Grammar Based Modular Level Generator for a Programming Puzzle Game

Chaima Jemmali
Northeastern University
jemmali.c@northeastern.edu

Carter Ithier
Northeastern University
ithier.c@northeastern.edu

Seth Cooper
Northeastern University
se.cooper@northeastern.edu

Magy Seif El-Nasr
UC Santa Cruz
mseifeln@ucsc.edu

Abstract

Procedural Content Generation is widely used in games, however, its use in educational puzzle games has been limited. These types of games present common challenges such as solvability and non triviality, but also the extra challenge of preserving intended learning goals. In this paper, we present a modular constructive approach to generate levels in a puzzle programming game. The approach uses a grammar to generate game elements from code and works backwards from the solution to ensure solvability, controllability over the solution, and variation, allowing for alternative solutions that preserve the learning goals.

Introduction

There has been a lot of research on Procedural Content Generation (PCG) and there are several techniques used to generate levels for different game genres. The most common two genres where PCG is applied for level generation are dungeons/rogue-likes and platformer games. However, the application to puzzle games has been limited (De Kegel and Haahr 2019). Educational Games are another area where PCG has not been adopted widely (Dong and Barnes 2017). Besides the common challenges of solvability and non-triviality of the solution, these types of games present the extra challenge of preserving intended learning goals (Smith, Butler, and Popovic 2013; Valls-Vargas, Zhu, and Ontañón 2017; Dong and Barnes 2017).

In this paper, we focus on two aspects of the generator: controllability over the educational goals and variation, meaning the designer has full control over the learning goals in the level to be generated. Further, generated levels vary in size, layout, and number of alternative solutions. In fact, we are not looking if a puzzle is merely solvable, we require it to be solvable with the solution code provided as input which provides greater control on what gameplay is generated. This approach is similar to generating levels from the gameplay as a vocabulary (Van der Linden, Lopes, and Bidarra 2013), where our gameplay is described through the solution code. To guarantee solvability, we use a grammar based approach which insures by construction that the level is solvable without post-processing or filtering out unsolvable solutions (Traichioiu et al. 2015; Valls-Vargas, Zhu, and Ontañón 2017; Font et al. 2016). We combine this approach with working backwards from a solved map similar to what has been done for the Sokoban game (Taylor and Parberry 2011). Controllability and variation are somewhat in opposition, especially when having such a hard constraint on how the level should be solved. To provide variation, we allow fluidity in our level construction, especially in the path creation section. This allows levels to have alternative working *equally difficult* solutions, of the same length as the provided solution code, but also introduces *shorter* solutions which means the player can solve the level by bypassing certain elements. We further evaluate the generator through the design/aesthetics lens by analyzing the expressive range (Smith and Whitehead 2010) using two metrics: percentages of walkable and interactable tiles. Our results show that the levels generated are 100% solvable by the provided solution code. On average, 47.4% of them had an alternative working solution of the same length, while 21.3% had shorter "easier" solutions.

The contribution of this paper is the design and application of a PCG system combining previously used techniques, as well as introducing new ones for path creation. The generator is applied to a programming puzzle game, but the modular aspect of the approach should make parts of it applicable to other educational or puzzle games.

Related Work

Research on games that teach programming is significant, especially at the introductory level (Harteveld et al. 2014; Miljanovic and Bradbury 2018). Generating content for these games can be challenging and time consuming, which is increasing the need for PCG to create content that can be tailored to specific player needs (Park et al. 2019). However, a major concern in PCG is the lack of reliability (Togelius et al. 2011), which makes the assessment of the generator of utmost importance. Controllability and variability are two of the major metrics used to assess PCG systems. In fact, the degree of control and the set of options are very important characteristics of any procedural generator (Shaker

Figure 1: Coding interface: bottom-left: Structure of the command with examples, bottom-right: output area shows syntax error or progress towards solution, top-right: Text area and buttons to run/restart/help/undo/redo, top-left: Programmable objects and animation/code execution area.

