Automatic Generation of Super Mario Levels via Graph Grammars

Eduardo Hauck
Department of Computer Science
University of Tsukuba
Tsukuba, Japan
eduardohauck@gmail.com

Claus Aranha
Department of Computer Science
University of Tsukuba
Tsukuba, Japan
caranha@cs.tsukuba.ac.jp

***Abstract*—Automatically generating game levels using Procedural Content Generation (PCG) is a challenging problem because of the necessity of attending both functional and nonfunctional requirements. In the particular example of level generation for Super Mario, Machine Learning approaches, such as GANs and Reinforcement Learning, have shown promise. However, these black-box approaches usually are not explainable, and thus difficult to integrate with human designers. We propose a different level-generation system that uses a reachable graph structure, automatic detection of level structures, and formal graph grammars. A human designer can also easily add or remove structures to use the system as a level co-creation tool. An experimental analysis shows that the proposed system can generate playable and visually pleasing levels, while revealing some limitations of current approaches on platformer level generation, such as the generation of backtracking segments.**

***Index Terms*—Procedural Content Generation, Graph Grammar**

I. Introduction

Procedural Content Generation (PCG) has been used by the video game industry to overcome technical limitations, create new ways of playing games and accelerate development. However, automatically generating levels with similar quality as to human-authored levels is still a challenge. Level generation is a complex task because levels have both functional and nonfunctional requirements that have to be met [1]. Creating high-quality levels is a creatively demanding task that can be difficult even for humans. Creating tools that can support or accelerate the work of designers can be the key to develop better games.

The application of Machine Learning to PCG (PCGML) [2] has shown promising results for level generation. For example, Generative Adversarial Networks (GANs) have been applied to generate certain types of levels [3] or to leverage the limited dataset usually available for games [1]. While these approaches are successful in generating levels automatically, the complexity of interpreting the results of a neural network makes it difficult to develop co-creation tools that can be used intuitively by players or designers.

When creating new levels, humans often play with the recombination of patterns seen or used previously. In this way, level design is done at a higher level, where the focus is on the role of each pattern, and how their interaction affects gameplay. We believe that a generation system working at this higher recombination level will be easier and more intuitive to collaborate with for a human designer, and this idea is the main motivation of our work.

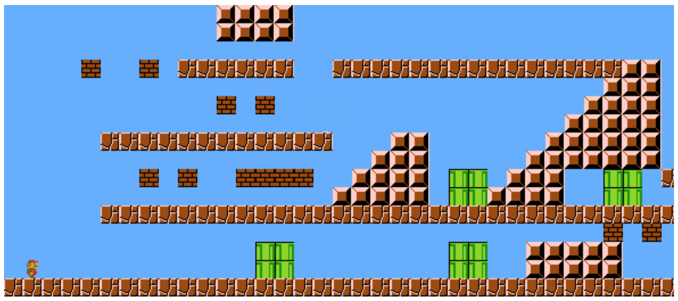


Fig. 1: Section of a generated level that offers multiple paths, where at least one of them is traversable. If the player decides to go through the bottom path, it will be forced to backtrack.

Londono and Missura proposed a system following the same idea of high level composition of a level from set patterns [4]. They sought to identify frequent patterns in a level, and to use these patterns to generate new levels. They propose a graph grammar learning system based on the minimum description length (MDL) principle. The system induces a graph grammar that can reproduce an input level and also generate new levels based on the patterns identified. However, although it was an interesting idea, only a superficial description of this system was provided, with no results or analysis.

Inspired by their work, we propose a new level generation system based on graphs. The system is capable of extracting, processing, and recombining patterns based on the data of one or more input maps. These patterns are stored as simple files that can be edited or even manually created by a human developer, and then used in the generation of new levels. By extracting patterns from user designed levels and recombining these patterns in different ways, we expect that the levels produced by the system will exhibit some of the features present in the original levels, while at the same time exhibit new features emerging from the automated recombination.

We implement and evaluate this system using the Mario

AI Framework [1] to generate levels for the Super Mario Bros. game. Experiments show that the system is capable not only of generating playable and often visually appealing levels, but also levels that present one or more paths leading to a dead-end that require the player to backtrack. Figure 1 presents an example of a generated level with this property. In the future, we plan to evaluate more generation methods using the same system as a basis, and then use the system to develop human-friendly design assistant tools that can be used by designers and players alike.

