Guided Game Level Repair via Explainable AI

Mahsa Bazzaz, Seth Cooper

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

Abstract

Procedurally generated levels created by machine learning models can be unsolvable without further editing. Various methods have been developed to automatically repair these levels by enforcing hard constraints during the post-processing step. However, as levels increase in size, these constraint-based repairs become increasingly slow. This paper proposes using explainability methods to identify specific regions of a level that contribute to its unsolvability. By assigning higher weights to these regions, constraint-based solvers can prioritize these problematic areas, enabling more efficient repairs. Our results, tested across three games, demonstrate that this approach can help to repair procedurally generated levels faster.

Introduction

Procedural Content Generation via Machine Learning (PCGML) involves generating game content using models trained on existing game content (Summerville et al. 2018). These models aim to produce solvable levels, which players can complete by moving from the start to the end of the level through a series of valid movements while completing tasks and avoiding obstacles.

However, due to the noisy nature of ML-based generation, a repair step is often necessary in PCGML approaches to make the generated levels usable (Chen et al. 2020). The initial outputs from these models often include levels that are impossible to complete or contain broken structures with missing or incorrectly arranged elements (e.g., pipes in platformer games or broken decorations in puzzle games). Consequently, recent research has focused on developing methods to repair these levels by making the necessary adjustments.

One popular approach to level repair involves using constraint satisfaction. This method represents the problem as a collection of constraints over variables and seeks an assignment of values to these variables that satisfies all constraints. Constraint solvers are software tools that search through possible assignments to find one that satisfies the constraints or they prove that no such assignment exists.

In the context of level repair, the process generally starts with the largest possible set of candidate values for each

tile. The solver then cycles between propagating the constraints and narrowing down the possible values until a fixed point — a complete level that satisfies all constraints — is reached. This process involves iteratively exploring possible assignments of values to variables. The number of possible assignments grows exponentially with the number of variables. Consequently, this iterative process, which may include backtracking, can be very time-consuming.

Recent work in interpretable/explainable AI has led to AI approaches (such as classifiers) that can explain their decisions. The goal of explainability in machine learning is to make the model's decision-making process understandable to humans, allowing users to trust and interpret the model's outputs. For example, in a model designed to recognize handwritten digits, an explainable model might highlight specific regions of the image, such as the curves and lines that form the shape of each digit, indicating that these features significantly contribute to the model's decision that the image contains a particular number. In our work, we propose training a binary classifier to distinguish between two classes of game levels: solvable and unsolvable. We then apply explainability methods to these classifiers to identify the regions of the level that contribute to their unsolvability. Subsequently, we provide meaningful weights that prioritize the change of these problematic regions to a constraint-based solver, guiding level repair and enhancing its performance.

In this work, we first generate a training set consisting of solvable and unsolvable levels from three different 2D tile-based games. These levels are used to train deep neural network classifiers that label levels as either solvable or unsolvable. Explainability methods enable the classifiers to provide per-location attributions, indicating the importance of each tile in the classification process. These attributions assign importance to the features (level tiles) contributing to the classifier's predictions (solvability). We utilize this attribution data as weights to guide a constraint satisfaction solver that repairs the level.

Subsequently, we construct a pipeline for our experiments to measure the performance of this approach. This pipeline includes a level generator, the explainable solvability classifier, a weight generator, and a constraint satisfaction solver. This pipeline generates an unsolvable level, obtains attributions for the unsolvability of each tile location from the ex-
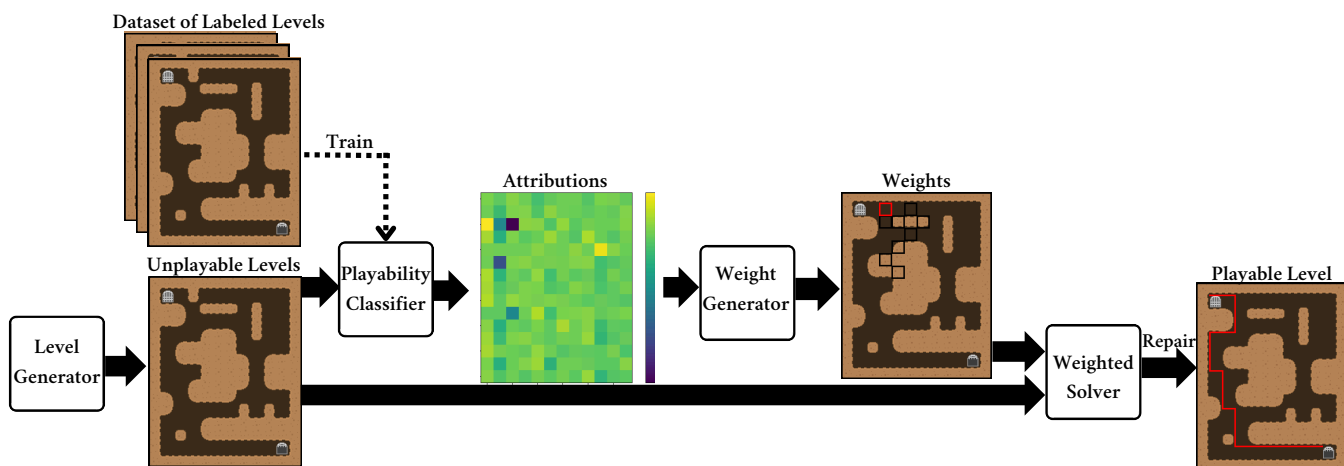
Figure 1: Visual Representation of the system pipeline.

plainable classifier, creates weights based on the attributions for the constraint solver, and attempts to repair the level by solving this constraint satisfaction problem, guided by the attributions.

