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 be 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 (PCG). While this field has been around for a very long time, progress in technology has allowed developers to think up of new algorithms that allow them to generate content better and more suited to their target demographic. 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 [5]. 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.

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 number 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]. However, developers shouldn't use PCG without thought. The more famous games that employ PCG usually have mechanics that take advantage of it. 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 more effectively.

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

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 the main feature in their methods, so it's a given that we need to go over certain Machine Learning concepts. Also, my paper references multiple video games, so we'll be going over a brief overview of said video games.

2.1 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.

2.2 Quake

Quake is a classic game first-person shooter 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 humanity's martial prowess. You play the game in first person, which means you play through the playable character's eyes (like in real life). This also means the game is in 3-d, as opposed to Super Mario Bros. 2-d. And of course, the shooter part means you're shooting things with guns; the game is quite violent.

2.3 Machine Learning

Machine learning is a scientific study that employs algorithms and statistical models to perform tasks without the need for specific instructions. Machine Learning algorithms use training data to build a mathematical model that allows it to recognize patterns. These algorithms can be used in many different applications like image recognition and filtering an email inbox.

2.4 Training

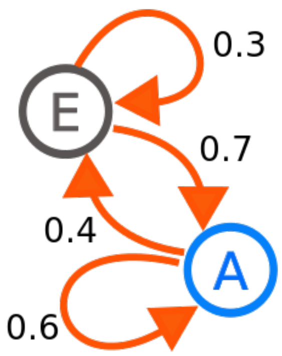

Figure 1: An example of a Markov chain. Transitioning from E to A has a 70% of occurring while staying in E is 30%. Going from A to E has a 40% chance while staying in A has a 60% chance. As we can see all these probabilities equal 100%. This ensures that the current state transitions into something. Taken from [6]

Machines are much better at processing and storing information and than humans. But, any problems not explicitly handled in the code can cause these programs to fail. Machines are not as good at adapting to situations than humans. They are completely dependent on being updated or pre-programmed to handle these situations. Training allows us to get around this. Training is a way to leverage a machine's ability to process information while also making machines more intelligent. By feeding machines with data relevant to a certain task (training data), they can look for patterns and relationships in the data, allowing it to solve many different situations that may arise while performing a task.

2.5 Encoding data

Encoding data is vital for training. Machines can't actually understand the human language. A machine learning algorithm isn't reading through your emails or recognizing your face. It's looking for patterns (learned in training) that allow it to fulfill its tasks. This is why it's necessary to take the real world data you've gathered and turn into something a machine can understand (0's and 1's). Encoding your data is key to getting good data for use in training and testing.

2.6 Markov Chains

Markov chains are a stochastic model (a collection of random variables) [6], which describes a sequence of events such that the current state is dependent on the previous states. Figure 1 shows this. As an example let's pretend we're building a 2D Super Mario Brothers stage, and the stage is broken up into multiple tiles (a square of arbitrary size). If we start at the top left and say the tile at that location represents a part of the sky. We can take the resulting tile (a sky tile) and try to generate the tile to the right of it. If we look at Figure 1, there is a 60% chance to remain in state A and a 40% chance to transition to another state. If sky tiles are represented by state A and cloud tiles are represented by state E, then according to our example, there would be a 40% chance to generate a cloud tile. If a cloud tile is generated then there would be a 30% chance to generate another cloud tile while there would be a 70% chance to generate a sky tile. While there are many more different features in a Super Mario Bros. level this example should hopefully help you understand Markov Chains.

