Proceedings of the Eighteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment
(AIIDE 2022)

Sturgeon: Tile-Based Procedural Level Generation via Learned and Designed Constraints

Seth Cooper
Northeastern University
se.cooper@northeastern.edu

Abstract

This work describes Sturgeon, a system for tile-based level generation using constraints. We present a small mid-level constraint API that can be instantiated with various low-level solvers, including portfolio solvers. We show how this mid-level API can be used to generate levels incorporating a variety of constraints, including constraints learned from example levels and constraints provided by a designer. We incorporate a flexible constraint-based approach within the system for ensuring level goals are reachable. Finally, we demonstrate the effectiveness of the system in a variety of games and show applications ranging from infilling and repair to expressive range coverage.

Background

Procedural content generation (PCG) (Shaker, Togelius, and Nelson 2016) is an approach used in games to automatically generate content. Recently, procedural content generation via machine learning (PCGML) (Summerville et al. 2018) has been proposed as a way to learn to generate game content from examples of existing content.

In this work we present Sturgeon, a flexible and efficient system for constraint-based PCG applied to the generation of 2D tile-based levels. The system uses a small, mid-level API to express constraints over Boolean variables, and translates these into low-level constraint satisfaction problems that can be solved with a variety of standard low-level solvers (e.g. SAT, SMT, or Answer Set). Constraints can be learned from example levels (such as tile patterns and distributions) or provided by a designer (such as number or locations of certain tiles).

Previous work in PCG has used tile patterns learned from example levels, such as WaveFunctionCollapse (Gumin 2016), Model Synthesis (Merrell and Manocha 2010) or n-grams (Dahlskog, Togelius, and Nelson 2014). Several of the patterns used in this work are directly inspired by these.

Other previous work has incorporated constraint-based approaches into PCG, often to ensure generated levels have desired properties. Some of this work incorporates constraints learned from example levels, while others are built specifically to prevent certain classes of problems. This includes implementing WaveFunctionCollapse using Answer

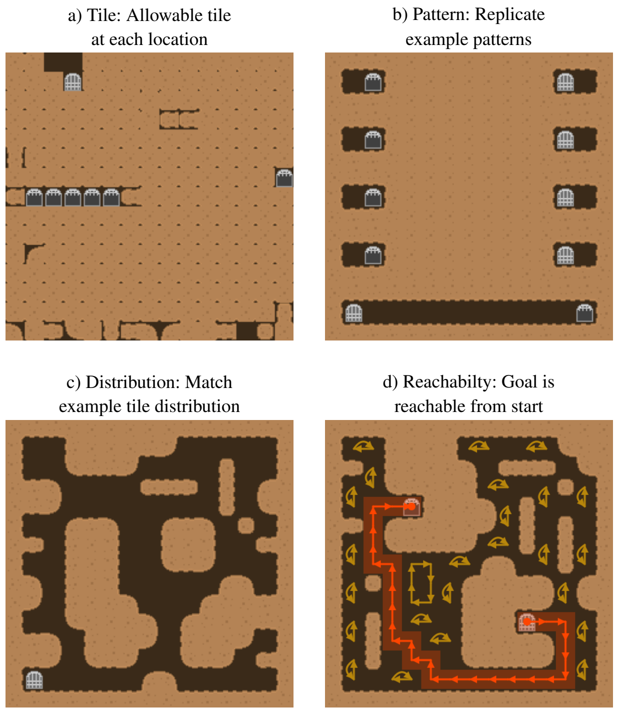

a) Tile: Allowable tile at each location
b) Pattern: Replicate example patterns
c) Distribution: Match example tile distribution
d) Reachabilty: Goal is reachable from start

Figure 1: Impact of adding each type of design rule, building on the previous, for the `cave` game. Reachability path shown in red (unused closed cycles shown in gold).

Set Programming (Karth and Smith 2017), generating mazes with reachable exits (Nelson and Smith 2016), constraining the possible solutions in puzzle games (Smith et al. 2012), and using probabilistic re-sampling to satisfy constraints (Snodgrass and Ontañón 2016).

We build on this previous work and demonstrate the flexibility of the proposed approach across seven different games and show how the mid-level API can be used to express useful constraints on the generated levels. We show how the use of more general constraint solvers can build on example- and pattern-based approaches to allow a wide variety of learned patterns and provide extra control through constraints. We formulate the problem of goal reachability as a graph problem that can be translated into constraints to ensure gener-

Function	Description	SAT-style implementation	Answer Set implementation	SMT implementation
MAKEVAR()	Create a new Boolean variable.			
MAKECONJ(ls)	Create a representation of the conjunction (*and*) of the given literals.	Create new conjunction variable.	Create new conjunction variable.	`And`
CNSTRCOUNT (vs, lo, hi, wt)	Add a constraint that between lo and hi of the given variables (or conjunctions) vs are true, with weight wt.	$atMostK$ native constraints if available, otherwise Boolean-encoded $atMostK$ constraints[†].	Frontend: constrained choice rule Backend: `add_weight_rule`	Pseudo-Boolean constraints `PbLe`, `PbGe`.
CNSTRIMPLIESDISJ (l, ms, wt)	Add a constraint that the literal (or conjunction) l implies the disjunction (*or*) of the literals (or conjunctions) in ms, with weight wt.	Clause	Frontend: rule Backend: `add_rule`	`Implies`, `Or`
SOLVE()	Run the solver.	Soft constraints directly supported; multiple hard constraints (e.g. encoded cardinality) can be converted to a single soft constraints by adding a label variable[‡].	Soft constraints supported via additional label variables[‡] added to rules and given to `#minimize` (frontend) or `add_minimize` (backend).	Soft constraints directly supported.
GETVAR(v)	Get the value (i.e. true or false) of a variable.			
GETOBJECTIVE()	Get the value of unsatisfied soft constraint weights.			

In many special cases, shortcuts can be used: e.g. if there is only one literal in MAKECONJ, it can be used directly; if lo is 1 in CNSTR-COUNT, a disjunction can be used; SMT can use `PbEq` when $lo == hi$; etc.

[†]In this work we use PySat's `kmtotalizer` encoding (Morgado, Ignatiev, and Marques-Silva 2014); since PySat only supported hard native $atMostK$ constraints, all such soft constraints must be encoded.

[‡]In this work, label variables are used as additional variables added to hard constraints which can then themselves be used in soft constraints or optimizations, e.g. (Belov, Järvisalo, and Marques-Silva 2013).

Table 1: Description of mid-level solver API and low-level implementations.

ated levels are possible to complete in games with a variety of player movement rules. We demonstrate support for designer-provided constraints in a variety of applications, including level infilling, linking level segments, and level repair, and show how constraints can be used to explore the expressive range coverage of the generator.

System Overview

The goal of the system is to generate 2D tile-based levels via a variety of constraints. A mid-level constraint API, consisting of only a few functions (described below and in Table 1), is used to express constraints over Boolean variables representing things such as tile placement and pathfinding properties of the level, which are then solved by a low-level solver. We show that the system can be used for a variety of applications; these applications can use constraints learned from example levels, constraints to ensure reachability of goals, and constraints provided by a designer.