We conducted our experiments on levels across three different 2D tile-based game domains: Super Mario Bros., Super Cat Tales, and a custom Cave game. Our results show using attributions to unsolvability gathered from the explainable solvability classifier helps the solver repair the levels faster.

In summary, our contributions are as follows:

- We introduce the use of explainability methods to identify regions in unsolvable levels that contribute to their unsolvability.
- We develop a pipeline to demonstrate the effectiveness of this approach in enhancing level repair performance.
- We present a Mixed Integer Linear Programming encoding of the constraints used for weighted level repair.

The GitHub repository containing the data, models, training code, and generated artifacts is available on GitHub[1].

Related Work

Level Repair

Solvability is a crucial aspect of procedural level generation. However, unsolvable levels are almost inevitable since current machine learning models cannot consistently ensure the satisfaction of a constraint like solvability. As a result, various "generate-then-repair" approaches have been developed to address this issue. Cooper and Sarkar (2020) introduced a pathfinding agent capable of repairing levels as part of testing levels for solvability. Their results show that the repair agent could improve the solvability of the level at the cost of an increase in time. Jain et al. (2016) leveraged the lossy nature of AutoEncoder reproduction to repair broken levels and classify them into different styles. Mott, Nandi, and Zeller (2019) applied long-short term memory recurrent

neural network (LSTM) to repair locally incoherent sections of GAN-generated levels. Zhang et al. (2020) constructed a framework to repair GAN-generated levels using a mixed-integer linear program (MILP) with solvability constraints. Chen et al. (2020) tried repairing the results of translating image representation (pixels) into a level representation (tiles) with Autoencoders and Markov chains. Most relevant to our work, Shu et al. (2020) attempted to repair errors in pipe tile placement using a combination of multi-layer perceptron and genetic algorithms; their approach, similar to ours, trains a model to directly find locations of misplaced tiles. However, in their approach, information about misplaced tiles is used by an evolutionary repairer, and focuses mainly on local tile placements. On the other hand, our work uses the attributions from explainable classifiers to inform weighted level repair and concentrates primarily on the global solvability of levels.

Solvability Analysis

Path-finding agents and level classifiers have been utilized in order to help generate solvable levels.
Snodgrass and Ontañón (2014), Volz et al. (2018), and other studies, used Robin Baumgarten's A* controller (Togelius et al. 2011) to divide the generated maps into solvable and unsolvable. The tile-based agent of Summerville, Philip, and Mateas (2015) has also been utilized in many projects in order to determine if levels are solvable or unsolvable (Summerville and Mateas 2016; Snodgrass and Ontañón 2016). More recently, deep neural network classifiers have been employed for level solvability classification as well. Bazzaz and Cooper (2023) used Pool-based Active learning to train solvability classifiers that require very few labeled levels.

Model Explainability

Given that recent work has applied deep learning methods to classifying level solvability, explainable classifiers become relevant. Model explainability refers to the degree to which a human is able to understand the reasons behind a choice or prediction made by the model. Explainable ma-
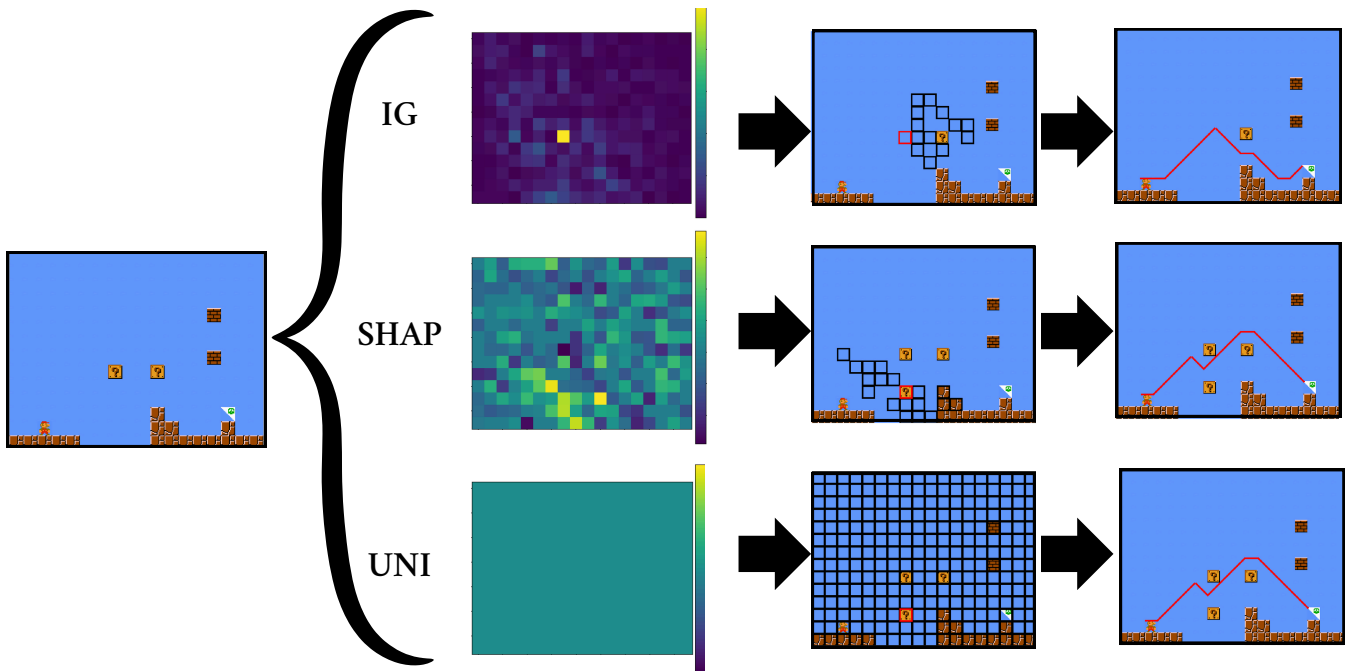
Figure 2: Visual representation of different repairs to a Mario level. The left side is the broken level. Each explainability method generates an attribution for all tiles, and these attributions are scaled to weights. The black squares show the low weights marked for the solver. The red squares show the tile that the solver chooses to change. The right side is the repaired level.

