficulty"$, which is further separated into five categories of varying difficulties for content categorization.

The problem for the ICQ model is finding the mapping for $g \in G$ so that $\phi_{ICQ} : g \rightarrow +1, -1$ such that $+1/-1$ indicates whether the parameter is acceptable or unacceptable. This specification brings about two problems: how to select the smallest number of representative games for use in training, and how to get a classifier for generalization. Roberts et. al. solve this by using an Active Learning(a subset of Machine Learning) algorithm based around clustering analysis. Using this algorithm we can separate the content space into multiple subspaces. Then a representative game is chosen in those subspaces and played by the developers. Then using their knowledge and experience they can label those games. These labels are then used for training.

The problem for the CC model is finding a mapping for $g \in G_a$ a subspace for all acceptable games such that $\phi_{CC} : c_f$ for $(f = 1...F)$ The CC model comes across the same problems as the ICQ model. This allows Jonathan Roberts et. al. to use the same solutions as the one the used in the ICQ model, along with a resource sharing idea. For resource sharing, developers can assign categorical labels whenever they label the game as acceptable.

The problem for the GPE model is how to infer is taking a public tester's feedback and estimating the popularity of the game. To do this Roberts et. al. achieve this by using a generic algorithm. This algorithm treats the popularity of a game as missing information. Then it looks at the feedback given by the testers and infers a game's popularity using a maximum likely hood estimator.

The problem for the PDC model is finding a mapping for $\phi_{PDC} : (l, c) \rightarrow y$. Because y is a binary value which indicates a player's positive or negative experience, we can use binary classification for learning.

4.3 Generating Stages using the IP Model

Jonathan Roberts et. al. carry out the IP model using a state machine with three stages; in this case they used: Categorize, Produce, Generalize as their stages. This allows them to detect preferred content, good game generation as well as system failure counter measures.

The Categorize state is responsible for detecting a player's content preference. To do this the player needs to play a few games used in the GPE learning process. As soon as the player has played a game, the PDC model uses the player's play-log along with content features to decide whether the player enjoyed the game or not. If the player shows any indication in the data that they enjoyed the game, the current state moves on to the Produce state

The Produce state is used to direct the ICQ and CC model to direct the content generator into producing content based on the category the Categorize state determined. This state is also responsible for detecting whether a player actually enjoys the game or not. This is done by giving the player the generated content and using the PDC model to produce logs the state determines if the player is enjoing the content. If the state determines a player isn't having fun with the newly generated content the state loops back to the previous state.

The Generalize state is responsible for system failures. System Failures in this case means that the IP model cannot find a player's preferred content. If the IP model cannot produce content for a player after many attempts, Jonathan Roberts et. al. proposes to exploit the ICQ, CC and the GPE model to generate more generalized content that the general public would enjoy

To test their method Jonathan Roberts et. al. ran a simulation utilizing Quake. Naturally they used their models to try and generate content that pleases the player base.

To start both the ICQ and CC models were trained by a single developer who played and labeled the games used in

the Active Learning process. For the GPE and PDC models Jonathan Roberts et. al. utilized the internet, they set up a client/Server architecture to collect data from their survey takers. To actually collect meaningful data for the GPE they chose 100 representative games via the ICQ model(games in this case are just individual levels in Quake). This means there 20 games per difficulty categories and they fixed the seed so all the survey takers play the exact same game.

The client was distributed via website like Reddit, essentially websites that attract a lot of gamers (both casual and hardcore). In total 895 surveys were submitted from 140 people. The survey merely contained two questions about the game: "Did you enjoy it? (yes/no)" and "How do you rate it? (Very Bad/Bad/Average/Good/Very Good)" [3]. The play-logs produced and the answers gained in the surveys were used to train the PDC model.

Further analysis of the surveys prove that the representative games chosen were actually pretty good because the caused controversy among the players. For example, as the difficulty of the levels increases the amount of "Very Good" labels also increases, but the hardest levels were also the levels that had the most "Very Bad" labels. Additionally the middle difficulties were also the least likely to receive "Very Bad" lables. This shows that parts of the player base has polarizing opinions, which should potentially allows the models to categorize players better.

4.4 Testing the IP Model

To test the IP model produced by the previous survey, Jonathan Roberts et. al. used two baseline algorithms to compare the IP model to. A random model which just generates games randomly using Oblige, a Quake map editor. And the other is a Skill model, which uses Oblige again to generate games by manipulating the skill sliders in Oblige.

