Visual Exploration of Tile Level Datasets

Seth Cooper[1], Faisal Abutarab[2], Emily Halina[2] and Nathan Sturtevant[2]

[1]Northeastern University, USA
[2]University of Alberta, Canada

Abstract

Datasets of game levels are increasingly used, for example as training data for PCGML algorithms, outputs from exhaustive PCG, or pre-computed parts of larger levels. However, tools for exploring these datasets are limited. In this work we describe an interactive tool for visual exploration of large tile level datasets. The level explorer currently supports level elements including both image- and text-based tile grids, path graph edges, and string tags. The level explorer visualizes the levels and allows the user to select elements that they would like to be present, and filters the dataset to find all levels consistent with the user's selection. We also describe an efficient encoding scheme used for level filtering, along with example cases of several games, demonstrating handling of datasets consisting of up to millions of levels.

Keywords

levels, datasets, visualization

1. Introduction

Many game AI algorithms are increasingly being built upon level datasets. This ranges from training data for Procedural Content Generation via Machine Learning (PCGML) [1], to the pre-computation of level segments or scenes for later use [2, 3]. Many PCG AI approaches also produce or analyze datasets of levels, such as results of exhaustive PCG [4] or expressive range analysis [5] and quality-diversity approaches [6].

Existing datasets include general repositories with levels from multiple games, such as the video game level corpus (VGLC) [7], as well as datasets for specific games such as the boxoban [8] dataset of Sokoban levels, The Windmill [9] database of levels for The Witness, or levels for Angry Birds [10].

However, tools that support visual understanding of such level datasets are limited. Thus, in this work we present an interactive tool for visualization and exploration of level datasets; the level explorer can scale to datasets that number into the hundreds of thousands or millions of levels. The level explorer has support for a variety of *level elements*: 2D grids of tiles containing images and/or text, path edges, and tags. The tool both visualizes these elements for levels in the dataset and allows interactive filtering based on levels containing selected elements.

While the level explorer is in its early stages, there are many potential uses. It could be useful for level designers to understand and explore levels in their games, as a supplementary visualization for expressive range analysis, or for the development of level sets for benchmarking algorithms. The level explorer itself could also be thought of as a level editor, where it is backed by levels that have been pre-generated and evaluated for particular properties. It may help address the "fundamental tension" of PCGML [11]: given a large enough training dataset, it may be practical to find desired levels in the dataset, as compared to training a generator.

2. Related Work

Recently, researchers have built level editors that use an AI system to support a mixed-initiative process in which human designers and the AI system work together to create levels [12, 13]. Although our tool was primarily designed to help designers explore datasets, it can also be considered a kind of level editor. The level explorer is similar to a number of existing mixed-initiative level editors. The mixed-initiative map editor Envoi relies on an Answer Set Programming (ASP) constraint-based generator that designers can modify in two ways: by adding rules (such as adjacency rules) on the layout of a map and by adding locked tiles that cannot be overwritten [14, 15]. Designers can press a button to generate a complete level that satisfies the designer-specified constraints. Similarly, our tool allows designers to filter a space of content (the input dataset), and then explore content from the filtered space. miWFC is a mixed-initiative editor that supports designers in using Wave Function Collapse (WFC) [16], a PCG technique closely related to constraint-based PCG methods [17]. The editor provides a "pencil" tool that allows designers to manually place tiles, and a side panel that shows the tiles possible for the cell the designer is

AIIDE Workshop on Experimental Artificial Intelligence in Games, October 08, 2023, University of Utah, Utah, USA
se.cooper@northeastern.edu (S. Cooper); abutarab@ualberta.ca (F. Abutarab); ehalina@ualberta.ca (E. Halina); nathanst@ualberta.ca (N. Sturtevant)
0000-0003-4504-0877 (S. Cooper); 0009-0004-3408-7092 (E. Halina); 0000-0003-4318-2791 (N. Sturtevant)