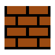
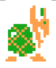
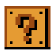
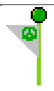
II. Related Work

A. Mario AI Framework

The Mario AI competition [5] was introduced in 2009, tasking participants to create AI agents and level generators for the Super Mario Bros videogame. It was built on top of Infinite Mario Bros (IMB), an open-source clone of Super Mario Bros. However, unlike the original game, the levels are procedurally generated using a handcrafted method developed by the creator of IMB, Markus Perrson.

The Mario AI framework stores and render levels from text files, where each symbol represents a tile to be rendered on the screen. Table I shows the symbols used for each sprite present in the first level of the original game, commonly referred to as World 1-1. In this work, we read and generate levels using this same format.

TABLE I: Symbols used for each sprite in the Mario AI framework. This table shows all the elements used in the first level of Super Mario Bros, World 1-1.

Sprite	Symbol	Type
	M	Spawn location for the character
	-	Air (empty tile)
	X	Ground
	#	Platform
	S or C	Platform (C symbol contain a coin)
	g	Goomba (enemy character)
	k	Koopa (enemy character)
	t	Pipe (formed by 2 tiles side by side)
	! or @	Question Block (contains coin or mushroom)
	F	Finish line (1 symbol covers the whole column)

B. Level Generation in Super Mario Bros

The framework for the Mario AI competition became a particularly useful tool in the PCG field, as it proposes a difficult challenge in a videogame environment that is simple enough to manipulate and test. For example, Shaker et al. proposed a grammatical evolution approach. They employ the combination of a grammatical representation of the level with an evolutionary algorithm to evolve the grammar. The grammar rules contain elements of the game and a position in which they should spawn. Candidates are then generated with a certain number of these string components, and an evolution is performed to obtain a certain type of level [6]. Another approach, by Khalifa et al., combines evolutionary algorithms with quality-diversity (QD) algorithms that maintains the diversity of the population and generate levels with varying characteristics [7].

The methods mentioned are categorized within the Search-based PCG [8] category. They employ a heuristic function that approximates the quality of a candidate solution to guide the generation. Although these methods have shown to be able to generate levels, they often require handcrafted functions that use domain knowledge on the subject. This can be challenging, as it is not clear how the overall quality of a level could be expressed by a single formula, and this value could even change between different games.

C. PCGML

Procedural Content Generation via Machine Learning (PCGML) [2] is an emerging field that aims to leverage all the recent development achieved in the ML field to improve content generation through the use of existing data. Unlike pure Search-based methods, which require some domain knowledge of the problem, PCGML methods extract and model the data as a basis for the generation.

One of the challenges in machine learning is that of the development of explainable methods. That is, given a certain method and a certain output, an explainable method allows for a human user to understand why or how the method came to produce that output. The effort towards ”Explainable AI” can lead to systems that are more human-friendly and allow more creative use of its functionalities. Recent works in PCG are starting to address this issue, such as the design pattern classification system proposed by Guzdial et al. [9].

D. Graph Grammar learning for Super Mario Bros

London and Missura [4] proposed a graph grammar learning system to generate Super Mario levels. This can be a better approach in terms of explainability, as graphs are visual structures and grammar rules based on them would allow for an intuitive understanding of the generation.

They follow the assumption that human-designed levels contain high-quality patterns that can be extracted and used