et al. 2016). When dealing with puzzle levels, solvability is an important constraint which can be achieved through different techniques. Some works have used generate-and-test techniques (Dong and Barnes 2017), constructive approaches where the content is generated only once and performance checks may be applied throughout (De Kegel and Haahr 2019), search-based algorithms (Togelius and Shaker 2016), or answer set programming (ASP) (Smith and Mateas 2011), which falls somewhere in between constructive and generate-and-test. Procedural content generation via machine learning (PCGML) (Summerville et al. 2018) is an increasingly popular approach to generating various content, however, it does not always guarantee solvability. In the context of learning games, another constraint is to preserve the intended learning goals or the intended difficulty of the levels and ascertain there are no trivial solutions. The definition of such solutions and the approaches to detect them vary depending on the game. For instance, in an educational game that teaches parallel programming (Valls-Vargas, Zhu, and Ontañón 2017) a trivial puzzle is a puzzle that doesn't require the player to make any changes to be solved and is identified through the use of a model checker that will run different scenarios to check if they pass or fail. In another puzzle game that teaches programming (Dong and Barnes 2017), solutions that violate educational goals are solutions that contain unintentional loops and unnecessary elements. However, the work evaluates a code synthesizer that creates a solution code from a template rather than the level generated for that code. To ensure the absence of trivial solutions, Smith, Butler, and Popovic (2013) used a modified version of ASP to successfully generate levels with no undesirable solutions, however, the search space is so large that it makes the generation time too long for online application.

In this work, we use a constructive approach to guarantee solvability in a computationally inexpensive manner (Shaker et al. 2016). We use a grammar to generate the appropriate game elements given an input code. Then, we work backwards from the solution similarly to Taylor and Parberry (2011) to ensure solvability with a specific solution while allowing for variability with equally difficult codes and minimizing trivial solutions.

Object Type	Symbol	Description
movable block	m / M / M & PP	can be walkable or not, may activate a pressure plate
rotating block	rr / RR	L shaped block that rotates around its center (rr/RR), can be walkable or not
stoppable block	mb / MB	moving block that can be stopped
fire statue	FS / FSR & reward	A rotatable or movable statue that blows wind/fire (see Figure 1)
statue	S / S & PP	statue that has the same characteristics as an above ground block
pressure plate	PP & reward	activates a reward (key, hidden door, hidden block, etc)
hidden block	bh	ground level block that can be revealed.
revealed block	BR	above ground block that can be hidden.
hidden door	DH	A door that can be revealed.
closed door	DC	A door that can be opened.
hidden key	KH	A key that can be revealed.
fire	FU / FL	A fire pit that can be lit or unlit.
lever	LV & reward	A lever that activates a reward if it's in the correct position.

Table 1: Description of game elements in the game; their symbols, mechanics, and features

Game

In this paper, we use *May's Journey*, a puzzle game that teaches the basics of programming by having learners type simple instructions in the game's custom programming language to interact with objects, solve puzzles, and navigate an environmental maze (Jemmali et al. 2019; 2018).

The game is structured in two major phases. In the first phase, players can move the avatar around using arrow keys, interact with different parts of the environment, talk to NPCs, or collect objects. In the second phase, they pull up a programming interface, as can be seen in Figure 1, where they interact with game objects through code to solve different puzzles, which allows them to progress through a level or reveal some rewards.

Each level in the game offers a coding challenge that can be *maze-based*, where the reward for solving it is getting to the next level, or *reward-based*, where solving it yields a physical reward (key, manuscript, secret area), or both. Each level has an entry and at least one exit, but could have multiple. If a level has more than one exit, only one of them can be an open door. The others have to be either hidden or closed and revealed by solving the level. The programming language is object oriented; it is similar to Java, but with less heavy syntax. The affordances of the language are: simple instructions (commands applied to objects), simplified for loops, if statements, variables, object attributes, and while loops. For our PCG system we considered 4 commands that can be applied to objects: Move, Rotate, Open and Stop. For attributes, we considered 2: isMoving (bool) and position (Vector3 or string). Each object can have multiple attributes, but only one type of command applied to it. For example, if it is movable it can't be rotated, but it could have more than one attribute. Table 1 shows the objects in the game with a small description of their functionality and the symbols used to describe them in our grammar.