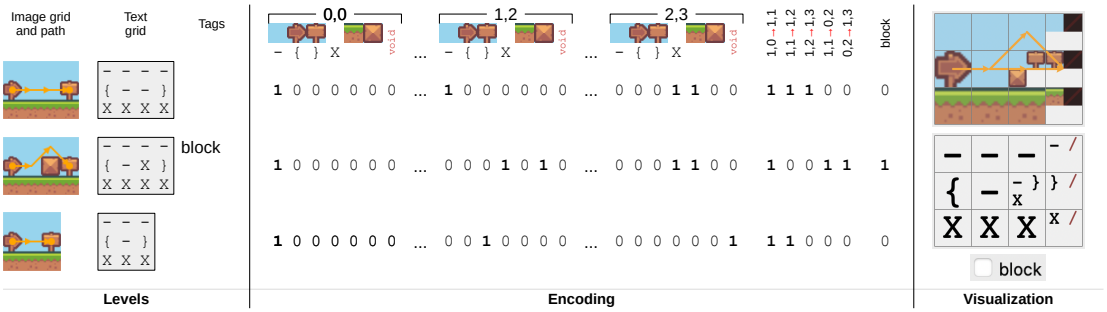


Figure 1: Example of encoding three small levels from a platformer game. The left side shows the levels; the middle, a summary of their encoded bit stings; and the right, how they would be visualized in the level explorer. Each location in the grid is encoded using 7 bits. There are 5 unique edges, each given their own bit, and the second level has a "block" tag, given 1 bit. As the third level is smaller than the others, it is padded with *void* tiles in the last column. Image tiles from Kenney [24].

Dataset	Levels	Rows×Cols	Bits	Tile	Edge	Tag	Encode	Filter	Source
`mario`	8,011,401	14 × 16	3082	2464 (11)	612	6	1.2h	4.57s	n-gram enumerated; VGLC 1-1 [7, 25]
`boxoban`	1,504,332	10 × 10	500	500 (5)	0	0	5.1m	0.25s	boxoban [8, 26]; Kenney [24]
`witness`	755,932	6 × 6	712	648 (18)	64	0	1.7m	0.15s	enumerated [27]
`cave`	36,000	15 × 15	7111	6525 (29)	586	0	37.3s	0.06s	generated [28]; Kenney [24]
`lode`	150	22 × 32	5632	5632 (8)	0	0	0.3s	0.03s	VGLC [7, 29]

Table 1
Summary of datasets. Tile bits shows total tile bits and, in parenthesis, per-location tile bits. Encode is time to encode, not including time to generate or convert level file formats (which in some cases could take considerably longer); average of three. Filter is time to filter based on a selection, also including time for encoding selection into bitstring and processing remaining bitstring for visualization; average of five after loading and the first filter (which was noticeably slower).

hovered over. Our tool also allows users to see which tiles are possible at each cell, but we present it in a different manner. The way we show what tiles are possible in each cell is influenced by Oskar Stålberg, who visualized WFC by displaying all tiles available in each cell [18]. The Anhinga level editor [19] uses exhaustive PCG to search though and evaluate all possible single-tile changes from the current level.

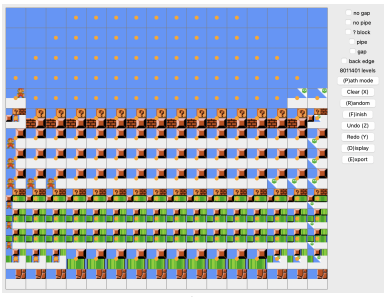
The level explorer also allows designers to filter levels by gameplay—that is, they can specify that levels must contain (partial) paths as a solution. This allows designers to work backwards from a path to a level design that can be solved by the specified full or partial path. Existing work has looked at generating levels from gameplay specifications (such as beat structure [20], streamlines that represent player traces [21], or taking path constraints as input to a generator [22]) and evolving gameplay specifications to create levels [23].