chine learning methods can be used to discover knowledge, to debug or justify the model and its predictions, and to improve the model. Machine learning models can be categorized as glass-box or black-box (Rai 2020). Glass-box models are inherently explainable to the user, but black-box models need explainability methods to generate explanations. There are multiple techniques used in explainable artificial intelligence (XAI) to understand the decisions made by these complex machine learning models, particularly deep neural networks.

Two widely adopted explainability methods in computer vision are *Deep SHAP* and *Integrated Gradients*, which help in understanding decisions made by classifiers.

Deep SHAP is an explainability technique that can be used for models with a neural network architecture (Shrikumar et al. 2016). It combines Shapley values with the DeepLIFT algorithm to approximate the conditional expectations of Shapley values using a selection of background samples (Lundberg and Lee 2017a). Shapley values (Shapley et al. 1953) are one of the measures of feature importance from cooperative game theory, to attribute the contribution of each feature to the model's output. It computes the importance of each feature by considering all possible combinations of features and their effects on the prediction.

Integrated Gradients provide a robust and interpretable method for understanding the contributions of input features to the predictions of the classifier. It computes the integral of the gradients of the model's output concerning its input along a straight path from a baseline to the input of interest. By integrating the gradients, Integrated Gradients assign im-

portance scores to input features, indicating how much they contribute to the model's decision (Sundararajan, Taly, and Yan 2017).

Zhu et al. identified Model Explainability as a valuable research area and proposed various research questions and case study scenarios. For example, one suggestion was to make the noise vector in a GAN model explainable, allowing users to control the output by modifying the noise vector.

System Overview

Traditionally in explainable AI, *feature attribution* refers to how much each feature (e.g., pixels of an image) influences the model's output. For example, in a model designed to detect cats in images, feature attribution might highlight specific areas of the image, such as the shape of the ears or the pattern of the fur, indicating that these features significantly contribute to the model's decision that the image contains a cat. This work proposes training a solvability classifier with a dataset of solvable and unsolvable levels. The classifier will then provide relative attributions indicating the contribution of each tile to the classification of a level's solvability. In out work, these attributions can be converted into penalty weights to guide the repair process, where tiles with *higher* attributions to the classification receive *lower* weights and are thus less penalized for changes during the repair. Ideally, this guidance helps the constraint solver to repair the level more efficiently.

To evaluate the effectiveness of weights derived from explainable classification, we developed a controlled pipeline for unsolvable level repair, allowing comparison of various

weight generation methods. Figure 1 illustrates the overall framework of the pipeline. The main components of our system are:

- *level generation* This can be a level generator like constraint-based generators, or any machine learning model like VAEs, GANS, etc.

- *explainable classification* Takes a level as input and assigns each tile location an *attribution* value of participation in the unsolvability of the level. To train the classifier, a set of *training levels*, both solvable and unsolvable, are needed.

- *weight generation* Creates meaningful penalty weights for the change of tiles at each location of the level to guide the repair.

- *level repair* Accepts a level to repair, along with the associated weights, as inputs, and finds a solvable level solution.

Here we describe the specific components and domains used in this work.

Domains

This work experiments on three different game domains of Super Mario Bros. (Mario) (Nintendo 1985), Cave, and Super Cat Tales (Supercat) (Neutronized 2016). The Mario levels are based on the level 1-1 from the VGLC (Summerville et al. 2016). Cave is a top-down cave map introduced by Cooper (2022a); this custom game is created with image tiles from Kenney (Kenney 2022). Supercat is also a platforming game with wall and ledge jumps; a simplified version of the actual game's movement patterns is used. The example levels as well as patterns and reachability templates used in this work are borrowed from the original work on Sturgeon (Cooper 2022a). The Mario levels are in size 14×18, Cave levels 15×12, and Super Cat Tales levels 20×20.

Level Generation

In this work, we used the Sturgeon constraint-based level generation system (Cooper 2022b), which generates levels by converting high-level design rules into constraint satisfaction problems and can use different "low-level" solvers to find solutions. This system is also capable of generating unsolvable levels by incorporating additional constraints that a level's goal is not reachable from its start (Cooper and Bazzaz 2024). This constraint-based level generator gave us the controllability to evaluate the performance of weight generation methods regardless of the level generation model.

