Procedural Content Generation using Pattern Extraction and Chunk-Based Methods for Level Generation in Mario

Sureshkumar, Harrishan (150294245), ec15191@qmul.ac.uk
Omkar Vinayak, Nadkarni (190758309), ec19577@qmul.ac.uk
Raihan, Khondoker Md Arif (190714383), ec19545@qmul.ac.uk

School of Electronic Engineering and Computer Science
Queen Mary University of London, UK

Abstract. This paper demonstrates the effect of n-grams (Chunk-Based Methods) trained on existing human made game levels for the 2D video game Super Mario Bros. Our experiments produced a variety of diverse playable game levels with similar features to that in the training corpus using simple methods with fast execution times. We investigate an alternative feature/pattern extraction for prediction method and discuss why a Chunk-Based method performs well in this use case. Furthermore, we discuss the differences when using varying chunk sizes as well as using different levels from the game as the training corpus to show how it effects the difficulty and variation in the generated level.

1 Introduction

The development of Procedural Content Generation (PCG) [18] algorithms for games has always been a useful but challenging task. The aspect of content in games include characteristics such as: levels, game rules, maps, stories etc. The goal of PCG in games is to generate these types of content in an algorithmic manner. A wide variety of PCG approaches exist which range from simple methods (i.e. constraint based generative algorithms) to more complex machine learning approaches (i.e. using Deep Neural Networks such as Generative Adversarial Networks) [11]

Out of the vast range of different approaches available we decided to implement a simple but effective chunk-based algorithm and a pattern extraction algorithm both which use existing human made levels of the game as the training corpus to extract features and generate a new level with the obtained features. In this report we will describe a background on related work, the approaches we took, the different generative algorithms we implemented and an analysis of which algorithm we preferred. Furthermore, we will also analyze different settings of our preferred level generator where we will discuss level difficulty and uniqueness.

2 Literature Review

Procedural Content Generation is the approach of using algorithms to create data which is typically produced by a human. PCG has gained widespread popularity in computer graphics and video games in recent times as it enables developers to be boundlessly creative and create content more quickly as opposed to manually handcrafting them. Procedural generation algorithms incorporate random numbers. The types of content that can be created in a game through this approach are – levels, gameplay, rules, audio, narrative and visuals. The use of procedural generation in games originated in table top role-playing games [2] where the game master found a way to generate dungeons and terrain through random die rolls. In 1980, a tile-based dungeon crawling game called Rogue [1] popularized roguelike games, a subgenre of role playing video games uses procedural generation in creating new generated levels with rooms, passages, enemies and treasure every time a game starts. One of the modern and very popular roguelike platformer games, Spelunky [3] strikes a perfect balance between authored and random content and successfully creates exciting and playable levels every time. It generates 16 rooms with 4 possible configurations in a 4x4 grid. No Man's Sky [4], a modern open-ended space exploration game, makes extreme use of PCG to generate a massive universe consisting of planets, ecosystems, flora, fauna, textures, ships and more. A small-scale application of procedural generation can also be very effective in working on certain aspects of a game. The role-playing first-person shooter game Borderlands [5], used procedural generation for creating weapons which enhances one part of the game and makes it more exciting.

There are many criteria based on which we can distinguish between procedural content generation algorithms. Out of those criteria, we will focus on the distinction [5] between the algorithm belonging to a constructive or a generate-and-test category. A constructive algorithm creates contents on the fly, based on certain constraints and elements of randomness. A generate-and-test algorithm generates content and then regenerates again if the content doesn't satisfy the specified condition. Some of the popular constructive algorithms worth mentioning are Cellular Automata [7], Perlin Noise [6], Generative Grammar [8] and Doran and Parberry [9]. Search based algorithms are a special case of generate-and-test algorithms that doesn't simply reject a piece of generated content but assigns a fitness number to generated contents and optimizes on top of it. Evolutionary algorithms are the common method of choice when it comes to search based procedural content generation. A type of evolutionary algorithm called the Quality diversity algorithm [10] looks for a set of good solutions with respect to behavior and performance. Machine learning approaches [11] like Generative Adversarial Networks (GAN), Long Short-Term Memory Recurrent Neural Networks, Markov Models and more have been used in the recent years to generate levels by training on existing content.

