Procedural Level Generation via Program Inversion

Harper Noteboom[1,†], Kalyani Nair[1,†] and Seth Cooper[2]

[1]Pomona College
[2]Northeastern University

Abstract

Procedurally generating levels for games often involves writing custom algorithms or providing large amounts of levels to learn from. In this work, we explore an approach to general video game level generation using a game's source program along with a single level, based on the concept of program inversion. Given the forward program for a game, we produce an inverted program in the same language that essentially runs the game backward. Starting from a solved level (which can be found searching with the forward program on a level), the inverted program can be run to generate variations on the level. We look at simple games written in a domain-specific rewrite-based language for describing tile- and turn-based games, where a single player repeatedly makes deterministic choices until they win. We verify that the level variations created by the inverted program can be solved by the original forward program.

Keywords

procedural content generation, program inversion, rewrite rules

1. Introduction

Procedural content generation (PCG) for creating game content, such as game levels, has become increasingly popular [1]. There are a wide variety of approaches to PCG, with claimed benefits such as various kinds of replayability and tailoring content to players [2].

However, the majority of PCG techniques require a fair amount of additional work above and beyond the work needed to create a game in the first place. "Classical" rule-, search- and constraint-based approaches require writing custom code just for generating content (e.g., a grammar, a fitness function, or constraints). More recent machine learning and "generative AI" [3] approaches often require large amounts of training data, potentially defeating the purpose of generating content in the first place [4]. If the content being generated is game levels, it is important for them to be completable and solvable by the player, and it is not uncommon to also apply a game-playing agent for testing content (e.g., [5, 6]).

With the goal of reducing the amount of additional work needed to generate content for a game, in this work we explore an approach to generating content for a game based on things a designer would need to create for the game anyway: the game's source code, along with at least one level. This can be considered general video game level generation [7]: generating levels for a whole class of games implemented in description language. Our approach is based on the concept of *program inversion*: the game's source code is *inverted* to create a new program, in the original description language, that undoes player moves in the game. The inverted program can then be run on a solved level to generate new starting levels that we know can be solved by the original program. We call this concept procedural level generation via program inversion.

In this work we take an initial step in this direction. We use programs written in a small variant of the domain-specific language *Tile Rewrite Rule Behavior Trees* (TRRBTs) [8]. In this language, 2D grids of tiles, called *boards*, are used to represent starting levels as well as the current game configuration, and the logic of a game's program is represented using a behavior-tree like structure, called a *tree*. Leaf

Joint AIIDE Workshop on Experimental Artificial Intelligence in Games and Intelligent Narrative Technologies, November 10-11, 2025, Edmonton, Canada.

[†]These authors contributed equally.

hjnf2022@mymail.pomona.edu (H. Noteboom); ksne2023@mymail.pomona.edu (K. Nair); se.cooper@northeastern.edu (S. Cooper)
0000-0003-4504-0877 (S. Cooper)

nodes of the tree can change the board by rewriting one pattern of tiles (the *left-hand side*) with another (the *right-hand side*), or check for the existence of a pattern of tiles on the board by matching it. We chose this language as inverting the changes to board configuration based on rewrites is straightforward: swap the left and right hand sides of the rule. Additionally, the modularity of the language allows us to limit the structure of supported games, and we do not use the full range of tree and node structures that TRRBTs can support. We limit to games with a simple tree structure: those where a single player repeatedly makes a choice with deterministic consequences until they win.

To generate levels for a game, given the game's TRRBT code *tree* and a board, we use the tree to solve the board (if needed — an already solved board can be provided), then invert the tree and run the inverted tree on the solved board to generate new boards. These boards are variations on the original board. These variations maintain certain features from the initial input board as there are elements of the boards that cannot be changed by rewrites (e.g., walls or the dimensions of the board). However, this approach is effective in generating interesting and solvable level variations. While the result of a program inversion with one starting board will have some underlying constants between all levels, these level variations have advantages from requiring many more steps to solve to having more interactive pieces on the board, depending on the game.

