Sturgeon-MKIII: Simultaneous Level and Example Playthrough Generation via Constraint Satisfaction with Tile Rewrite Rules

Seth Cooper
se.cooper@northeastern.edu
Northeastern University
Boston, USA

ABSTRACT

Completability is a key aspect of procedural level generation. In this work, we present a constraint-based approach to level generation for 2D tile-based games that simultaneously generates a level and an example playthrough of the level demonstrating its completability. The approach represents game mechanics as tile rewrite rules, which allows a variety of games and mechanics (beyond simple pathfinding) to be incorporated. The mechanics are represented as constraints in the same problem along with the constraints used to generate the level itself. Thus, the solution to the constraint problem contains both a level and a playthrough of the level. We demonstrate the flexibilty of the system and of tile rewrite rules in several applications, including lock-and-key dungeons, platformers, puzzles, and match-three style games.

CCS CONCEPTS

• **Human-centered computing**;

KEYWORDS

procedural content generation, constraints, tile rewrite rules

ACM Reference Format:
Seth Cooper. 2023. Sturgeon-MKIII: Simultaneous Level and Example Playthrough Generation via Constraint Satisfaction with Tile Rewrite Rules. In *Foundations of Digital Games 2023 (FDG 2023), April 12–14, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3582437.3587205

1 INTRODUCTION

When procedurally generating levels, it should be possible for players to complete generated levels. In this work, we present a constraint-based approach to level generation for 2D tile-based games that ensures generated levels are completable and supports a variety of game mechanics. Along with level design constraints, the system encodes the game mechanics themselves as constraints, and thus by solving a single constraint problem, simultaneously generates a level and an example playthrough of the level demonstrating its completability.

The system extends the existing Sturgeon system [3] by adding support for *timesteps* that can represent gameplay as changes to the tiles over time. The system represents game mechanics as tile *rewrite rules*. The rewrite rules in this work are directly inspired by Maxim Gumin's recent work on MarkovJunior [10] (the system's name began as an abbreviated form of MarkovIII). However, the tile rewrite rules are similar to "graphical rewrite rules" used in some visual programming environments [5, 21, 31].

Rewrite rules represent what kind of moves can be made in the game. For example, if an `@` tile represents the player, and a `-` tile represents an empty space, allowing `@-` to be rewritten with `-@` could represent the player moving into an empty space. Further, if an `&` tile is the player holding a key, and a `%` is a key, a rule rewriting `@%` with `-&` could be the character picking up the key. More complex games and mechanics can be built up this way.

We describe the system and how it organizes rewrite rules to include mechanics for players, interactive objects, enemies, and simple physics. We show how the system can reproduce and extend an example from MarkovJunior. We demonstrate applications in a variety of games, including lock-and-key dungeons, platformers, puzzles, and match-three style games. We provide some timing information to show how long it takes to generate levels with playthroughts of the complexity shown.

2 RELATED WORK

In research in procedural content generation [22], there have been many approaches taken towards ensuring that generated levels are completable.

One technique is a "generate-and-test" approach, where potentially uncompletable levels (or parts of levels) are generated, and then are evaluated or filtered for completability in a postprocessing step. Completability testing is often done by a gameplaying agent [1, 25, 30]. Search-based approaches [27] to level generation can incorporate completability into their objective functions, or employ approaches such as Feasible–Infeasible Two-Population to improve search for completable levels [29, 32].

Rather than only testing generated levels for completability, some work attempts to repair levels that are not completable to make them so. Zhang et al. [33] used mixed integer linear programming to repair dungeon levels and Cooper and Sarkar [4] used pathfinding agents that could modify platformer levels to make them completable. Jain et al. [13] used autoencoders that could attempt to repair platformer levels.

Some grammar-based approaches take a more constructive approach, creating grammars that ensure that generated levels are completable. Font et al. [8] use grammars that ensured completable

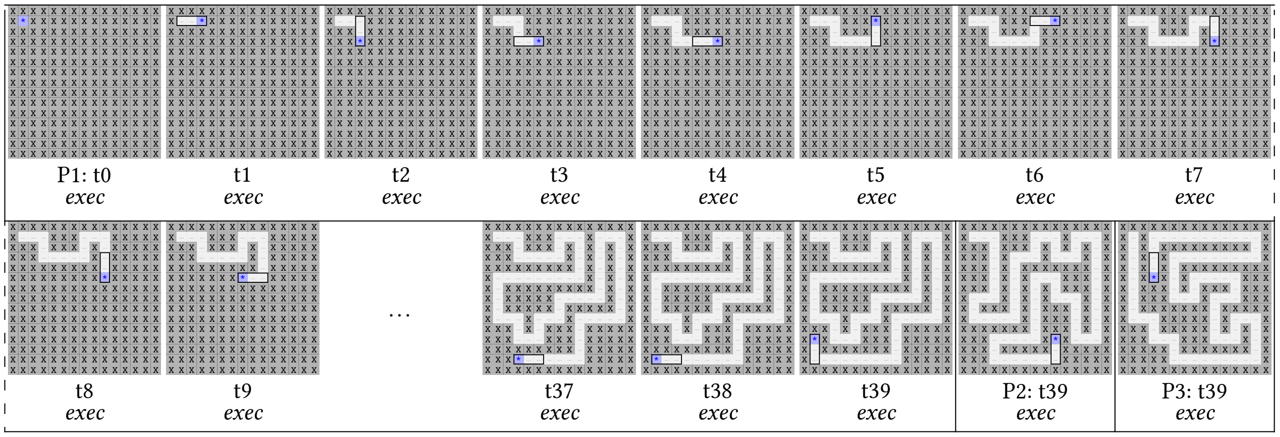

Figure 1: Example playthroughs and two additional final timesteps generated by `mkjr-walk`. Outlined tiles show tiles rewritten from previous timestep.

