Sturgeon-MKIV: Constraint-Based Level and Playthrough Generation with Graph Label Rewrite Rules

Seth Cooper, Mahsa Bazzaz

Khoury College of Computer Sciences, Northeastern University
se.cooper@northeastern.edu, bazzaz.ma@northeastern.edu

Abstract

Procedurally generated game levels should be completable. The representation used for levels and game mechanics impacts the types of games for which different techniques can be applied. Previous work used a constraint solving approach to simultaneously generate levels with example playthroughs, showing they can be completed using the game's mechanics. However, that work used 2D grid-based rewrite rules. In this work, we extend previous approaches by representing levels as more general graphs, and game mechanics as rewrites on node and edge labels of subgraphs. Using this approach, graph-based levels with playthroughs are generated. We describe the approach and demonstrate its application in some games with graph-based levels.

Introduction

Video game levels are a common type of content to procedurally generate (Shaker, Togelius, and Nelson 2016). A key aspect of procedurally generated levels is to ensure that they can be completed by the player. While there are a variety of level representations, a large focus of research has been on tile-based levels. Graph-based levels have also received attention.

A popular approach to generating graph-based levels for games is the use of graph grammars. To ensure completability and other desired features of levels, these grammars may be manually designed (Dormans 2010; Dormans and Bakkes 2011), or incorporate evolutionary algorithms (Font et al. 2016). However, such approaches require either careful design of grammars or fitness functions, which may require more in-depth familiarity with these technical systems.

Recently, systems such as WaveFunctionCollapse (Gumin 2016) and variations on it (Karth and Smith 2017, 2019; Langendam and Bidarra 2022; Alaka and Bidarra 2023; Nie et al. 2024), which learn to generate levels from examples, have grown in popularity. As these systems can learn to generate levels from a few examples, they can be considered more approachable than more technical systems. They generally work on grid-based levels, although there have been some extensions to generating levels on more general, though fixed, graph structures (Kim et al. 2020).

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Can we bring the same kind of example-based level generation to graph-based levels, while also ensuring the completability and controllability of generated levels?

To explore this question, we developed Sturgeon-MKIV, a constraint-based system for generating graph-based levels and playthroughs, i.e. a sequence of graphs where the first graph is the level and the rest show the game mechanics being used to complete the level. It builds on an existing system for learning to generate graph-based levels from examples, and represents game mechanics as rewrites of the node and edge labels on subgraphs.

Label rewrite rules consist of a graph where nodes and edges have left-hand side (LHS) and right-hand side (RHS) labels. If the structure of the label rewrite rule and its LHS labels can match a subgraph in a graph, the labels can be rewritten with the RHS labels. For example, if the label P represents the player, and the label _ represents a blank space, the label rewrite rule (P→_)—(_→P) could represent the player moving onto an empty node. If it were applied to the graph (P)—(_)—(_), it would result in the graph (_)—(P)—(_). If edges are also labelled, a similar rule (P→_)-_→⋆-(_→P) would also mark that edge as visited (and not allow the player to visit edges more than once, as the edge label ⋆ on the LHS would no longer match, unless another rule was added).

While the local graph patterns are learned from examples, the label rewrite rules are manually authored, and any additional constraints on gameplay, such as starting and win conditions for levels, are also manually authored. In this work, although the initial graph structure is generated, we only allow the labels to change, and thus the structure of the graph cannot change during gameplay (e.g., nodes and edges cannot be added or removed).

This work is based on the Sturgeon constraint-based level generation system (Cooper 2022) and is in a large part a combination of the previously-developed techniques in the Sturgeon-GRAPH (Cooper 2023a; Cooper and Balema 2023) graph level generator and the Sturgeon-MKIII (Cooper 2023b) tile-based playthrough generator. While Sturgeon-GRAPH generates graph-based levels without playthroughs, and Sturgeon-MKIII generates tile-based levels with playthroughs, Sturgeon-MKIV generates playthroughs for graph-based levels.

We found that the current system could be applied primarily to single-player, turn-based puzzle-style games. We gen-

erated graph-based levels in a variety of game applications, including Lights Out, a generic adventure game map, Code Master, and Sokoban. This work contributes to the use of graph label rewrite rules as game mechanics in a constraint-based approach to generating completable graph game levels.

Related Work

Procedural Level Generation

Much research in PCG has focused on the generation of levels (Shaker, Togelius, and Nelson 2016), and a wide variety of techniques have been developed. As our work generates levels in part by learning graph patterns from examples, it can be considered a form of procedural content generation via machine learning (PCGML) (Summerville et al. 2018). It is also related to work using constraint-based approaches for PCG, e.g. (Smith et al. 2012; Nelson and Smith 2016; Snodgrass and Ontañón 2016; Carpenter et al. 2021; Cooper 2022; Katz, Bateni, and Smith 2024).

Ensuring that generated levels are completable — that is, it is technically possible for a player to complete them — is often incorporated into level generators. Constraint-based methods like those mentioned can often incorporate constraints ensuring completability. In other approaches, gameplay agents may be incorporated to check completability (Biemer and Cooper 2022; Volz et al. 2018), and search-based or evolutionary approaches (Togelius et al. 2011) may use these to guide the search for completable levels (Zafar, Mujtaba, and Beg 2020; Viana et al. 2022). Yet other work has explored taking generated levels and repairing them so that they are completable (Jain et al. 2016; Zhang et al. 2020; Cooper and Sarkar 2020; Shu et al. 2020). Closely related to this work is the Sturgeon-MKIII system, which ensures completability of grid-based levels by representing mechanics as tile rewrite rules (Cooper 2023b).

Tile Rewriting

Tile rewrite rules, which can be applied to grids, have been used in many systems to represent changes for the creation of animations, simulations, and games. One arrangement of tiles (LHS) in a grid can be rewritten with another arrangement (RHS) to update the grid. The BITPICT system (Furnas 1991) used pixel rewrites to create animations. Due to tile rewrites' expected approachability, much of this work has also focused on making game development more approachable to younger learners (Repenning 1995; Cypher and Smith 1995; Wright 2006). However, tile rewrites have also been used in other game authoring systems such as PuzzleScript (Lavelle 2013) and combined with behavior trees (Zhou, Martens, and Cooper 2024).

In addition to being used to represent game mechanics themselves, tile rewrites have been used to generate images or levels by starting from a basic grid and rewriting until some criteria are met (Tows 2009; van Rozen and Heijn 2018; Gumin 2022).

Graph Rewriting

When generating graph-based levels, oftentimes graph grammars are used. Graph grammars have a long history (Ehrig, Pfender, and Schneider 1973). These can be thought of as rewrite rules on subgraphs, where the grammar defines arrangements of nodes and edges (LHS) that can be rewritten with a different arrangement of nodes and edges (RHS).

While level-generating graph grammars are most often manually authored (Dormans 2010; Jemmali et al. 2020; Karavolos, Bouwer, and Bidarra 2015), some work has examined learning grammars from examples (Londoño and Missura 2015; Hauck and Aranha 2020; Merrell 2023) or providing designer controls over the generated graphs (Linden, Lopes, and Bidarra 2013; Valls-Vargas, Zhu, and Ontañón 2017; Madkour et al. 2021). Often in these cases, the goal of rewriting the graph is to create a level by starting with a small graph and adding nodes and edges until a desired level is created. In contrast to much prior work in graph level generation, in this work we used learned local graph patterns to generate a level; label rewrite rules are only used to generate the accompanying playthrough. As these rewrite rules cannot change the graph structure, they can take a simplified form of a single graph with a LHS and RHS on each node and edge (as opposed to the LHS and RHS each being a separate graph themselves).