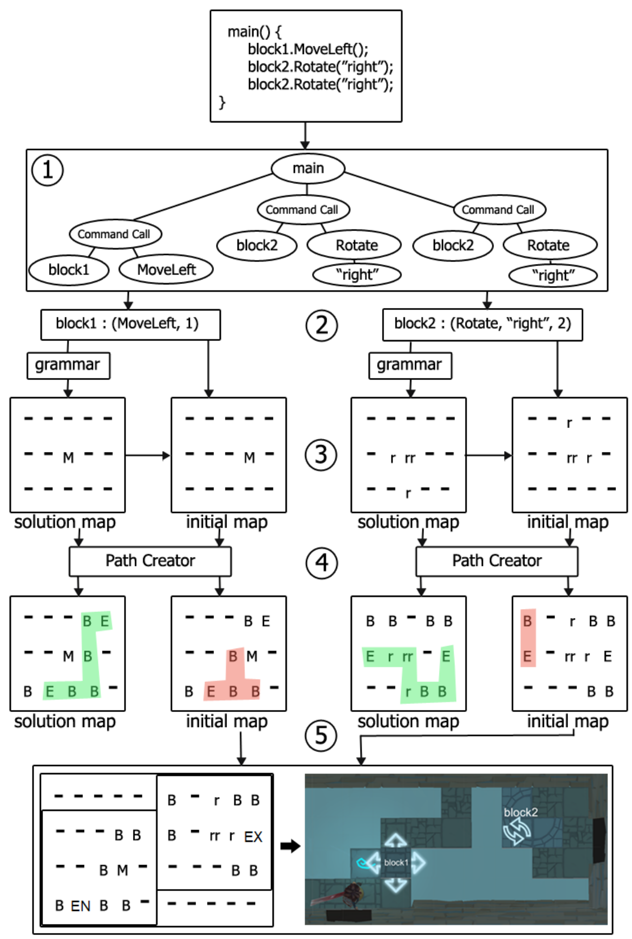

Figure 2: Generation Process: 1) getting an abstract syntax tree from input code 2) extracting objects and actions 3) creating solution and initial mini-maps 4) building paths for each map 5) merging mini-maps

Methodology

The generator takes as input a solution code and outputs a level layout that can be solved using that code. The generator is divided in modules to allow for flexibility, modification, and reuse for different purposes. In fact, it is recommended to break the process down into multiple steps to design successful grammar based systems (Togelius, Shaker, and Dormans 2016). The full process can be seen in Figure 2. In this section, each module is described in detail using the example in the figure.

Abstract Syntax Tree Generation

This step takes the input code as a string and extracts the abstract syntax tree (AST) from it. This step is simple, but allows the system to be more generalizable since from this step forward, the generation does not depend on the source language but only on the AST. In the first step of Figure 2, we can see that the code presents three command calls: MoveLeft once, and Rotate("right") twice.

Game objects and actions extraction

From the AST, we extract each object, a list of commands applied to it, as well as a list of attributes. The same command will be merged together in this format: (command_name, arguments, n) where n is the number of times that command is repeated. If the command takes no arguments, that field will be empty. For attributes, the format is (attribute_name, initial_state, desired_state). In general, the desired state is extracted from comparison operators in an if statement or while loop. In the example, we obtain block1(MoveLeft, 1) and block2(Rotate, "right", 2) . An example of an attribute would be if we had a stoppable block, we would have (isMoving, true, false) where the block is initially moving and should be stopped in the desired state.