[1]http://marioai.org/

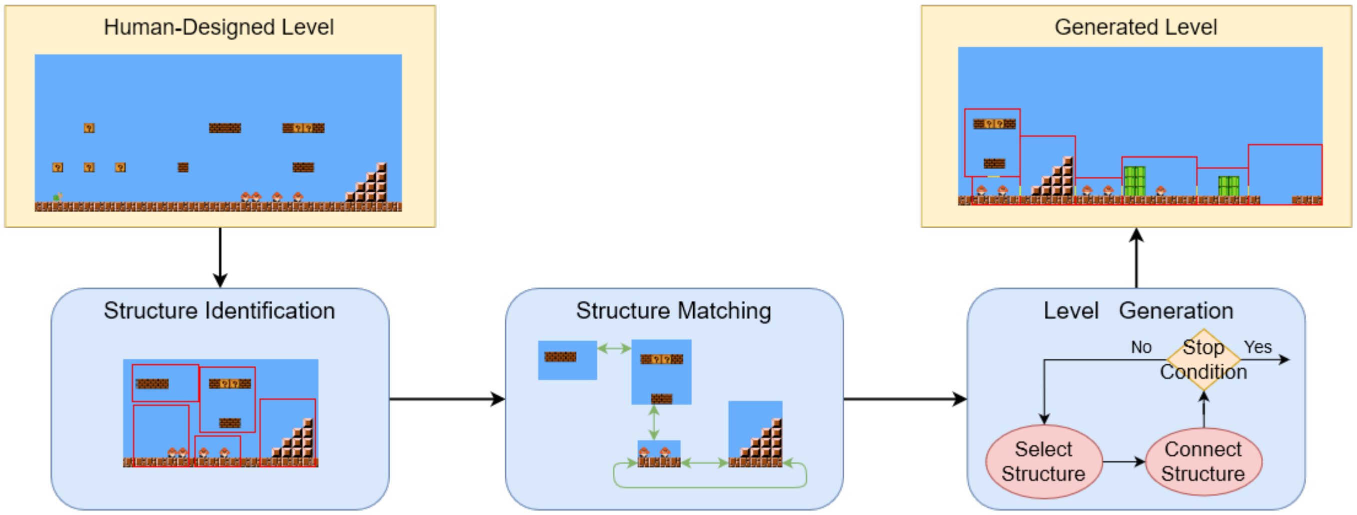


Fig. 2: The three main stages of the proposed system.

to generate new levels. To achieve this, they implement the SubdueGL algorithm [10], an algorithm that can obtain a context-free graph grammar based on the discovery of frequent structures in a graph.

They propose to model the elements of a level as nodes of two types:

- Platform nodes: represent solid tiles on which the character can stand on. A contiguous sequence of platform tiles on the horizontal axis can be represented by two nodes connected by an edge.
- Cluster nodes: represents non-solid tiles such as coins or enemies that the character can interact with. Clusters are formed from these elements by employing a simplified version of the clustering algorithm $GDBSCAN$ [11]. All the nodes of a cluster form a connected component.

A third type of edge is also employed based on the concept of reachability. If a character can navigate from a platform node to a cluster node (e.g. through a jump), then a reachability edge is added between these two nodes. The reachability concept is used to connect platform and cluster nodes together and ensure that the whole graph representing the level is a connected component. As our system use this same concept, a more detailed explanation of how reachability between two nodes is computed is given on section III, B.

After converting the level to its graph representation, the SubdueGL algorithm is applied and frequent structures are identified. The most frequent structures are abstracted from the graph and used as rules for a graph grammar. This process is repeated until a grammar capable of reproducing the original level is obtained.

Although an overview of the system was provided, no implementation or analysis have ever been published. Moreover, the system is based on the assumption that high-quality design patterns would frequently appear in human-designed levels. While this may be true, video game levels datasets are known for being often scarce, when available at all.

Nevertheless, we believe that the expression power of graphs can be used to learn patterns and generate high-quality levels from them. We draw inspiration from their work by adopting a similar graph representation. However, we use a different approach for learning the patterns to ensure that a desired number of patterns can be extracted from a map regardless of their frequency.

III. Proposed System

We propose a system for the automatic generation of platformer levels that extract structures from existing levels and identifies how they can be recombined with each other. This information is then used to create a graph grammar that can output levels using different recombinations of these structures while ensuring that the level can be traversed by the player. Source code is available online [2].

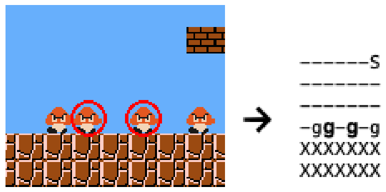
A level or a pattern in a level is represented as a graph, where each tile is associated with a node. A node will contain information such as the (x, y) coordinates of the tile and its tile ID. Air tiles (tiles where the player can move through freely) are also represented as nodes since they can be an important part of a structure. Edges of a node represent tiles that are adjacent to it.

The proposed framework is divided in three stages, illustrated in Figure 2:

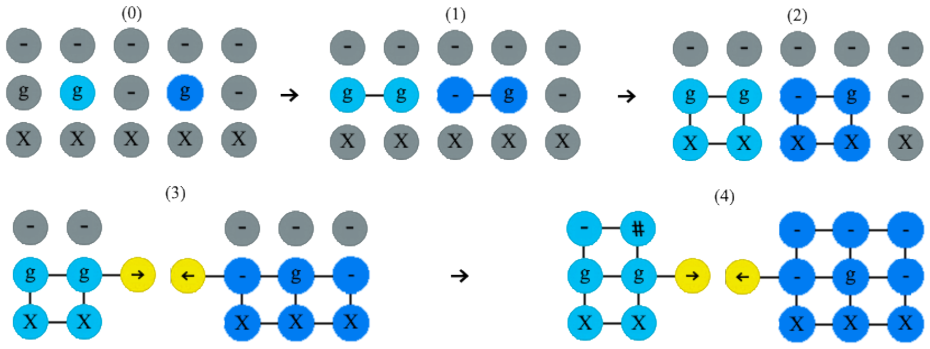
- **Structure Identification:** structures are identified and extracted from the set of input maps;
- **Structure Matching:** different ways of recombining these structures are identified;
- **Level Generation:** structures are connected to each other until reaching certain criteria.

Note that a human designer could be added to any of the three stages to help achieving a certain kind of result. Particularly, on the first stage, a user could input their own

[2]https://github.com/ehauckdo/mario



(a) selected points on a level



(b) expansion of a graph from the initial points

Fig. 3: Identification of a structure from two initial points (a). (b) shows the order of expansion (left, down, right, up). A connector node is added in step 3 to both structures due to an overlap. They no longer expand towards the overlapping direction.

structures to be mixed with the identified ones. Also, on the last stage, a user could try different combinations of structures and experiment with the level. Nevertheless, to get a baseline for the potential of the method, in this work we only investigate computational methods for each of the three stages.

A. *Structure Identification*

A structure is defined as a subsection of a level (a window of $d \times d$ size). This stage receives 3 parameters as input, as follows:

- L: a set of levels l;
- n: the minimum number of structures to be identified in each level $l \in L$;
- d: the base size (width of each side) of the structures;

For each level l in the set L, a minimum number n of structures are identified. In the current implementation, a simple strategy is employed to extract random chunks from the level and identify how they connect to each other is adopted, as described below.

The Structure identification stage starts by selecting n non-air tiles in the level. These non-air tiles are selected such as they are as equally spaced as possible from each other by using the Suppression via Disc Covering algorithm proposed by Gauglitz et al. [12]. The implementation of the algorithm was provided by Georgy Skorobogatov and is available online [3].

After n non-air tiles are selected in the level, a node is created for each one of them to represent the tile at that (x, y) coordinates. The next step is expanding, one direction at a time (left, down, right and up), each of these nodes until reaching a width and height of d. The expansion to each side is done simultaneously for every structure. Figure 3 illustrates how the expansion occurs.

Because the structures are expanded simultaneously, an overlap between two growing structures may happen during expansion. When this happens, the expansion to the corresponding directions on both structures is halted (i.e. the actual overlapping is not allowed). A connector node is created for each structure, and placed in the (x, y) position of the node of the opposite colliding structure. These connector nodes do not contain any tile information. They are temporary nodes used in the next steps of the system to connect two structures together. Their function is analogue to that of a non-terminal node of a graph grammar.

After all structures have reached the base d size for their width and height or had they growth halted due to an overlap, the selection of structures from the input levels is complete. The next step is deciding how structures can be joined to one another.

B. *Structure Matching*

Once a list of structures is obtained, it is interesting to evaluate which pairs of structures can be connected and from which nodes this connection can happen. This step is performed for two important, inter-connected reasons: to decide which connections are good and which should be avoided; and to speed up generation by disregarding connections that are known beforehand to violate a set of desired constraints.

[3]https://gist.github.com/LostFan123/63c7a1a26945ffaf115dc6886b69e862

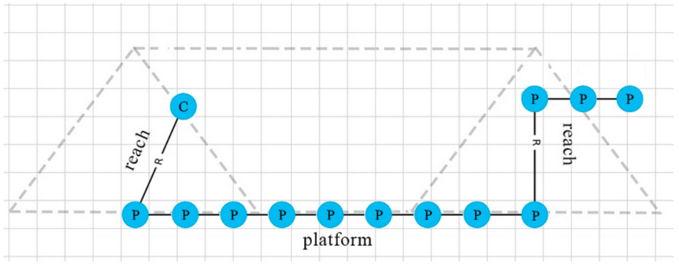


Fig. 4: Illustration of the reachable area given a platform. Nodes with label P represent a platform, while nodes with label C represent a non-solid tile (such as a coin). The area given by the trapezium and the two triangles represents the reachable area of the player when at any point in the bottom platform.

In the current work, when verifying whether two structures a and b are joinable, two constraints are evaluated:

- Structural consistency: the connector nodes of a and b must have been created due to an overlapping in the same axis (i.e. either horizontal or vertical) and opposing directions;
- Playability ensurance: the player character should be able to move from a to b.