Some systems for modeling games have incorporated even more general rewriting systems (Martens 2015) which can thus also be used to represent graphs to (Martens et al. 2024). Other approaches have used graphs as a visual approach to representing game mechanics, such as the Machinations system (Adams and Dormans 2012) and Petri nets (Reuter, Göbel, and Steinmetz 2015).

System Overview

As mentioned above, the Sturgeon-MKIV system is technically an extension of the Sturgeon-GRAPH (Cooper 2023a; Cooper and Balema 2023) graph generation system, adding support for simultaneous playthrough generation similar to the Sturgeon-MKIII (Cooper 2023b) system. However, rewrite rule mechanics are represented as rewrites on the labels of subgraphs, rather than on grids. As more details of the previous systems are provided in referenced papers, here we summarize the most relevant points to the current work.

At a high level, Sturgeon-MKIV takes as input example graphs; a collection of rewrite rules; custom constraints on the playthrough; and various problem setup parameters (such as minimum and maximum generated graph size, maximum number of timesteps, etc). Local node and edge patterns are extracted from the example graph(s), and along with the rewrite rules and other inputs, are combined into a constraint satisfaction problem. When solved, this produces a sequence of graphs representing a playthrough solving the level with the rewrite rules.

Sturgeon-GRAPH generates graphs by specifying a constraint problem representing the constraints on the graph (e.g. node and edge labels and connectivity). Each *potential* node and *potential* edge has a boolean variable for each of its possible labels, including a special label indicating

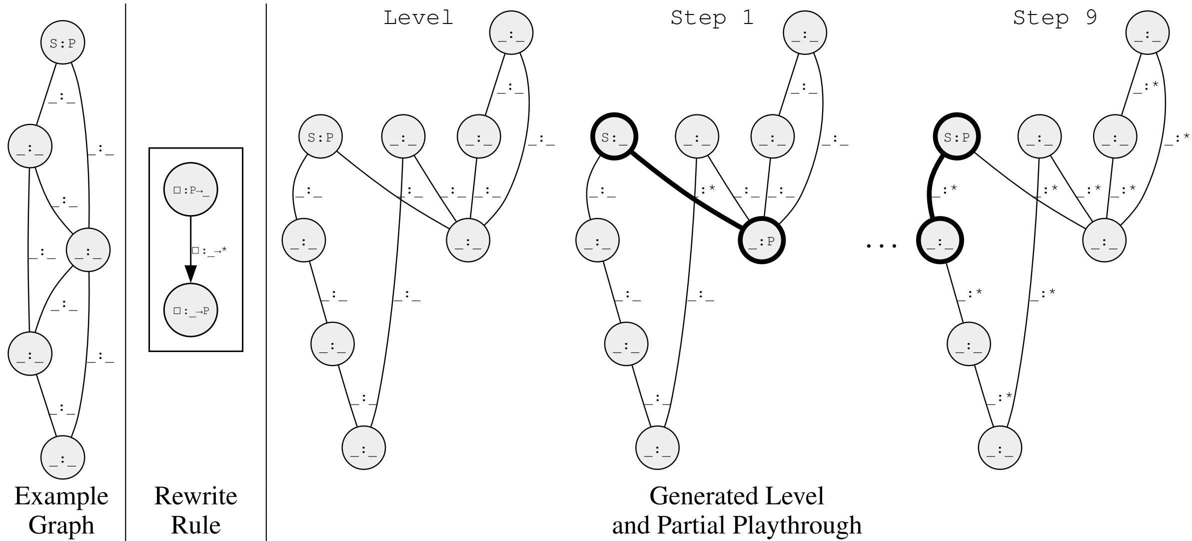

Figure 1: Using the system to generate graphs where each edge can be visited exactly once and return to the start. Local patterns for generating the level are learned from the example graph. There is a single rewrite rule, which moves the player from one node to another along an unmarked edge, and marks the edge. The generated level and playthrough show the sequence of generated graphs showing how the player would move through the graph to solve the level. The subgraph where the rule is applied is highlighted in each timestep.

that it is not present. Local *patterns* of nodes and edges are learned from example graphs by extracting the unique patterns from examples, and generated graphs are constrained such that they only contain those patterns. There can also be constraints on the type of graph generated, e.g. a directed acyclic graph or an undirected graph. In this work we also added support for general directed graphs. An *edge setup* can also be used to specify what edges are possible in the generated graph (i.e. an edge is not necessarily possible between each pair of nodes). In some cases where the solver used has support, it is possible to also solve for the position of nodes.

Similar to how Sturgeon-MKIII adds timesteps to grid-based levels, Sturgeon-MKIV adds timesteps to graph-based levels. By adding timesteps, the system generates not just an individual graph, but a sequence of graphs representing gameplay over time. In Sturgeon-MKIV, each node and label has both a *static* label, that does not change over time, and a *dynamic* label, that can change over time.

While the first timestep is the generated graph itself, each timestep after the first is essentially an assignment to the dynamic labels of the nodes and edges in the graph. Thus, for each timestep after the first, each potential node and edge has a Boolean variable for each possible dynamic label it could have (again, including a special label meaning the node or edge is not present). In this work we do not allow nodes or edges to be added or removed dynamically. Each timestep also has a variable for if it is *terminal* or not. A terminal timestep does not change dynamic labels from the previous timestep, and all following timesteps are also terminal.

How the dynamic labels of nodes and edges can change from one timestep to the next is specified by label rewrite rules. A label rewrite rule consists of a (usually small, possibly unconnected) graph, where each node and edge has a static label, a LHS dynamic label, and a RHS dynamic label. If all of the static and LHS dynamic labels match a subgraph of the current graph, the dynamic labels can be rewritten to their corresponding RHS dynamic labels. Rewrite rules can be either directed or undirected. If both the graph being rewritten and the rewrite rule are directed, then the edge direction matters, otherwise, it does not.

In the constraint problem, rewrite rules are added as constraints on the dynamic labels between adjacent timesteps. There is also a constraint that exactly one rule must be applied at each non-terminal timestep[1]. Constraints are added between each timestep to enforce this, and that any labels that are not part of a change remain the same.

Additional constraints can be added that are specific to the game, such as initial conditions applied to the first timestep and win conditions applied to the final timestep.

Once the constraint problem is set up, it is given to a *low level* solver to get an assignment for each variable. The assignment is then interpreted as labels for each potential node and edge (or their absence) as well as the dynamic labels across timesteps. These can be interpreted as a sequence of graphs representing gameplay and a solution to the level according to the rewrite rules. There is no guarantee there is not a shorter solution to the level, just that there is one.

As a basic explanatory application, we describe generating levels for a game where the goal is to find a path through a graph that visits all the edges exactly once and returns to the starting node (i.e. a Eulerian cycle). This is shown in

[1]This is similar to the *single choice* grouping order of Sturgeon-MKIII, though other groupings and orders might be added in the future, such as alternating among available moves each timestep.