3 Benchmark

Super Mario Bros [12] is a side-scrolling platform game where the player has to take on the role of the character named Mario in a two-dimensional world. The main objective for Mario is to reach the flag at the end of each level. Optional goals include collection of coins, killing more enemies, finishing a level as fast as possible and getting the highest score. Mario can take the following actions – move left, move right, jump and shoot fire-balls which depends on what state he is in. The game world has coins scattered all over the level and contains normal, special and hidden blocks that reveals coins and power ups such as super mushrooms. Eating a super mushroom evolves the state of Mario from small to big and eating a fire flower power up provides him with the ability to shoot fire-balls. Mario loses his power up if he touches an enemy instead of stomping it down. Mario turns from fire-ball state to big and big to small respectively when he touches an enemy in these states.

Among some of the recent works in level generation for Super Mario Bros, a method worth mentioning is Higher order Markov chains [13], which was employed to determine the probabilistic distribution of every tile in a level map by learning the probability of a tile given the previous tiles to the immediate left, immediately above, and the one to the left and above. By changing the order of Markov chain to high or low when it's necessary, improved the quality of the generated levels. Another piece of work that demonstrated procedural generation of Mario levels using Long Short-Term Memory Recurrent Neural Networks [14], treated every tile as a point in a sequence as LSTMs excel at predicting the next item in a sequence. They trained the neural network from 39 original levels from Super Mario Bros and Japanese Super Mario Bros. They applied player path information during training to get more playable levels of better quality. There was a competition [15] in 2010 for gameplay, learning and level generation in Super Mario Bros. Some of the participants in the competition generated highly challenging levels that had an excess number of rocks and gaps, levels that had very wide gaps that are placed in an unpredictable manner, levels that are characterized by many coins and levels that contained high number of powerups. It was seen from the results that, few numbers of coins, rocks, gaps and enemies were preferred and considered to generate more fun.

4 Background

The two algorithms used in this report for level generation in Super Mario Bros are N-gram Style Capture and a method using pattern extraction which are explained in detail in the following subsections. The levels generated by these algorithms are evaluated by an A star agent for playability before displaying the level.

4.1: N-gram Style Capture (Chunk-Based approach) [17]

This algorithm proposes a simple strategy where by it randomly selects a chunk (vertical strip of specific size) from an existing human made level and places it into the new level being generated. It then randomly obtains another chunk from an existing human made level and places it beside the first chunk and so on until the whole level is generated. This method is described to be surprising for the first few games for the player but the repeated chunks are said to soon become recognizable as the generator is used more and more.

This method is referred to as n-gram style capture where ‘n’ is a sequence of items from a text/training corpus [17]. In this case ‘n’ refers to the number of columns (vertical strips) in a level chunk.

4.2 Pattern Extraction

Pattern extraction is loosely based on the idea of Markov chains [13]. Pattern extraction uses a pattern of preceding tiles to determine the type of current tile. This is done by first extracting the patterns from different levels already available in the framework. More number of levels allow for a better map generation. The number of preceding blocks that are responsible for determining the current block depend on the pattern length specified in the algorithm. This method gives very unpredictable results sometimes as the placement of blocks and enemies in the game is not logical, but the levels are diverse every time and with the use of a generate-and-test function a playable level is generated.

5 Techniques Implemented

5.1 Chunk-Based Generator

This generator takes in a specific number of human-made levels (training corpus) and a specified chunk size. The chunk size specifies the number of consecutive columns which are extracted as a chunk from the training corpus as shown in *Figure 1*. It creates a pool of all the extracted chunks in a consecutive order from all levels in the training data and stores it in the form of a two-dimensional array. When the new level is being generated the algorithm randomly chooses chunks from the generated pool and assigns them to a location in the new level. Once the two-dimensional array representing the new generated level is filled the level is then returned.

Problems mentioned in section 4.1 regarding repeated chunks becoming more and more obvious with time is an issue that was overcome by increasing the number of human-made levels in the training corpus used for extraction of chunks when generating the pool and by decreasing the chunk size to an optimal range.

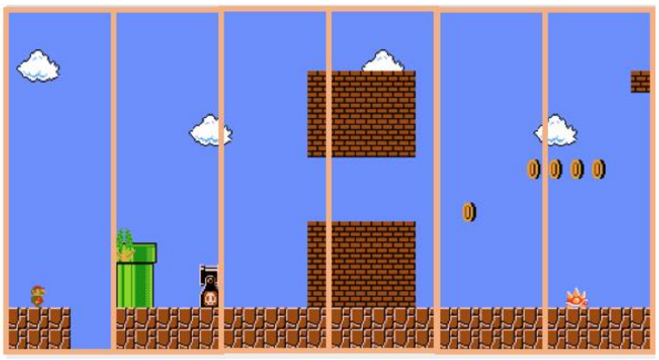


Figure 1: Each orange chunk represents a random chunk taken from the generated pool. A series of chunks then make up the new level

5.2 Constraint Based Smoothening

Due to the randomized selection of the chunks, we found that sometimes, a tall column of solid objects or other large overlapping objects prevented the level from being playable. Also due to the intersection between level chunks in the pool, half a pipe could be placed instead of a full one (as pipes take up two blocks). When there was a flower on top of the half pipe, half of the flower would be sticking out. Both problems were overcome by implementing constraint-based smoothing on top of the generated levels to create a smooth playable transition between the chunks.