In this work, a *tile* is simply an entity that can have functional (i.e. gameplay) and/or image information associated with it, and can be placed at a location in the 2D level grid. A *level* is then a 2D *grid* with locations where tiles are placed. Each level can consist of both a *functional grid* that defines gameplay for the level (e.g. using solid or goal functional tiles) and an *image grid* that defines what the level looks like (e.g. using brick or finish line image tiles).

This work also uses *tags*, which are labels associated with one or more tiles and can be used to limit what tiles can be placed at a location. A default tag allows any tile to be placed. Sometimes the tile/tag distinction is intentionally blurred: functional tiles can be used as tags to constrain image tile placement. For example, a solid functional tile could be used as a tag to only allow brick or stone image tiles to be placed at that location. Each level exists in an infinite grid of special *void* tiles and tags, wherever they are not otherwise defined.

It is possible to generate functional and image grids *simultaneously* by associating tiles with both functional and image information, or *sequentially* by first generating a functional grid and then using that to limit image tile placement. We found the sequential approach can be more efficient, as reachability does not have to consider what a tile looks like; it is also possible to generate only a functional or image grid.

Flows for simultaneous and sequential generation are given in Figure 2. The system takes example functional and image grids, from which it extracts tiles. It then takes a pattern template to use for extracting patterns, a region definition (discussed below) to extract tile distributions, and op-

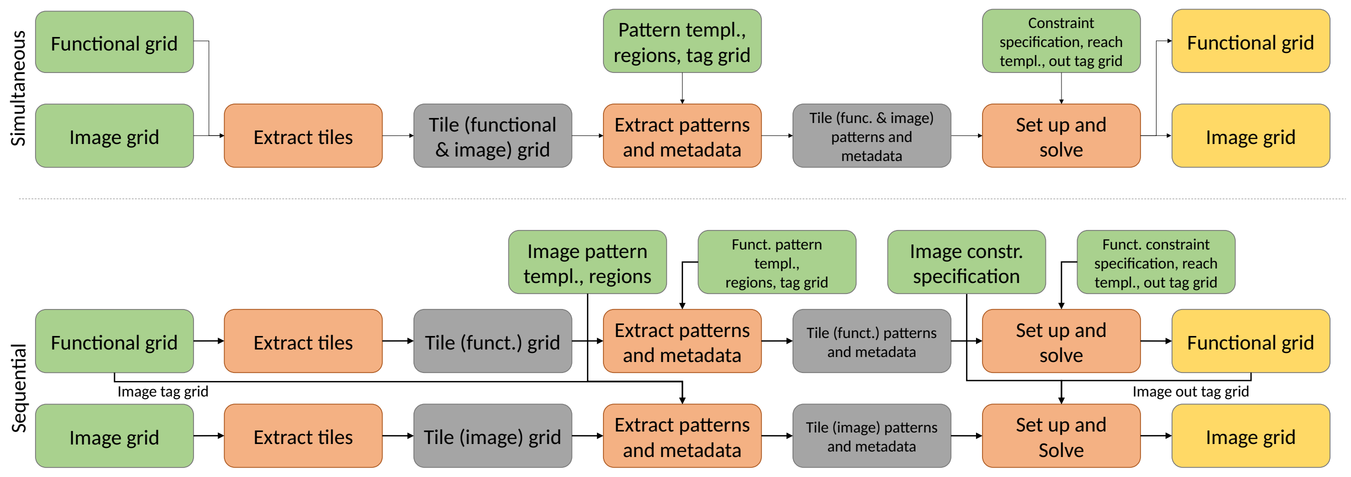

Simultaneous
Functional grid
Image grid
Extract tiles
Tile (functional & image) grid
Pattern templ., regions, tag grid
Extract patterns and metadata
Tile (func. & image) patterns and metadata
Constraint specification, reach templ., out tag grid
Set up and solve
Functional grid
Image grid
Sequential
Functional grid
Extract tiles
Tile (funct.) grid
Image pattern templ., regions
Funct. pattern templ., regions, tag grid
Extract patterns and metadata
Tile (funct.) patterns and metadata
Image constr. specification
Funct. constraint specification, reach templ., out tag grid
Set up and solve
Functional grid
Image tag grid
Image out tag grid
Image grid
Extract tiles
Tile (image) grid
Extract patterns and metadata
Tile (image) patterns and metadata
Set up and Solve
Image grid

Figure 2: Flows in learning and generation.

tionally, a *tag grid* to associate tags with tiles. At this point the generator is ready to generate levels. The generator uses information extracted from the example levels. It also uses other design rules, and can incorporate a reachability template, an out tag grid, and custom constraints. The mid-level API is used to create a constraint problem, which is then solved to generate a level.

We used four types of *design rules* for level generation. The design rules are collections of constraints that can be expressed using the mid-level API, which then uses a low-level solver to generate the level. For this work we used: tile rules, requiring one tile placed at each location; pattern rules, that provide relationships between nearby tiles learned from example levels; distribution rules, preferring the output tile distribution to be near the input distribution; and reachability rules, requiring a path through the level. Figure 1 shows the incremental effect of each type of rule.

The system is implemented in Python; the mid-level solver API is implemented as Python functions and uses the Python modules of the low-level solvers; the design rules are written in Python using the mid-level API. The system is run as a collection of command-line scripts.

In the following, we describe the mid-level solver API and solvers used to implement the API, the various design rules implemented using the API, and the games used in this work.

Mid-Level API and Low-Level Solvers

The API provides basic functions for creating Boolean variables, running the solver, and getting the resulting variable assignments. The API also provides functions for creating conjunctions of variables, along with functions for adding constraints on counts and adding implications (which can be hard or soft). The API and a description of how it uses the various low-level solvers is given in Table 1.

One benefit of supporting multiple low-level solvers is that *portfolio solvers* that combine multiple individual solvers can be used. The low-level solvers used are:

• `clingo-fe`, `clingo-be`: Answer Set (AS) based solvers. We used the Python interface to the clingo (Gebser et al. 2011) solver. We used two approaches in this work. For `clingo-fe`, we used the standard text-based "frontend" Answer Set Programming language with grounding and solving. In practice we found this approach to run quite slowly. Thus, for `clingo-be`, we used clingo's "backend" API directly to construct the program for the solver, and bypass the grounding step.

• `z3`: SMT solver. We use the Python Z3 solver (de Moura and Bjørner 2008). Although SMT solvers are more general, we used only Boolean variables.

• `pysat-rc2`, `pysat-fm`: Weighted partial max-SAT solvers, optionally with native cardinality constraints. For brevity we will refer to these as SAT-style solvers. In particular, these types of solvers support a mix of hard and weighted soft constraints, and optionally native $atMostK$ cardinality constraints. In this work we used solvers provided by the Python PySat (Ignatiev, Morgado, and Marques-Silva 2018) library: `pysat-rc2`, which uses a solver based on the "relaxable cardinality constraints" or RC2 approach (Ignatiev, Morgado, and Marques-Silva 2019), and `pysat-fm`, which is based on the "Fu & Malik" approach (Fu and Malik 2006; Manquinho, Marques-Silva, and Planes 2009). Native cardinality constraints are supported through their use of MiniCard (Liffiton and Maglalang 2012).

• `rc2&be`: A portfolio solver that combines `pysat-rc2` and `clingo-be`. Both solvers are run in parallel and the first solver to find an optimal solution is used.