We present the program inversion approach we used for TRRBTs having this structure, and implemented four games: *Peg Solitaire* (jumping pegs to remove them), *Merge* (merging neighboring numbers into higher numbers), *Sokoban* (pushing crates onto targets), and *TwoDoor* (a simple lock-and-key dungeon). We show empirically that the inverted trees only generate solvable boards, by enumerating them and checking their solvability. We also show that generating boards using an inverted tree compares favorably to a similar approach using only the original (forward) tree.

This work contributes the general concept of program inversion as an approach to generating content for games from a game's source code, as well as an exploration of this concept in a domain-specific language, generating levels across several games.

2. Related Work

Procedural Content Generation. A variety of approaches have been developed for procedural content in games [1]. These range from "classic" techniques, such as constructive- or rule- based (e.g., [9, 10]), to search-based (e.g., [11, 12]), and constraint-based (e.g., [13, 14]) methods. Such approaches generally require implementing a generator in addition to implementing the game itself.

Recently, machine learning approaches that can learn from example content have been developed in the area of PCGML [3], as well as "generative AI" [15]. Such approaches can require large amounts of data for training or pre-training, and may not guarantee solvability of generated levels (sometimes relying on an agent test or repair step). Some learning approaches, however, can learn from a single example level (e.g., WaveFunctionCollapse [16] or Sturgeon [17]). These approaches could thus potentially require only the game's code and an example level to work, as they could learn to generate levels from the example level, and then use the code to check solvability, in a generate-and test framework. Comparing that to our program inversion approach would be interesting future work.

In this work we enumerate all the possible boards found by the inverted tree. This is essentially a form of Exhaustive PCG [18]; however, we do this mainly to confirm the levels' solvability. Some work has looked at interactively visualizing and filtering such large datasets of generated levels [19].

General Video Game Level Generation and Game DSLs. A number of domain-specific languages have been developed over the years for describing both board and video games, such as Ludii [20], Game Description Language [21], Video Game Description Language [22], Ceptre [23], and PuzzleScript [24]. Some have been used as targets for general video game level generation: that is, given a game's code in some description language along with a player agent, generate levels for the game [7].

One approach in this area approached generating levels for games written in VGDL by translating it into Answer Set Programming [25]. This approach also applied an evolutionary process with agents verifying that levels are solvable. Other work developed a framework for evaluating general VGDL

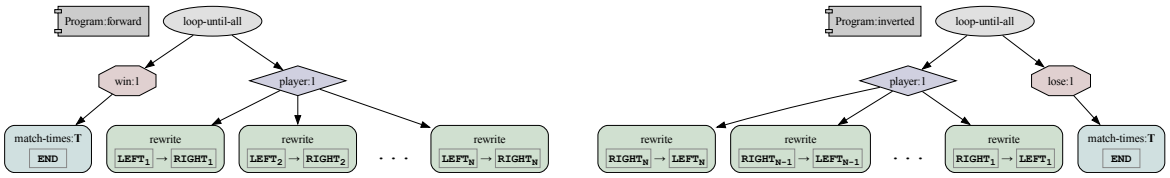

Figure 1: Generic approach to TRRBT program inversion. Forward tree on the left and inverted tree on the right.

level generators [7]. VGDL has also been used for evaluating general level generators [26].

PuzzleScript has also been a target for general level generation. For example, one approach has used constructive and search based techniques based on an analysis of a game's rules [27]. While most prior work in general video game level generation creates levels, our work aims to create a level generator in the original language. Notably, earlier work in TRRBTs [8] generated randomly shuffled levels using the player's moves from within the tree, but only if the individual moves themselves were reversible.

Game Mechanic Inversion. Some previous work has looked at inverting moves or mechanics in specific games as a way to ensure the solvability of generated levels. This is a not uncommon approach to generating *Sokoban* [28] levels, by letting the character pull rather than push crates from a solved configuration (e.g., [29, 30, 31]), as well as similar specific games. A similar approach, that of undoing moves, has also been applied to generating Sudoku number puzzles [32]. Some work on *Sokoban* has also looked at using reversed moves to *solve* puzzles, starting from the solution and searching backwards to the start [33]. While usually these works look at specific games, we are looking at a more general technique for a whole category of games.