Grammar-based solution mini-map generation

For each game object, there are possible shapes, or possible gameplay elements it can incorporate. Depending on the game object name and its actions and attributes, the grammar rules are traversed until a final shape is found. The grammar rules are extended with probabilities so that some rules are more favored than others. The probability distribution can be chosen as input and can be either uniform (all rules have the same probability of being chosen), favors complexity, or favors simplicity where, respectively, rules with more complex/simple shapes will be favored. Further, when a specific rule is applied, a penalty value is added. The penalty value can also be changed as input and can range between 0 (no penalty) and 1 (each rule can only be applied once). When the penalty value for a rule reaches 1, that rule is no longer considered. This penalty is to guarantee that the same rule cannot be applied too many times and that the layout is diverse. The shape obtained from the grammar traversal is placed in the solution mini-map, meaning the shape is placed as it would be when the map is solved. Figure 2 shows the solution maps for each of the objects in step 3. Having separate mini-maps for each object maintains the intended gameplay since players have to solve the level in one submission, however, this limits some gameplay opportunities where objects can influence each other.

Initial mini-map creation

Next, the actions obtained from step 2 are applied to the solution maps in reverse. For example, if the action is MoveUp, then the object is moved down. This is repeated until all actions are applied.

Path Creation

To create a path in step 4, we use two different algorithms depending on the shape of the object we get from the grammar. If the shape's gameplay is focused on revealing a reward, such as fire statues or levers, we use the *reward path* algorithm, which mostly fills the map with walkable tiles and then removes tiles from the corners up until a predefined threshold.

At the start of the reward path algorithm, all empty tiles in both initial and solution maps are filled with walkable blocks B. Then, we get all four corner tiles of the map, choose a valid one to remove, and move to its neighbors until no valid neighbors are available. A tile is considered valid if it does not break the path in the solution map. We chose this approach to have the tiles removed in a systematic manner and not randomly in order to obtain a map that looks closer to the hand-designed maps.

If the shape's gameplay is more maze-based, such as a block that can be walkable or that can obstruct a path, we use the *maze path* algorithm, which takes as input both the solution map and initial map to make sure that 1) there is a path between the entry and exit in the solution map, and 2) there is no path between entry and exit in the initial map. If there is no solution that satisfies these conditions, the first rule is prioritized to assure that, when a level is solved, the player can make their way from entry to exit. If the second rule is not satisfied, this means the player may be able to walk to the exit without having to solve the puzzle.

In the maze path algorithm, the entry and exit, which we refer to as EN and EX, are fixed at the start. After choosing these points, we find all walkable tiles in the solution map. Then, we find points (a) and (b) which are respectively the closest tile to EN and EX in the walkable tiles. Finally, the shortest paths will be drawn from EN to (a), EX to (b), and (a) to (b). The same path is also drawn in the initial map. If the resulted path is walkable from EN to EX in the initial map, we remove tiles that break that path as long as they do not break the solution path. This again ensures the first condition of solvability of the puzzle. Finally, the last part of the maze path algorithm, which adds tiles back into the path, is purely for aesthetics reasons and variability to add different shapes of paths and not just narrow ones.

Mini-map Combination

Since we could have many mini maps to combine depending on how many programmable objects are in the code, in step 5, we consider the best way to combine them. We determine the best combination based on the goal of minimizing the cost of combination. If two maps can be combined without modification, they have a 0 cost, while the need for modifications can result in costs depending on the positions of the doors E (EN or EX). Combining two maps needs modifications when they don't have opposing edges that share a door. In this scenario, the merged map is made bigger and a path is created between the entry and exit. This process in this step is minimized by finding the best combination of the maps that guarantees the minimum cost.

Miscellaneous Placements

To have a fully functional map, we add the Player tile in front of the entry. The coding tile is placed in a way that is accessible from the entry. Finally, rewards are placed depending on their type. Closed and hidden doors are placed along walls that are accessible. If there are no available, accessible walls, the doors are placed and then a path is created to make sure they are accessible. Hidden keys are placed anywhere on the walkable path. Finally, headers are added to the file, which determine the actions of pressure plates, levers, and fires, as well as the objects that can activate them.