Description of Design Rules

Here we describe the design rules used. In some cases they use game-specific information.

Tile Rules — The basic tile rules create a Boolean variable for each allowable tile at each location, and then require that exactly one allowable tile is placed in each location. This is similar to a "one-hot" style encoding. An out tag grid limits the allowable tiles at each location. Using the tile rules, the solver will generate levels with tiles that respect the tags, as

For all locations in the grid, for all potential tiles at that location:		
	MakeVar() $\rightarrow$ $tile$	Each location has a variable for each tile allowed by the tag at that location.
For all locations in the grid:		
	CnstrCount($tiles, 1, 1, HARD$)	Exactly one tile variable is true per location.

Table 2: Description of tile rules. $tile$ is the variable for an individual tile and $tiles$ are the variables for all tiles at a location. For convenience, all void tiles share a single variable that is constrained to be true (using CnstrCount).

When a pattern is needed:		
	MakeConj($tilesInPattern$) $\rightarrow$ $pattern$	A pattern is the conjunction of its individual tiles.
For all locations in the grid where patterns should be applied, using learned patterns:		
	CnstrImpliesDisj ($inputPattern, outputPatterns, HARD$)	If a placeable input pattern has output, and any output patterns are placeable: that input pattern implies at least one of its placeable output patterns.
	CnstrCount($inputPattern, 0, 0, HARD$)	If a placeable input pattern has output, and no output patterns are placeable: that input pattern is false.
For all locations in the grid where patterns should be applied:		
	CnstrCount($allInputPatterns, 1, \infty, HARD$)	At least one placeable input tile pattern is true.

Table 3: Description of pattern rules. The dotted line indicates that one of the two functions is used depending on if there are placeable output patterns or not.

For all regions, for all tags, for all tiles in that tag:		
	CnstrCount($countedTiles, min, max, SOFT$)	The number of tiles for a region and tag should be similar to the number desired, based on the example level.

Table 4: Description of distribution rules. In this work we use a low-weight soft constraint where min and max are $\pm 50\%$ of the desired number of such tiles.

General:		

	CnstrCount($validStartTiles, 1, 1, HARD$)
	CnstrCount($validGoalTiles, 1, 1, HARD$)
	CnstrCount($invalidStartTiles, 0, 0, HARD$)
	CnstrCount($invalidGoalTiles, 0, 0, HARD$)
		There must be exactly one start tile and one goal tile in valid locations, and none in any invalid locations.

For all edges:		
	MakeVar() $\rightarrow$ $edge$	Each edge has a variable for if it is reachable.
For all nodes:		
	MakeVar() $\rightarrow$ $node$	Each node has a variable for if it is reachable.
	MakeVar() $\rightarrow$ $open$ $\forall openTile$ CnstrImpliesDisj($openTile, open, HARD$) CnstrImpliesDisj($open, openTiles, HARD$)	Each node has a variable for if it is open; a node is open iff its corresponding tile is an open tile.
	$\forall outEdge$ CnstrImpliesDisj($\neg node, \neg outEdge, HARD$)	A node being not reachable implies all its out edges are not reachable.

	$\forall outEdge$ $\forall needOpen$
	CnstrImpliesDisj($\neg open_{needOpen}, \neg outEdge, HARD$)
	$\forall outEdge$ $\forall needClosed$
	CnstrImpliesDisj($open_{needClosed}, \neg outEdge, HARD$)
		For an edge, a node required to be open being closed, or a node required to be closed being open, implies the edge is not reachable.

	CnstrCount($inEdges, 0, 1, HARD$)
	CnstrCount($outEdges, 0, 1, HARD$)
		A node has at most one reachable in edge and one reachable out edge.

	MakeConj($\neg startTile \oplus \forall inEdge\ \neg inEdge$) $\rightarrow$ $noIn$ CnstrImpliesDisj($noIn, \neg node, HARD$)	A node that is not the start node and has no incoming reachable edges is not reachable.
	CnstrImpliesDisj($startTile, node, HARD$) $\forall inEdge$ CnstrImpliesDisj($startTile, \neg inEdge, HARD$)	The start tile locations's node is reachable and has no incoming reachable edges.
	CnstrImpliesDisj($goalTile, node, HARD$) $\forall outEdge$ CnstrImpliesDisj($goalTile, \neg outEdge, HARD$)	The goal tile locations's node is reachable and has no outgoing reachable edges.

Table 5: Description of reachability rules. The $\forall$ symbol represents a loop or list over all the relevant variables, and $\oplus$ is concatenation of variables into a list. For nodes, tiles referred to are those at the node's location.

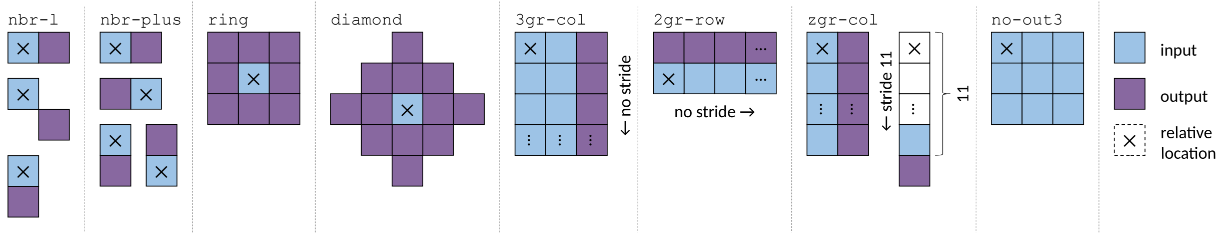


Figure 3: Pattern templates.

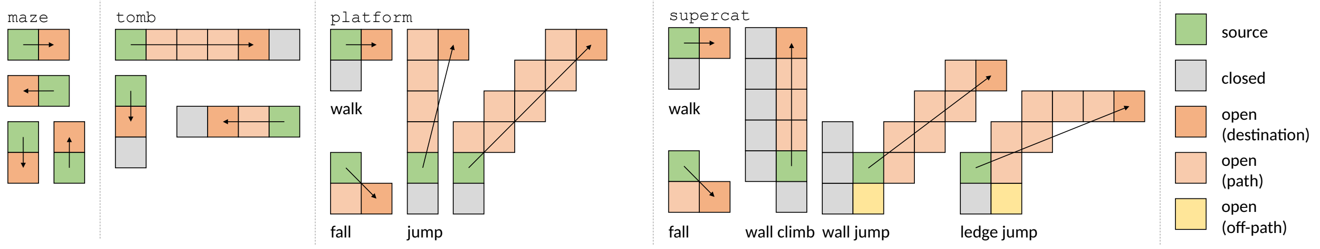


Figure 4: Example reachability templates (not a complete list).

in Figure 1(a). Table 2 describes API use.

Pattern Rules — With only tile rules, the tiles will be placed haphazardly. The addition of pattern rules begins to make the level look like the example level by replicating local patterns, as in Figure 1(b). Table 3 describes API use.

Pattern rules use a *pattern template*, which consists of a collection of corresponding input and output templates. Roughly speaking, the presence of an input pattern of tiles constrains the possible output patterns of tiles that can be present, and some input pattern must be present. The specific tiles and tags for the patterns used are learned from the example levels. Patterns can be learned and applied at each location, per-row, per-column, or with an arbitrary stride. For a specific learned input or output pattern to be *placeable* at a location, the tags for the generated level must match those from the example level for that pattern. Pattern templates used in this work, shown in Figure 3, are:

• `nbr-l`, `nbr-plus`: An individual tile independently constrains some of it neighbors.