The playability ensurance constraint is verified by employing the reachability concept initially proposed by Londono and Missura [4]. It models the reach of the player character given its mechanics (in the case of Super Mario Bros., walking and jumping) and use this information to create edges between non-adjacent nodes in the graph. Figure 4 illustrates how the reachability is computed, given a certain platform.

The reachability concept is used to ensure that for some platform node of a structure a, the player can reach at least another node of structure b. This approach helps to alleviate with some of the limitations of the standard approach in the literature regarding employing an A* agent to verify the playability of a level, as it will be seen later in the experiments section.

However, since it is a simple model of the mechanics of the character, it has limitations. For example, it does not consider that the character is able to get a power-up item and destroy a block that is obstructing the path. Nevertheless, if reachability is used to assert that all nodes from a certain platform are reachable, then the player should be able to traverse the whole level.

If these two constraints are satisfied for a pair of connector nodes, then the two structures are deemed joinable through these pair of nodes. This evaluation is done for every pair of connector nodes. Note that, consequently, structures with more than one connector node can be joined with one another in different ways, as long as they do not violate the desired constraints.

After this step is complete, every structure will have a list of other structures it can be connected to, and from which connector nodes these connections can be made. A weighted distribution can be added to bias the likelihood of a certain connection being made.

C. *Level Generation*

In the level generation stage, a grammar is built from the information obtained in the previous two stages. This grammar is then used to generate new levels. The generation can be executed freely by applying the substitutions obtained from the grammar rules, or following some constraint. These generation constraints are complimentary to the constraints evaluated in the structure matching stage. They are concerned with the properties of the whole generating map as opposed to the properties of two connected structures.

A simple hand-coded starting structure is used as a start for the generation, to ensure the player always spawns at a safe location. It will contain only the ground blocks for the first three columns of the game, the symbol for the spawn position of the character, and a connector node.

The generation is performed by iterating over a structure joining process, given by:

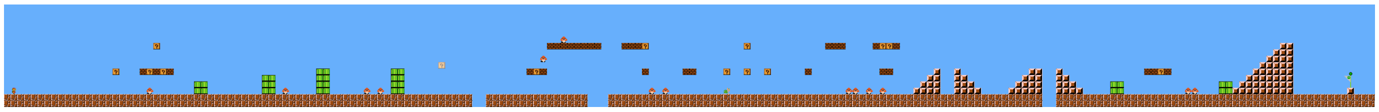
- Obtain all possible substitutions for the currently available connector nodes;
- Scale the probabilities of each connector nodes;
- Draw structure given the probability distribution and add it to the graph;
- Flag the connector nodes used as connected.
- If any constraint is violated, backtrack to previous state

Two constraints are considered for the generation process. First, is the availability of connector nodes (ensure at least one entry point for substitution at every step of the generation). Second, preventing the overlap of a new joining structure with a structure joined at a previous step of the generation.

Regarding the second constraint, the option of allowing overlap of nodes during generation to support the emergence of new and unexpected patterns was considered. However, early experiments showed that the generated levels were too structurally inconsistent, such as growing outside of the screen boundaries.

An additional matter to be considered is whether or not the overlapping of air tiles is considered a constraint violation. Air tiles can be seen as ”empty” tiles, thus their overlapping could be disregarded so novel patterns could be discovered during generation. However, it is known that negative space, i.e. the empty spaces between material objects in a space, plays an important role in game design. Therefore, choosing to disregard the overlapping of air nodes should be done with care. To ensure higher structural consistency of the level, the base system prevents any overlaps to happen during generation, but experiments with the relaxation of this constraint were also performed, as shown in the next section.

The iteration described for the generation process is repeated until reaching a stopping criteria. Since every pair of structures used during the generation are reachable from one another, the resulting graph is guaranteed to be a connected component. Finally, the output graph is converted back into text format and is ready to be played using the MarioAI framework.