3. Description

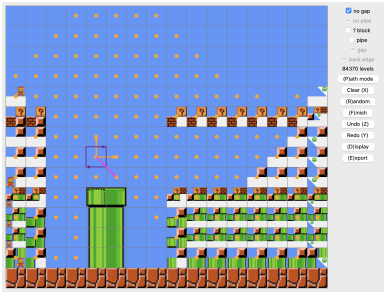
Here we give a general description of the level explorer, including level encoding, filtering, and visualization.

Level Encoding — For efficient filtering by level elements, levels are encoded as bitstrings. The encoding process takes all the levels to be encoded as input, as certain properties of the encoding need to know information about all levels (e.g. all the edges and tags used across all levels). The encoding process outputs the bitstring encoding of each level, as well as metadata needed to decode the bitstrings back into levels. An example of the encoding technique is shown in Figure 1, showing three levels, parts of their encodings, and how they would be visualized. The level representation is based on that used by the Sturgeon level generation system [28] but is flexible enough to support a variety of tile-based games. Level elements are encoded into the bitstring as follows.

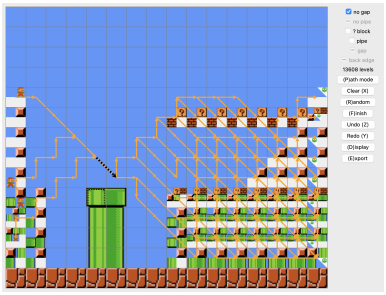
Tile grids: Levels are defined by a 2D grid of tiles. Each tile can be associated with an image and/or text "function" (e.g. start, goal, solid, passable, etc). During encoding, each location in the grid gets its own bit substring. In a location's substring, each bit can represent an image, text, or both. The encoding is done such that any tiles that have a unique image-text pair get both encoded in single bit, but tiles that share their image or text with another tile have each encoded as a separate bit. In the example, the sky image and - text only ever occur together, and thus are encoded together as a single bit. However, the X text can occur with either the ground or block images, and so each of these text or image possibilities gets its own bit (3 in all). Thus, when present in a level,



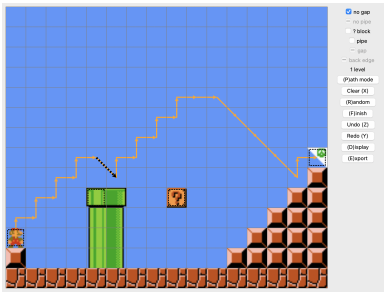
Initial view.



User selects a tile (pipe top-right) and the "No gap" tag; mouses over path edge (pink).



User selects edge, remaining edges appear.



User selects more tiles (Mario , flag , and ? block).

Figure 2: Sample `mario` interaction.