• `ring`, `diamond`: An individual tile jointly constrains some of it neighbors.

• `3gr-col`, `2gr-row`: Meant to emulate 3-grams and 2-grams, as in (Dahlskog, Togelius, and Nelson 2014), constraining a following column or row based on the previous one(s). There is no stride along the axis perpendicular to the direction of the n-gram, so that, e.g. column constraints are applied only once at the top of each column.

• `zgr-col`: Crafted specifically for `zelda`. Similar to a column n-gram, but each column is 11 tiles long with a stride of 11 (the height of a room). This causes the n-gram to move across each row of rooms. The bottom tile of each column also constrains the top tile of the column below it, which causes each room to match up with the one below it (primarily for doors).

• `no-out3`: A 3×3 input and no output. Since an input pattern must be present at each location where patterns are applied, this is meant to emulate WaveFunctionCollapse (Gumin 2016).

Distribution Rules — Using tile and pattern rules, the overall distribution of tiles may be quite different from the example level, with tiles that are rare in the example level occurring frequently in the output, or vice versa. Distribution rules constrain the output tile counts to be similar to those in the example data, as in Figure 1(c). Table 4 describes API use.

We use distributions by tag (i.e. tiles in a tag in the output should be similar to the tiles in that tag in the input), as well as spatially by regions (i.e. tiles in each corresponding input/output region should be similar). We use two types of spatial distributions. `global` is simply a single region over the entire level. `div` divides levels into a coarse $N \times M$ grid of regions, re-scaling if the level sizes are different. For example, `div` regions should capture the fact that cloud images appear at the top of the level but not at the bottom.

Reachability Rules — Thus far, the levels may not be possible to complete since previous rules do not take into account player movement and whether the player can reach the goal of the level from the start. Adding reachability rules ensures generated levels are completable, as in Figure 1(d). Table 5 describes API use.

The reachability rules are based on a graph constructed over the level grid. Roughly speaking, each location gets a node, and there is a directed edge from each node to nodes that the player could *potentially* reach from that node. However, edges can only be part of the reachability path if certain criteria of *open* (i.e. traversable by the player) and *closed* (i.e. not traversable) functional tiles are met at locations in the level. As an example, a player might only be able reach a potential destination node along an edge that represents a jump if the location under the start of the jump is closed and all the locations along the arc of the jump are open. We also assume there are functional tiles that represent the *start* and *goal* of the level. The reachability rules require a path in the graph from the start to the goal. This approach is similar to Aloul et al.'s pathfinding using SAT work (Aloul, Rawi, and Aboelaze 2006). However, their work is on an undirected graph, with all edges always usable, and uses optimization to find the shortest path.

We noted some potentially interesting side-effects of this approach. First, it is only required that a path can be found; it does not have to be short or direct. Thus, the solver can find quite circuitous paths. Second, in addition to the path from start to goal, the solver can include additional closed cycles off the main path in the solution. For clarity, these cycles are not shown in most figures, but are shown in Figure 1(d).

Reachability rules use a *reachability template*, which defines the directed edges and their associated open and closed locations. Reachability templates used in this work are (examples shown in Figure 4):

• `maze`: Simple template where the player can move in 4 directions to adjacent open tiles. Used for `cave` and `zelda`.

• `tomb`: Template for `tomb`, where the player moves in 4 directions, but must move in a direction until a closed tile stops them.

• `platform`: Basic platforming template, based on the work of Summerville et al. (Summerville, Philip, and Mateas 2015). The player can fall through open tiles, and walk or jump from above closed tiles. Used for `mariobros`, `marioland`, and `icarus` (which allows column wrapping).

• `supercat`: Platforming template for `supercat`. The player can fall and walk, but also wall climb, wall jump, and ledge jump; there is no "standard" jump. These are a simplification of the actual game's movement, ignoring things like velocity and the variations between cats.

Games

The games used in this work are:

• `cave`: A simple top-down cave map made for this work. Image tiles from Kenney (Kenney 2022). Inspired partly by the mazes in (Nelson and Smith 2016).

• `tomb`: Tomb of the Mask (Happymagenta 2016), a top-down action/puzzle game where the player must navigate a maze while avoiding hazards. When the player starts moving in one direction, they continue in that direction until they hit an obstacle. Patterns from level 1 were used.

• `zelda`: The Legend of Zelda (Nintendo 1986), with tile patterns learned from dungeon 1-1 from the VGLC (Summerville et al. 2016) with minor cleanup.

• `icarus`: Kid Icarus (Nintendo 1987), with tile patterns learned from level 1 from the VGLC (Summerville et al. 2016) with minor cleanup.

• `mariobros`: Super Mario Bros. (Nintendo 1983), with tile patterns learned from level 1-1 from the VGLC (Summerville et al. 2016) with minor cleanup.

• `marioland`: Super Mario Land (Nintendo 1989), with tile patterns learned from level 1-1 from the VGLC (Summerville et al. 2016) with minor cleanup.

• `supercat`: Super Cat Tales (Neutronized 2016), a platform game where the player controls a cat. The game does not have a "standard" jump, but the the player can wall climb, jump off walls, and leap off ledges. Tile patterns learned from the tree section of level 1-7 were used.