(a) World 1-1, level used as input

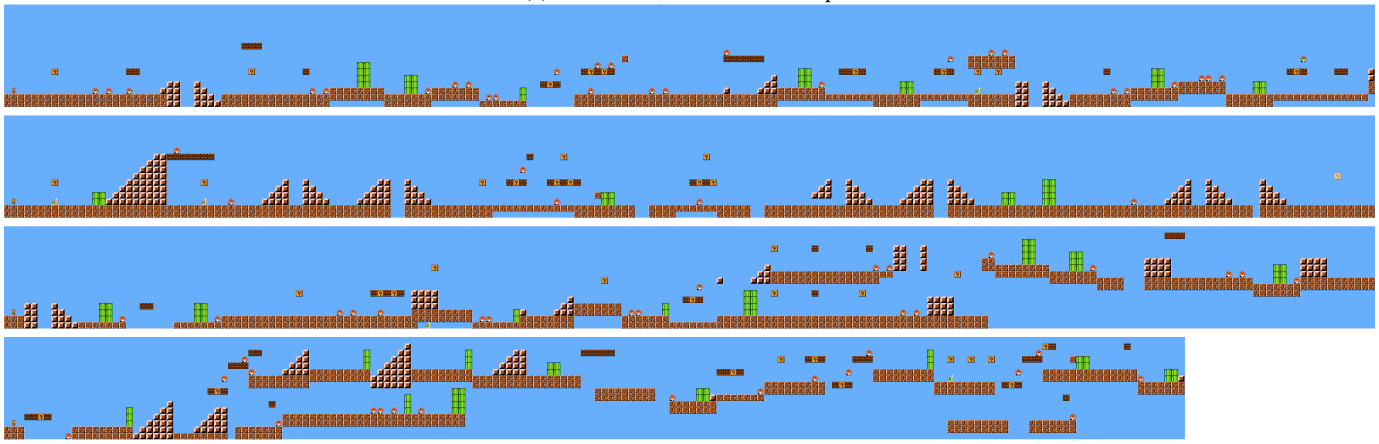

(b) Output levels

Fig. 5: A sample of the generated levels. The two top levels on (b) were generated by avoiding overlap of air tiles, resulting in more organized levels, while the two bottom levels were generated allowing for the overlap of air tiles, and present less linearity.

IV. Experiments

In this section, we evaluate the generation results of the proposed system. For that, we assessed whether our system is capable of generating levels, whether the generated levels are playable, and how did they fare visually in comparison to the input level. After this baseline evaluation, we assess how the constraint on preventing the overlap of structures during generation affects the generated levels. We use the first level of the Super Mario Bros game (World 1-1) as the input for the system in both experiments. This level is shown in Figure 5a, and is available for as part of the Mario AI framework.

A. Experiment Setup

We set the parameter n of structures to be selected as 35, and the minimum size d of each structure as 4. In the generation process, we adopted a uniform probability distribution for the selection of new structures. That is, given all possible connections from a connector node, the structure to be connected to it is drawn randomly from the list of possibilities. We set the stopping condition for the generation process as when 30 structures have been added to a level under generation. We generate 100 levels under two configurations: one using the baseline system, and another relaxing the constraint on overlapping of air nodes. We evaluated the expressivity range of the generator for the two configurations by employing the leniency and linearity metrics proposed previously in the literature [6]. We also evaluated whether levels are playable or not. We define a level as playable if it is possible for the player to get to the end of the level. To that end, we employed the Robin Baumgarten A* agent that was shown to perform at a super-human level in previous MarioAI competitions [5]. The agent was executed 5 times for each map over the period of 90 seconds, and if at least one execution results in a win we deem the level as playable.

B. Results

The proposed system created a very small number of unplayable levels. When overlap of air tiles is not allowed, only 3% of the levels generated were unplayable, while only 10% of the generated levels were unplayable when overlap of air tiles were allowed.

Some examples of the levels generated are on Figure 5b. The reader can observe that the levels generated without allowing the overlap of air tiles (top two levels of figure 5b) are much more similar to the input level than the levels generated while allowing the overlap of air tiles (bottom two levels of figure 5b). Allowing the overlap of air nodes in different patterns tends to generate less linear, more chaotic levels.

We can see the same trend by observing the distribution of leniency and linearity of the levels generated in both configurations (Figure 6). In the first configuration, all levels had high linearity (≥ 0.6), often as high or higher than the original level, while the levels in the second configuration had more varied linearity scores. All the levels generated in this experiment are available in our repository.

We also investigated what kind of levels generated by our system were being labelled as unplayable. We identified two situations in which a level was deemed unplayable.