Program Inversion. The general concept of program inversion [34], roughly speaking, is to derive a program that does the opposite of some given program, mapping outputs back to inputs. A number of approaches to program inversion focus on theoretical proofs of the correctness of the inversion [35], for example of tree traversals [36, 37] or certain types of matrix multiplication [38]. The program inversion concept is promising for programs that naturally pair in this way, such as compression/decompression programs [39].

In some cases a non-deterministic mapping from outputs back to inputs may be acceptable [40] or even desirable. A similar approach to ours, for example, has been used to generate TIFF images for testing implementations of image clients [41]. They use a *normalization program* to create a normalized representation of an image, and then invert that program and use the *inverse program* to generate test images. This can be seen as analogous to our approach of using the original (forward) program to solve a level and then the inverse program to generate new levels from that.

3. System Overview

Here we describe the approach we take to generating levels based on a game's source code.

Game Representation. Our inversion method for level generation uses Tile Rewrite Rule Behavior Trees (TRRBTs) as the underlying game representation [8], although we only support a very specific type of game, and not the full range of games that TRRBTs allow. Games consist of a *board* and a *tree*.

The board is a 2D array of tiles — strings of characters — that represent the current configuration of a game. A board can have multiple named layers of tiles that are stacked on top of each other. This makes it easier for something like foreground tiles to move over background tiles (e.g. pushing crates over targets in *Sokoban*, discussed below). Terminology-wise, "board" can be considered the term for a "level" in TRRBTs. A layered 2D array of tiles that does not represent the game board is also called a *pattern*. A board can also have a *result* associated with it, either *win* or *lose* if the game has ended in one of those outcomes, *stalemate* if the game has ended without an explicit outcome, or *nil* if the game is still running.

The tree determines the logic of the game and makes changes to the board. The tree is run as a behavior tree, with nodes succeeding or failing based on specific conditions for their node type, and

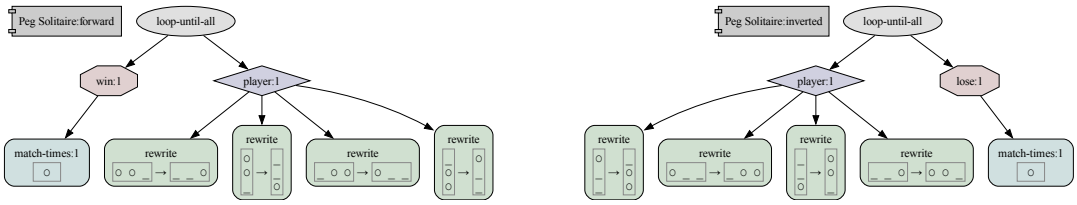

Figure 2: Inversion of tree for *Peg Solitaire*. Forward tree on the left and inverted tree on the right.

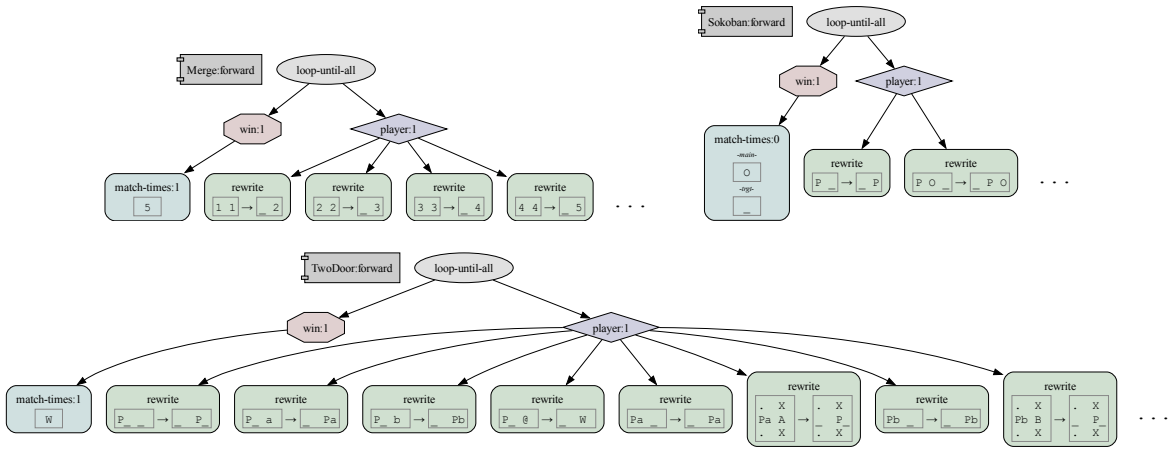

Figure 3: Forward trees for other games used. For space, *rewrite* nodes that are rotations of others are omitted.