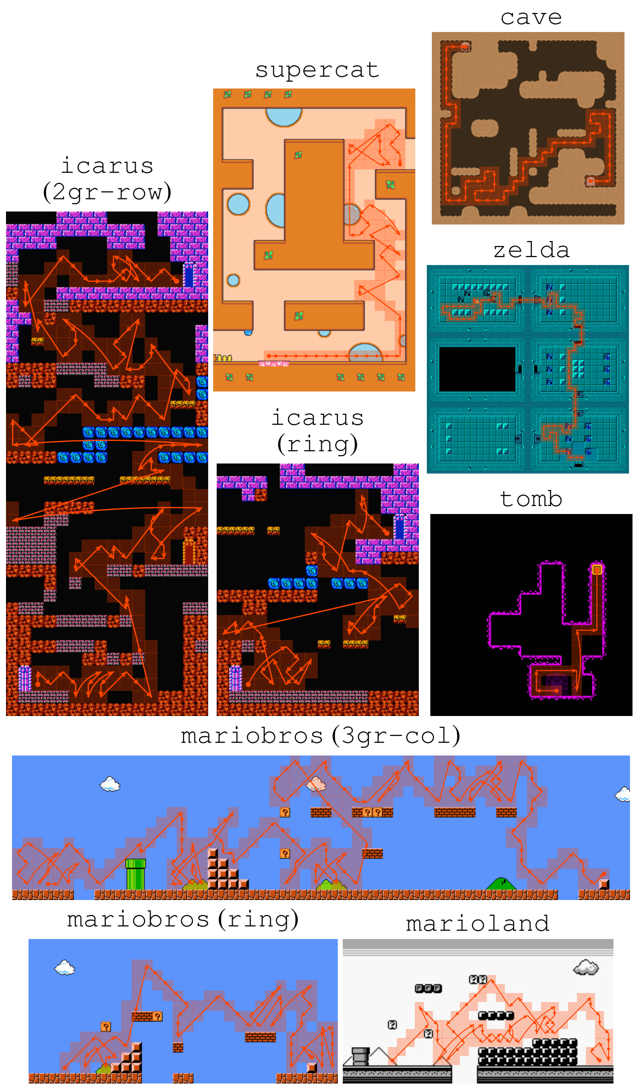


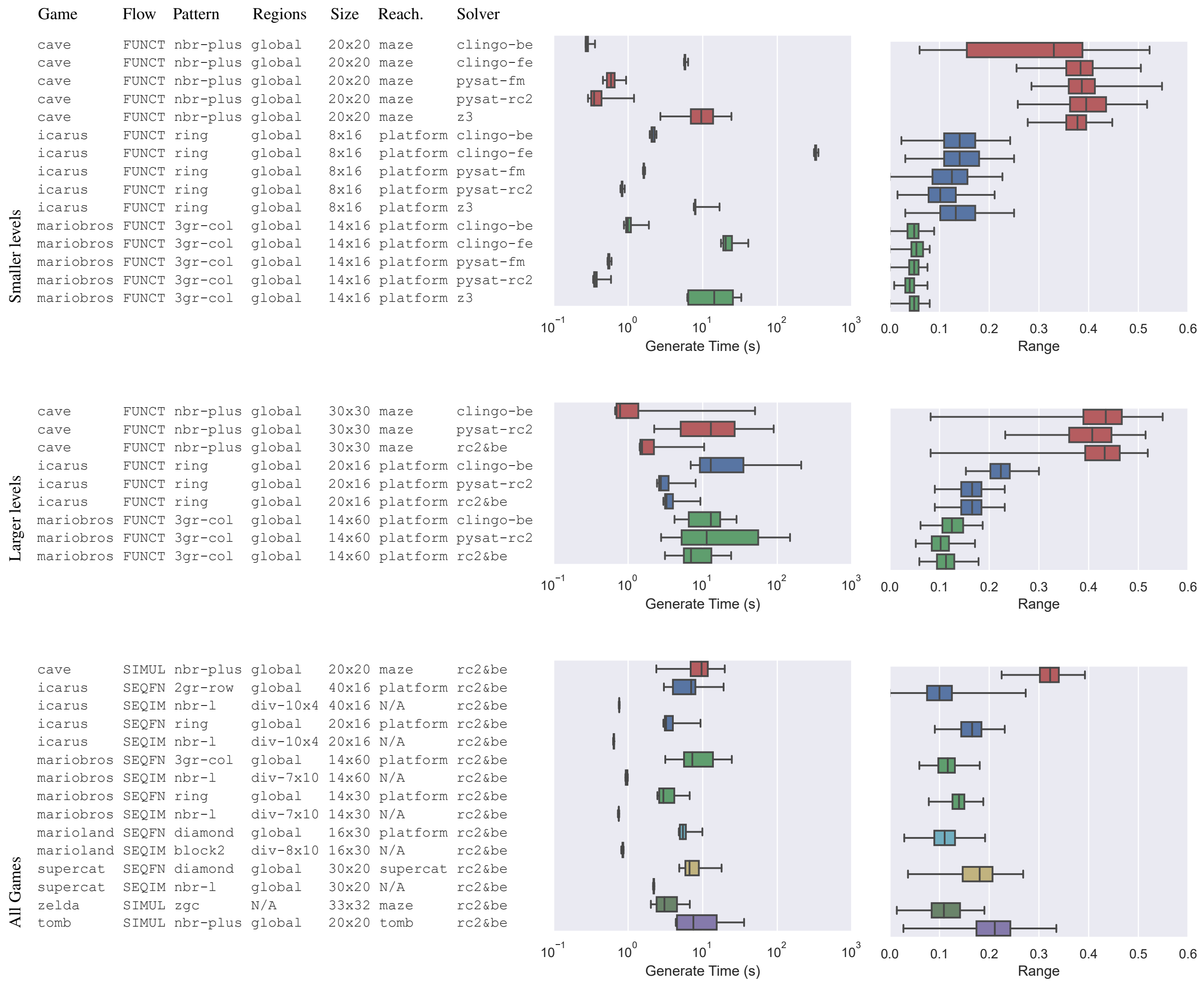
Figure 5: Example generated levels. Reachability path shown in red. More examples shown in Appendix A.

Solver Comparison and Level Generation

By using multiple low-level solvers, we can compare their performance. In the following evaluations, for each game setup (game, flow, pattern and reachability template, region definition, level size, and solver), we generated 25 levels. We looked at generation time and *range*, which we computed as fraction of tiles different, across all pairs of generated levels. We aimed for level sizes that could be generated in about 10s or less by most solvers. Evaluations were run on a 2018 MacBook Pro.

As an initial evaluation, we compared all individual solvers on generating small, functional levels for `cave`, `icarus`, and `mariobros`.

Results are in Figure 6(top). `clingo-fe` and `z3` were



Game	Flow	Pattern	Regions	Size	Reach.	Solver
cave	FUNCT	nbr-plus	global	20x20	maze	clingo-be
cave	FUNCT	nbr-plus	global	20x20	maze	clingo-fe
cave	FUNCT	nbr-plus	global	20x20	maze	pysat-fm
cave	FUNCT	nbr-plus	global	20x20	maze	pysat-rc2
cave	FUNCT	nbr-plus	global	20x20	maze	z3
icarus	FUNCT	ring	global	8x16	platform	clingo-be
icarus	FUNCT	ring	global	8x16	platform	clingo-fe
icarus	FUNCT	ring	global	8x16	platform	pysat-fm
icarus	FUNCT	ring	global	8x16	platform	pysat-rc2
icarus	FUNCT	ring	global	8x16	platform	z3
mariobros	FUNCT	3gr-col	global	14x16	platform	clingo-be
mariobros	FUNCT	3gr-col	global	14x16	platform	clingo-fe
mariobros	FUNCT	3gr-col	global	14x16	platform	pysat-fm
mariobros	FUNCT	3gr-col	global	14x16	platform	pysat-rc2
mariobros	FUNCT	3gr-col	global	14x16	platform	z3

Game	Flow	Pattern	Regions	Size	Reach.	Solver
cave	FUNCT	nbr-plus	global	30x30	maze	clingo-be
cave	FUNCT	nbr-plus	global	30x30	maze	pysat-rc2
cave	FUNCT	nbr-plus	global	30x30	maze	rc2&be
icarus	FUNCT	ring	global	20x16	platform	clingo-be
icarus	FUNCT	ring	global	20x16	platform	pysat-rc2
icarus	FUNCT	ring	global	20x16	platform	rc2&be
mariobros	FUNCT	3gr-col	global	14x60	platform	clingo-be
mariobros	FUNCT	3gr-col	global	14x60	platform	pysat-rc2
mariobros	FUNCT	3gr-col	global	14x60	platform	rc2&be