Explainable Classification

Figure 4 illustrates sample attributions of a Mario level and the rescaled values into weights. We used a unified classifier for the solvability classification of levels in all three domains. We used a minimal representation version of games to train the classifier that consists of 4 tile types. We created a unified simple affordances that include Solid, Empty, Start, End and we simplified all games into these affordances. Figure 3 shows this new representation for all three domains.
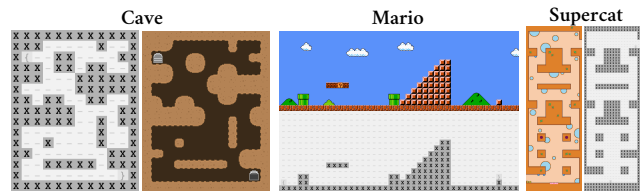

Figure 3: The minimal representation of the levels with 4 tile types (start: {, goal: }, solid: X, and empty: −) used for training the classifiers.
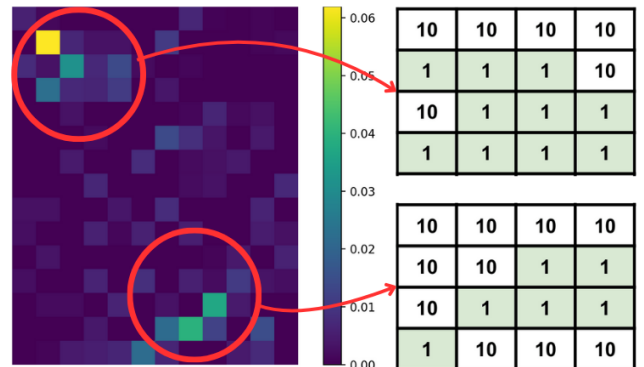

Figure 4: Representation of how attributions are scaled to meaningful (positive integers) weights for the solver. Lower values indicated a higher priority to change.

After this transformation, the levels are converted into one-hot vectors for the classifier.

The classifier architecture consists of three fully connected layers followed by a ReLU activation layer to introduce non-linearity, and a Dropout layer to to prevent overfitting. Lastly, the sigmoid activation function is used to squash the output into the range [0, 1] for binary classification.

To train the classifier, we generated a corpus of $3,000$ unsolvable levels and solvable levels from each domain game. Both solvable levels and unsolvable levels were generated using Sturgeon (Cooper 2022b). These levels all maintain the correct game structure, with the only difference being the existence of a path between the start and end points. This makes them ideal for training a classifier to distinguish between solvable and unsolvable levels.

We split the dataset into training and testing sets with an $80 - 20$ ratio. We used a weight decay of $1e - 3$ and a learning rate of $1e - 2$ with the Adam optimizer and CrossEntropy loss criterion. The classifier was trained for 10 epochs, achieving an accuracy of 98% for Cave, 97% for Mario, and 100% for Supercat.

We used the implementation of Deep SHAP by the Shap Python package (Lundberg and Lee 2017b) and implementation of Integrated Gradients from the Captum Python package (Kokhlikyan et al. 2020) to obtain explanations about tile location attributions in level solvability. These packages are among the most widely used tools for model explainability.

Weight Generation

The tile attributions are processed into weights for the repair solver. The primary goal at this stage is to identify regions of the level with the highest attribution values. To achieve this, the 80th percentile threshold of the attributions is calculated. This threshold captures the top 20% of values, effectively filtering out lower values.

Next, a binary map is created by comparing each attribution value against the percentile threshold. Connected component labeling is then employed to extract regions from the original level, focusing exclusively on components that are connected. Using OpenCV's connectedComponentsWith-Stats function (Bradski 2000), we analyze the connected components within the binary map to identify the label corresponding to the largest connected component in terms of area.

Solvers that accept weights as an additional input consider the weights as penalties for changing a tile. Consequently, we assign the largest connected component within the attributions the lowest weight, increasing the likelihood that the solver will modify this area of the level. After identifying the regions with higher attribution values and determining the connected region from the original level, we scale the high values to weight 1 and the others to weight 10.

Level Repair

Sturgeon is capable of repairing unsolvable levels by making minimal changes to the original level (Cooper 2022b). This is accomplished by converting the repair task into a constraint satisfaction problem and using constraint solvers to either find a solution or prove that none exists.

Unlike agent-based repair methods, this approach ensures the level is in a definite state after repair. The solver either finds a solution, which may take a considerable amount of time, or confirms that no solution exists, eliminating the need for agents to play the level to verify the repair.

The repair process asks the solver to create a solvable level while imposing soft constraints to match the original level's tile placements as closely as possible. Because this procedure can be more time-consuming than generating a new level, we introduce an enhancement to Sturgeon's original method: the option to assign per-location tile weights. This allows each tile in the level to have a different penalty for deviating from the original layout. These weights are integrated into the constraint problem for the solver.

Essentially, Sturgeon applies a reachability constraint from the start to the end of the level as a hard constraint and uses per-location tile weights as soft constraints to guide the solver in finding a solvable level.

Mixed-Integer Linear Programming Solver

Sturgeon uses a small "mid-level" API to translate constraint satisfaction problems into "low-level" solvers, which do the actual solving (Cooper 2022b). Although previous to this work, Sturgeon had support for several SAT- and ASP-based solvers, in the preliminary development of this project we explored the potential of using Linear Programming approaches for a new low-level solver. While Linear Program-

ming has been used previously for level repair by Zhang et al. (2020), in their work, they directly encoded the specific problem of level repair, whereas we indirectly encode it through the functions available in Sturgeon's mid-level API in such a way that the encoding could be used for other applications using Sturgeon (e.g. level generation).

Because Sturgeon's use of a MILP solver is introduced in this work, we provide more information about its implementation. However, the details of the MILP solver are not necessary to follow the other contributions of this work.

Here we describe how Sturgeon formulates its constraint problems for a low-level Mixed-Integer Linear Programming (MILP) solver (The SciPy community 2024). The goal of a MILP is to find assignments for a vector of numerical variables $\mathbf{x}$ that minimizes the weighted sum of the variables, where the weights are given by another vector $\mathbf{c}$, i.e. finding $\text{argmin}_{\mathbf{x}} \mathbf{c}^\mathsf{T} \mathbf{x}$. The variables can have bounds placed on them directly, i.e. $\mathbf{s} \le \mathbf{x} \le \mathbf{t}$, and bounds can be placed on weighted sums of the variables, i.e. $\mathbf{u} \le A\mathbf{x} \le \mathbf{v}$. Additionally, given elements of $\mathbf{x}$ can be required to be integers.