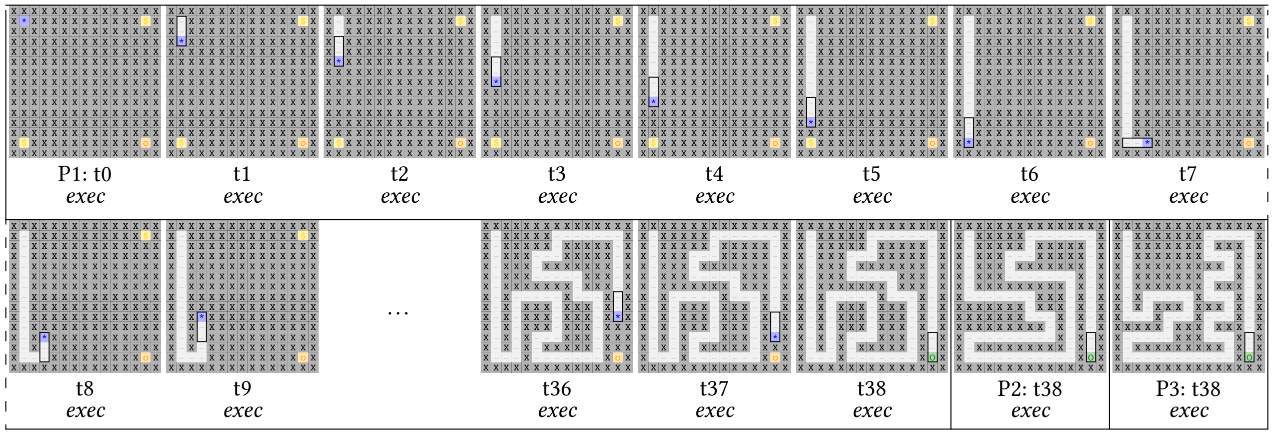

Figure 2: Example playthroughs and two additional final timesteps generated by `mkjr-walk-thru`. Outlined tiles show tiles rewritten from previous timestep.

generated dungeons. Dormans and Bakkes [7] used transformation rules on keys and locks in mission graphs that ensured missions stayed completeable.

More closely related to our work are techniques that generate levels using constraints and encode completability as additional constraints. However, this is often limited to pathfinding. For example, Nelson and Smith [18] encode constraints that simple mazes are solvable. The original Sturgeon system [3] supports constraints that there is a path between the start and goal of generated levels for a variety of player movement types. In an educational puzzle game, Smith at al. applied custom approaches to constraining solvable puzzle generation [23].

Graphical and pixel rewrite rules have been used to represent changes in simulations, games, and images. For example, Furnas' BITPICT system [9] used rewrite rules to edit bitmap images. Systems such as Agentsheets [21], Stagecast Creator/Cocoa/KidSim [5], and PatternProgrammer [31] have incorporated graphical rewrite rules, often aiming to make programming education more approachable. The PuzzleScript [15] game engine incorporates tile rewrite rules, and has been used in work on generating games and levels [14, 17].

Tile rewrite rules have also been used for generation of the levels themselves; notable Gumin's MarkovJunior work [10]. Van Rozen and Heijn also used a 'grammar' of tile rewrite rules to generate dungeon rooms [28]. This use of tile rewrite rules for level generation can be viewed similarly to generating levels via grammars [7] or transformations [6].

3 SYSTEM OVERVIEW

The goal of the system, Sturgeon-MKIII, is to generate levels with example playthroughs via constraint satisfaction problems. Mechanics are represented as rewrite rules, specifying how tiles can change from one timestep to the next. They consist of a left-hand side (LHS) and a right-hand side (RHS). If the LHS of a rule is present in one timestep, it can possibly be rewritten with the RHS of the rule in the next timestep — although different groupings and orderings (discussed below) impact how rules can be applied.

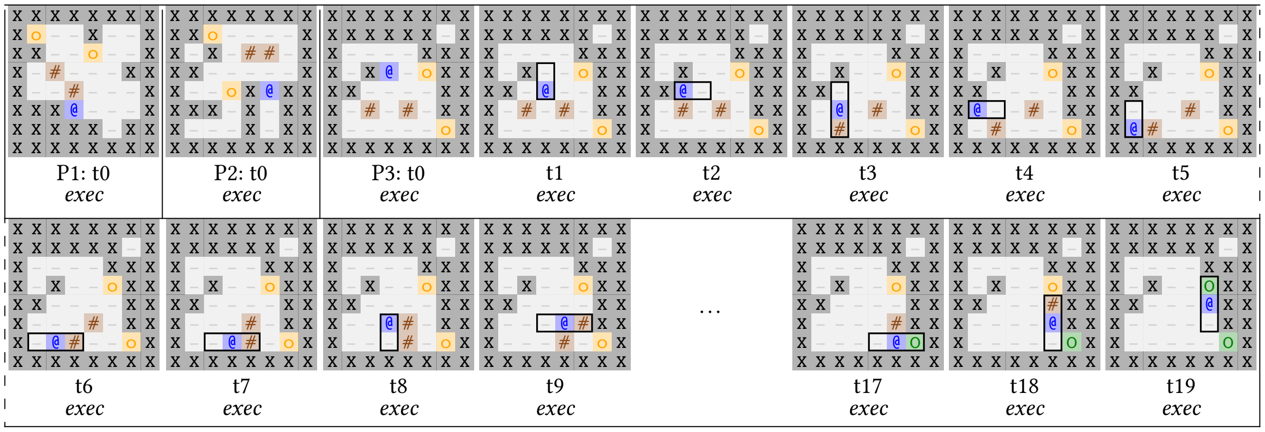

Figure 3: Example levels and playthroughs generated by soko. The first three show initial timesteps for a playthrough; the third shows a partial playthrough. Outlined tiles show tiles rewritten from previous timestep.