returning that information to their parent node, which adjusts its behavior based on its children.

Both a board and a tree are required for a game to be played. Although in previous work boards were included directly in the tree through a *set-board* node, in our method, we separated the board from the tree so that we could more easily invert the tree and and allow for different starting boards to be passed into the same tree.

Supported Tree Structure and Inversion. In this work we use only a few node types and require them to be in a particular arrangement. The node types are:

- *loop-until-all*: Iteratively runs children in order until all children fail on one iteration. Succeeds if any child succeeds, otherwise fails.
- *win* and *lose*: Runs children in order. If any child succeeds, immediately ends the game with the respective result for its associated player; otherwise fails.
- *player*: All children of a player node must be rewrite nodes. If any LEFT pattern of a child is present in the current board, the player can chose one LEFT to be replace with its associated RIGHT and the player node succeeds. Fails if there are no LEFT patterns present.
- *rewrite*: Contain the rewrites, where the associated LEFT can be replaced with the associated RIGHT. In this work, these are always the child of a player node and so not run directly.
- *match-times*: Succeeds if its pattern is present in the current board its associated number of times, otherwise fails.

Our program inversion approach takes a tree representing a game's (forward) logic as an input. These trees are required to fit the specific tree structure shown in Figure 1(left). This structure essentially represents a game loop that alternates checking for a win and the player making a move. The root is a *loop-until-all* node with two children. The first child of this loop is a *win* node that has a child *match-times* node that succeeds if a pattern END is matched on the board **T** times. The second child of loop-until-all is required to be the *player* node with a series of **N** *rewrite* nodes with patterns $\texttt{LEFT}_\texttt{i}$ and $\texttt{RIGHT}_\texttt{i}$, $1 \leq \texttt{i} \leq \mathbf{N}$ as children. Note that in this structure, checking for a win before the player moves allows starting from a board that is already solved.

`Inverted-Enumerate`
In: $tree$: (forward) game tree
$board$: starting board
Out: $boardsGen$: generated boards
$boardSolved \leftarrow solve(program, board)$
$treeInv \leftarrow invert(tree)$
$boardsEnum \leftarrow enumerate(treeInv, boardSolved)$
$boardsGen \leftarrow \{b \in boardsEnum \mid b_{steps} > 0 \wedge b_{result} \in \{nil, stalemate\}\}$
foreach $b \in boardsGen$:
$assert(solve(tree, b) \neq nil)$
end

Algorithm 1: Inverting tree and inversely enumerating boards

`Forward-Enumerate`
In: $tree$: (forward) game tree
$board$: starting board
Out: $boardsGen$: generated boards
$boardsEnum \leftarrow enumerate(tree, board)$
$boardsGen \leftarrow \{b \in boardsEnum \mid b_{result} = nil \wedge solve(tree, board) \neq nil\}$

Algorithm 2: Forward enumerating boards

The inverted version of such a tree is shown in Figure 1(right). The root remains a *loop-until-all* node, with the order of the children reversed. First is the *player* node, with the order of its *rewrite* children reversed, and the sides if these individual *rewrite* nodes reversed, and so that $\texttt{RIGHT}_\texttt{i}$ is on the left and $\texttt{LEFT}_\texttt{i}$ is on the right. This allows individual rewrites to be done backward. Second is a *lose* node, taking the place of the *win* node and its *match-times* child and associated END. A lose node is used so that the inverted game won't output a board that is already winning. Note that here losing is checked for after making a move, as the inverted tree presumes that it is starting from a board that won from the forward tree.