To convert the Boolean constraint satisfaction problem into a MILP, each Boolean variable v in the original problem is given a corresponding numerical variable x_v, as well as a corresponding entry in the weight vector c_v. A numerical variable value of 1 corresponds to a Boolean value of true, and a numerical value of 0 corresponds to false. As Boolean constraints include Boolean *literals* — both a variable and a *polarity* (i.e. v or $\neg v$) — for a given literal l, we also use x_l to refer to the numerical variable for a literal's Boolean variable x_v. Note that additional variables can be allocated as needed (described below).

Then, all variables $x_i \in \mathbf{x}$ have the constraint that they are integers ($x_i \in \mathbb{Z}$) and are either zero or one ($0 \le x_i \le 1$), thus $\mathbf{s} = \mathbf{0}$ and $\mathbf{t} = \mathbf{1}$. Although this setup could be formulated as a zero-one integer program, in practice we use MILP solvers, which support real-valued variables, but constrain them to be integers.

What remains to complete the MILP is then to fill in the values for the matrix A and the weight vector $\mathbf{c}$. The weight for all the variables in the initial problem is set to 0 as there is no preference for their being true or false, i.e.

$$c_v = 0$$

except for the additional weighting variables described below. To fill in A, we specify additional linear constraints on the variables, also described below.

The core of the approach is that, when using the numerical variables to constrain counts of Boolean literals that evaluate to true, positive variables can be included in the constraint directly as x_v, and negative variables can be included as $1 - x_v$ with the 1 then incorporated into the bounds. For example, a constraint that at most 1 of the literals i and $\neg j$ are true would be:

$$x_i + (1 - x_j) \le 1$$

or,

$$x_i - x_j \le 0$$

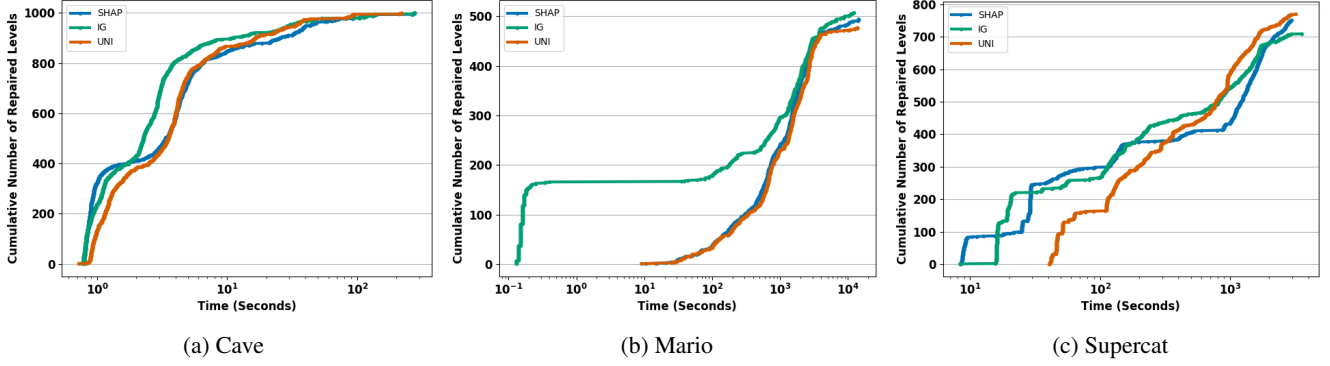|	(a) Cave	(b) Mario	(c) Supercat

Figure 5: Comparison of number of repaired levels in times between different weight generation methods. The time is shown on a logarithmic scale.

Dataset	Cave			Mario			Supercat		
	SHAP	IG	UNI	SHAP	IG	UNI	SHAP	IG	UNI
Mean	9.7	7.6	7.8	1192.2	1198.4	1523.2	747.2	511.4	650.5
Median	3.3	2.2	3.5	723.4	722.6	1161.9	148.3	128.5	348.2
Std Dev	25.6	25.3	19.6	1531.9	1533.0	348.2	828.7	688.3	679.6

Table 1: Statistics of repair times for each domain by each generation method in each game.

Dataset	Cave			Mario			Supercat		
	SHAP	IG	UNI	SHAP	IG	UNI	SHAP	IG	UNI
Mean	2.9	2.9	2.8	5.1	3.6	1.1	5.2	5.7	4.7
Median	2.0	2.0	2.0	1.0	1.0	1.0	5.0	5.0	5.0
Std Dev	2.0	1.9	1.9	9.5	7.0	2.3	2.6	4.2	0.4

Table 2: Statistics of the number of changes made by each generation method in each game.

Thus, if i is true and j is false, both i and $\neg j$ will be true, and the constraint will not be satisfied; with other assignments of the i and j, it will be satisfied. For this constraint's row of A, the entry for i is set to 1, for j to -1, and the rest to 0; this constraint's entry in $\mathbf{v}$ is set to 0, and entry in $\mathbf{u}$ effectively set to $-\infty$ as there is no lower bound.