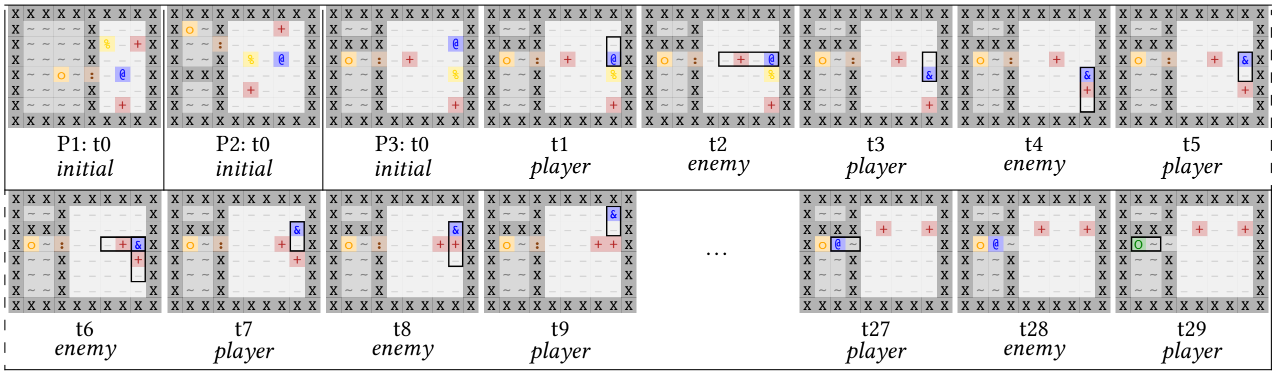

Figure 4: Example levels and playthroughs generated by lock. The first three show initial timesteps for a playthrough; the third shows a partial playthrough. Outlined tiles show tiles rewritten from previous timestep.

Each timestep has its own set of variables for each location and possible tile at that location. Constraints on the initial design of the level are applied to the first timestep, and constraints on the level being completed can be applied to the final timestep. The rewrite rules are translated into constraints on how the tiles can change from one timestep to the next; any tiles that are not part of a change remain the same. In this way, the entirety of the level generation, how the tiles can move to play the level, and the level being completed by the end, are encoded into a single constraint problem. The solution is a series of levels, the first of which is the level itself, and the rest provide an example playthrough of that level being completed. Note that the playthough does not have to be the shortest or optimal in some way.

In this work, rewrite rules are limited to being zero-dimensional (i.e. rewriting of individual tiles) or one-dimensional (i.e. rewriting of tiles along a line going north, south, east, or west, represented by N, S, E, or W). For example, a one-dimensional rule with a LHS of `@ -` and a RHS of `- @`, that can go in all four directions (NSEW), could represent the player moving their character to a neighboring blank space. A one-dimensional rule with a LHS of `7 -` and a RHS of `- 7` going only EW could represent allowing a player to only move a piece to the side, while a rule with the same LHS `7 -` and RHS `- 7`, that goes only S could represent pieces falling. A zero-dimensional rule with LHS of `-` and RHS of `1` could represent the player placing a piece.

The system can organize individual rewrite rules into *rule groups*, which determine how a set of rules should be applied during a timestep. The current rule group types are:

- *choice*: Exactly one rule from the group must be applied. This is useful for representing player moves, where the player can decide from different options.
- *all*: All rules that can be applied must be. This is useful for representing game systems such as enemies or physics that can all act at once.

Further, rule groups can be organized into *rule orders*, which specify which rule groups should be used over multiple timesteps. The current rule order types are:

- *single*: There is just one rule group.
- *sequential*: There are multiple rule groups, and they are applied sequentially on each timestep. The first group is applied on the first timestep, the second group on the second timestep, and so on. Once there are no more groups, they