Evaluation

To evaluate our system, we look at metrics related to both the education aspects and game aspects. For the educational requirements, we first check that levels generated can be solved using our input code. Further, we check if the level can be solved using alternative codes with the same code length. Finally, we check if the level can be solved with shorter alternative codes. We are not worried with longer codes since any level can theoretically be solved using a longer code than what was intended. For a code to be considered working, there should be a path from entry to exit, and from entry to any reward in the map, after the code is applied. If one path is missing, the code is not considered working.

Alternative codes are generated from the original code, by 1) searching for alternative commands that can be applied to a game object and 2) constructing codes using each combination of alternative commands. For example, if our code contains $object1.MoveLeft()$ and $object2.Rotate(\text{“}right\text{”})$, there are 4 Move commands (one in each direction) that can be applied to object1 and 2 Rotate commands (left/right) that can be applied to object2. This results in 8 different codes: 7 alternative codes and the original input code. More broadly, the number of alternative codes can be written as $\prod_{i=1}^{n} comb(O_i)$ where n is the number of objects and $comb(O_i)$ is the number of combinations for object O_i.

Alternative shorter codes are generated by 1) creating a list of all commands and constructs in the input code and 2) finding all code combinations that use at most $n-1$ of commands and constructs, with n being the size of the list extracted in step 1. For example if our input code contains a while loop, an if statement, and two commands, the list length would be 4, and the number of possible combinations would be 15. More generally, the number of alternative shorter codes can be written as $\sum_{r=0}^{n-1} \binom{n}{r}$. If $(n = 1)$ the only shorter code possible is an empty code.

To evaluate the design of the levels, we look at three metrics: map size, percentage of interactable objects over map size, and percentage of walkable tiles over map size. It is difficult to find more appropriate metrics to evaluate the levels since the gameplay is decided through the input code. In fact, the input code has the biggest impact over what kind of levels would be generated. Further, the variety of objects and game mechanics is limited by what's afforded in the original game. Evaluating the map shape is also not interesting since

Lvl	Elements	Gen. Time (s)	Input Sol.	Alt. Sol.	Alt. w/ Sol.	Short Sol.	Short w/ Sol.	Size
0	1 moving block	0.11 ± 0.03	100%	4	49.5%	1	13.5%	99.34
1_1	2 moving blocks	0.39 ± 0.07	100%	16	72.1%	3	24.8%	240.98
1_2	moving block, moving statue	1.17 ± 0.6	100%	64	67.2%	7	27.4%	318.24
2_3	1 rotating fire statue	0.12 ± 0.03	100%	2	0%	1	0%	160.62
3_1	3 rotating blocks	0.51 ± 0.1	100%	8	91.9%	7	37.2%	464.72
3_2	for loop and 2 moving blocks	0.54 ± 0.16	100%	16	70.6%	7	36.4%	350.38
4_2	lever, door, conditional	0.06 ± 0.01	100%	NA	NA	NA	NA	89.04
4_6	stoppable block, while loop, conditional, pressure-plate	0.12 ± 0.02	100%	NA	NA	1	9.9%	249.87
avg		0.38 ± 0.13	100%	18.33	47.35%	3.85	21.31%	246.64

Table 2: Results of 1000 runs for 7 different game levels; with uniform distribution, threshold = 2, grammar penalty = 0.5. Input Sol. = the input code is a solution; Alt. Sol. / Short Sol. = number of possible alternative / shorter codes; Alt. w/ Sol. / Short w/ Sol. = percentage of times there was at least 1 alternative/shorter code that is a solution; Size = average size of the map in tiles.)