The first situation is the generation of sections that are untraversable. Figure 7 shows a level generated with this problem. In this case, the level is impossible to complete. This kind of level is generated due to a limitation on our

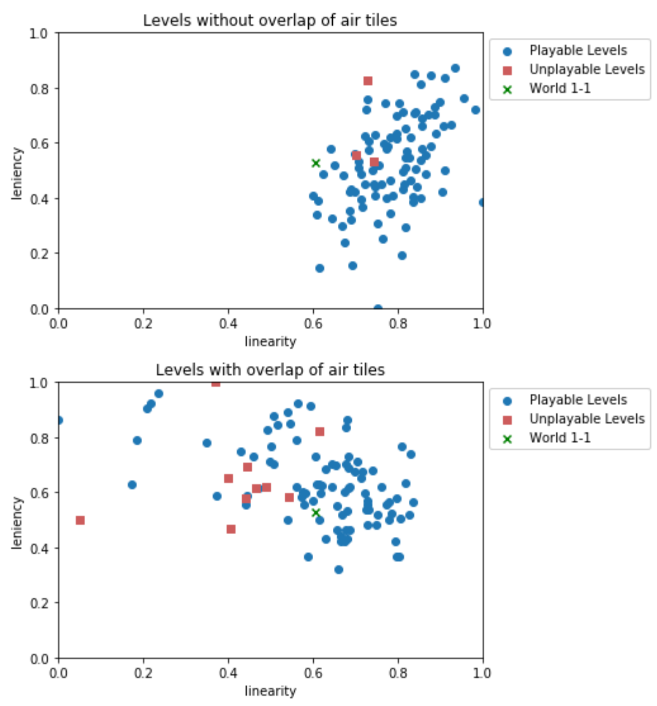


Fig. 6: Expresivity of the generator

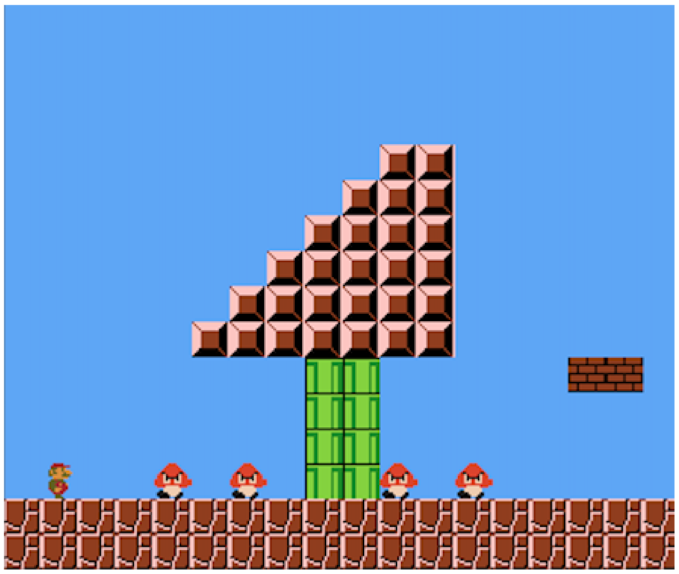


Fig. 7: Screenshot taken from an unplayable generated level. The reachability fails to identify that jumping on top of the pipe is impossible due to the obstacle on top of it.

design of reachability between two structures. Although we are modeling the reach of the character's jump, we are not taking into account the possibility of obstacles that prevent the jump from being successfully executed. That is, solid blocks are considered traversable and do not change the shape of the reachability area, resulting in levels such as the one shown in the figure.

The second situation is the generation of levels that required some sort of backtracking. Figure 1 shows the example of a level that presents the player with two paths that, if taken, would require the player to backtrack. In this case, the level is playable but it is incorrectly labelled due to the agent failing to backtrack once it reached a dead end.

C. Discussion

The expressivity analysis in Figure 6 and the qualitative assessment of the levels (such as the ones shown in Figure 5b) show that the system is capable of generating levels and that they are playable in general. The results of the first experiment show that the levels generated are visually similar to the original input level. The parameters and constraints selected for the experiment were able to properly capture structures from the input level and then recombine then in an orderly manner, resulting in Mario-like levels.

In the second experiment, we tried relaxing some of the constraints of the generation. This change resulted in less linear and more diverse levels, at the cost of structural consistency. Ignoring the empty space around the structures allowed the generation of interesting, unexpected or just faulty levels. This shows a trade-off on the level of tolerance in which we enforce this constraint.