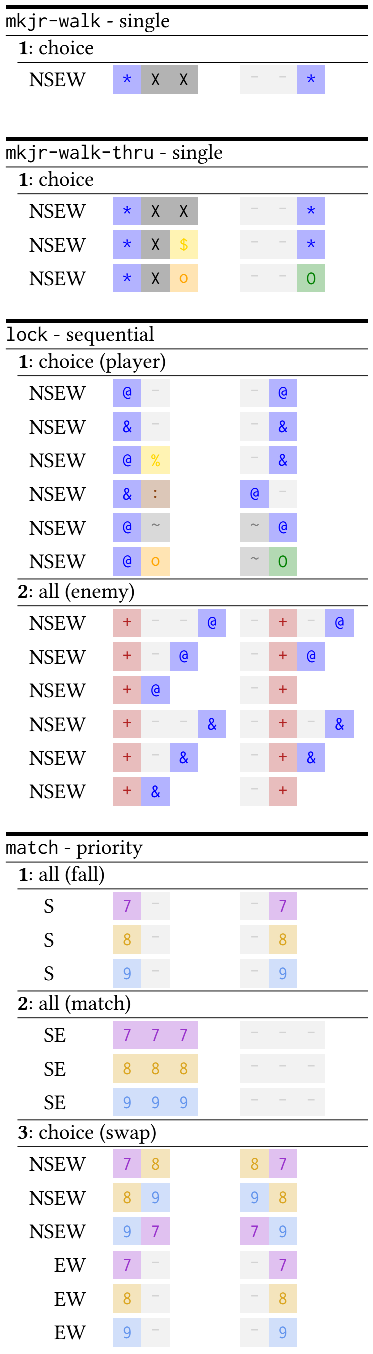

Figure 5: Replacement rules (directions, LHS, and RHS) for selected examples. In `mkjr-walk` and `mkjr-walk-thru`, the *choice* rules represent the `*` moving while clearing a path. In `lock`, the groups represent the player *choice* of actions and *all* the enemies moving. Using *sequential* ordering the player and enemies alternate turns. In `match`, groups represent *all* pieces falling, *all* matches being cleared, and the player's *choice* of move. With *priority* order, for example, the player cannot move while pieces are falling.

App.	Med.	Max.	App.	Med.	Max.
`mkjr-walk`	3	8	`plat`	14	23
`mkjr-walk-thru`	18	59	`vvv`	8	12
`soko`	4	9	`match`	89	173
`fill`	12	33	`doku`	116	454
`lock`	18	67			

Table 1: Level with playthrough generation times (in seconds). Median and maximum times given for 25 of each application.

are started over from the beginning. For example, this can allow the player and enemies to alternate moves.

- *priority*: There are multiple rule groups, and only the rule group with the highest priority that can be applied at each timestep is. If the first rule group can be applied it is, and none of the following are; if it cannot, and the second rule group can be, it is applied and none of the following are, and so on. For example, this can prevent the player from moving until some other sequence of changes is complete.

In addition to rewrite rules, the system also takes the width and height of the level to be generated, as well as the maximum number of timesteps allowed for the playthrough.

It is possible to optionally allow *early termination* of rules. This allows changes to stop being applied before the final timestep; however, once a playthough has terminated no further changes can be applied.

For the constraint problem setup, in addition to using standard Sturgeon tile variables and some standard constraints in timestep 0 (which are described in prior work [3], along with Sturgeon's mid-level solver API), the constraint problem is extended with the following variables (using Sturgeon's MakeVar):

- A variable is made for each possible tile, at each location, at each timestep.
- If the rule group ordering is *priority*, a variable is made for each priority group for each timestep (other than the last) to determine which group is used in that timestep.
- If early termination is allowed, a variable for each timestep (other than the first) that it is terminal (i.e. the playthrough has ended and no more changes are made).

Sturgeon's MakeConj is used to make conjunctions of variables:

- For each timestep (except the last), for each rule, a conjunction is made of the LHS tile variables (representing that the LHS is present).
- Across each timestep, for each rule, a conjunction is made of both the LHS tile variables in a timestep and the RHS tile variables in the next timestep (representing that the rule has been applied).

Sturgeon's CnstrCount and CnstrImpliesDisj are used to add constraints. If using *single* or *sequential* group ordering, only the relevant rewrite rules are available to be applied each timestep. If using *priority* group ordering, all rules are available, and the priority variables are used to determine which group is used. Constraints are applied at each timestep (except the first or last where not needed):

- Exactly one tile at each location is True.

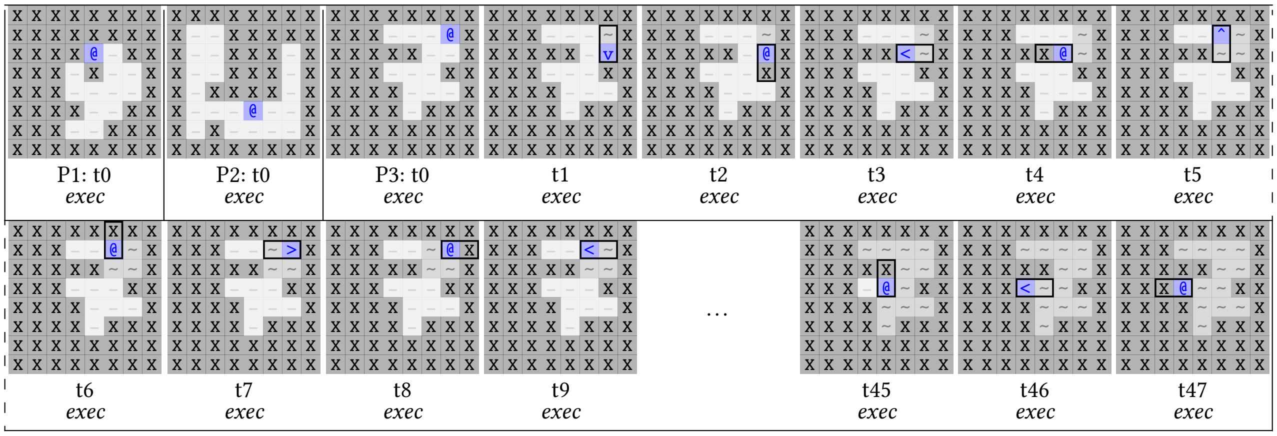

Figure 6: Example levels and playthroughs generated by `fill`. The first three show initial timesteps for a playthrough; the third shows a partial playthrough. Outlined tiles show tiles rewritten from previous timestep.