To test the performance of the IP model against the two other models Jonathan Roberts et. al. ran another survey. This survey involves getting players who didn't participate in the previous survey. The player is then asked to play 30 games, 10 generated by each model. Before actually playing the games the players are asked some preliminary questions about their amount of experience with video games and their perceived skill level. After playing the games, the players are then asked questions about their overall enjoyment.

To evaluate the performance of the three models Jonathan et. al. defined three metrics. Since the first question asked after a games is played is the same as the one used in the public test, the answer to that question is which was yes or no is either a 0 or 1, this is going to be considered as metric 1. Metric 2 is defined using the answers given by the answers about the player's overall gameplay experience. "Very bad" and "Bad" answers result in 0, while the other three answers are scored with a 1. Since one of the goals of the IP model is to automatically acquire a player's preferred difficulty, we acquire this by looking at whether a player enjoyed levels of a certain difficulty.

4.5 Results

For evaluating the IP model Jonathan Roberts et. al. were able to gather 14 people of varying experience with games and skill levels. This spread of players should be able to adequately represent the gaming community fairly well.

Figure 5 shows the results of the models based on the metrics described above. In figure 5(a) we can see that 10

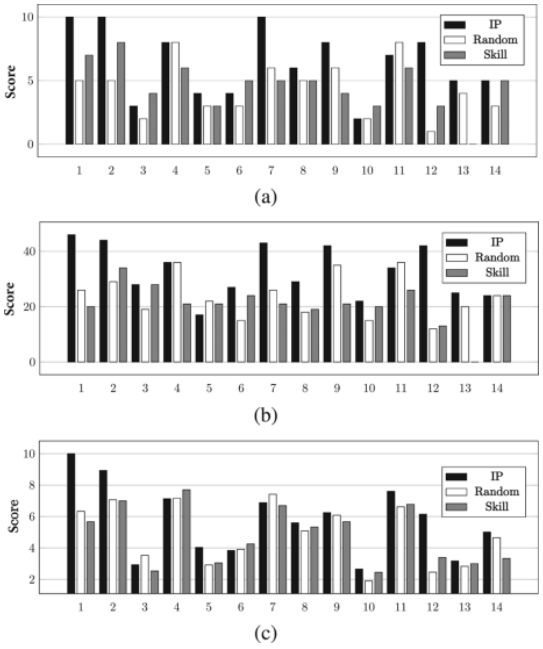

Figure 5: The IP model vs other methods. Metric 1(a) Metric 2(b) Metric 3(c). Taken from [3]

players gave the highest score to the IP model. The random and skill models on the other hand only received the highest scores two and three times respectively. This pattern can be seen again in figure 5(b) where 11 players gave the IP model the highest scores while the random and skill models got it from three and two players respectively. The results for figure 5(c) were essentially the same; the IP model wins by a landslide while the other two get bad results.

In summary this experiment works as a proof of concept, because the IP model worked well against other models in these experiments. The simulations suggests that the method was able to make enjoyable levels that target content appealing to the players

5. CONCLUSIONS

Machine Learning is an unexplored field in the context of Procedural Content Generation. But as we witnessed with both of the methods in this paper, it can be used to generate interesting content in video games. The first method introduced is able to model training data well while also picking nuances in said data. This allows it to create levels with interesting interactions between the level and the player. The second method on the other hand is able to create levels tailored to the players playing the game. This is important because it creates a more immersing gameplay experience. If we're looking at whether we can use Machine Learning to create content; we definitely can, these methods prove this. Machine Learning in this field has an amazing amount of potential. Hopefully in the future we can see these methods evolve and actually see practical use.

Acknowledgements

Thank you to Nic McPhee, Elena Machkasova, and Max Magnuson for their support and feedback.

6. REFERENCES

[1] Markov chain, May 2019.

[2] N. S. T. J. Nelson. *Procedural Content Generation in Games*. Springer, 2016.

[3] J. Roberts and K. Chen. Learning-based procedural content generation. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(1):88–101, March 2015.

[4] S. Snodgrass and S. OntaÃśÃşn. Learning to generate video game maps using markov models. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(4):410–422, Dec 2017.

[5] S. Snodgrass and S. OntaÃśÃşn. Procedural level generation using multi-layer level representations with mdmcs. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 280–287, Aug 2017.

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