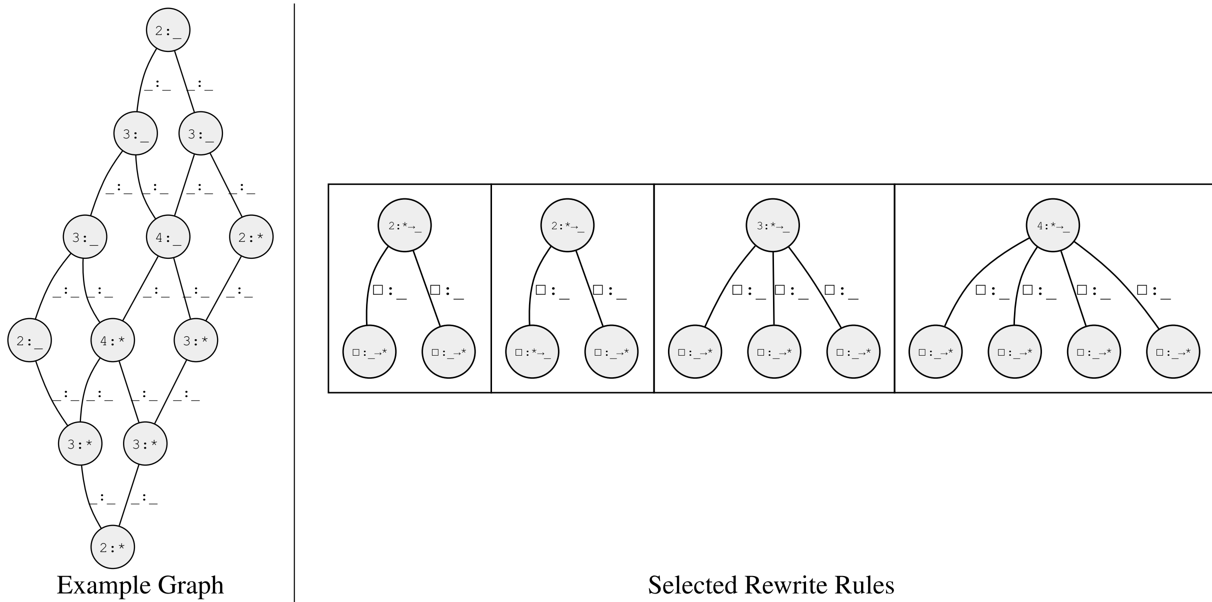

Figure 2: Example graph and selected rewrite rules for the Lights Out application. In the example graph, each node is statically labelled with its number of connected edges, some are dynamically labelled as on and some as off. The rewrite rules provide various configurations of toggling dynamic labels. For example, the first rule toggles a node from on to off and its two connected nodes from off to on.

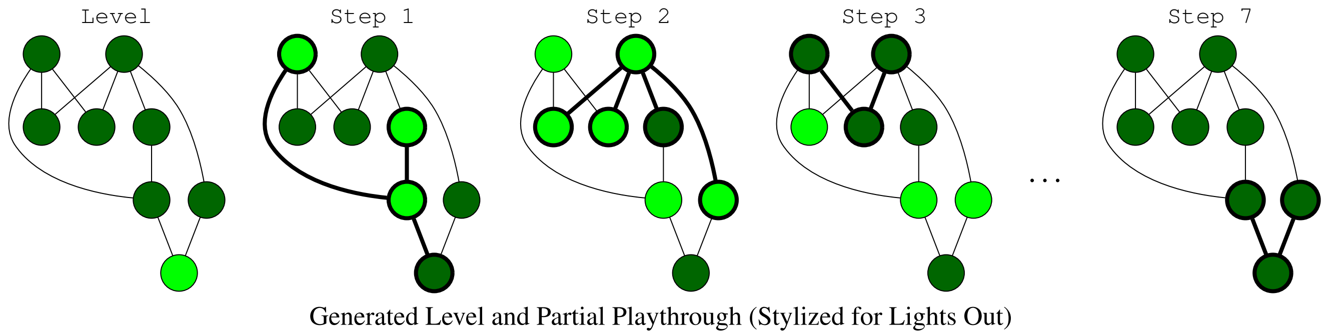

Figure 3: Sample generated level (top-left) and partial playthrough for the Lights Out application.

Figure 1.

In figures, labels are shown as `static:dynamic`. On the left of Figure 1 is the example graph used. It is undirected, and there is one node labelled `S:P`, using the static `S` label to represent the starting node for the cycle, and the dynamic label `P` to represent the player. The remaining nodes, and all edges, are labelled with blanks `_:_`.

In the middle, there is one directed rewrite rule. This rule essentially moves the player along an unvisited edge and marks the edge as visited. A box □ represents an unconstrained label (i.e. there is no constraint applied and the label could be anything). The top node matches any static label with the player `P` dynamic label, and rewrites the player with a blank `_`. The bottom node matches a blank `_`, and rewrites it with the player `P`. The edge matches a blank `_` and marks it as visited by rewriting with a `*`.

Additional constraints are added so that the level starts with one `S` and `P` label, and the level finishes with the player at the start, and no unvisited (`_`) edges.

The right of the figure shows a generated level and playthrough. In playthroughs, nodes and edges with labels changed from the previous timestep are highlighted.

Applications

In this section we describe game applications as a demonstration of the system. For each application, we generated five levels to get some rough timing information, and selected an example generated playthrough to show.

The patterns learned from examples only include the labels of a node and the edges directly connected to that node. To solve the constraint problems, all applications used PySAT's (Ignatiev, Morgado, and Marques-Silva 2018) MiniCard (Liffiton and Maglalang 2012) solver, except for the Adventure Game Map application, which used the z3

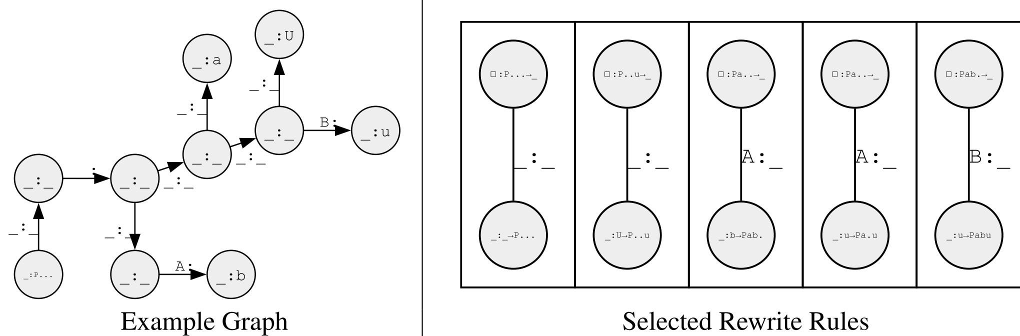

Figure 4: Example graph and selected rewrite rules for the adventure game map application. For example, the first rule is the player, with no pickups, moving to a new room; the second is the player with just the u powerup defeating the U boss; the third is the player with the a key moving through an A lock to get the b key.