Game	Flow	Pattern	Regions	Size	Reach.	Solver
cave	SIMUL	nbr-plus	global	20x20	maze	rc2&be
icarus	SEQFN	2gr-row	global	40x16	platform	rc2&be
icarus	SEQIM	nbr-l	div-10x4	40x16	N/A	rc2&be
icarus	SEQFN	ring	global	20x16	platform	rc2&be
icarus	SEQIM	nbr-l	div-10x4	20x16	N/A	rc2&be
mariobros	SEQFN	3gr-col	global	14x60	platform	rc2&be
mariobros	SEQIM	nbr-l	div-7x10	14x60	N/A	rc2&be
mariobros	SEQFN	ring	global	14x30	platform	rc2&be
mariobros	SEQIM	nbr-l	div-7x10	14x30	N/A	rc2&be
marioland	SEQFN	diamond	global	16x30	platform	rc2&be
marioland	SEQIM	block2	div-8x10	16x30	N/A	rc2&be
supercat	SEQFN	diamond	global	30x20	supercat	rc2&be
supercat	SEQIM	nbr-l	global	30x20	N/A	rc2&be
zelda	SIMUL	zgc	N/A	33x32	maze	rc2&be
tomb	SIMUL	nbr-plus	global	20x20	tomb	rc2&be

Figure 6: Times for generation (setup and solve only) and ranges. For sequential generation in All Games, images were generated for the functional setup directly preceding.

much slower than the other solvers, and `pysat-fm` appeared generally slower than `pysat-rc2`; thus we excluded them from further evaluations. Additionally, there does not appear to be much difference in the range of levels generated, other than `clingo-be` generating more similar `cave` levels.

Next, we compared generation of larger functional levels on these games using the more efficient solvers `clingo-be` and `pysat-rc2` and their combined portfolio solver `rc2&be`. Results are in Figure 6(middle). This example highlights a benefit of portfolio solving: `clingo-be` appears faster for `cave` while `pysat-rc2` appears faster for `icarus`; however, using both, the portfolio solver performs well across all these games (though there was some overhead in managing the multiple solvers). Thus, from this point on we used this portfolio solver.

Finally, we generated levels for all games, using either the simultaneous or sequential flow, using multiple patterns for some games.

Solver results are in Figure 6(bottom), and example levels generated for each game setup are shown in Figure 5 and Appendix A. These examples were selected to show the types of levels generated and also highlight some artifacts of the generated levels. In `icarus` levels, note the use of wrapping by the pathfinding, as well as the change in the block visuals as one moves up the level. Similarly in `mariobros`, clouds only appear near the top of the level. There are some potentially undesirable artifacts. When using n-gram patterns for `icarus`, there is little variation in the levels as the beginning and end are essentially memorized. In both `cave` and `tomb` levels, some unreachable areas are generated. In `marioland` and `supercat`, there are some visual artifacts, in the form of incomplete pyramids and non-circular windows, respectively.

Controllability Applications

Here we describe several example applications that demonstrate the controllability allowed by the system. For each example, five levels or images were generated. Example grids, guides and generated levels and images are given in Appendix B. The first examples used the `mariobros` (`ring`) setup to generate levels; image generation took <1s for all these. The remaining WFC-inspired examples were image only.

• `tall-pipe`: A 14×24 level must have exactly one pipe that reaches the 3rd-from top row (other pipes are okay). Median and maximum functional generation times were 3.1s and 3.8s.

• `3?`: A 14×24 level must have exactly 3 question blocks. Note that these blocks are not necessarily reachable. Median and maximum functional generation times were 2.0s and 2.1s.

• `max-gap`: A 14×24 level where the number of blank tiles in the bottom row should be maximized. Median and maximum functional generation times were 2.5s and 3.7s.

• `void`: A 16×40 out tag grid that uses some tags that only allow void tiles, to carve out where the level can go. Uses soft pattern constraints. Median and maximum functional generation times were 3.1s and 25.2s.

• `infill`: A 14×32 output guide grid that uses hard constraints on output tiles, but has an empty area cleared out in the middle that needs to be filled in. Could be used in co-creative tools (Partlan et al. 2021) where a designer could request suggestions for specific areas. Uses soft pattern constraints. Median and maximum functional generation times were 2.6s and 3.0s.

• `link`: A 14×37 output guide grid that uses hard constraints on output tiles for two level segments, with an empty area in between that can be used to link them. This could be used to link level segments together in approaches where levels are generated segment-wise and then linked together into a larger, full level, e.g. (Khalifa et al. 2019; Sarkar and Cooper 2020). Uses soft pattern constraints. Median and maximum functional generation times were 3.0s and 3.1s.

• `repair`: A 14×18 output guide that uses soft constraints. The input level is not completable and the pipe is missing tiles near its top-right (highlighed in the red box). The solver makes minimal changes to repair the level and make it completable: adding the missing tiles to the pipe and adding a block so that the pipe can be jumped over. Such an approach could be used to repair levels generated by other means, e.g. as in (Jain et al. 2016; Cooper and Sarkar 2020; Zhang et al. 2020). Uses soft pattern constraints. These examples took notably longer; median and maximum functional generation times were 10.5m and 24.8m. Also, although different paths were found, all solutions added the block in the same place.

• `flowers-1`, `flowers-7`, `skyline-15`, `skyline-20`: The Flowers and Skyline examples from WFC (Gumin 2016). These applications all use the `no-out3` pattern rule to emulate WFC's behavior, along with the `global` count constraint, to generate an image only. Additional constraints are applied to specify the number of flowers (1 and 7) and windows (15 and 20). These applications demonstrate the additional control that can be applied above the standard WFC pattern approach when using a constraint solver. Median and maximum times were 30.3s, 3.6s, 6.9s, 6.0s and 36.2s, 4.3s, 14.9, 7.0s respectively.

Expressive Range Coverage

Here we highlight a use of constraint-based generation for exploring the expressive range of the generator. Common approaches to expressive range analysis generate many levels and then analyze them according to properties such as linearity or leniency (Smith and Whitehead 2010). Additionally, quality-diversity search such as MAP-Elites (Mouret and Clune 2015) optimize populations of points using similar behavioral characteristics.

For certain properties of a level, the proposed system can constrain the allowable range of that property and then generate a level; when done over a number of properties and ranges, this can demonstrate the expressive range coverage of the generator.

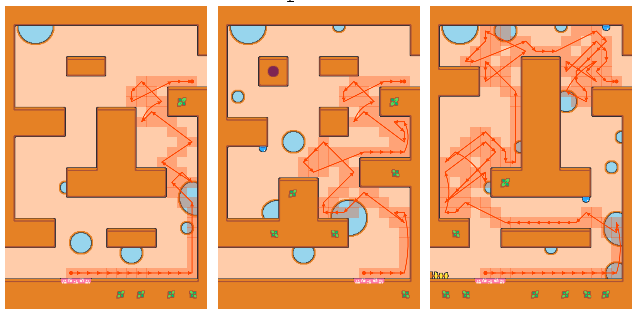
Using the `mariobros` (`ring`) setup, we explored the coverage along two dimensions, using number of gap tiles and number of solid tiles, with 6 ranges per dimension. A time limit of 10m was used for each 14×24 level. Of the 36 possible levels, 19 were found. Most of the levels not found required few or many solid tiles; 7 timed out. The total functional level generation took 86.2m, or 2.4m per possible level. Images are in Appendix C.