The resulting inverted tree can be run on a solved board, and each board that would be presented to the player will be a solvable starting board for the game. This can be used to generate new levels that are variations on the same underlying level structure.

4. Evaluation

Here we describe evaluations of the program inversion approach. Game trees and enumerated boards from the evaluation can be found at https://osf.io/er5wh/.

Inverted Enumeration of Boards. To empirically verify that the inverted trees generate only boards that can be solved, and to gather descriptive information on the generated boards, we enumerate all boards that are produced by an inverted tree, starting from a solved board.

The overall approach is shown in Algorithm 1. It starts by taking the (forward) tree for a game in the structure described above and a solvable starting board for the game. The function *solve* runs a breath-first search with the forward tree until it reaches a solved board that wins the game; this also associates the number of *steps* it took to solve with the board. Then *invert* produces the inverted tree as described above. Next, *enumerate* finds all boards reachable by the inverted tree, starting from the solved board; each enumerated board also gets associated with the number of *steps* it took to reach. From these enumerated boards, we keep only those that took at least one step to reach (so they are not the solved board from which inverted enumeration began) and either have not ended or ended in a stalemate. These enumerated boards are checked by asserting that it is possible to win each of then using the forward tree.

We applied this approach to four games:

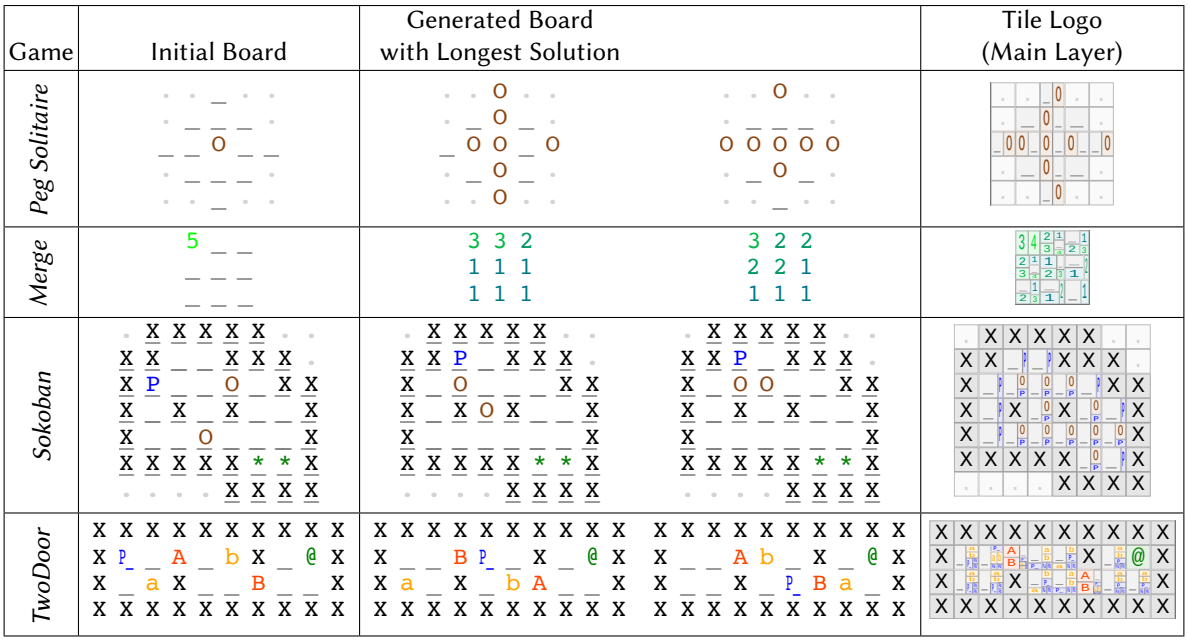

Figure 4: Starting and selected generated boards from each game. The tile logo visualizes the proportions (relative to the square root of the actual tile count) of different tiles at each location of the enumerated levels.