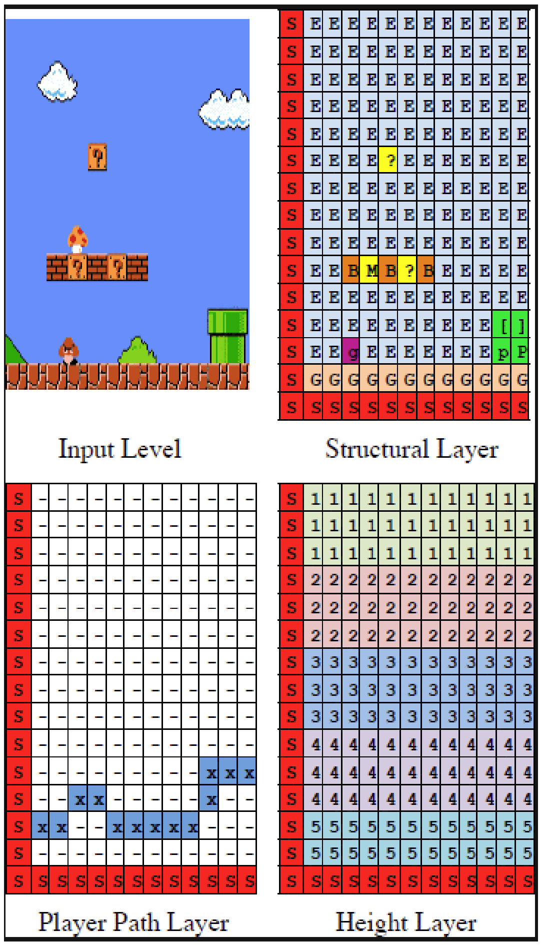

Figure 2: Structural layer (top right), player path layer (bottom left), and height layer (bottom right). Taken from [4]

A multi-dimensional Markov chain is an extension of Markov chains, wherein a multi-dimensional representation is used to represent the Markov Chain. The main feature of this is that any state in the graph can be dependent on any other state [3]. This is important to know because one of the studies found in this paper utilizes matrices to encode levels. However, the current state doesn't have to be dependent on all the other states. In the context of our example above, the current tile may depend on surrounding tiles, far away tiles, or even tiles from other matrices.

3. REPRESENTING LEVELS VIA LAYERS

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 set of tiles, t_i, which varies between these layers [4]. This makes it so that each layer can represent the different parts of the level. As an example, there can be a layer that represents 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 2 shows a section of a Super Mario Bros. level represented using the layers described above.

These layers aren't bound to what is being described in this section. An example this can be a difficulty layer, which, as the name suggests would represent the difficulty through a level (kind of like a heat map). This method seems very straight forward, but it is very flexible; allowing developers to create more complicated, and consistent levels.

3.1 Training

In this section, we will discuss how to train our models, both of which are Multi-dimensional Markov Chains (MdMC). The difference being, one only has dependencies based off of the structural layer, while the other has dependencies spanning multiple layers. At the end of training, we should have a complete Markov Chain with a Conditional Probability Distribution (CPD) which are the probabilities for a state to transition to another state.

To start training a single layer MdMCs we need two things: a network structure, and the training levels (training data). The network structure shows what the current state is dependent on. This is necessary because the levels are represented via matrices and levels are generated using a matrix. The training levels are created by the developers themselves. The structural layer and height layer are generated by hand and the player path layer is generated using an algorithm that traverses the level. The CPD can then be estimated using the patterns found in the training data.

Training the multi-layer MdMC is largely similar to training a single layer MdMC. The requirements are the same. The main differences are that the training levels are being represented using multiple layers, and the resulting Markov Chain may have dependencies from the other layers.

3.2 Sampling

This section we will talk about sampling (generating) new 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, we need our desired level's dimensions and the complete Markov Chain with the CPD as we trained it. To begin sampling we first must pick a starting point. Snodgrass et. al. [4] picked the bottom left corner. We then move from our starting point, 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 at the CPD of our Markov Chain and generate the tile depending on that.

To avoid errors during sampling we employ two procedures: A look ahead and a 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 strange if the top of the pipe was dirt or incomplete. The 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 [4]. During our training process, the researchers 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 an unavoidable, unseen state is observed 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 CPD models the probability of tiles in the main layer, and the network structures contain states from the other layers. This means that the model can have 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 Snodgrass et. al. generated Super Mario Bros. levels. They were especially looking at whether their multi-layered approach was able to recreate levels accurately and create more interesting situations than the 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 using a set of 34 tile types to represent objects (the ground, platforms, pipes, enemies, etc.). There is also a tile used to represent the boundaries of a 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 2 illustrates this set-up.