5.3 Pattern Extraction Based Generator

This method extracts patterns from the levels that already exist in the framework to make predictions in the newly generated map. The algorithm works in the following way.
All the levels from which patterns are to be extracted are divided into three sections horizontally as shown in *Figure 2* and the pattern's prediction characters of each section are stored separately.

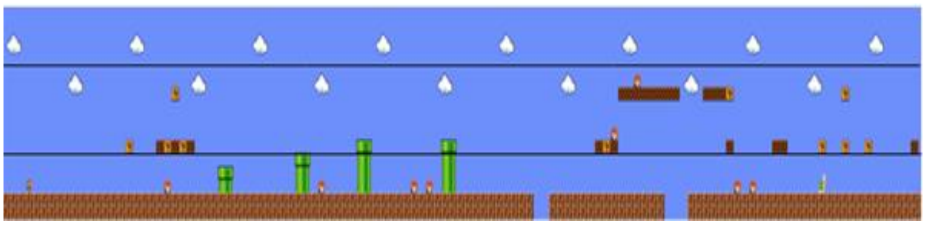


Figure 2: Division of levels into horizontal sections (sky (0-4), intermediate (5-11), ground (12-15))

The patterns are extracted row wise and each pattern consists of specified fixed length. The character that appears after this pattern is the prediction character which is stored in a list as shown in *Figure 3*. The patterns and respective prediction characters are stored in HashMap in java as key-value pairs where the pattern is the unique key and the value is an array of possible character values that could occur after this pattern. Whenever this pattern is encountered again the character array is updated.

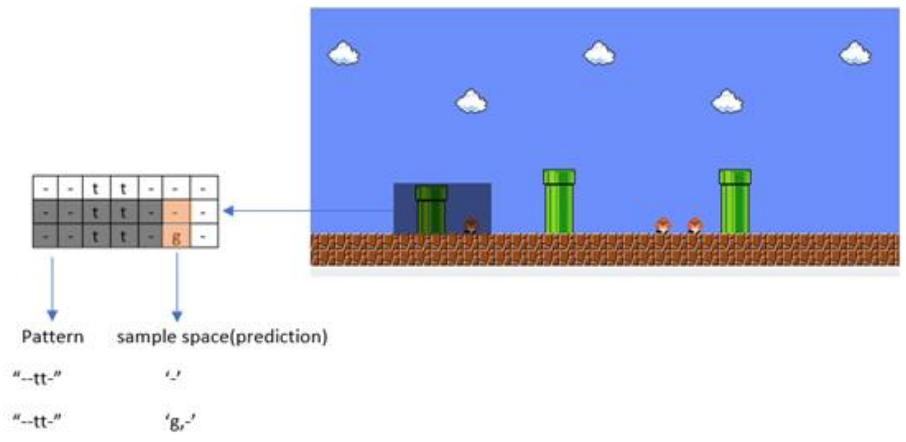


Figure 3: Extraction of pattern and characters that represent tiles, from a sample level

During level generation whenever a certain pattern is encountered in the newly generated level a random character is picked from the sample space to fill that tile position. The list does not keep count of each character and therefore the list can contain repetitive characters. For example, if a list contains 5 characters - (‘X’,’X’,’X’,’-’,’g’) and one character is picked at random, there is a probability of 60% that the character picked is ‘X’ (ground) and the probability of ‘-’ (empty) and ‘g’ (goomba) is 20% each. This is why frequency of each character in the sample space is important and therefore they are not discarded. A final level generated from extracted patterns is shown in *Figure 4.*

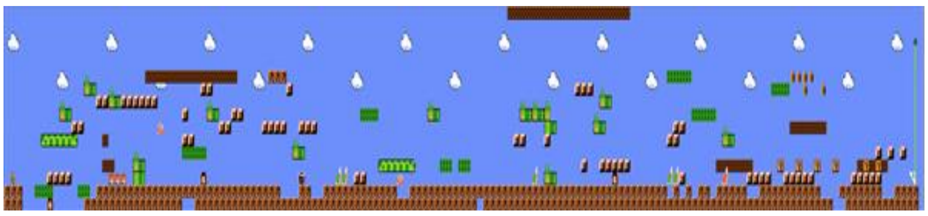


Figure 4: Example of level generation from extracted patterns approach

5.4: Generate-and-test method

After multiple level generations using the chunk-based generator, we still encountered a small chance of producing an unplayable level. This was overcome by implementing a function which simulates multiple runs on the generated level to confirm if the level could be completed by an AI agent in a given amount of time. If the level is unplayable, it regenerates and tests again. This goes on until it finds a generated level that is playable.