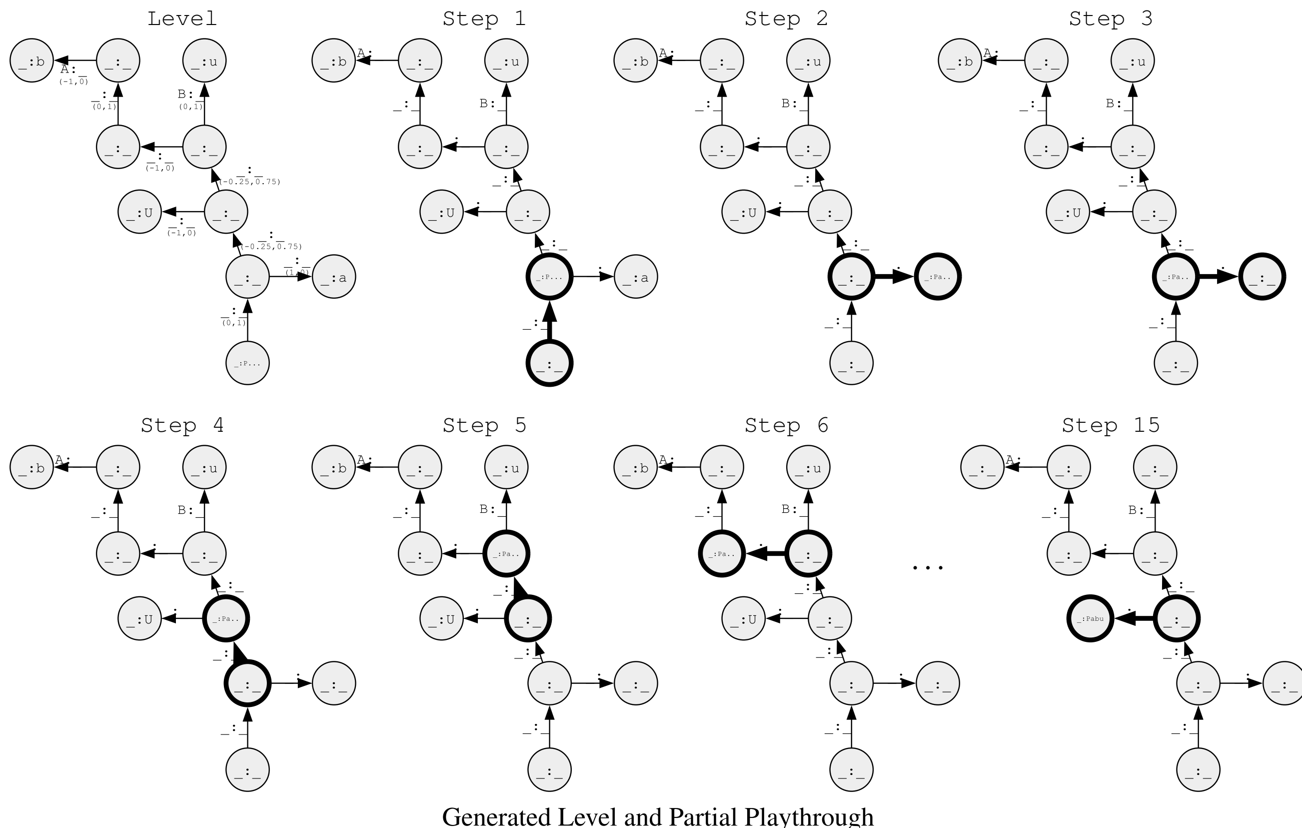

Figure 5: Sample generated level and partial playthrough for the adventure game map application. Note that initial level in top-left shows relative positions of nodes used in the constrained layout.

solver (de Moura and Bjørner 2008).

The example levels, rewrite rules, and generated levels and playthroughs are available on OSF at https://osf.io/fgj9u/.

Lights Out

Lights Out is a single-player puzzle game developed by Tiger Electronics (Tiger Electronics 1995). The standard version consists of a 5×5 grid of lights. Initially, some of the lights are on. Pressing a light will toggle that light as well as the (up to 4) neighboring lights from on to off (or vice versa). The goal is to turn off all the lights.

In our variation of Lights Out, each node in an undirected graph represents a light with a dynamic label of on ($\star$) or off (_), connected to 2–4 neighbors. The rewrite rules provide all possible light toggles for all possible numbers of neighbors. In the undirected rewrite rules, the node representing the light to be pressed has a static label of the number of

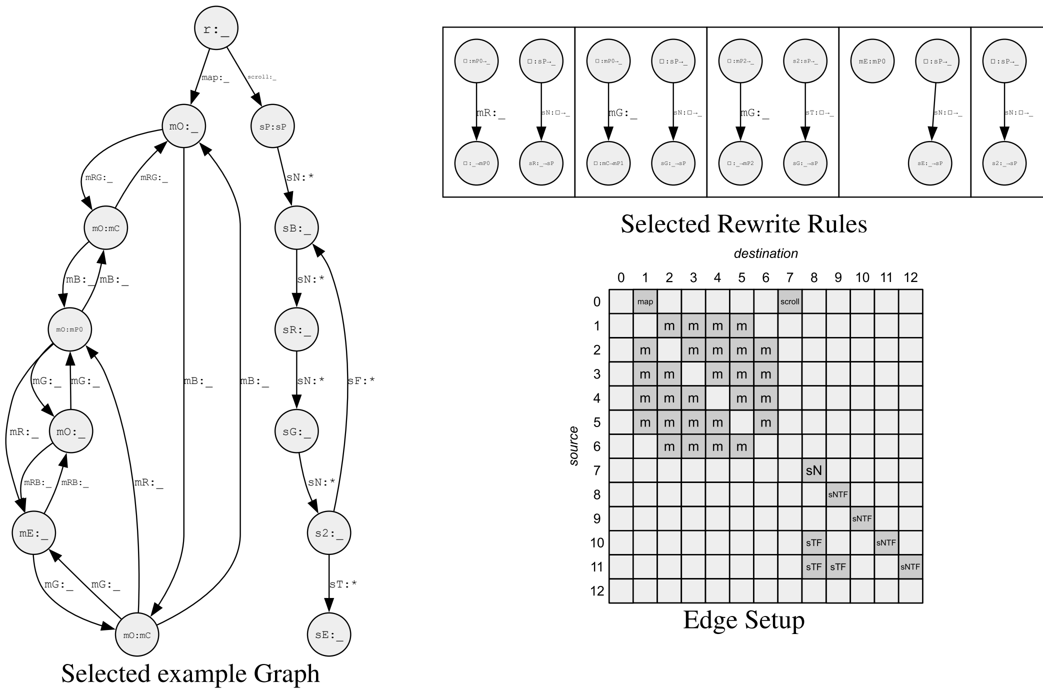

Figure 6: Selected example graph, selected rewrite rules, and edge setup used for the Code Master application. The graphs are from levels 1, 2, and 23 of the game. The first three rules are the player moving along the map and scroll on matching colors, potentially picking up a crystal; the fourth is the player moving onto the scroll exit while on the map exit; the fifth is the player moving onto the scroll branch. The edge setup shows an example of the possible edges and associated labels in generated graphs. In each cell, ‘map’ and ‘scroll’ represent edges from the root node to the two parts of the graph; ‘m’ represents any map-related edge; ‘sN’, ‘sTF’, and ‘sNTF’ represent scroll-related next, true, and/or false edges as indicated.

connected edges. This prevents matching on a subset of the edges and toggling fewer connected lights than required. In the first timestep, there must be exactly one light on, and in the last timestep, no lights on.