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 involves a springboard, 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 springboards mentioned above come into play. Springboards 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 number of springs required to complete the sampled level. This allows us to see if the spring is placed there by chance or if the springs are being utilized to com-

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 [4]

plete a level. Another interesting point Snodgrass et. al. is interested in is their approach's 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 levels 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 copies though, it just means that the levels look alike. This is where the similarities end. The single layer MdMC had a hard time placing springboards with intent (relevant to completing a level). In contrast, the multi-layer model was able to place springboards with intent more reliably. This shows that the multi-layer MdMC was able to capture nuances better than the single layer MdMC.

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 were 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 throughout the Figure and even when they don't cross, they are close to each other and are shaped similarly. Snodgrass et. al. were able to measure this by having an algorithm that traverses the level.

The multi-layer MdMC model was able to create levels with more nuance while also allowing for pre-set player paths throughout a level. The single layer MdMC was able to only really mimic structural similarities in the training levels, but

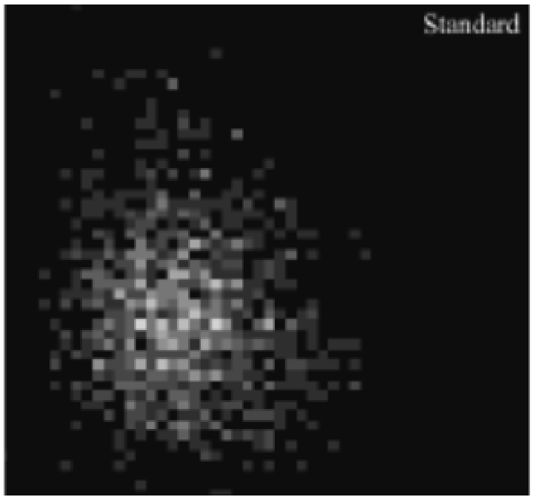

Figure 3: Shows the expressive range of the different models. The y-axis is the leniency and the x-axis is the linearity. Taken from [4]

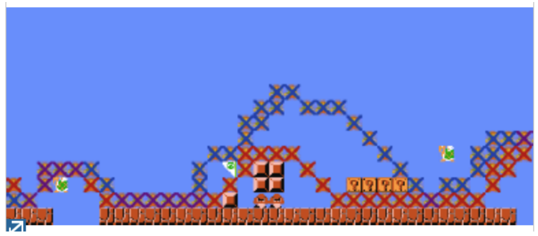

Figure 4: Shows the similarities between the path generated by the model vs the player path given by the researchers. Blue is the generated path, red is the intended path and purple is when the paths cross. Taken from [4]

lose out on everything else.

4. LEARNING BASED GENERATION

A challenge in procedural content generation is the chance of unplayable content being generated [1]. Situations where payers are put into impossible situations, like being placed in a room with enemies that kill the player in one hit. There are existing methods that get around this, however this involves manually changing the algorithm to account for these situations. This can be very expensive especially when the developer has a schedule to follow.

Additionally player feedback is also used to personalize content for a given target 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. To avoid this problem, developers use public playtesters. A generic play style can then be found to design around. But this method is expensive and time-consuming.

A solution 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 a public test phase, and the adaptive stage (involves target players) [1]. 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 Development Stage

In this section, we will start with the development stage, which aims to encode a developer's knowledge. We will look at a brief overview of the stage then take a deeper look into the techniques utilized in the stage, and how those techniques are enabled. 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 based on the content features found in the games [1]. 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.

As described above the ICQ model's job is to look at a set of games and decide which games are good or bad. In the context of Machine Learning, we want this model to learn what the developer deems as a good or bad game. To do this a set of representative games must be chosen by the developers before everything else. These games are then split up into multiple subspaces. Each subspace can have multiple features, Roberts et. al. treated these subspaces as vectors for the model to evaluate. Then the developer plays a representative game from each subspace and decides the quality of the game. This should allow the model to observe patterns and learn which games are good or bad. However, this comes with a caveat that the model is making general decisions about the games in the subspaces. It's completely possible that good games with bad representative games could be overlooked.