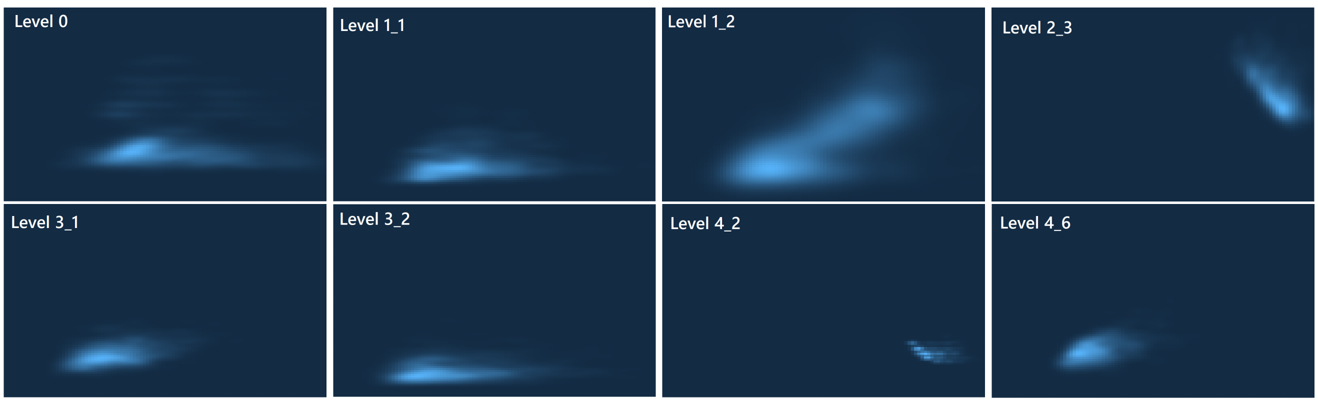

Figure 3: Expressive range for each level in terms of percentage of walkable and interactable tiles (1000 runs, 0.5 penalty, 2 threshold). X axis: % walkable (0 to 0.5). Y axis: % interactable (0 to 0.1)

player movements are restricted to the path and the player cannot fall off, meaning there is no difficulty attached to the layout of the map.

Results

We ran the generator with the solution codes from 8 different levels in the game, including a variety of game objects and constructs. Every code was run 1000 times and the results are presented in Table 2. Overall, every level generated was solvable using the input code, which is expected since it is guaranteed by construction. Further, on average 47.35% of the levels generated had an alternative working solution (Alt. w/ Sol.), which means about half of the levels generated provide different equally difficult ways to solve the problem. This can be a nice balance between levels that have a unique solution vs levels that offer the players multiple ways to solve them. From Table 2, we can see that the percentage varies considerably across levels, which is again expected. For example, in level 2_3 the game object can only be reward based, which makes it impossible to have alternative solutions since the only way to get the reward would be to use the input code. On the other hand, levels that are purely maze-based and have no rewards such as 3_1 have a much higher rate of alternative solutions (91.9%). These levels would also have the highest rate of short solutions working. Overall, 21.31% of the levels generated had working shorter solutions (Short w/ Sol.), with again variations across levels. Levels with more lines of code and more objects would have more possible alternative solutions and therefore more chances for them to succeed. For example, levels (1_2, 3_1, 3_2) with the highest number of shorter solutions (7), unsurprisingly, have the highest rate of shorter solutions solving the puzzle. Looking at specific levels, such as level 4_2, alternative codes and shorter codes are not applicable. This comes from the original design of such levels. This level's code cannot be modified in the game. The gameplay consists of understanding the code and making appropriate changes in the environment so that when the code executes, it reveals a reward. Another particular level is level 4_6 where we haven't defined alternative codes for the Stop or Open commands which makes alternative codes not possible.