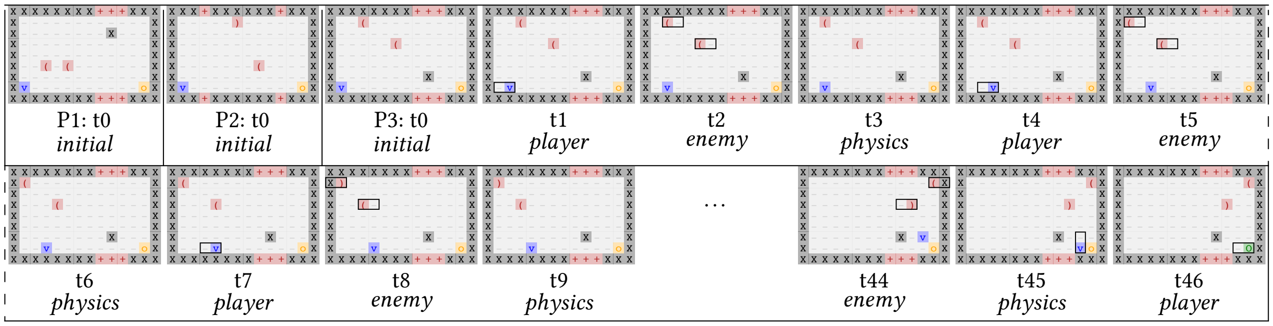

Figure 7: Example levels and playthroughs generated by `vvv`. The first three show initial timesteps for a playthrough; the third shows a partial playthrough. Outlined tiles show tiles rewritten from previous timestep.

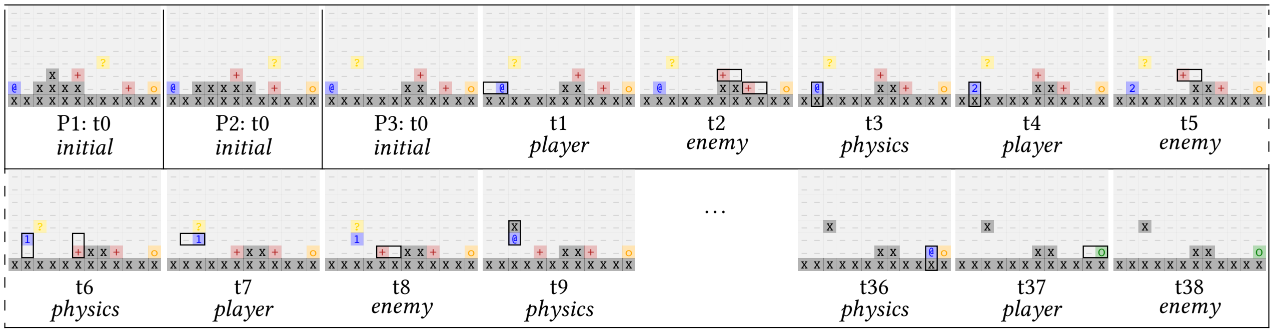

Figure 8: Example levels and playthroughs generated by `plat`. The first three show initial timesteps for a playthrough; the third shows a partial playthrough. Outlined tiles show tiles rewritten from previous timestep.

- If early termination is allowed, a timestep being terminal implies the following timestep is also terminal.
- If using *priority* group ordering, a priority group has been applied if and only if one if its rules has been applied.
- If using *priority* group ordering, exactly one of any priority group being applied, or the next timestep being terminal, must be True.
- For a *choice* rule group, exactly one of any rule being applied, the next timestep being terminal, or a higher-priority group being applied must be True.

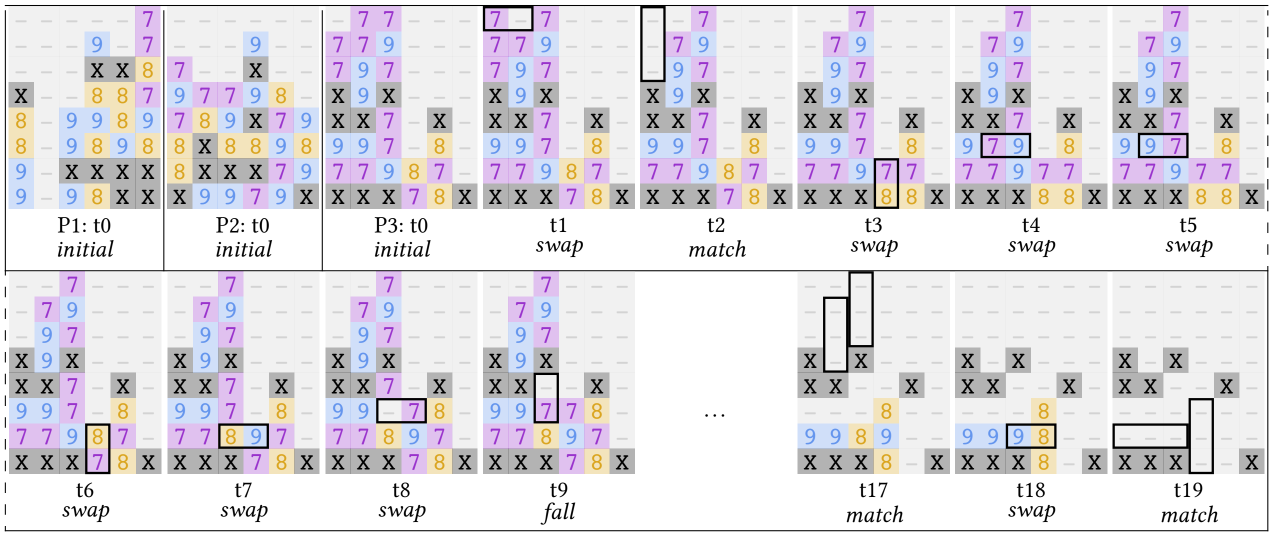

Figure 9: Example levels and playthroughs generated by `match`. The first three show initial timesteps for a playthrough; the third shows a partial playthrough. Outlined tiles show tiles rewritten from previous timestep.