To construct constraints using larger numbers of literals, we introduce a few utility functions. These are functions based on the polarity of literals and list of literals:

$$\text{literal polarity: } \mathcal{P}(l) = \begin{cases} x_l & \text{if } l \text{ is positive} \\ -x_l & \text{if } l \text{ is negative} \end{cases}$$

$$\text{list polarity: } \quad \mathcal{P}(L) = \sum_{l \in L} \mathcal{P}(l)$$

and the negativity of literals and list of literals:

$$\text{literal negativity: } \mathcal{N}(l) = \begin{cases} 0 & \text{if } l \text{ is positive} \\ 1 & \text{if } l \text{ is negative} \end{cases}$$

$$\text{list negativity: } \quad \mathcal{N}(L) = \sum_{l \in L} \mathcal{N}(l)$$

For example with these functions, the above constraint could be written as:

$$\mathcal{P}(\langle i, \neg j \rangle) \leq 1 - \mathcal{N}(\langle i, \neg j \rangle)$$

For Sturgeon's mid-level API, there are several functions to be implemented using a combination of MILP variables (using $\mathbf{x}$), weights (using $\mathbf{c}$), and constraints (using A). The most straightforward are: MAKEVAR(), which simply creates a new numerical variable; SOLVE(), which calls the MILP solver; and GETVAR(v) and GETOBJECTIVE(), which access the results of the solver. Our current implementation uses the SciPy MILP solver (Virtanen et al. 2020; Huangfu and Hall 2018).

The first mid-level API function that requires additions to the MILP is MAKECONJ(L), which returns a representation of the conjunction (and) of the literals $l \in L$. MAKECONJ creates a new variable x_κ representing the conjunction, and adds a pair of linear constraints such that x_κ will be 1 if and only if all l are also 1:

$$|L|x_\kappa - \mathcal{P}(L) \leq \mathcal{N}(L)$$
$$-|L|x_\kappa + \mathcal{P}(L) \leq |L| - 1 - \mathcal{N}(L)$$

The next mid-level API function is CNSTRIMPLIES-DISJ(i, L, w), which adds a Boolean constraint that the literal i implies the disjunction (or) of the literals $l \in L$. This function optionally takes a weight w. If no weight is provided, the Boolean constraint is considered hard, and a linear constraint is added:
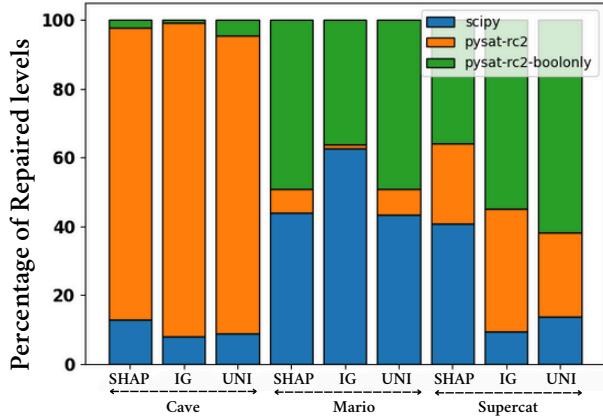
144

Figure 6: Comparison of Percentage of repaired levels by each solver between each weight generation method in each game. Our results show each game has a separate winning solver regardless of the method of weight generation.

$$\mathcal{P}(i) - \mathcal{P}(L) \leq -\mathcal{N}(i) + \mathcal{N}(L)$$

If a weight w is provided, the Boolean constraint is considered soft, and a new weighting variable x_ω is created and included in the constraint:

$$\mathcal{P}(i) - \mathcal{P}(L) - x_\omega \leq -\mathcal{N}(i) + \mathcal{N}(L)$$
$$c_\omega = w$$

Finally, the CNSTRCOUNT(L, a, b, w) adds a Boolean constraint that between a and b (inclusive) of the literals $l \in L$ are true. This function also optionally takes a weight, which, if not provided, treats the Boolean constraint as hard and adds the linear constraint:

$$a - \mathcal{N}(L) \leq \mathcal{P}(L) \leq b - \mathcal{N}(L)$$

If a weight w is provided, two new weighting variables x_α and x_β are created and included in the constraints:

$$a - \mathcal{N}(C) \leq \mathcal{P}(L) + |L|x_\alpha$$
$$\mathcal{P}(L) - |L|x_\beta \leq b - \mathcal{N}(L)$$
$$c_\alpha = w; \; c_\beta = w$$

Although there are two separate weighting variables, it should not be necessary to set both of them to 1 at the same time.

Experiments

Here we describe the experiments we ran to evaluate the pipeline and different components. As described above, the pipeline of level generation and level repair uses Sturgeon's impossible level generator to generate unsolvable levels by incorporating constraints that the level's goal is not reachable from the start (Cooper and Bazzaz 2024). The classifier then creates a grid of attribution values of the same size as the level passed into the weight generator, which translates the attributions to a grid of meaningful weights for repair via Sturgeon's weighted constraint solver.

The low-level solvers that are included in this study, `pysat-rc2`, `pysat-rc2-boolonly`, and `scipy`, accept positive integer weights. The solvers `pysat-rc2` and `pysat-rc2-boolonly` are Sturgeon's pre-existing PySAT RC2 solver (Ignatiev, Morgado, and Marques-Silva 2018; Ignatiev, Morgado, and Marques-Silva 2019); whereas `pysat-rc2-boolonly` uses only Boolean constraints by encoding cardinality constraints using the PySAT's `kmtotalizer` encoding (Morgado, Ignatiev, and Marques-Silva 2014), `pysat-rc2` uses cardinality constraints directly when possible through MiniCard (Liffiton and Maglalang 2012). The `scipy` solver uses the SciPy MILP solver (Virtanen et al. 2020), which is itself based on HiGHS (Huangfu and Hall 2018). The `scipy` MILP solver uses the problem formulation mentioned above.