To analyze the variety of the levels generated, we looked at the expressive range of the generator in accordance to the percentage walkable (X axis, 0 to 0.5) and interactable (Y axis 0 to 0.1) tiles over the size of the map. In figure 3, we can see how the heat maps change depending on the level. We notice that the expressive range tightly follows the design and the possibilities allowed by the game, as well as the size of the map. For example, level_0 has a moving block which has 1/3 chance of yielding a reward thus making the

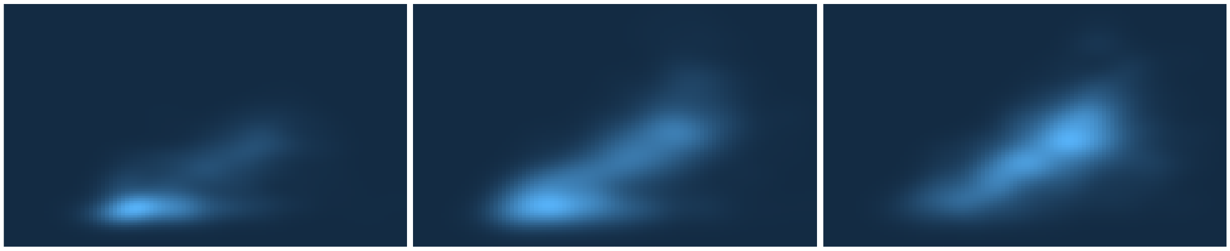

Figure 4: Expressive range with probability distribution variation for level 1_2. From left to right: favors simplicity, uniform, favors complexity. X axis: % walkable (0 to 0.5). Y axis: % interactable (0 to 0.1)

Figure 5: Examples of generated levels for level 1_2. From left to right, solvable with only input code, solvable with input and alternative, solvable with shorter code.

rate of interactable objects mostly low with a few scattered higher rates. On the other hand, levels 1_2 and 4_6 are reward based levels which gives a higher rate of walkable tiles. Levels with many interactable objects (level 3_2 and 4_6) tend to have bigger maps which makes the percentages of interactable and walkable much more concentrated. Figure 4 shows the effects of varying the probability distribution in level 2_1. This level was chosen because it has a combination of objects that can be maze-based or reward-based which makes it more representative. In this level, players need to move a statue to the right twice and move a block up once. We can observe significant changes between the distribution that favors simplicity and the uniform one. The change is not that noticeable between the uniform distribution and the one that favors complexity. However, while the heat maps seem similar, the brightest area on the complex one is on the higher end and the brightest area for the uniform one is on the lower end of the shape.

Figure 5 shows some examples of generated levels for the code of level 2_1. The left example shows a level that can only be solved with input code, players need to move the fire statue to get a key and move block1 on a pressure plate to reveal a hidden block. In the middle example, block1 can be moved either up or right to complete the path allowing for one alternative solution. However, in the right example, block1 does not need to be moved at all and only the statue needs to be moved to clear the path, which results in a shorter solution.

Discussion & Limitations

The results show that the generator successfully creates levels that are solvable using the desired input code, however, some of them are still solvable by shorter "less difficult" codes. These levels may or may not be desirable. In some cases, the designer may want levels that have an obvious longer solution and perhaps a shorter cleverer one. We do not claim that this is the case with the levels generated. But, we point out that it may be desirable in some special case, and the decision can be left up to the designer. If we want to completely remove the shorter solutions, we could add constraint checkers at different stages of the generation and discard parts of the map that violate certain conditions. Another way is to include the maps created by the shorter codes in the generation process and make sure none of them has a solution when creating the paths. This will increase the generation time, but we believe it will still be reasonable for in-game generation given that it is now at 0.38 seconds on average. While our approach is applied to a programming game, the same concept can be reproduced in a puzzle game where instead of code, the input would be game play vocabulary as in (Van der Linden, Lopes, and Bidarra 2013).

One limitation of this work is that the creation of alternative solutions is specific to the gameplay and affordances of the game and cannot be easily imported into another game. Another limitation is that while inputting code grants full control to the designer, it is not accessible to non-programmers or even programmers who are not familiar with the programming language of the game. One way to tackle this is to build a code generator that will take as input coding constructs and synthesize valid code that can be input to this generator. That way, the user would only select the learning constructs they want.