- *Peg Solitaire*: A game where the player jumps over and removes pegs `O`; the goal is to end up with one peg left. The tree structure for Peg Solitaire can be seen in Figure 2; the *match-times* node represents having one peg left, each rewrite represents jumping over and removing a peg. This game used a solved starting board.
- *Merge*: The player can merge two neighboring tiles with the same number to create one tile with a number one larger. The goal is to end up with one tile with the number `5`. The tree can be seen in Figure 3; each *rewrite* node represents the combining of neighboring tiles and the *match-times* checks for a single tile with `5`. This game used a solved starting board.
- *Sokoban*: Based on the puzzle game Sokoban [28], the player's character `P` can push crates `O` around the floor. The goal is to get all the creates onto target locations `*`. This game makes use of TRRBT's layers to have a separate layer for the targets under the main layer so the player and crates can move over them. The use of layers can be seen in the tree for Sokoban in Figure 3; the *match-times* node ensures there are no crates on the main layer that don't line up with the targets on the target layer.
- *TwoDoor*: The player's character `P_` can navigate a simple area with two keys, `a` and `b`, and two doors, `A` and `B`. They can carry one key at a time, (e.g., `Pa`). Their goal is to reach the exit `@`; the player can only go to the exit when not carrying a key. Notably in this game, the rewrite rule to remove a door includes the neighboring walls, which thus requires the inverted tree to only place doors between two walls (rather than anywhere on the board); thus the forward game tree was partially written with the inverted tree in mind. The tree can be seen in Figure 3.

The forward and inverted trees for *Peg Solitaire* are shown in Figure 2, and the forward trees for the other games are shown in Figure 3. Figure 4 shows the initial boards used for each game and selected boards generated with the longest solutions, created with `level2image`. It also shows a "tile logo" visualizing the proportions of different tiles at each location of the enumerated levels, created with Level Explorer [19], which uses `squarify`'s [42] implementation of squarified treemaps [43]. A summary description of the inverse enumerated boards is given in Table 1 under "Inverted".

Comparison to Forward Enumeration of Boards. We considered it may also be informative to compare the boards enumerated by the inverse tree to those enumerated by simply running the forward tree from the starting board. This was done using Algorithm 2, which enumerated the boards using the

	Inverted			Forward		
Game	Gen. Board Count	Avg. Soln. Length	Max Soln. Length	Gen. Board Count	Avg. Soln. Length	Max Soln. Length
Peg Solitaire	38	3.632	6	0	—	—
Merge	273	5.073	8	0	—	—
Sokoban	820	26.280	57	315	20.959	42
Twodoor	1455	8.451	18	37	7.081	11

Table 1
Summary of enumerated boards using inverted and forward enumeration.

forward tree from the starting board, and keeps those that have not yet come to a result and can also be solved. A summary description of the enumerated boards is given in Table 1 under "Forward".

5. Discussion

Checking that all the boards enumerated by the inverted tree when starting from a solved board are themselves solvable shows, empirically in the cases that were tried, that running the inverted tree will always result in a solvable board that could thus be used in a game.

One potentially useful feature of the approach is that it works from a solved board or a starting board. This provides flexibility, as a designer can either use a starting board (which they may already have if they are working on a game) and get variations on that board, or start from a desired solution and generate boards directly from that.

We also found that, in comparison to just forward enumeration from a solvable board, inverse enumeration find more boards. When starting from a solved board there are, unsurprisingly, no new boards to be found. But even in the games starting from unsolved boards, inverted enumeration is able to find a larger number of boards with higher average and maximum solution lengths.

We also think, subjectively, inverse enumeration finds some interesting levels. For example, the generated *Sokoban* boards appear to include the provided starting board as a step along the solution path. The generated *TwoDoor* boards involve backtracking a bit after collecting a key: one to collect the other key behind a door, and one just to use up a key in a door as the player can only exit without a key.

6. Conclusion

In this work we have presented the concept of program inversion for procedural content generation. Although other work has applied inverted mechanics to content generation in specific games, we show an implementation of that concept using a small version of the Tile Rewrite Rule Behavior Trees domain-specific language that captures a class of simple games. This allows new boards to be generated for these games directly from their source code and a single example board.