6 Experimental Study

6.1: Chunk-Based vs Pattern Extraction Based Generator

To reach our desired goal of making a PCG algorithm which generates a playable level every time it is executed, we compared the 2 algorithms we implemented. When running each generator 50 times we found that the pattern-based generator produced unplayable levels most of the time whereas the Chunk-Based Generator produced playable levels most of the time.

We tested the consistency of playable levels generated by both algorithms.

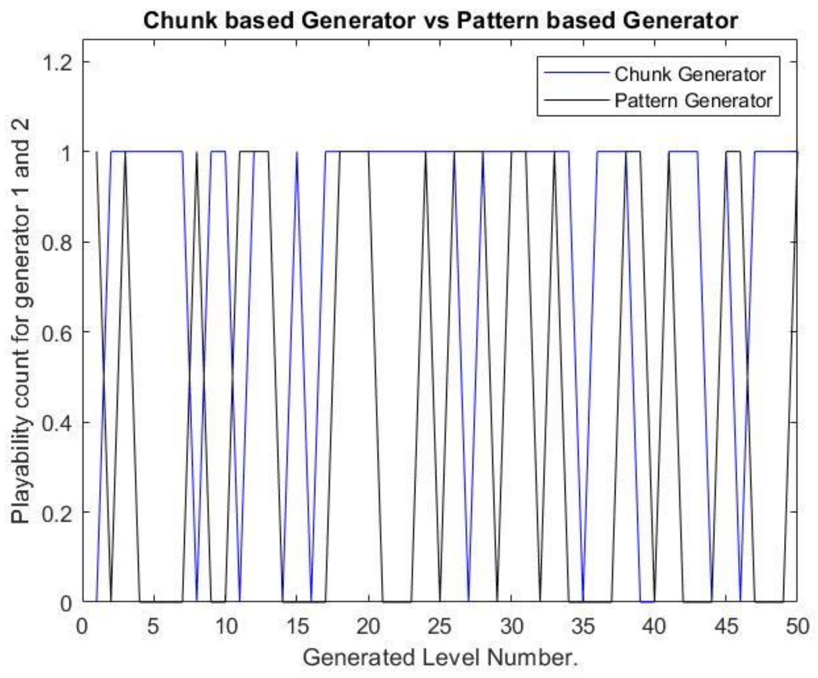


Figure 5 – Plot which shows count of playable levels vs generated level

The chunk-based generator produced 39 playable levels out of 50 generations and the Pattern-Based generator produced only 22 playable levels. In conclusion the Chunk-Based generator had a success rate of 78% whereas the Pattern-Based generator had a success rate of 44% thereby proving the Chunk-Based generator produced better and more consistent results. Due to the poor performance of the Pattern-Based generator we decided to use only the Chunk-Based Generator. The consistency in the generation of playable levels by the chunk-based generator can be seen in *Figure 5*.

6.2: Chunk-Based Generator: Difficulty vs Uniqueness
The Chunk-Based Generator takes in 2 parameters, 1 is the chunk size and the other is the number of levels to be used as the training corpus. By running an A* based search agent [19] to control Mario on our generated level we needed a useable performance metric as a measure of difficulty. After analyzing the performance metrics after each execution, we found the number of jumps Mario made in each generated level varied with chunk size. We recorded the average number of jumps across 10 runs for each configuration which is shown in Table 1.

Chunk Size	Levels used as training corpus	Jumps (Average)
1	1-5	15
2	1-5	13
3	1-5	11
1	1-10	16
2	1-10	13
3	1-10	11
1	1-15	15
2	1-15	14
3	1-15	13

Table 1 – Average number of jumps of agent for varying chunk sizes and number of levels used as training corpus

When increasing the chunk size above 3 we noticed the time remaining and the number of jumps made by the agent remained somewhat consistent thereafter. Additionally, we found chunk sizes larger than 4 started to cause visible repetition of chunks after multiple plays so did not record results beyond a chunk size of 3 for this experiment.