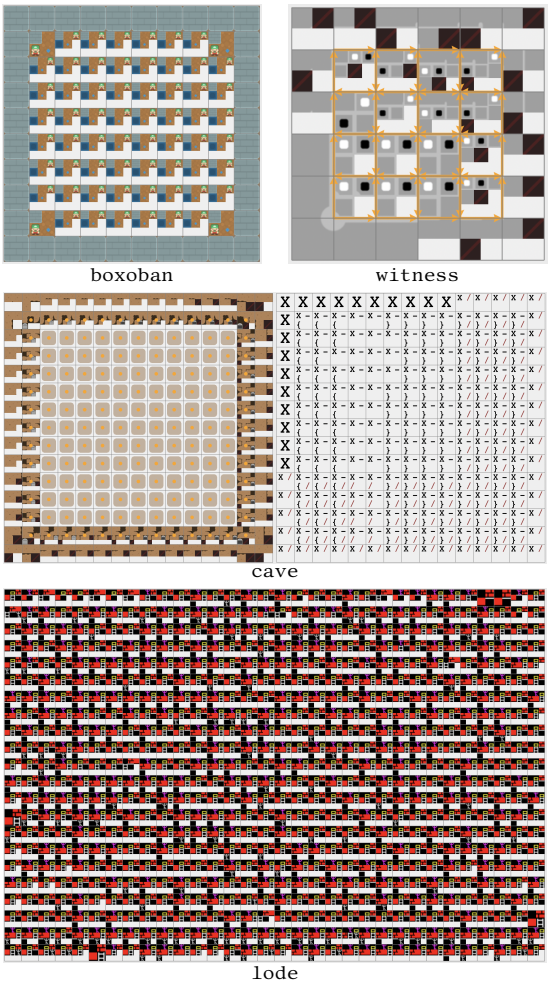


Figure 3: Initial views for different games. Notably, in `witness`, paths go along edges between tiles. In `cave`, both image and text tiles are displayed; the `X` text corresponds to several different images, and the image tiles and path edges are abstracted.

tiles encoded as a single bit will have that bit set to 1, and those that have their text and image in different bits will have both corresponding bits set to 1. Variable level sizes are handled by padding the grid with a special *void* tile, which gets its own bit.

Path edges: Levels can also be associated with paths. Paths are directed sequences of edges through the level grid (although they do not have to use integer coordinates), typically used to represent a possible solution to the level. During the encoding, unique edges across all levels are tracked. Each edge gets its own bit, set to 1 if present in the level.

Tags: Levels can also be tagged with strings. In the example, the second level was tagged with "block" since there is a block in it. Similar to edges, during encoding each unique tag gets its own bit, set to 1 if present in the level.

Level Filtering — Once levels have been encoded into bitstrings, the level explorer can filter them to select levels

that contain certain elements. An efficient implementation is built in Python using numpy [30].

The core of the filtering algorithm is shown in Algorithm 1. It takes as input a bitstring-encoded selection (sel) of elements to be present in the levels, along with the level bitstrings themselves (lev). First, all level bitstrings are bitwise AND-ed with, and compared to, the selection. This results in a Boolean array with True corresponding to levels that pass the filter ($rmnLev$). This array is then multiplied with the level bitstrings and the result is bitwise OR-ed together. This results in a bitstring with 1s set for elements that exist in any remaining level ($rmnBit$).

Level Visualization — The visualization displays the bitstring representing all the possible elements that remain. For example, a single location can have multiple tiles remaining. The level explorer visualization is interactive, and the basic form of interaction is using the mouse to select or de-select level elements, which launches a filtering based on the new selection and an update of the display. As filtering is not instantaneous for larger datasets, the computation happens in a thread, and recent selections are cached. Here we describe the basic approach to visualization.

Tile grids: At each location in the grid, all the possible tiles at that location are shown as a subgrid. Separate grids are displayed for image and text grids; however, if all the tiles are unique image-text pairs, only the image grid is shown by default. If there are too many tiles in one location, they are shown as an abstracted tile. If an abstracted location is moused over, all the tiles at that location are shown again so they can be selected.

Path edges: Directed edges are shown as arrows over the grid. If there are too many edges in the entire visualization to show at once, all edges are abstracted by only showing edge "source" positions as dots. If the mouse gets close to a dot, the edges coming out of that dot are shown.

Tags: Tags are shown as checkboxes in the interface. Checkboxes that do not correspond to remaining bits in the filtered levels are greyed out.

Additionally, the interface contains buttons to clear the selection, undo/redo, make a random selection, or finish the level by selecting a remaining level at random. The user can also change *modes* and *displays*, which impacts how tiles and edges are shown and selected. The filtered levels can also be exported.

As the user mouses over the level visualization, elements that would be selected are highlighted. The user's tile and edge selection is shown as dotted lines at the location of the tile or edge. For tiles, the lines are different depending on if it was directly selected by the user, indirectly selected by selecting a tile of another type (e.g. an image that is indirectly selected by the selection of text), or selected by finishing the level. The number of remaining levels is also shown.

Algorithm 1 Filtering level bit strings

In: sel: bitstring of selected elements
lev: array of level bitstrings
Out: $rmnLev$: array of remaining levels
$rmnBit$: bit string of remaining elements

$rmnLev \leftarrow mapEqual(mapAND(sel, lev), sel)$
$rmnBit \leftarrow reduceOR(rmnLev \times lev)$

4. Datasets

Here we describe some sample datasets we have used with the level explorer. A video of interactions with the datasets is available at https://osf.io/fcyjt/.