We conducted experiments for 1000 levels across each domain. The pipeline initially generates a new unsolvable level, and then computes the attributions using Deep SHAP, Integrated Gradients, and a uniform attribution as a baseline. We refer to these three weight generation methods as *SHAP*, *IG*, and *UNI* in our figures and tables to simplify our notation and enhance readability. The three solvers run in parallel for each repair attempt, and the first successful solver terminates the others. We record the winning solver, the time to repair, the time to run the classifier for attributions and weights in the case of *SHAP* and *IG*, and the number of changes made to the original level. Figure 2 presents the attribution grids generated by each method alongside the corresponding repaired outputs.

Evaluation

We evaluated the performance of this level repair approach based on the duration that it takes to repair a level (including running an explainable classifier if used), the number of changes made to the level to be repaired, and the winning solver that repaired the level. Figure 5 shows the cumulative number of repaired levels in time. We used a time limit of 4000 seconds for each level to be repaired. Our results indicate that repair assistance with attributions can accelerate the repair process, particularly for larger and more complex game levels. The Cave levels, being generally easy and quick to repair, did not show as large an improvement with aided repair. On the other hand, Mario and Supercat levels, which are more challenging to repair, showed a reduction in repair time with the aid of attributions. Note that not all the 1000 has been repaired as some of the repairs were cut off by the time limit. Table 1 shows the corresponding detailed statistics of the repair times of different games with different methods, excluding the repair times of levels that were not completed due to the time limit.

We also examined the number of changes made to repair each level to determine if any method requires fewer changes or if any method is prone to making more unnecessary changes. Table 2 demonstrates no major difference in the number of changes among the methods. Across different games, the median number of changes is consistent, although the means and standard deviations vary. This might be due to the inclusion of different sets of levels, with some methods repairing levels not addressed by others.
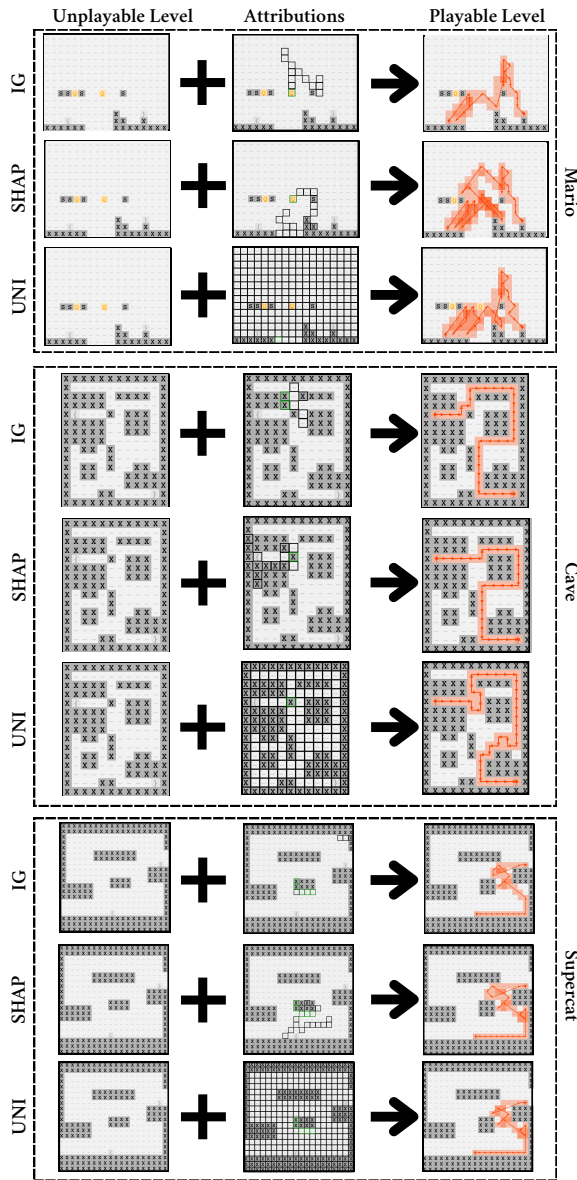
Figure 7: Example repaired levels of Mario, Cave, and Supercat. The black squares represent the lowest weights set for repair. The plotted path is just a path between the start and end of the level that the solver came up with during the repair (Not necessarily the shortest path). Sometimes the repaired level is the same between the methods and sometimes it is not.

Overall, the results indicate that the repaired levels of all three methods typically involve the same number of tile location changes, although in some cases the attribution-guided repair can result in more tiles being changed. Using attributions as weights for solver results leads to faster repairs without usually affecting the scope of the repair.

Lastly, we investigate which solvers are more successful in repairing the levels with the guide of attributions. As seen in Figure 6 `pysat-rc2` is the best-performing

solver for Cave levels, `scipy` excels in Mario levels, and `pysat-rc2-boolonly` performs best in Supercat levels. Interestingly, each solver appears to be particularly well-suited to a specific game. The results of Figure 5 and 6 highlight that this system serves as a guide for constraint solvers to improve the speed of level repair. Thus, the system's effectiveness is inherently tied to the overall performance of the solvers. While levels that these solvers can handle are repaired more quickly, those beyond their capability may remain with no solutions even with additional guidance.

Discussion

This work focuses exclusively on repairing levels using attribution values derived from an explainable solvability classifier. We see potential in leveraging attribution values from classifiers for other features, such as amusingness, linearity, and more, to generate new levels from existing ones.

It is important to note that to obtain valuable attribution values, it is crucial to use high-quality classifiers. Although training the classifier adds some overhead, this investment pays off by enabling faster level repairs. This approach could be particularly advantageous for larger, more complex segments, where solvers typically encounter the greatest difficulties. In these challenging scenarios, providing the solver with weighted inputs can offer significant benefits. While this project did not address working on larger levels, future research should focus on exploring this area.