In the future we are interested in expanding the approach to larger, more complex games that use more of the TRRBT language, or other other existing languages such as Ludii or PuzzleScript; also developing approaches to generating levels based just on the game's code without a provided starting level. By creating a level generator in the same language as the original game, there is the possibility of integrating the two and adding a runtime level generator back into the original game (perhaps as a language feature in a TRRBT *transform node*). Since the inverted programs are themselves games, it may be possible to play them to generate levels for the original game.

We would also like to perform a user study of game designers to see how useful this approach is to them. We are also interested in methods to verify inversion works for generating solvable levels theoretically rather than empirically. Finally, program inversion may be useful for generating other types of content.

Acknowledgments

The authors would like to thank Chris Martens, Cynthia Li, and Kaylah Facey for their work on the underlying TRRBT language and input on this work.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools in preparing this work.

References

[1] N. Shaker, J. Togelius, M. J. Nelson, Procedural Content Generation in Games, 2016.
[2] G. Smith, Understanding procedural content generation: a design-centric analysis of the role of PCG in games, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2014, pp. 917–926.
[3] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, J. Togelius, Procedural Content Generation via Machine Learning (PCGML), IEEE Transactions on Games 10 (2018) 257–270.
[4] I. Karth, A. M. Smith, Addressing the fundamental tension of PCGML with discriminative learning, in: Proceedings of the 14th International Conference on the Foundations of Digital Games, 2019.
[5] A. J. Summerville, S. Philip, M. Mateas, MCMCTS PCG 4 SMB: Monte Carlo tree search to guide platformer level generation, AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (2015).
[6] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. Smith, 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.
[7] A. Khalifa, D. Perez-Liebana, S. M. Lucas, J. Togelius, General video game level generation, in: Proceedings of the Genetic and Evolutionary Computation Conference 2016, 2016, pp. 253–259.
[8] J. Zhou, C. Martens, S. Cooper, Authoring games with tile rewrite rule behavior trees, in: Proceedings of the 19th International Conference on the Foundations of Digital Games, 2024.
[9] J. Dormans, Adventures in level design: Generating missions and spaces for action adventure games, in: Proceedings of the 2010 Workshop on Procedural Content Generation in Games, 2010.
[10] G. Smith, J. Whitehead, M. Mateas, M. Treanor, J. March, M. Cha, Launchpad: a rhythm-based level generator for 2D platformers, IEEE Transactions on Computational Intelligence and AI in Games 3 (2011) 1–16.
[11] B. M. F. Viana, L. T. Pereira, C. F. M. Toledo, S. R. dos Santos, S. M. D. M. Maia, Feasible–infeasible two-population genetic algorithm to evolve dungeon levels with dependencies in barrier mechanics, Applied Soft Computing 119 (2022) 108586.
[12] A. Alvarez, S. Dahlskog, J. Font, J. Holmberg, C. Nolasco, A. Österman, Fostering creativity in the mixed-initiative evolutionary dungeon designer, in: Proceedings of the 13th International Conference on the Foundations of Digital Games, 2018, pp. 1–8.
[13] M. J. Nelson, A. M. Smith, ASP with applications to mazes and levels, in: N. Shaker, J. Togelius, M. J. Nelson (Eds.), Procedural Content Generation in Games, 2016, pp. 143–157.
[14] I. Karth, A. M. Smith, Wavefunctioncollapse is constraint solving in the wild, in: Proceedings of the 12th International Conference on the Foundations of Digital Games, 2017, pp. 1–10.
[15] S. Sudhakaran, M. González-Duque, C. Glanois, M. Freiberger, E. Najarro, S. Risi, MarioGPT: open-ended text2level generation through large language models, 2023. ArXiv:2302.05981 [cs].
[16] M. Gumin, Wavefunctioncollapse, https://github.com/mxgmn/WaveFunctionCollapse, 2016.
[17] S. Cooper, Sturgeon: tile-based procedural level generation via learned and designed constraints, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 18 (2022) 26–36.