`mario`: Super Mario Bros. [25] levels generated by enumerating all possible via a 3-gram [31] based on level 1-1 from the VGLC [7], resulting in around 8 million levels. Paths through levels were added using A* pathfinding [32], and tags were added for the presence and absence of some tiles and level and path features.

`boxoban`: The boxoban [8] dataset of around 1.5 million Sokoban [26] levels; image tiles from Kenney [24].

`witness`: Black and white square puzzles based on The Witness [27]. Levels 4×4 and smaller, going from bottom-left to top-right, with exactly one solution were enumerated [4], resulting in around 750 thousand levels. In this dataset, path edges go along the borders between tiles. To make it look more like puzzles in the game, a border of "padding" tiles was added around each level.

`cave`: A custom cave exploration game. Levels were generated by Sturgeon [28], generating a thousand levels each with rows and columns ranging from 10–15, resulting in 36 thousand levels. Generated levels also provide a solution path, though not necessarily the shortest. The solid text tile `X` has a several image tiles depending on its location relative to other solid tiles. Image tiles from Kenney [24].

`lode`: As a smaller dataset, the VGLC [7] Lode Runner [29] dataset, consisting of 150 levels.

A sample interaction with the `mario` dataset is shown in Figure 2. Initial views for the other games are shown in Figure 3. A summary of the datasets, including time to encode and filter them during interaction, is provided in Table 1.

5. Discussion and Conclusion

Thus far we have implemented the level explorer, worked out some technical aspects of the user interface, and supported datasets into the millions of levels. When

interacting with the level explorer, we found that it can be useful for a quick visual summary of datasets; for example, what areas are reachable by paths, available pipe heights in `mario`, and the fact that there are never boxes in corners in `boxoban`. We found that in smaller datasets (i.e. `lode`) it is possible to very quickly end up with only one level remaining, e.g. when selecting player location. Thus a tool like this may be more useful when datasets are larger, or benefit from indicating how many remaining levels contain each element. In the current setup, the level explorer only determines which elements remain, but not how many remaining levels each element is in; though possible, we we found counting remaining levels for each element to be notably slower. We are also interested in support for more complex filters, such as the absence of elements. In the future, we are interested in a user study to understand the usefulness of the level explorer and how it compares to other tools.

Acknowledgements

We would like to thank Matthew Guzdial and Eugene Chen for their discussions and feedback on the project.

References

[1] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, J. Togelius, Procedural Content Generation via Machine Learning (PCGML), IEEE Transactions on Games 10 (2018) 257–270.

[2] M. C. Green, L. Mugrai, A. Khalifa, J. Togelius, Mario level generation from mechanics using scene stitching, arXiv:2002.02992 [cs] (2020).

[3] C. F. Biemer, S. Cooper, On linking level segments, in: 2022 IEEE Conference on Games (CoG), 2022, pp. 199–205.

[4] N. R. Sturtevant, M. J. Ota, Exhaustive and semi-exhaustive procedural content generation, AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment; Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference (2018).

[5] G. Smith, J. Whitehead, Analyzing the expressive range of a level generator, in: Proceedings of the 2010 Workshop on Procedural Content Generation in Games, PCGames '10, 2010, pp. 1–7.

[6] D. Gravina, A. Khalifa, A. Liapis, J. Togelius, G. N. Yannakakis, Procedural Content Generation through Quality Diversity, in: 2019 IEEE Conference on Games (CoG), 2019, pp. 1–8.

[7] A. J. Summerville, S. Snodgrass, M. Mateas, S. Ontañón, The VGLC: The Video Game Level Corpus, arXiv:1606.07487 [cs] (2016).

[8] A. Guez, M. Mirza, K. Gregor, R. Kabra, S. Racaniere, T. Weber, D. Raposo, A. Santoro, L. Orseau, T. Eccles, G. Wayne, D. Silver, T. Lillicrap, V. Valdes, An investigation of model-free planning: boxoban levels, https://github.com/deepmind/boxoban-levels/, 2018.