The simulations showed that the smaller the chunk size, the more jumps the agent had to make possibly due to the roughness of the terrain (i.e. shorter horizontally flat regions/ high random variation for placement of solid blocks) and the larger the chunk size the less jumps the agent had to make due to the smoothness of the terrain (i.e. longer horizontally flat regions/ lower random variation for the placement of solid blocks).
However, as we increased the number of levels used as the training corpus this pattern began to fade and became more random. This is due to the fact that, there is more

random variation and a larger number of combinations for the new generated level thereby concluding that when using more levels as the train corpus the generated map is much more unpredictable and more unique.
In general, we found that a smaller chunk size produced a harder level with more variation (more unique) and when a smaller chunk size is used with more original levels in the training corpus (see figure 6), then the level tends to be harder due to the rapidly changing structure in the level between chunks. However, when using a larger chunk size (see figure 7) the generated level was easier to play as the structure throughout the level doesn't rapidly change as much as when using a smaller chunk size. When using a larger chunk size with a smaller number of original levels for the training corpus we found the generated level to be more predictable and have less random variation (less unique).

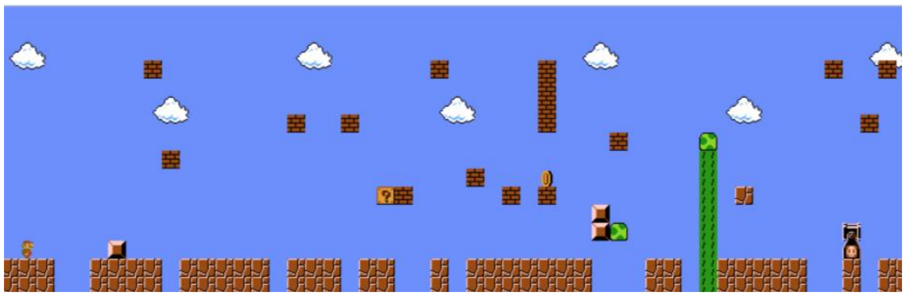


Figure 6: Generated Level, Chunk Size 1, Train Corpus Levels 1-15
(More random variation, More potential jumps, More inaccessible bricks, Harder than level below)

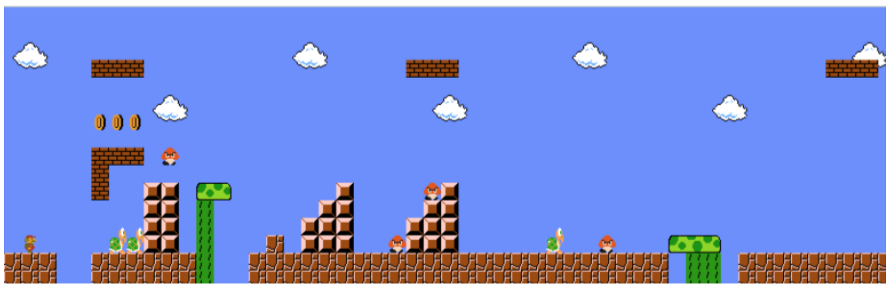


Figure 7: Generated Level, Chunk Size 3, Train Corpus Levels 1-5
(Less random variation, Less potential jumps, Less in accessible bricks, Easier than above level)

6.3: Player Performance: Agents vs Humans

From the results obtained in Table 2 we can see that even the best AI agent (Robin Baumgarten, A* search-based agent [19] provided in the framework does not win every time. The agent seems to only complete 8 out of 15 of the original levels.

Levels	Number of Jumps (Average)	Win Rate over 10 iterations (%)	Causes of Death	Loss Rate (%)
1	15.5	100%		0%
2	12.5	100%		0%
3	6	10%		90%
4	7	0%	Died on pipe flower 6 times Ran out of time 4 times	100%
5	11.7	80%	Died on pipe flower twice	20%
6	6	0%	Died near grass by falling	100%
7	16.5	90%	Died on pipe flower	10%
8	18.5	100%		0%
9	17	100%		0%
10	7	0%	Died near grass by falling	100%
11	20.5	100%		0%
12	1	0%	Died on pipe flower every time	100%
13	11.3	10%	Died near grass by falling	90%
14	14	100%		0%
15	7	0%		100%

Table 2 - Agents Performance on Original Levels over 150 iterations

The weakness of the agent is analysed to be unable to handle some “pipe flowers” and “grass platforms” in the game as these are the 2 main ways it dies. Upon playing the game it was observed that the difficulty increased with each level and this is not reflected when the agent plays the original levels. The agent only dies when it faces a weakness and does not experience difficulty in the same way a human player does. We found that the agent struggles with the levels which include ‘grass platforms’ and sometimes with ‘pipe flowers’ but performs exceptionally well in other levels within a much shorter time frame than a human.
Human players on the other hand had no issues with the levels that the agent died on, including ‘grass platforms’ and ‘pipe flower’. Similarly controlling Mario’s momentum in each run proved to be a difficult task for humans but was not an issue with the agent. From this we concluded that using the agent to check if our Chunk-Based generator produced playable levels was not a perfect measure as sometimes the agent can lose on playable levels meaning that our Chunk-Based generator would actually perform better than discovered in section 6.1 as some playable levels would be declared as unplayable when using the generate-and-test method (see section 5.4) which makes use of the agent to confirm whether the level is playable. (See Table 3).