Conclusion

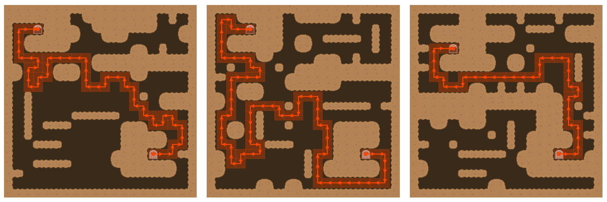
In this work we presented a system that uses a mid-level constraint-based API for generation of 2D tile-based levels and demonstrated it in a variety of games and application cases, evaluating its performance and controllability over generated levels. In the future we would like to expand to 3D grids and other structures such as graphs; add support for multiple sequential goals and other mechanics such as lock-and-key; and explore further performance improvements, particularly to level repair.

Appendix A: Level Images

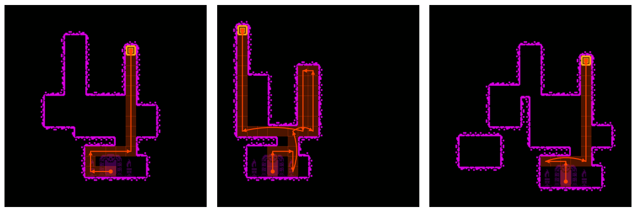
`supercat`



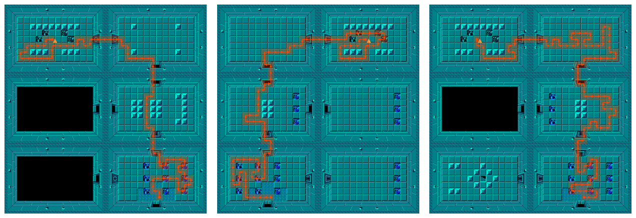
cave



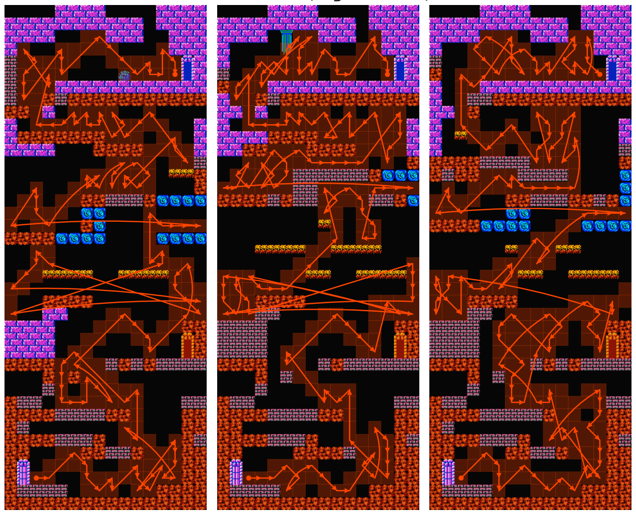
tomb



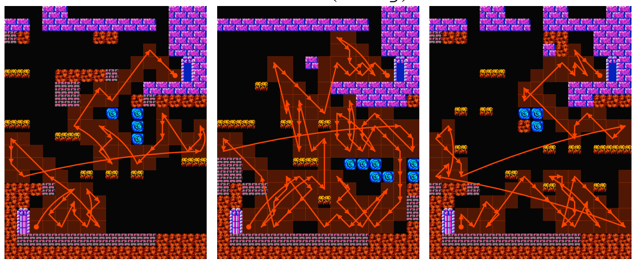
zelda



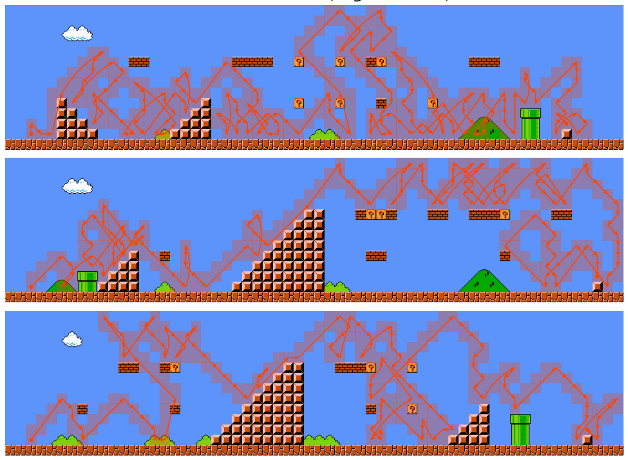
icarus (2gr-row)



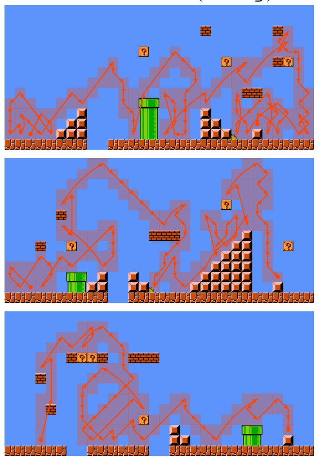
icarus (ring)



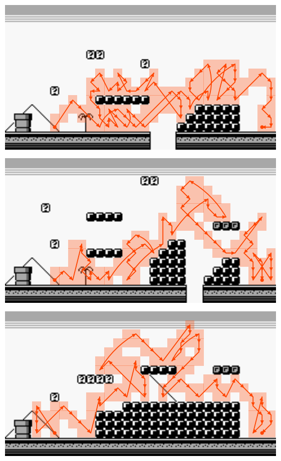
mariobros (3gr-col)



mariobros (ring)