The CC model's job is to partition the accepted games into categories. This process is largely similar to the ICQ model's learning process. When the developer is playing the representative games, they are also tasked with labeling the category and content features. The model should be able to use this information to label games.

4.2 The Public Test Stage

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 [1]. To model this 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 of games in categories that the developers chose. 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 GPE model's job is to look at player feedback, estimate the popularity of a game, and find outlier players based on a conformity score given to each player. This should allow for the generation of content based on a target player demographic. To fulfill these representative games are handed to the public to play. These games are separated into different categories (the same ones used for the CC model labels). After a game has been, the player is then directed to indicate whether they enjoyed the game or not. These scores are then used to allow the model to learn the patterns between the feedback given and the category and features of a game to decide the popularity of the game and the conformity of players against the rest.

The PDC model attempts to label whether an experience is positive or negative based on the play-logs recorded during a player's run through a game. By looking at the category, feedback, and play-logs, the model should be able to find patterns and make a binary decision about whether the experience of the players was fun or not.

4.3 The Adaptive Stage

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 should 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 [1].

4.4 The IP model

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 countermeasures.

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 enjoying 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, mean that the IP model cannot find a player's preferred content. If the IP model cannot produce content for a player after many attempts, Roberts et. al. proposes to exploit the ICQ, CC and the GPE model to generate more generalized content.

To test their method 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 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 were 20 games per difficulty categories and they fixed the random number generator so all the survey takers play the exact same game.

The client was distributed via a 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)" [1]. The play-logs produced and the answers gained in the surveys were used to train the PDC model.

Further analysis of the surveys proves 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 the least likely to receive "Very Bad" labels. This shows that parts of the player base have polarizing opinions, which potentially allow the models to categorize players better.

4.5 Testing the IP Model

To test the IP model produced by the previous survey, Roberts et. al. used two baseline algorithms to compare the IP model to. A random model that just generates games using Oblige, a Quake map editor, and the other a Skill model, which uses Oblige again to generate games by manipulating the skill sliders (difficulty, amount of enemies, etc.).

To test the performance of the IP model against the two other models 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 Roberts et. al. defined three metrics [1]. The first question asked after a game is played is the same as the one used in the previous test. The answer to that question is either a yes or no. This scores a 0 or 1 respectively, this is going to be considered as metric 1. Metric 2 is defined using 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. Metric 3 is based on a player's preferred difficulty. This is measured by looking at the answers for metric 1 and looking at the difficulty category.

4.6 Results

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

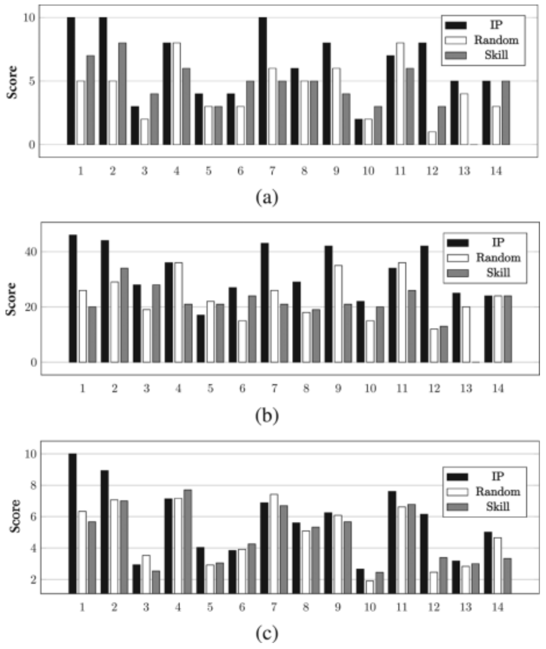

Figure 5: The IP model vs other methods. Metric 1(a) Metric 2(b) Metric 3(c). Taken from [1]

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

This experiment works as a proof of concept. The experiment suggest 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] 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.

[2] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games*. Springer, 2016.

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

[4] S. Snodgrass and S. Ontañòn. 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.

[5] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, 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.

[6] Wikipedia. Markov chain. https://en.wikipedia.org/wiki/Markov_chain.