Training Corpus (Original Levels)	Average Number of Jumps for levels won	Win Rate out of 10 Generations	Causes of Death	Loss (Humanly Playable)	Loss (Humanly unplayable)
1	12.5	100%	-	0%	0%
1-2	13.3	70%	-	10%	10%
1-3	14.0	60%	Died near grass	20%	20%
1-4	14.5	70%	Hard level missed jump near grass	10%	20%
1-5	14.3	90%	Hard level missed jump near grass	10%	0%
1-6	15.7	90%	-	0%	10%
1-7	15.5	60%	Died on pipe flower twice and once near grass	30%	10%
1-8	16.0	80%	Died near grass twice	20%	0%
1-9	15.7	90%	-	0%	10%
1-10	15.5	80%	Died near grass once Died on pipe flower once	20%	0%
1-11	15.7	90%	-	0%	10%
1-12	16.3	90%	Died on pipe flower	10%	0%
1-13	17.0	70%	Died on pipe flower	20%	10%
1-14	17.3	80%	Died on pipe flower	10%	10%
1-15	16.7	90%	Died on pipe flower	10%	0%

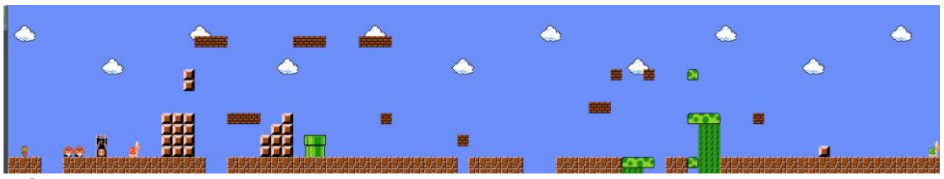
Table 3 - Agents Performance on Generated Levels using Chunk-Based Generator without generate-and-test evaluation function to check playability (For comparison to the original levels we chose a chunk size of 3 as that would produce outputs which are more similar to the training corpus but still diverse. Each row in the table used 10 different generated levels for evaluation)

By looking at Table 2 and Table 3 we can compare the performance of the agent on the original levels to the performance on the generated levels by analysing the average over 10 iterations for each configuration. The weakness of the agent remains the same as demonstrated in Table 3. When taking into account the original levels completed by the agent, we can calculate the average number of jumps taken by the agent to win to be about 15.8 and by getting the average number of jumps for the generated levels we can see it is a similar value of around 15.3. When using the number of jumps as a performance metric for difficulty we can say that the Chunk-Based generator reproduces a similar overall difficulty across different training corpuses.

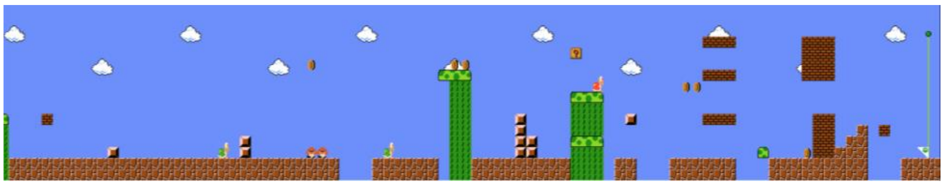
6.4: Quality of Generated Levels

Generated Level 1 (Chunk Size:3 and Training Corpus: Levels 1-15)

1st half of level

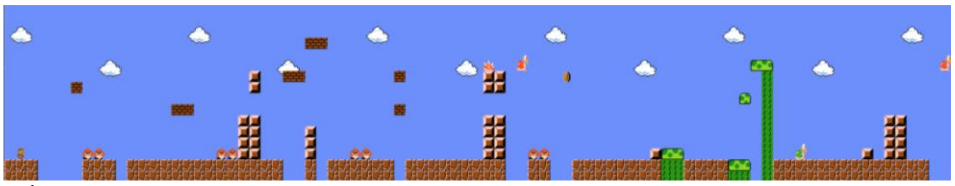


2nd half of level

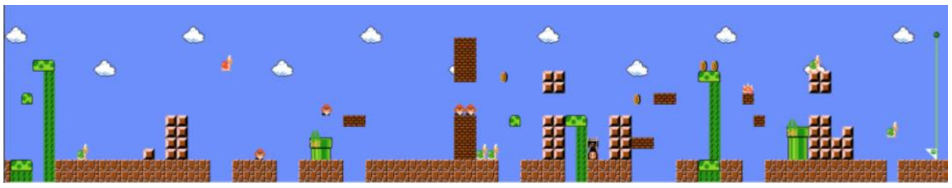


Generated Level 2(Chunk Size: 2 and Training Corpus: Levels 1-15)