For the sake of controllability, this study concentrated on the repair pipeline and employed Sturgeon's impossible level generator to produce unsolvable levels. We acknowledge that using Sturgeon in this pipeline is a current limitation. However, it provided a reliable method for running experiments across a large number of levels and evaluating the performance. We believe this approach can be extended to other level generators that produce undesirable outputs beyond solvability, such as local patterns, structures, and style.

Although this work utilized 2D grid-based levels, we believe the general approach can be applied to other repair contexts where an explainable classifier can attribute features that inform the repair process, such as edges on a mesh or notes in a melody.

Conclusion

Level repair using constraint solvers is a practice applied in Procedural Content Generation via Machine Learning (PCGML). However, these solvers often require significant time to complete the repair process. To address this issue, we propose the use of explainable solvability classifiers to expedite the repair process. By calculating the contribution of each tile location to the level's unsolvability, we assist the constraint solver with weighted inputs that correspond to these contributions. Our results indicate that this method reduces the repair time while typically changing same number of tiles from the levels.

Acknowledgments

References

Bazzaz, M.; and Cooper, S. 2023. Active Learning for Classifying 2D Grid-Based Level Completability. arxiv:2309.04367.

Bradski, G. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

Chen, E.; Sydora, C.; Burega, B.; Mahajan, A.; Abdullah, A.; Gallivan, M.; and Guzdial, M. 2020. Image-to-level: Generation and repair. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, 189–195.

Cooper, S. 2022a. Constraint-based 2d tile game blending in the sturgeon system. In *Proceedings of the Experimental AI in Games Workshop*.

Cooper, S. 2022b. 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.; and Bazzaz, M. 2024. Literally Unplayable: On Constraint-Based Generation of Uncompletable Levels. In *Proceedings of the 19th International Conference on the Foundations of Digital Games*, 1–8.

Cooper, S.; and Sarkar, A. 2020. Pathfinding Agents for Platformer Level Repair. In *AIIDE Workshops*.

Huangfu, Q.; and Hall, J. A. J. 2018. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1): 119–142.

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.

Kenney. 2022. Free game assets. https://www.kenney.nl/assets.

Kokhlikyan, N.; Miglani, V.; Martin, M.; Wang, E.; Alsallakh, B.; Reynolds, J.; Melnikov, A.; Kliushkina, N.; Araya, C.; Yan, S.; et al. 2020. Captum: A unified and generic model interpretability library for pytorch. *arXiv preprint arXiv:2009.07896*.

Liffiton, M. H.; and Maglalang, J. C. 2012. A cardinality solver: more expressive constraints for free. In Cimatti, A.; and Sebastiani, R., eds., *Theory and Applications of Satisfiability Testing – SAT 2012*, Lecture Notes in Computer Science, 485–486. Berlin, Heidelberg: Springer. ISBN 978-3-642-31612-8.

Lundberg, S. M.; and Lee, S.-I. 2017a. A Unified Approach to Interpreting Model Predictions. In Guyon, I.; Luxburg, U. V.; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Lundberg, S. M.; and Lee, S.-I. 2017b. A Unified Approach to Interpreting Model Predictions. In Guyon, I.; Luxburg, U. V.; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 30*, 4765–4774. Curran Associates, Inc.

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.

Mott, J.; Nandi, S.; and Zeller, L. 2019. Controllable and coherent level generation: A two-pronged approach. In *Experimental AI in games workshop*.

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

Nintendo. 1985. Super Mario Bros. Game [NES].

Rai, A. 2020. Explainable AI: From black box to glass box. *Journal of the Academy of Marketing Science*, 48: 137–141.

Shapley, L. S.; et al 1953. A value for n-person games.

Shrikumar, A.; Greenside, P.; Shcherbina, A.; and Kundaje, A. 2016. Not just a black box: Learning important features through propagating activation differences. *arXiv preprint arXiv:1605.01713*.

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.

Snodgrass, S.; and Ontañón, S. 2014. Experiments in map generation using Markov chains. In *International Conference on Foundations of Digital Games*.

Snodgrass, S.; and Ontañón, S. 2016. Controllable Procedural Content Generation via Constrained Multi-Dimensional Markov Chain Sampling. In *IJCAI*, 780–786.

Summerville, A.; and Mateas, M. 2016. Super mario as a string: Platformer level generation via lstms. *arXiv preprint arXiv:1603.00930*.

Summerville, A.; Philip, S.; and Mateas, M. 2015. Mcmcts pcg 4 smb: Monte carlo tree search to guide platformer level generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 11, 68–74.

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

Sundararajan, M.; Taly, A.; and Yan, Q. 2017. Axiomatic Attribution for Deep Networks. arXiv:1703.01365.

The SciPy community. 2024. scipy.optimize.milp. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.milp.html.

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.

Virtanen, P.; Gommers, R.; Oliphant, T. E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods*, 17(3): 261–272.

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.

Zhang, H.; Fontaine, M. C.; Hoover, A. K.; Togelius, J.; Dilkina, B. N.; and Nikolaidis, S. 2020. Video Game Level Repair via Mixed Integer Linear Programming. In *Artificial Intelligence and Interactive Digital Entertainment Conference*.

Zhu, J.; Awiszus, M.; Cook, M.; Dockhorn, A.; Eberhardinger, M.; Loiacono, D.; Lucas, S. M.; Matran-Fernandez, A.; Liebana, D. P.; Thompson, T.; et al. ???? 3.13 Explainable AI for Games. *Human-Game AI Interaction*, 73.