The example graph and selected rewrite rules are shown in Figure 2. We generated levels with between 6 and 8 nodes, with 8 timesteps. We used the *band-4* edge setup, where each node can have an edge to the 4 nodes following it, based on node id. An example level and playthrough, stylized to look more like Lights Out, is shown in Figure 3. The minimum, mean, and maximum generation times were 29.4s, 56.6s (SD=33.4s), and 113.7s.

Adventure Game Map

This application is a basic generic adventure game map. The map is a directed acyclic graph, where nodes represent rooms and edges connections between rooms. Dynamic node labels represent the player and their inventory (`P...`), and pickups or bosses in the room. The player can pick up two keys (`a` and `b`) and one powerup (`u`); if the player has a pickup it is in the player label (e.g. a player with both keys, but not the powerup, would be `Pab.`). Edges can be statically labelled with a lock, and the player must have the corresponding key to traverse that edge. If a node is dynamically labelled with a boss, the player must have the corresponding powerup to enter that room, and entering removes the boss; this represents the player needing the powerup to defeat the boss.

In this application, we also make use of Sturgeon-GRAPH’s constrained layout feature (Cooper and Balema 2023), so that the constraint problem also includes solving for the node positions based on their relative locations along edges. As noted above, because of this we used the z3 solver (de Moura and Bjørner 2008) for this application, because it has support for the real-valued variables used in positional constraints.

The rewrite rules generally represent the player moving from one room to another, when it is possible for them to do so, and collecting any pickup there. In the custom-made example level, the player generally needs to get key `a`, then key `b`, then the powerup `u`, then defeat the boss `U`. The example level and selected rules are shown in Figure 4.

Generated levels must start with one player and one of each pickup and boss, and end with no bosses left. Levels could be between 8 and 10 nodes, with 16 timesteps. We used *stripe-1,4* edges, where each node can connect to the first and fourth following nodes, by node id. An example level and playthrough is shown in Figure 5. The minimum, mean, and maximum generation times were 38.1s, 52.9s (SD=11.1s), and 65.4s.

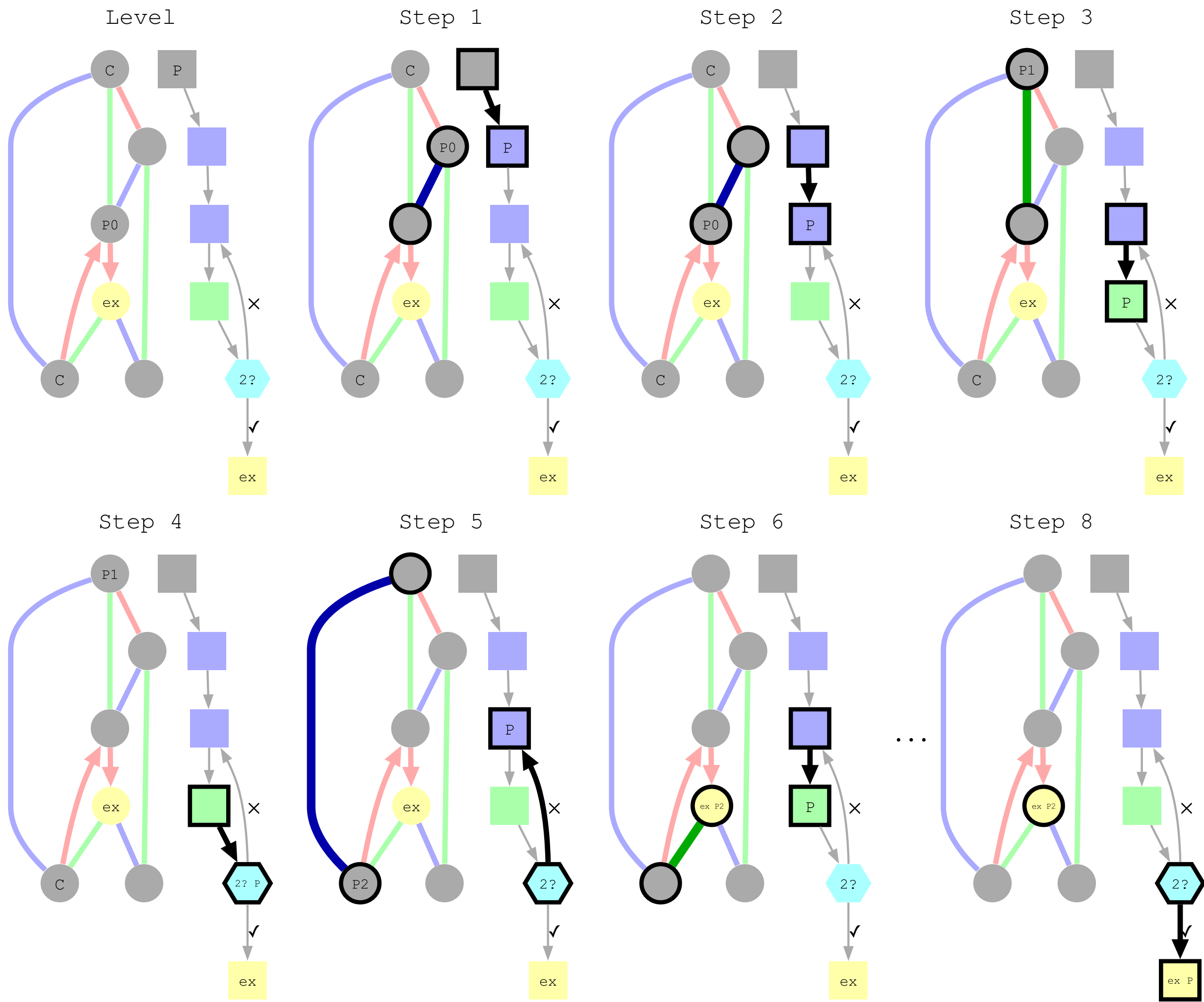

Figure 7: Sample generated level and partial playthrough for Code Master, with a branch.

Code Master

Code Master is a single-player puzzle game developed by ThinkFun, Inc (ThinkFun, Inc 2015). Each level of the game consists of both a map and a scroll. The map represents the world in which the map player moves and can collect crystals; nodes on the map are connected by red, green, or blue edges. The scroll represents a "program" that controls how the map player moves on the map. There is a collection of red, green, or blue tokens that are placed on the nodes of the scroll. The scroll player also moves along the nodes of the scroll, and the order of color tokens encountered by the scroll player on the scroll determines the color order that edges are followed on the map by the map player. Some scroll nodes can also hold "branch" tokens that cause a different scroll edge to be taken, depending on if the condition (e.g. the map player has collected exactly 2 crystals) is met or not; this causes loops in the program. The goal is to place color and branch tokens on the scroll in an order such that the map player moves through the map to the exit at the same time the exit is reached by the scroll player, potentially collecting any necessary crystals along the way.