1st half of level

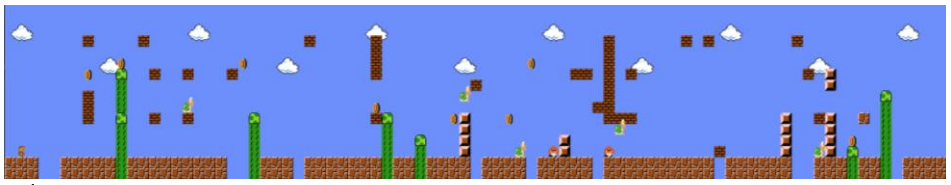


2nd half of level

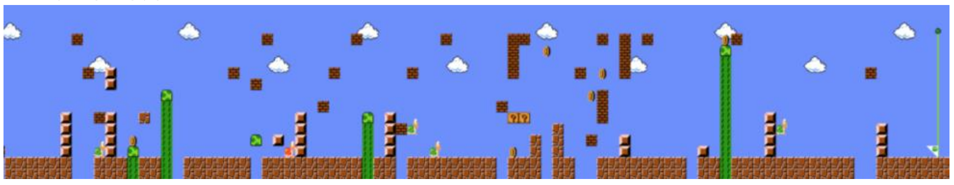


Generated Level 3: (Chunk Size:1 and Training Corpus: Levels 1-15)

1st half of level



2nd half of level



We asked 6 human players to play the 3 generated levels and to fill out a questionnaire. Players 1 to 3 had played the game before and players 3 to 6 had not.

Generated Level 1 (Chunk size:3) Feedback

Player	Difficulty (1-10)	Diversity (1-10)	Enjoyability (1-10)	Number of tries to Complete the level	Comments & Criticism
1	3	9	10	2	Cannon came too early on in the level
2	6	8	9	4	Coins in unreachable places
3	4	7	9	3	-
4	5	10	10	6	Unreachable question box
5	4	8	8	7	-
6	6	7	8	9	-
Average	4.7	8.2	9.0	5.2	

Generated Level 2 (Chunk size: 2) Feedback

Player	Difficulty (1-10)	Diversity (1-10)	Enjoyability (1-10)	Number of tries to Complete the level	Comments & Criticism
1	5	10	10	3	-
2	7	8	9	13	One coin unreachable
3	6	8	9	3	-
4	6	10	8	8	-
5	7	9	7	8	-
6	7	7	6	10	-
Average	6.3	8.7	8.2	7.5	

Generated Level 3 (Chunk size: 1) Feedback

Player	Difficulty (1-10)	Diversity (1-10)	Enjoyability (1-10)	Number of tries to Complete the level	Comments & Criticism
1	7	10	6	12	Many unreachable blocks
2	9	9	6	Couldn’t	-
3	8	9	7	10	-
4	10	10	5	19	-
5	9	9	7	27	Too hard to jump onto single blocks
6	9	7	6	Couldn’t	-
Average	8.7	9.0	6.2	17	

7 Discussion

From the collected feedback from the human players we can see that the average difficulty and ‘number of tries’ increased with the decrease of chunk size. This confirms our findings in section 6.2 where we record the number of average jumps of the agent when playing levels generated for varying chunk sizes. By using the number of jumps as a performance metric to measure the difficulty of the level we could see a link between chunk size and difficulty of the level.

When using a smaller chunk size, the levels were harder than the generated levels with larger chunk size. This is mainly due to the difficulty to control the momentum of Mario when jumping to small block platforms. Through direct observation we found that when players face a challenge which involves jumping and landing on a small block platform, they either jumped too far or too little thereby missing the block completely.

By using all 15 original levels in the training corpus we were able to make the generated levels more diverse. The diversity of each level increased, as the chunk size decreased. Additionally, we found that the average enjoyability decreased as the difficulty increased therefore showing it is important to get the right balance of enjoyability and difficulty.