[18] N. R. Sturtevant, M. J. Ota, Exhaustive and semi-exhaustive procedural content generation, AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (2018).
[19] S. Cooper, F. Abutarab, E. Halina, N. Sturtevant, Visual exploration of tile level datasets, in: Proceedings of the Experimental AI in Games Workshop, 2023.
[20] M. Stephenson, É. Piette, D. J. Soemers, C. Browne, An overview of the Ludii general game system, in: 2019 IEEE Conference on Games (CoG), 2019, pp. 1–2. ISSN: 2325-4289.
[21] N. Love, T. Hinrichs, M. Genesereth, General game playing: Game Description Language specification, Technical Report LG-2006-01, Stanford Logic Group, 2006.
[22] T. Schaul, A video game description language for model-based or interactive learning, in: 2013 IEEE Conference on Computational Inteligence in Games (CIG), 2013, pp. 1–8. ISSN: 2325-4289.
[23] C. Martens, A. Card, H. Crain, A. Khatri, Modeling game mechanics with Ceptre, IEEE Transactions on Games 16 (2024) 431–444.
[24] S. Lavelle, Puzzlescript, https://www.puzzlescript.net/, 2013.
[25] X. Neufeld, S. Mostaghim, D. Perez-Liebana, Procedural level generation with answer set programming for general video game playing, in: 2015 7th Computer Science and Electronic Engineering Conference (CEEC), 2015, pp. 207–212.
[26] O. Drageset, M. H. M. Winands, R. D. Gaina, D. Perez-Liebana, Optimising level generators for general video game AI, in: 2019 IEEE Conference on Games (CoG), 2019, pp. 1–8. ISSN: 2325-4289.
[27] A. Khalifa, M. Fayek, Automatic puzzle level generation: A general approach using a description language, in: Computational Creativity and Games Workshop, 2015.
[28] Thinking Rabbit, Sokoban, 1982. Game.
[29] J. Taylor, I. Parberry, Procedural generation of Sokoban levels, in: Proceedings of the International North American Conference on Intelligent Games and Simulation, 2011.
[30] K. Cho, Automatic level generation for puzzle games, https://abagames.github.io/joys-of-small-game-development-en/procedural/puzzle_level.html, 2023. Accessed: 2025-08-26.
[31] M. Brzozowski, Sokoban level generator, https://github.com/miki151/sokoban/, 2016. Accessed: 2025-08-26.
[32] T. Boothby, L. Svec, T. Zhang, Generating Sudoku puzzles as an inverse problem, Mathematical contest in modeling (2008).
[33] F. W. Takes, Sokoban: Reversed Solving, Master's thesis, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2008.
[34] E. W. Dijkstra, Program inversion, in: Selected Writings on Computing: A Personal Perspective, 1982, pp. 351–354.
[35] B. J. Ross, Running programs backwards: the logical inversion of imperative computation, Form. Asp. Comput. 9 (1997) 331–348.
[36] W. Chen, J. T. Udding, Program inversion: more than fun!, Science of Computer Programming 15 (1990) 1–13.
[37] B. Schoenmakers, Inorder traversal of a binary heap and its inversion in optimal time and space, in: R. S. Bird, C. C. Morgan, J. C. P. Woodcock (Eds.), Mathematics of Program Construction, 1993, pp. 291–301.
[38] W. Chen, A formal approach to program inversion, in: Proceedings of the 1990 ACM annual conference on Cooperation, 1990, pp. 398–403.
[39] S. Srivastava, S. Gulwani, S. Chaudhuri, J. S. Foster, Path-based inductive synthesis for program inversion, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011, pp. 492–503.
[40] D. Gries, J. v. d. Snepscheut, Inorder traversal of a binary tree and its inversion, The Formal Development and Proofs, Addison-Wesley Publishing Company (1989).
[41] A. Kanade, R. Alur, S. Rajamani, G. Ramanlingam, Representation dependence testing using program inversion, in: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, 2010, pp. 277–286.
[42] U. Laserson, squarify, https://github.com/laserson/squarify/, 2013. Accessed: 2025-08-20.

[43] M. Bruls, K. Huizing, J. J. van Wijk, Squarified treemaps, in: W. C. de Leeuw, R. van Liere (Eds.), Data Visualization 2000, 2000, pp. 33–42.