In this work we used a simplified version of the game, with at most 2 crystals per level and 1 crystal per node, no trolls, only branching based on crystals. We also did not use self-edges on the map. To turn a generated graph into a puzzle, the color and branch tokens would be removed from the scroll.

We represent levels as a directed graph. To separate out the map part from the scroll part, there is a special root node with one child that leads to each part, conceptually dividing the nodes into two types. Map nodes have static labels for if they are ordinary (`mO`) or the exit (`mE`). They have dynamic labels for the player and how many crystals they have (`mP0`, `mP1`, `mP2`) as well as the crystals themselves (`mC`). Map edges are statically labelled with their (potentially multiple) colors (e.g., `mR`, `mGB`). Scroll nodes are dynamically labelled with the player (`sP`). They are statically labelled with either the player start, their color, or the exit (e.g. `sP`,

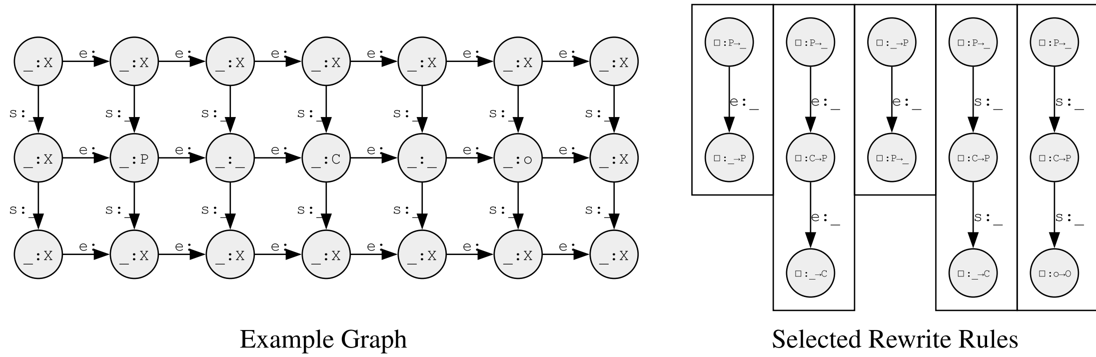

Figure 8: Example graphs and selected rewrite rules for the Sokoban application. The rewrite rules show the player moving east, pushing a crate east, moving west (backwards along an east edge), pushing a crate south, and pushing a crate onto a target south.

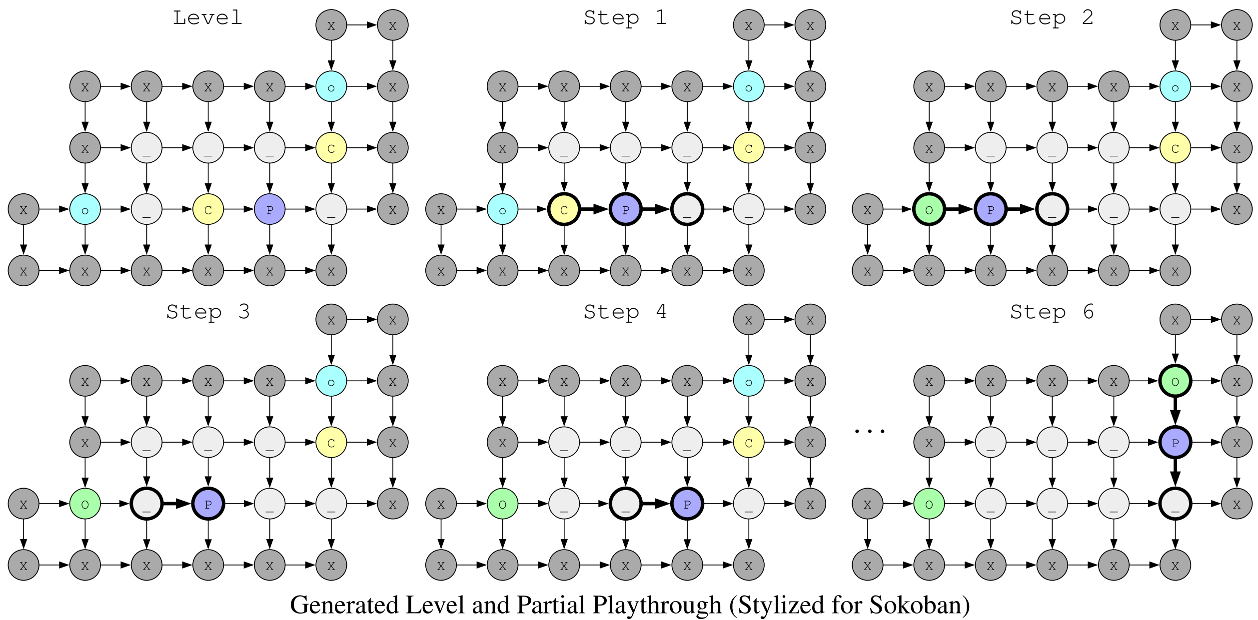

Figure 9: Sample generated level and partial playthrough for Sokoban.

`sR`, `sE`), or with a branch on exactly two crystals node (`s2`). Scroll edges are statically labeled with whether the player advances to the next scroll node (`sN`) or whether the result of a branch is true or false (`sT`, `sF`). Scroll edges are also dynamically labelled with if they have not yet been visited ($\star$).

The majority of the rewrite rules consist of paired subgraph rewrites, where the scroll player moves onto a node of a particular color, and the map player follows an edge of the matching color, picking up a crystal if there is one. Additionally, the scroll player can move onto the scroll exit if the map player is at the map exit, and the scroll player can independently move onto a branch node. Moves along scroll edges rewrite their dynamic label to mark them as visited.

For Code Master, we used a specialized edge setup *codemaster-m,s* that divides the possible edges into map and scroll edges based on up to *m* map nodes and *s* scroll nodes. An example of *codemaster-6,6* is shown in Figure 6.

We used levels 1, 2, and 23 from the game as examples. The example levels and selected rewrite rules are shown in Figure 6. Generated levels must generally have one root, map player, map exit, scroll player, scroll exit, at least four color or branch scroll nodes, and at least 2 different color scroll nodes. At the end, the map and scroll players must be at their respective exits, and all scroll edges must be visited.

When generating levels for Code Master, we generated levels both without and with a branch in the scroll. For levels without a branch, there must be no crystals and no branch nodes; levels could have between 10 and 13 nodes and 6 timesteps, using the *codemaster-6,6* edge setup. The minimum, mean, and maximum generation times were 27.6s, 28.0s (SD=0.2s), and 28.1s.

For levels with a branch, there must be 2 crystals and 1 branch node; levels could have between 12 and 15 nodes and 12 timesteps, using the *codemaster-7,7* edge setup. A generated level and playthrough is shown in Figure 7. The minimum, mean, and maximum generation times were 121.5s, 210.7s (SD=114.6s), and 398.6s.

Sokoban

This application is included as a demonstration that Stugeon-MKIV can also incorporate grid-based games, as grids can be considered a special case of graphs.

Sokoban (Thinking Rabbit 1928) is a puzzle game where the player's goal is to push crates onto targets at specific locations in the level. In this work we use a version where the player cannot walk on targets and crates cannot be pushed off targets once on them. This version was also used in Stugeon-MKIII (Cooper 2023b).