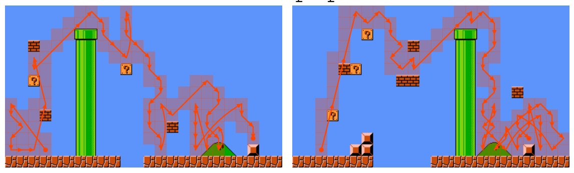


marioland

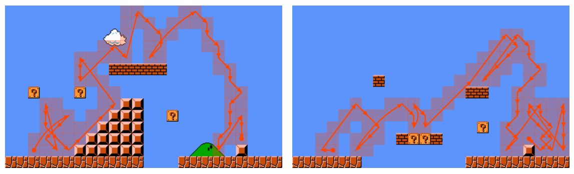


Appendix B: Applications

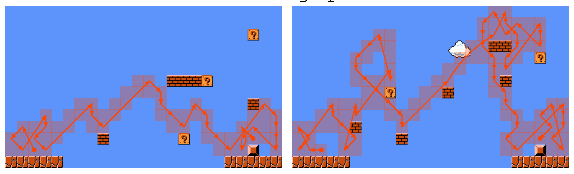
tall-pipe

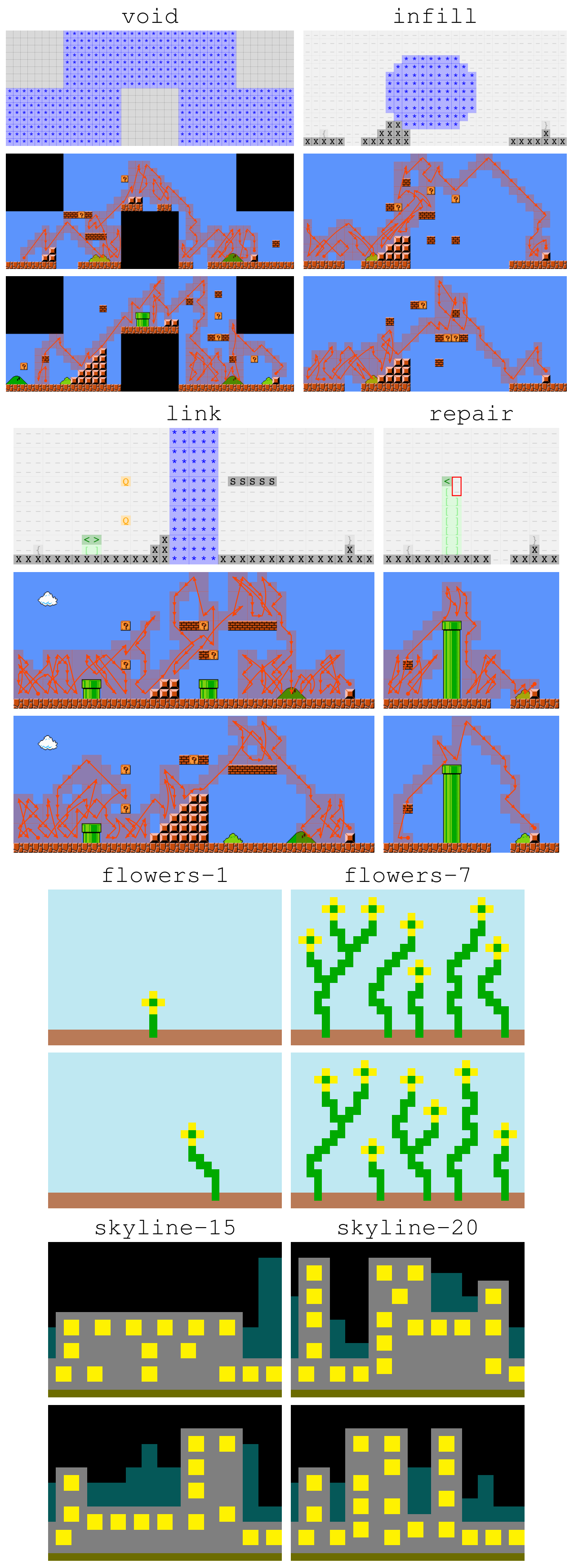


3?

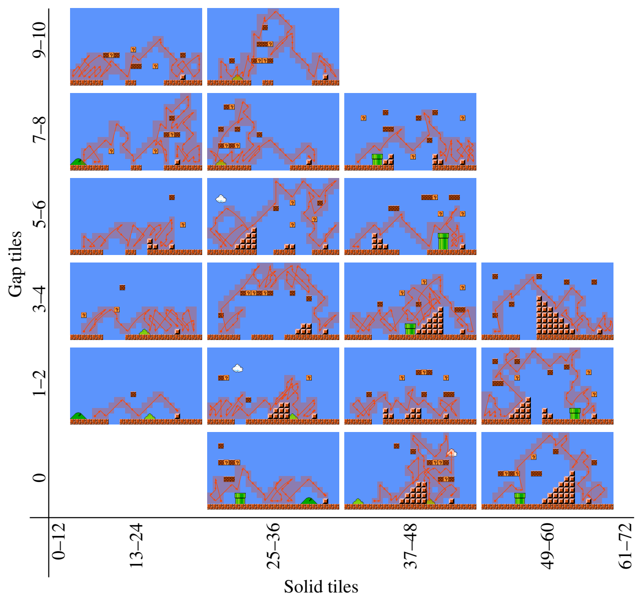


max-gap





Appendix C: Expressive Range Coverage



Acknowledgments

The authors would like to thank Adam Smith for discussion of Clingo's backend interface, and Colan Biemer and Anurag Sarkar for their feedback.

References

Aloul, F. A.; Rawi, B. A.; and Aboelaze, M. 2006. Identifying the shortest path in large networks using Boolean satisfiability. In *2006 3rd International Conference on Electrical and Electronics Engineering*, 1–4.

Belov, A.; Järvisalo, M.; and Marques-Silva, J. 2013. Formula preprocessing in MUS extraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, 108–123.

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

Dahlskog, S.; Togelius, J.; and Nelson, M. J. 2014. Linear levels through n-grams. In *Proceedings of the 18th International Academic MindTrek Conference*, 200–206.

de Moura, L.; and Bjørner, N. 2008. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 337–340.

Fu, Z.; and Malik, S. 2006. On solving the partial MAX-SAT problem. In *Theory and Applications of Satisfiability Testing - SAT 2006*, 252–265.

Gebser, M.; Kaufmann, B.; Kaminski, R.; Ostrowski, M.; Schaub, T.; and Schneider, M. 2011. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2): 107–124.

Gumin, M. 2016. WaveFunctionCollapse. https://github.com/mxgmn/WaveFunctionCollapse. Accessed: 2022-02-04.

Happymagenta. 2016. Tomb of the Mask. Game [iPhone].

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.

Ignatiev, A.; Morgado, A.; and Marques-Silva, J. 2019. RC2: an efficient MaxSAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1): 53–64.

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.

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*, 68:1–68:10.

Kenney. 2022. Free game assets. https://www.kenney.nl/assets. Accessed: 2022-01-07.

Khalifa, A.; Green, M. C.; Barros, G.; and Togelius, J. 2019. Intentional computational level design. In *Proceedings of The Genetic and Evolutionary Computation Conference*, 796–803.

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.

Manquinho, V.; Marques-Silva, J.; and Planes, J. 2009. Algorithms for weighted Boolean optimization. In *Theory and Applications of Satisfiability Testing - SAT 2009*, 495–508.

Merrell, P.; and Manocha, D. 2010. Model synthesis: A general procedural modeling algorithm. *IEEE transactions on visualization and computer graphics*, 17(6): 715–728.

Morgado, A.; Ignatiev, A.; and Marques-Silva, J. 2014. MSCG: robust core-guided MaxSAT solving. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1): 129–134.

Mouret, J.-B.; and Clune, J. 2015. Illuminating search spaces by mapping elites. *arXiv:1504.04909 [cs, q-bio]*.

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.

Neutronized. 2016. Super Cat Tales. Game [iPhone].

Nintendo. 1983. Super Mario Bros. Game [NES].

Nintendo. 1986. The Legend of Zelda. Game [NES].

Nintendo. 1987. Kid Icarus. Game [NES].

Nintendo. 1989. Super Mario Land. Game [Game Boy].

Partlan, N.; Kleinman, E.; Howe, J.; Ahmad, S.; Marsella, S.; and El-Nasr, M. S. 2021. Design-driven requirements for computationally co-creative game AI design tools. *arXiv:2107.13738 [cs]*.

Sarkar, A.; and Cooper, S. 2020. Sequential segment-based level generation and blending using variational autoencoders. In *International Conference on the Foundations of Digital Games*, 1–9.

Shaker, N.; Togelius, J.; and Nelson, M. J. 2016. *Procedural content generation in games*. Springer.

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.

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

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.

Summerville, A. J.; Philip, S.; and Mateas, M. 2015. MCMCTS PCG 4 SMB: Monte Carlo tree search to guide platformer level generation. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

Summerville, A. J.; Snodgrass, S.; Mateas, M.; and Ontañón, S. 2016. The VGLC: The Video Game Level Corpus. *arXiv:1606.07487 [cs]*.

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.