- For an *all* rule group, for every change that can be applied (determined using the LHS pattern conjunctions), it must be applied, or the next timestep must be terminal, or a higher-priority group must be applied; and, at most one of the change being applied, the next timestep being terminal, or a higher-priority group must being applied can be True.
- Every tile at every location must either keep its value in the next timestep or be part of a change.

Individual applications (described below) can apply their own additional constraints:

- The "level design" constraints from Sturgeon can be applied in timestep 0. However, in this work we do not use Sturgeon's default tile count or reachability constraints to generate levels, and only some applications discussed use patterns learned from example levels.
- Applications can apply additional constraints on level generation in the first timestep, as well as other constraints throughout timesteps.
- Applications can add constraints that the level is solved in the last timestep.

Regarding these additional constraints, for example, a specific tile at a specific location and timestep can be required by using CnstrCount to constraint its varible to True, or a certain number (or range) of a specific tile begin present in a timestep using CnstrCount on all the variables for that type of tile in the timestep. Requiring distances between types of tiles can be accomplished by using CnstrImpliesDisj to constrain a tile being present at a location to imply another type of tile is not present at all nearby locations. Requiring a tile to "move" uses CnstrImpliesDisj to make a tile being present at a location in the first timestep imply that is is not present at that location in some subsequent timestep.

4 APPLICATIONS

Here we describe applications of the system to several different games. The games are meant to be small examples that explore some of the breadth of mechanics that are possible to represent and solve using rewrite rules. The first two applications emulate and extend MarkovJunior's self-avoiding walk example.

For each application, 25 levels with playthroughs were generated. As all constraints were hard, we used Sturgeon's low-level portfolio solver consisting of three instances of PySAT's [12] MiniCard [16] solver. Timing information is given in Table 1. Example levels and playthroughs in figures were selected for demonstrating the output of the systems. Generation parameters were selected to be able to run in a short time, generally less then a few minutes. Full rewrite rules and generated examples, including playthrough animations, are available at https://osf.io/sbew9. Selected example rewrite rules are shown in Figure 5.

— `mkjr-walk`: This application is based on the self-avoiding walk example from MarkovJunior [10]. This illustrative application does not produce a game level, but just a random walk. The levels are 15×15 with 40 timesteps, with no early termination. The level is constrained to start completely solid except for one location with a "player" `*`, which has the choice to move in any direction. There is one *single* rewrite rule group (it is *choice*, but there is only one rule) allowing the player to move. Generated examples are given in Figure 1.

— `mkjr-walk-thru`: This application builds on the `mkjr-walk` application, but with additional tiles and *choice* rules (in the *single* group) for tiles `$` that the walk must pass through (placed in the other corners) and a tile `o` it must end at (placed opposite the start). In the final timestep these must be removed. This demonstrates controllabilility over the walk generated. Generated examples are given in Figure 2.

— `soko`: An application inspired by the game Sokoban [26]. MarkovJunior [10] also presents a Sokoban-like example. The player `@` has to push crates `#` around the level into slots `o`. The levels

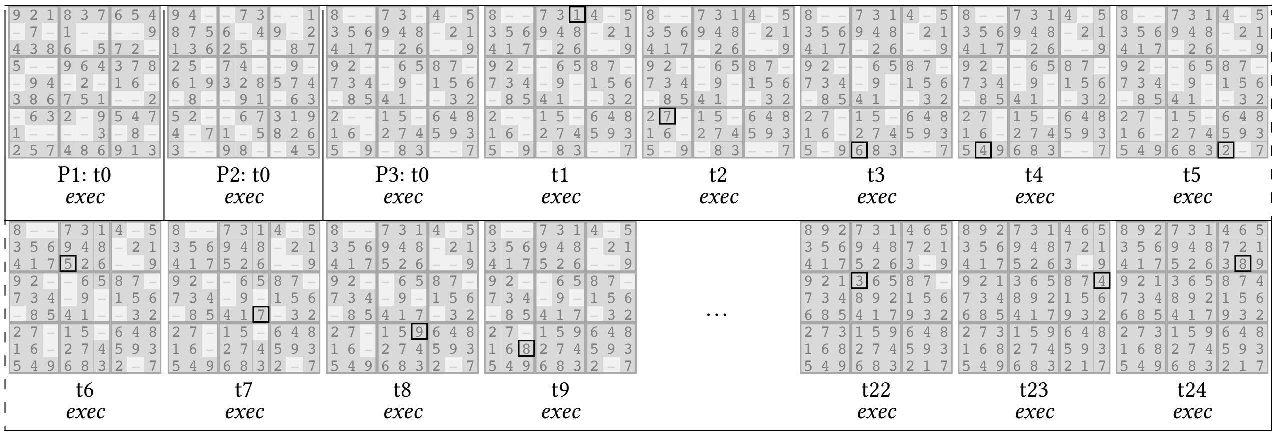

Figure 10: Example levels and playthroughs generated by doku. The first three show initial timesteps for a playthrough; the third shows a partial playthrough. Outlined tiles show tiles rewritten from previous timestep.

are 8 × 8 with 20 timesteps and early termination. The generated level must contain one player, two crates, and two slots. There must also be a solid border around the level. There are additional constraints to have the area around the crates clear and prevent the from starting too close to the slots. There must be no crates left at the end. There are is a *single choice* rule group for the player to move, push crates, and push crates into slots. Generated examples are given in Figure 3.