This application uses the *grid-5* edge setup. In a directed acyclic graph, nodes can have *east* (statically labeled `e`) and *south* (statically labeled `s`) edges, which among other constraints, places them on a grid. The "tiles" on the grid use dynamic labels. The player `P` can move into empty space `_` or push crates `C` into empty space or targets `o`. Crates pushed onto targets change to `O`. There are also walls `X`. The rewrite rules use the edge label to make sure that crate pushes happen along a straight line.

A simple custom level was used as an example. The example level and selected rewrite rules are shown in Figure 8. Generated levels must start with one player, two crates, and two targets, and end with no targets without crates on them. Generated levels could have between 25 and 36 nodes and 8 timesteps. A generated level and playthrough is shown in Figure 9. The minimum, mean, and maximum generation times were 2.23s, 2.32s (SD=0.11s), and 2.50s.

Discussion

We applied the approach to several different types of games, including those that directly represent the player in the level and those that don't (e.g. Lights Out). There are still several limitations and areas for future work remaining.

In this work the games represented are, essentially, single-player turn-based puzzle games, where a single choice is made at each turn. Future work may consider extending to e.g. multiplayer games where each player has a different selection of moves they can choose from, or games where different types of choices can be made.

The generated levels highly depend on the example graphs. Different example graphs would likely lead to different generated levels, and exploring this impact could be an area of future work. However, this is more generally a property of techniques that learn from examples (such as the WaveFunctionCollapse algorithm (Gumin 2016)) and not just our approach.

The levels themselves used in this work are fairly small, and the system could be more scalable as the size of the level, number of timesteps, and number of rewrite rules grows. However, there are many approaches that generate levels in a more segment-wise manner and then combine them together (Green et al. 2020; Biemer and Cooper 2022), although this is usually done in grid-based levels. It may be interesting to try approaches to improving performance that have worked in similar applications, such as iterative constrained extension (Mao and Cooper 2023). Potentially such approaches could be applied in the temporal dimension as well as spatial ones.

Some of the generated levels would still need further processing to be usable. Particularly the adventure game map would need to be converted into actual rooms, connected as they are in the graph, and that respect the requirements specified by the labels (e.g. key and locks in the right place). A number of techniques for hierarchical, multi-step, or ensemble level generation have been developed, e.g. Li et al. (2021), including those that can work directly on graphs, e.g. Dormans (2010).

Often the solutions themselves are fairly simple. This is particularly evident in the Sokoban application where crates start fairly close to their targets. One approach to address this might be to put more constraints on the starting graph (e.g. relative crate and target locations), or possibly in the form of negative example graph patterns, like the negative example patterns in Karth and Smith (2019). It may also be possible to apply constraints to the steps in the playthrough itself, which might lead to more interesting levels. We also explored changes to the rewrite rules in the case of Sokoban, i.e. the crate label itself changes to count how many times it has been pushed, and can only be pushed onto a target after a certain number of pushes, although this obfuscates the rules.

The generation approach presented also does not rule out shorter or other solutions being possible. For example, in the actual Code Master game there is generally only one way to arrange the tokens that results in a solution, whereas in our generated levels there may be several. Future work may explore ruling out alternate solutions or shortcuts, e.g. as in Smith, Butler, and Popovic (2013).

Conclusion

In this work we described and demonstrated Sturgeon-MKIV, a constraint-based system for generating graph-based levels along with playthroughs demonstrating their compatibility. The system represents game mechanics as graph label rewrite rules. The system provides flexibility beyond previously used grid-based rewrite rule mechanics.

There are number of areas for extending and improving the approach. We are interested in addressing the limitations mentioned in the discussion. In many cases there is nothing inherently 2D about the generated graphs, and thus exploring the generation of 3D graphs could be promising. We would also like to explore approaches to learning the rewrite rules themselves from example playthroughs, as well as studying how designers might author graph rewrite rules more easily.

Acknowledgments

The authors would like to thank Hudson Shaw Cooper for his help with the Code Master puzzles.

References

Adams, E.; and Dormans, J. 2012. *Game mechanics: advanced game design*. New Riders.

Alaka, S.; and Bidarra, R. 2023. Hierarchical Semantic Wave Function Collapse. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*, FDG '23, 1–10. New York, NY, USA: Association for Computing Machinery. ISBN 978-1-4503-9855-8.

Biemer, C. F.; and Cooper, S. 2022. On Linking Level Segments. In *2022 IEEE Conference on Games (CoG)*, 199–205.

Carpenter, D.; Bacher, J. T.; Crain, H.; and Martens, C. 2021. Casual creation of tile maps via authorable constraint-based generators. In *presented at 1st Workshop on Programming Languages and Interactive Entertainment*.

Cooper, S. 2022. Sturgeon: tile-based procedural level generation via learned and designed constraints. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 18(1): 26–36.

Cooper, S. 2023a. Sturgeon-GRAPH: Constrained Graph Generation from Examples. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*, FDG '23, 1–9. New York, NY, USA: Association for Computing Machinery. ISBN 978-1-4503-9855-8.

Cooper, S. 2023b. Sturgeon-MKIII: simultaneous level and example playthrough generation via constraint satisfaction with tile rewrite rules. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*, FDG '23, 1–9. New York, NY, USA: Association for Computing Machinery. ISBN 978-1-4503-9855-8.

Cooper, S.; and Balema, E. 2023. Learning constrained graph layout for content generation. In *Proceedings of the Experimental AI in Games Workshop*.

Cooper, S.; and Sarkar, A. 2020. Pathfinding Agents for Platformer Level Repair. In *Proceedings of the Experimental AI in Games Workshop*.

Cypher, A.; and Smith, D. C. 1995. KidSim: end user programming of simulations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 27–34.

de Moura, L.; and Bjørner, N. 2008. Z3: An Efficient SMT Solver. In Ramakrishnan, C. R.; and Rehof, J., eds., *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, 337–340.

Dormans, J. 2010. Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 1–8. ISBN 978-1-4503-0023-0.

Dormans, J.; and Bakkes, S. 2011. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3): 216–228.

Ehrig, H.; Pfender, M.; and Schneider, H. J. 1973. Graph-grammars: an algebraic approach. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, 167–180.

Font, J. M.; Izquierdo, R.; Manrique, D.; and Togelius, J. 2016. Constrained Level Generation through Grammar-Based Evolutionary Algorithms. In Squillero, G.; and Burelli, P., eds., *Applications of Evolutionary Computation*, Lecture Notes in Computer Science, 558–573. Cham: Springer International Publishing. ISBN 978-3-319-31204-0.

Furnas, G. W. 1991. New graphical reasoning models for understanding graphical interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 71–78.

Green, M. C.; Mugrai, L.; Khalifa, A.; and Togelius, J. 2020. Mario Level Generation from Mechanics Using Scene Stitching. *arXiv:2002.02992 [cs]*.

Gumin, M. 2016. WaveFunctionCollapse. https://github.com/mxgmn/WaveFunctionCollapse.

Gumin, M. 2022. MarkovJunior. https://github.com/mxgmn/MarkovJunior/.

Hauck, E.; and Aranha, C. 2020. Automatic Generation of Super Mario Levels via Graph Grammars. In *2020 IEEE Conference on Games (CoG)*, 297–304.