Conclusion

In this paper, we presented a grammar based modular approach to generate levels in a programming puzzle game. The approach works backwards from a solution code and uses both the solution map and the initial map to ensure that levels are solvable using the input code. The levels generated allow variation in the solution space through alternative codes while minimizing shorter, more trivial solutions. However, some of them still allow shorter codes. In the future, we want to improve on the approach, build a user-friendly interface and conduct a user-study with designers. Further, we would like to work on integrating procedurally generated levels within the game according to some player model that will inform us about the coding constructs that the player needs practice with.

Acknowledgements

This research is supported by NSF AISL (Advancing Informal STEM Learning) Award Id: 1810972.

References

De Kegel, B., and Haahr, M. 2019. Procedural puzzle generation: a survey. *IEEE Transactions on Games* 12(1):21–40.

Dong, Y., and Barnes, T. 2017. Evaluation of a template-based puzzle generator for an educational programming game. In *Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Font, J. M.; Izquierdo, R.; Manrique, D.; and Togelius, J. 2016. Constrained level generation through grammar-based evolutionary algorithms. In *European Conference on the Applications of Evolutionary Computation*, 558–573. Springer.

Harteveld, C.; Smith, G.; Carmichael, G.; Gee, E.; and Stewart-Gardiner, C. 2014. A design-focused analysis of games teaching computer science. *Proceedings of Games+ Learning+ Society* 10:1–8.

Jemmali, C.; Bunian, S.; Mambretti, A.; and El-Nasr, M. S. 2018. Educational game design: an empirical study of the effects of narrative. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, 34. ACM.

Jemmali, C.; Kleinman, E.; Bunian, S.; Almeda, M. V.; Rowe, E.; and El-Nasr, M. S. 2019. Using game design mechanics as metaphors to enhance learning of introductory programming concepts. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, 1–5.

Miljanovic, M. A., and Bradbury, J. S. 2018. A review of serious games for programming. In *Joint International Conference on Serious Games*, 204–216. Springer.

Park, K.; Mott, B. W.; Min, W.; Boyer, K. E.; Wiebe, E. N.; and Lester, J. C. 2019. Generating educational game levels with multistep deep convolutional generative adversarial networks. In *2019 IEEE Conference on Games (CoG)*, 1–8. IEEE.

Shaker, N.; Liapis, A.; Togelius, J.; Lopes, R.; and Bidarra, R. 2016. Constructive generation methods for dungeons and levels. In *Procedural Content Generation in Games*. Springer. 31–55.

Smith, A. M., and Mateas, M. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3):187–200.

Smith, G., and Whitehead, J. 2010. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 1–7.

Smith, A. M.; Butler, E.; and Popovic, Z. 2013. Quantifying over play: Constraining undesirable solutions in puzzle design. In *FDG*, 221–228.

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):257–270.

Taylor, J., and Parberry, I. 2011. Procedural generation of sokoban levels. In *Proceedings of the International North American Conference on Intelligent Games and Simulation*, 5–12.

Togelius, J., and Shaker, N. 2016. The search-based approach. In *Procedural Content Generation in Games*. Springer. 17–30.

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):172–186.

Togelius, J.; Shaker, N.; and Dormans, J. 2016. Grammars and l-systems with applications to vegetation and levels. In *Procedural Content Generation in Games*. Springer. 73–98.

Traichioiu, M.; Bakkes, S.; Roijers, D. M.; et al. 2015. Grammar-based procedural content generation from designer-provided difficulty curves. In *FDG*.

Valls-Vargas, J.; Zhu, J.; and Ontañón, S. 2017. Graph grammar-based controllable generation of puzzles for a learning game about parallel programming. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 1–10.

Van der Linden, R.; Lopes, R.; and Bidarra, R. 2013. Designing procedurally generated levels. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.