8 Conclusions and Future Work

We demonstrated through experimental study that when using a Chunk-Based generator to generate levels in ‘Super Mario Bros’ it is best to use a wide range of levels to be used as the training corpus for a greater diversity measure. It was observed that an ideal chunk size should be greater than or equal to 1 and less than or equal to 3 to have good balance of playability and diversity. When a chunk size larger than 3 is used, it starts to become obvious over multiple iterations where the chunks repeat and the levels start to look more closely to the levels in the training corpus.

Improvements can be made to the constraint-based smoothing used to smoothen out transitions between chunks in the generated levels thereby making it easier to travel from chunk to chunk. By implementing a better constraint based smoothing process we could prevent the generator from producing unplayable levels thereby allowing us to get rid of the generate-and-test function. This would save time and computation as the agent will no longer be needed to run simulations to check if the level is playable before outputting it to the human player.

In addition, based on some criticism from our questionnaire feedback we saw players encountered problems such as unreachable objects in the generated levels. This could also be improved by running an algorithm which checks that all blocks/objects are reachable by generating a path between blocks/objects if they are not accessible.

In conclusion, we found that even though the Chunk-Based generator makes use of a rather simplistic approach it produces good results and provides an enjoyable experience for human players. It is a fast, efficient and usable approach for procedural content generation in games like ‘Super Mario Bros’.

References

1. Toy, M., Wichman, G., Arnold, K. and Lane, J., 1980. Rogue (PC Game).
2. Brown, J.A. and Scirea, M., 2018, June. Procedural Generation for Tabletop Games: User Driven Approaches with Restrictions on Computational Resources. In *International Conference in Software Engineering for Defence Applications* (pp. 44-54). Springer, Cham.
3. Yu, D., 2016. *Spelunky: Boss Fight Books# 11* (Vol. 11). Boss Fight Books.
4. Games, H., 2016. No man's sky, 2016. URL: http://www. no-mans-sky. com/about/(visited on 05/30/2016).
5. Togelius, J., Yannakakis, G.N., Stanley, K.O. and Browne, C., 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*(3), pp.172-186.
6. Perlin, K., 2002, July. Improving noise. In *ACM transactions on graphics (TOG)* (Vol. 21, No. 3, pp. 681-682). ACM.
7. Adamatzky, A., 2010. *Game of life cellular automata* (Vol. 1). London: Springer.
8. Van der Linden, R., Lopes, R. and Bidarra, R., 2013, November. Designing procedurally generated levels. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.
9. Doran, J. and Parberry, I., 2010. Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games*, *2*(2), pp.111-119.
10. Gravina, D., Khalifa, A., Liapis, A., Togelius, J. and Yannakakis, G.N., 2019, August. Procedural content generation through quality diversity. In *2019 IEEE Conference on Games (CoG)* (pp. 1-8). IEEE.
11. Summerville, A., Snodgrass, S., Guzdial, M., Holmgård, C., Hoover, A.K., Isaksen, A., Nealen, A. and Togelius, J., 2018. Procedural content generation via machine learning (PCGML). *IEEE Transactions on Games*, *10*(3), pp.257-270.
12. Miyamoto, S., Yamauchi, H. and Tezuka, T., 1985. Super Mario Bros. *Nintendo Entertainment System. Nintendo*.
13. Snodgrass, S. and Ontañón, S., 2014. Experiments in map generation using Markov chains. In *FDG*.
14. Summerville, A. and Mateas, M., 2016. Super mario as a string: Platformer level generation via lstms. arXiv preprint arXiv:1603.00930.
15. Shaker, N., Togelius, J., Yannakakis, G.N., Weber, B., Shimizu, T., Hashiyama, T., Sorenson, N., Pasquier, P., Mawhorter, P., Takahashi, G. and Smith, G., 2011. The 2010 Mario AI championship: Level generation track. IEEE Transactions on Computational Intelligence and AI in Games, 3(4), pp.332-347.

16. Dahlskog, S. and Togelius, J., 2014, April. Procedural content generation using patterns as objectives. In European Conference on the Applications of Evolutionary Computation (pp. 325-336). Springer, Berlin, Heidelberg.
17. Dahlskog, S., Togelius, J. and Nelson, M.J., 2014, November. Linear levels through n-grams. In Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services (pp. 200-206). ACM.
18. Dahlskog, S. and Togelius, J., 2014, April. Procedural content generation using patterns as objectives. In European Conference on the Applications of Evolutionary Computation (pp. 325-336). Springer, Berlin, Heidelberg.
19. Hart, P.E., Nilsson, N.J. and Raphael, B., 1968. A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics, 4(2), pp.100-107.