Ignatiev, A.; Morgado, A.; and Marques-Silva, J. 2018. PySAT: a Python toolkit for prototyping with SAT oracles. In *Theory and Applications of Satisfiability Testing – SAT 2018*, 428–437.

Jain, R.; Isaksen, A.; Holmgård, C.; and Togelius, J. 2016. Autoencoders for level generation, repair, and recognition. In *Proceedings of the ICCC workshop on computational creativity and games*, volume 9.

Jemmali, C.; Ithier, C.; Cooper, S.; and El-Nasr, M. S. 2020. Grammar Based Modular Level Generator for a Programming Puzzle Game. In *Proceedings of the Experimental AI in Games Workshop*, 7.

Karavolos, D.; Bouwer, A.; and Bidarra, R. 2015. Mixed-initiative design of game levels: integrating mission and space into level generation. *Proceedings of the 10th International Conference on the Foundations of Digital Games*.

Karth, I.; and Smith, A. M. 2017. WaveFunctionCollapse is constraint solving in the wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 1–10.

Karth, I.; and Smith, A. M. 2019. Addressing the fundamental tension of PCGML with discriminative learning. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, FDG '19, 1–9. New York, NY, USA: Association for Computing Machinery. ISBN 978-1-4503-7217-6.

Katz, J.; Bateni, B.; and Smith, A. 2024. You Only Randomize Once: Shaping Statistical Properties in Constraint-based PCG. In *Proceedings of the 19th International Conference on the Foundations of Digital Games*.

Kim, H.; Hahn, T.; Kim, S.; and Kang, S. 2020. Graph based wave function collapse algorithm for procedural content generation in games. *IEICE Transactions on Information and Systems*, E103.D(8): 1901–1910.

Langendam, T. S. L.; and Bidarra, R. 2022. miWFC-Designer empowerment through mixed-initiative Wave Function Collapse. In *Proceedings of the 17th International Conference on the Foundations of Digital Games*, 1–8.

Lavelle, S. 2013. PuzzleScript. https://www.puzzlescript.net/.

Li, B.; Chen, R.; Xue, Y.; Wang, R.; Li, W.; and Guzdial, M. 2021. Ensemble learning for mega man level generation. In *Proceedings of the 16th International Conference on the Foundations of Digital Games*, FDG '21, 1–9. New York, NY, USA: Association for Computing Machinery. ISBN 978-1-4503-8422-3.

Liffiton, M. H.; and Maglalang, J. C. 2012. A cardinality solver: more expressive constraints for free. In *Theory and Applications of Satisfiability Testing – SAT 2012*, 485–486.

Linden, R.; Lopes, R.; and Bidarra, R. 2013. Designing procedurally generated levels. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Londoño, S.; and Missura, O. 2015. Graph Grammars for Super Mario Bros Levels. In *Sixth FDG Workshop on Procedural Content Generation*.

Madkour, A.; Marsella, S.; Harteveld, C.; Seif El-Nasr, M.; and van de Meent, J.-W. 2021. Guiding Generative Graph Grammars of Dungeon Mission Graphs via Examples. In *Experimental AI in Games Workshop*.

Mao, H.; and Cooper, S. 2023. Segment-wise level generation using iterative constrained extension. In *2023 IEEE Conference on Games (CoG)*, 1–7.

Martens, C. 2015. Ceptre: a language for modeling generative interactive systems. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 11(1): 51–57.

Martens, C.; Card, A.; Crain, H.; and Khatri, A. 2024. Modeling game mechanics with Ceptre. *IEEE Transactions on Games*, 16(2): 431–444.

Merrell, P. 2023. Example-Based Procedural Modeling Using Graph Grammars. *ACM Transactions on Graphics*, 42(4): 60:1–60:16.

Nelson, M. J.; and Smith, A. M. 2016. ASP with applications to mazes and levels. In Shaker, N.; Togelius, J.; and Nelson, M. J., eds., *Procedural Content Generation in Games*, 143–157. Springer International Publishing.

Nie, Y.; Zheng, S.; Zhuang, Z.; and Togelius, J. 2024. Nested Wave Function Collapse enables large-scale content generation. *IEEE Transactions on Games*, 1–11.

Repenning, A. 1995. Bending the rules: steps toward semantically enriched graphical rewrite rules. In *Proceedings of Symposium on Visual Languages*, 226–233.

Reuter, C.; Göbel, S.; and Steinmetz, R. 2015. Detecting structural errors in scene-based Multiplayer Games using automatically generated Petri Nets. In *Proceedings of the 10th International Conference on the Foundations of Digital Games*.

Shaker, N.; Togelius, J.; and Nelson, M. J. 2016. *Procedural Content Generation in Games*. Springer International Publishing.

Shu, T.; Wang, Z.; Liu, J.; and Yao, X. 2020. A novel cnet-assisted evolutionary level repairer and its applications to Super Mario Bros. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, 1–10. IEEE.

Smith, A. M.; Andersen, E.; Mateas, M.; and Popović, Z. 2012. A Case Study of Expressively Constrainable Level Design Automation Tools for a Puzzle Game. In *Proceedings of the International Conference on the Foundations of Digital Games*, 156–163. ISBN 978-1-4503-1333-9.

Smith, A. M.; Butler, E.; and Popovic, Z. 2013. Quantifying over Play: Constraining Undesirable Solutions in Puzzle Design. In *Proceedings of the 8th International Conference on Foundations of Digital Games*, 221–228.

Snodgrass, S.; and Ontañón, S. 2016. Controllable procedural content generation via constrained multi-dimensional Markov chain sampling. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 780–786.

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.

ThinkFun, Inc. 2015. Code Master. https://www.thinkfun.com/products/code-master/. Accessed: 2024-06-24.

Thinking Rabbit. 1928. Sokoban. Game.

Tiger Electronics. 1995. Lights Out.

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.

Tows, G. S. 2009. Imagegram: image grammar for procedural generation.

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*, 7:1–7:10. ISBN 978-1-4503-5319-9.

van Rozen, R.; and Heijn, Q. 2018. Measuring Quality of Grammars for Procedural Level Generation. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, 1–8. ISBN 978-1-4503-6571-0.

Viana, B. M. F.; Pereira, L. T.; Toledo, C. F. M.; dos Santos, S. R.; and Maia, S. M. D. M. 2022. Feasible–infeasible two-population genetic algorithm to evolve dungeon levels with dependencies in barrier mechanics. *Applied Soft Computing*, 119: 108586.

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

Wright, T. 2006. PatternProgrammer: yet another rule-based programming environment for children. In *Proceedings of the 7th Australasian User interface conference - Volume 50*, 91–96.

Zafar, A.; Mujtaba, H.; and Beg, M. O. 2020. Search-based procedural content generation for GVG-LG. *Applied Soft Computing*, 86: 105909.

Zhang, H.; Fontaine, M.; Hoover, A.; Togelius, J.; Dilkina, B.; and Nikolaidis, S. 2020. Video game level repair via mixed integer linear programming. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 16(1): 151–158.

Zhou, J.; Martens, C.; and Cooper, S. 2024. Authoring Games with Tile Rewrite Rule Behavior Trees. In *Proceedings of the 19th International Conference on the Foundations of Digital Games*.