— `lock`: An application of a top-down dungeon-style game. The player `@` has to avoid enemies `+`, collect a key `%` to open a door `:`, and reach the exit `o`. The lock-and-key mechanic is accomplished by changing the player's tile (to `&`) when they collect the key, and only allowing the door to be opened by the player tile with the key. The levels are 8 × 10 with 31 timesteps and early termination. Levels have patterns learned from an example; different floor tiles are used to make sure the player and exit start in different rooms. The level must have one player, exit, key, door, and two enemies. Enemies are constrained to start near other tiles of interest and move during the playthrough (so they are not placed off by themselves), the level must have be somewhat solid. The exit must be entered at the end. There are two *sequential* rule groups, one *choice* for the player and one *all* for the enemies. Generated examples are given in Figure 4.

— `fill`: A top-down game where the player slides until they hit a wall (e.g. Tomb of the Mask [11]). As the player moves they change the floor tile from an 'empty' tile to a 'filled' tile. A different tile is used for the player depending on the direction they are going. The levels are 8 × 8 with 50 timesteps and early termination. There must be one player, no filled tiles, a solid border, and some empty floor tiles. At the end there must be no empty floor tiles and the player. There is a *single choice* rule group for player movement and filling. Generated examples are given in Figure 6.

— `vvv`: A gravity-switching side-view game inspired by VVVVVV [2]. The player can move left and right and switch gravity when standing. They must avoid a patrolling enemy and spikes and reach the exit. A different tile is used for the player depending on the direction of gravity. The levels are 9 × 14 with 47 timesteps and early termination. Levels have patterns learned from an example. There must be one player and exit on opposite sides. There must be two enemies and at least four spikes. The exit must be entered at the end. There are three *sequential* rule groups, one *choice* for the player, one *all* for the enemies, and one *all* for the "physics" of the player moving due to gravity. Generated examples are given in Figure 7.

— `plat`: A platform-style game inspired by Super Mario Bros. [19], with enemies the player can defeat from above and bonus blocks they can collect by hitting from below. Jumps are accomplished by using a different tile for the player depending on how much upward motion remains in the jump. The levels are 8 × 12 with 40 timesteps and early termination. Levels have patterns learned from an example. There must be one player and one exit on opposite sides, and two enemies and one bonus block. The exit must be entered at the end, and the enemies defeated and bonus collected. There are three *sequential* rule groups, one *choice* for the player, one *all* for the enemies, and one *all* for the "physics" of the player moving due to jumps or gravity. Generated examples are given in Figure 8.

— `match`: A match-three style game where the player can swap tiles around, groups of three matching tiles are removed, and tiles with nothing under them fall. The levels are 8 × 6 with 20 timesteps and early termination. The level has constraints such as the number of blanks and tiles of each type, and should not start with three in a row. All matchable tiles must be cleared at the end. There are three *priority* rule groups, in order of priority: one *all* for tiles falling, one *all* for removing matches, and one *choice* for the player. Generated examples are given in Figure 9.

— `doku`: The game Sudoku, where the numbers 1 through 9 must be placed in a grid so each row, column, and 3 × 3 box contains all nine numbers. However, this example is meant to generate "easy" Sudoku puzzles, where it is possible to solve the puzzle by only making moves where the move is fully determined by a missing number in a box having only one possible location, based on the other rows and columns in that box being ruled out. It takes advantage of the flexibility of the system to add additional variables and constraints on each move made to do this. The levels are 9×9 with 25 timesteps; and early termination is not allowed. There are constraints on the symmetry and number of blanks in the initial level. Each timestep along the playthrough must be valid, and the grid must be filled in correctly at the end. There is a *single choice* rule group for placing numbers. Generated examples are given in Figure 10.

5 DISCUSSION

Regarding timing and scalability, in this work we presented relatively small levels with few timesteps that could be generated fairly quickly. For example,doku puzzles had only 25 blanks. However, this may suffice for puzzle games such as soko. The system may work best for smaller puzzle games with short, but involved, solutions. Scalability to larger problems may require future work.

Some of the applications (e.g. plat and vvv) had relatively low variety in their generated levels. This may be a consequence of the example levels and patterns used to generate them, and it is possible more extensive training examples, or more flexible patterns, would expand the potential levels.

During the development process, we found that the solver would often prefer to generate levels with very short, simple solutions. Thus several of the examples have constraints on, for example, placement of tiles in the initial timestep, to try to prevent these types of levels. More general approaches to preventing undesirable solutions [24] would be useful.

6 CONCLUSION

In this work, we presented a constraint-based system for generating 2D tile-based levels and playthroughs in a variety of games. The system models mechanics as tile rewrite rules and encodes them into constraints in the same problem as level generation. With this approach, generated levels are completable.

There is also more to explore with using rewrite rules as game mechanics. We would like to explore (graphical) tools for authoring and games whose mechanics are defined by rewrite rules, with support from a solver that could, for example, be used to check the completability of hand-authored levels.

In this work the rewrite rule structure was limited, which in turn limited the complexity of the games. We would like to explore more flexible rewrite rules, such as 2D rules or the use of wildcards, to allow creation of more complex games. It would also be interesting to explore such a system in 3D using voxels.