An interesting outcome of the experiment were the generation of levels deemed unplayable in the second configuration. Our method found levels with paths that lead to a dead-end, or paths that lead into a gap, requiring the player to backtrack. The generation of these kind of levels are only possible due to the reachability concept, as it ensures that there will be some path to the end of the level, even if one or more paths lead to dead-ends. For these levels, the agent always fails to finish the level, and therefore the levels are labeled as unplayable. This highlights a limitation in the current methodology for platformer level generation, as a considerable number of works published in the literature use this same A* agent as a tester to decide whether a level is playable or not. Consequently, we believe that levels with any sort of backtracking were probably deemed unplayable and discarded by other level generation methods so far.

Despite the mislabeling of these levels during the experiments, they validate our initial expectations that new patterns would be discovered from the input data by using the proposed method. The AI agent can play the original level flawlessly, yet from this original level, we were able to generate new patterns that revealed some of the problems with this agent.

V. Conclusion & Future Work

We proposed a level generation system based on the extraction and recombination of patterns from human-designed levels. In this paper, we generated a large number of levels by extracting and recombining patterns from the firs level in Super Mario Bros. A majority of these levels were playable, and some presented a natural-looking placement of elements throughout the level, reflecting the patterns obtained from the human example.

Our system can be easily extended to become a useful tool for level design, as it allows new patterns to be manually entered or edited by a user or designer who wish to interact directly with the system. Another benefit of the proposed system is the capability of generating levels with that contain paths leading to dead-ends, requiring the player to backtrack.

The system, however, does show a few limitations. Our reachability design was not capable of modeling all possible scenarios where the player would be unable to access a certain area, leading to some of the unplayable levels that were generated. We also did not perform a complete analysis of the system parameters to understand how different combinations of number of structures and base size affect the generated levels. Additionally, despite using a graph representation for the levels, we are make little use of graph concepts to support the generation. However, we believe that future iterations of the system could make use of such concepts to improve the generation.

Future directions of this work include both improvements to the system as well as an expansion of it. The reachability design should be changed to take into account obstacles that prevent the player from passing. Other methods for generating both levels and patterns can be explored. For instance, we randomly extract chunks from the input levels, but some patterns in the levels have semantic purposes to the game, such as serving as introduction to a new mechanic. These patterns could be taken into account when extracting the chunks. Additionally, we evaluated a stochastic generation of levels, but more directed approaches can be employed to generate levels targeting certain features or mechanics.

Another direction of research would be developing an interface tool that allows the user to select patterns and create levels with them. Such a tool would make use of one of the main advantages of the system, namely its ease to interface with humans. This tool could also present co-creation features, where parts of the level are created by the player, and parts are created by the system, such the co-creation system proposed by Guzdial et al. [13].

References

[1] R. R. Torrado, A. Khalifa, M. C. Green, N. Justesen, S. Risi, and J. Togelius, "Bootstrapping conditional gans for video game level generation," *arXiv preprint arXiv:1910.01603*, 2019.

[2] 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*, vol. 10, no. 3, pp. 257–270, 2018.

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

[4] S. Londoño and O. Missura, "Graph grammars for super mario bros levels." in *FDG*, 2015.

[5] J. Togelius, S. Karakovskiy, and R. Baumgarten, "The 2009 mario ai competition," in *IEEE Congress on Evolutionary Computation*. IEEE, 2010, pp. 1–8.

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

[7] A. Khalifa, M. C. Green, G. Barros, and J. Togelius, "Intentional computational level design," *arXiv preprint arXiv:1904.08972*, 2019.

[8] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[9] M. Guzdial, J. Reno, J. Chen, G. Smith, and M. Riedl, "Explainable pcgml via game design patterns," *arXiv preprint arXiv:1809.09419*, 2018.

[10] I. Jonyer, L. B. Holder, and D. J. Cook, "Mdl-based context-free graph grammar induction and applications," *International Journal on Artificial Intelligence Tools*, vol. 13, no. 01, pp. 65–79, 2004.

[11] J. Sander, M. Ester, H.-P. Kriegel, and X. Xu, "Density-based clustering in spatial databases: The algorithm gdbscan and its applications," *Data mining and knowledge discovery*, vol. 2, no. 2, pp. 169–194, 1998.

[12] S. Gauglitz, C. Lee, M. Turk, and T. Höllerer, "Integrating the physical environment into mobile remote collaboration," in *Proceedings of the 14th international conference on Human-computer interaction with mobile devices and services*, 2012, pp. 241–250.

[13] M. Guzdial, N. Liao, and M. Riedl, "Co-creative level design via machine learning," *arXiv preprint arXiv:1809.09420*, 2018.