[9] thefifthmatt, The Windmill, https://windmill.thefifthmatt.com/, 2023.

[10] A. Zafar, S. Hassan, Q. S. uddin, Corpus for Angry Birds Level Generation, in: 2019 2nd International Conference on Computing, Mathematics and Engineering Technologies, 2019, pp. 1–4.

[11] I. Karth, A. M. Smith, Addressing the fundamental tension of PCGML with discriminative learning, in: Proceedings of the 14th International Conference on the Foundations of Digital Games, 2019, pp. 1–9.

[12] G. N. Yannakakis, A. Liapis, C. Alexopoulos, Mixed-initiative co-creativity, in: M. Mateas, T. Barnes, I. Bogost (Eds.), Proceedings of the 9th International Conference on the Foundations of Digital Games, 2014.

[13] G. Lai, F. F. Leymarie, W. Latham, On mixed-initiative content creation for video games, IEEE Transactions on Games 14 (2022) 543–557.

[14] D. Carpenter, J. T. Bacher, H. Crain, C. Martens, Casual creation of tile maps via authorable constraint-based generators, in: 1st Workshop on Programming Languages and Interactive Entertainment, 2021.

[15] H. Crain, D. Carpenter, C. Martens, Evaluating a casual procedural generation tool for tabletop role-playing game maps, in: 2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), IEEE, 2022, pp. 1–6.

[16] T. S. L. Langendam, R. Bidarra, miWFC - designer empowerment through mixed-initiative wave function collapse, in: Proceedings of the 17th International Conference on the Foundations of Digital Games, 2022, pp. 1–8.

[17] I. Karth, A. M. Smith, WaveFunctionCollapse is constraint solving in the wild, in: Proceedings of the 12th International Conference on the Foundations of Digital Games, 2017, pp. 1–10.

[18] O. Stålberg, Wave Function Collapse in Bad North, URL: https://www.youtube.com/watch?v=0bcZb-SsnrA, 2018.

[19] N. R. Sturtevant, N. Decroocq, A. Tripodi, C. Yang, M. Guzdial, A demonstration of Anhinga: A mixed-initiative epcg tool for snakebird, in: Artificial Intelligence and Interactive Digital Entertainment, 2020, pp. 328–330.

[20] G. Smith, J. Whitehead, M. Mateas, Tanagra: A mixed-initiative level design tool, in: Proceedings of the Fifth International Conference on the Foundations of Digital Games, 2010, pp. 209–216.

[21] L. N. Ferreira, Streamlevels: Using visualization to generate platform levels, ACM Computers in Entertainment (2015).
[22] S. Cooper, M. Guzdial, path2level: Constraint-based level generation from paths, in: 2023 IEEE Conference on Games (CoG), 2023.
[23] D. Karavolos, A. Liapis, G. N. Yannakakis, Evolving missions to create game spaces, in: 2016 IEEE Conference on Computational Intelligence and Games, IEEE, 2016, pp. 1–8.
[24] Kenney, Free game assets, https://www.kenney.nl/assets, 2022.
[25] Nintendo, Super Mario Bros., 1985. Game [NES].
[26] Thinking Rabbit, Sokoban, 1928. Game.
[27] Thekla, Inc., The Witness, 2016. Game.
[28] S. Cooper, Sturgeon: Tile-based procedural level generation via learned and designed constraints, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 18 (2022) 26–36.
[29] D. Smith, Lode Runner, 1983. Game.
[30] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant, Array programming with NumPy, Nature 585 (2020) 357–362.
[31] S. Dahlskog, J. Togelius, M. J. Nelson, Linear levels through n-grams, in: Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services, 2014, pp. 200–206.
[32] A. J. Summerville, S. Philip, M. Mateas, MCMCTS PCG 4 SMB: Monte Carlo tree search to guide platformer level generation, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (2015).