It might be promising to explore constraint-based approaches to level generation with more continuous physics. We note that although a tile-and-turn based platformer may seem unusual, there do exist some games in this style, such as Turn Undead [20].

Since the system generates an example playthrough of the level, it may be useful for tutorial or training purposes. For example, the playthrough could be used in-game to demonstrate to a player how to solve the level if they get stuck.

ACKNOWLEDGMENTS

We would like to thank Colan Biemer for feedback.

REFERENCES

[1] Colan F. Biemer and Seth Cooper. 2022. On linking level segments. In *2022 IEEE Conference on Games (CoG)*. 199–205.
[2] Terry Cavanagh. 2010. VVVVVV. Game [PC].
[3] Seth Cooper. 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 (2022), 26–36.
[4] Seth Cooper and Anurag Sarkar. 2020. Pathfinding Agents for Platformer Level Repair. In *Proceedings of the Experimental AI in Games Workshop*.
[5] Allen Cypher and David Canfield Smith. 1995. KidSim: end user programming of simulations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 27–34.
[6] Joris Dormans. 2011. Level design as model transformation: a strategy for automated content generation. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. 1–8.
[7] Joris Dormans and Sander Bakkes. 2011. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (Sept. 2011), 216–228.
[8] Jose M. Font, Roberto Izquierdo, Daniel Manrique, and Julian Togelius. 2016. Constrained level generation through grammar-based evolutionary algorithms. In *Applications of Evolutionary Computation (Lecture Notes in Computer Science)*, Giovanni Squillero and Paolo Burelli (Eds.). 558–573.
[9] George W. Furnas. 1991. New graphical reasoning models for understanding graphical interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 71–78.
[10] Maxim Gumin. 2022. MarkovJunior. https://github.com/mxgmn/MarkovJunior/.
[11] Happymagenta. 2016. Tomb of the Mask. Game [iPhone].
[12] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. 2018. PySAT: a Python toolkit for prototyping with SAT oracles. In *Theory and Applications of Satisfiability Testing – SAT 2018*. 428–437.
[13] Rishabh Jain, Aaron Isaksen, Christoffer Holmgård, and Julian Togelius. 2016. Autoencoders for level generation, repair, and recognition. In *Proceedings of the ICCC workshop on computational creativity and games*, Vol. 9.
[14] Ahmed Khalifa and Magda Fayek. 2015. Automatic puzzle level generation: A general approach using a description language. In *Computational Creativity and Games Workshop*.
[15] Stephen Lavelle. 2013. PuzzleScript. https://www.puzzlescript.net/.
[16] Mark H. Liffiton and Jordyn C. Maglalang. 2012. A cardinality solver: more expressive constraints for free. In *Theory and Applications of Satisfiability Testing – SAT 2012*. 485–486.
[17] Chong-U. Lim and D. Fox Harrell. 2014. An Approach to General Videogame Evaluation and Automatic Generation Using a Description Language. In *2014 IEEE Conference on Computational Intelligence and Games*. 1–8.
[18] Mark J. Nelson and Adam M. Smith. 2016. ASP with applications to mazes and levels. In *Procedural Content Generation in Games*, Noor Shaker, Julian Togelius, and Mark J. Nelson (Eds.). Springer International Publishing, 143–157.
[19] Nintendo. 1985. Super Mario Bros. Game [NES].
[20] Nitrome. 2017. Turn Undead. Game [iPhone].
[21] Alexander Repenning. 1995. Bending the rules: steps toward semantically enriched graphical rewrite rules. In *Proceedings of Symposium on Visual Languages*. 226–233.
[22] Noor Shaker, Julian Togelius, and Mark J. Nelson. 2016. *Procedural Content Generation in Games*. Springer International Publishing.
[23] Adam M. Smith, Erik Andersen, Michael Mateas, and Zoran Popović. 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.
[24] Adam M. Smith, Eric Butler, and Zoran Popovic. 2013. Quantifying over Play: Constraining Undesirable Solutions in Puzzle Design. In *Proceedings of the 8th International Conference on Foundations of Digital Games*. 221–228.
[25] Sam Snodgrass and Santiago Ontañón. 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.
[26] Thinking Rabbit. 1928. Sokoban. Game.
[27] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. 2011. Search-based procedural content generation: a taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 172–186.
[28] Riemer van Rozen and Quinten Heijn. 2018. Measuring quality of grammars for procedural level generation. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*. 1–8.
[29] Breno M. F. Viana, Leonardo T. Pereira, Claudio F. M. Toledo, Selan R. dos Santos, and Silvia M. D. M. Maia. 2022. Feasible–infeasible two-population genetic algorithm to evolve dungeon levels with dependencies in barrier mechanics. *Applied Soft Computing* 119 (April 2022), 108586.
[30] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M Lucas, Adam Smith, and Sebastian Risi. 2018. Evolving Mario levels in the latent space of a deep convolutional

generative adversarial network. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 221–228.

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

[32] Adeel Zafar, Hasan Mujtaba, and Mirza Omer Beg. 2020. Search-based procedural content generation for GVG-LG. *Applied Soft Computing* 86 (Jan. 2020), 105909.

[33] Hejia Zhang, Matthew Fontaine, Amy Hoover, Julian Togelius, Bistra Dilkina, and Stefanos Nikolaidis. 2020. Video game level repair via mixed integer linear programming. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 16, 1 (Oct. 2020), 151–158.