Proceedings of the Nineteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2023)

Tree-Based Reconstructive Partitioning: A Novel Low-Data Level Generation Approach

Emily Halina, Matthew Guzdial

Department of Computing Science, Alberta Machine Intelligence Institute (Amii)
University of Alberta, Edmonton, Alberta, Canada
ehalina@ualberta.ca, guzdial@ualberta.ca

Abstract

Procedural Content Generation (PCG) is the algorithmic generation of content, often applied to games. PCG and PCG via Machine Learning (PCGML) have appeared in published games. However, it can prove difficult to apply these approaches in the early stages of an in-development game. PCG requires expertise in representing designer notions of quality in rules or functions, and PCGML typically requires significant training data, which may not be available early in development. In this paper, we introduce Tree-based Reconstructive Partitioning (TRP), a novel PCGML approach aimed to address this problem. Our results, across two domains, demonstrate that TRP produces levels that are more playable and coherent, and that the approach is more generalizable with less training data. We consider TRP to be a promising new approach that can afford the introduction of PCGML into the early stages of game development without requiring human expertise or significant training data.

1 Introduction

Procedural Content Generation (PCG) refers to the algorithmic generation of video game content (Shaker, Togelius, and Nelson 2016). By utilizing human-authored examples and heuristics, PCG approaches can generate many types of game content, from 3D models of trees (Interactive Data Visualization 2023) to entire game worlds (Xbox Game Studios 2011). In particular, the task of level generation has been well explored across multiple games, including platformers (Awiszus, Schubert, and Rosenhahn 2020), dungeon crawlers (Van Der Linden, Lopes, and Bidarra 2013), and puzzle games (Guzdial, Sturtevant, and Yang 2021). Many games have drawn on PCG to generate level structure live within a game or offline during development. However, many games do not employ PCG, or have found it negatively impacts development (Schreier 2017).

A major challenge developers face in integrating PCG approaches into game development is generating quality content with minimal human authoring. The human-authoring required for PCG approaches ranges from example levels for Wave Function Collapse (WFC), to components and rules for constructive systems, or functions for Search-Based PCG (SBPCG). The pace of game development is blistering, with rapid ideation and prototyping often leading to major changes in game balance and mechanics throughout the development process, potentially requiring frequent re-authoring of these human authored components (Ramadan and Widyani 2013). This makes more human-authoring-intensive PCG approaches a burden, potentially sinking some game development projects (Schreier 2017). An ideal system would generate quality, novel content with minimal human-authoring required.

While there have been a number of PCG methods which can generate levels using minimal human-authored examples, these methods may not be generally appropriate for active game development. WFC can generate content from only a single human-authored example, but struggles with generating levels that adhere to global constraints (Karth and Smith 2017). This makes it difficult to ensure WFC creates playable or soundly constructed levels, especially for game domains which have multiple requirements for playability, such as requiring both a key and door in a level. A potential workaround to this problem is to hand-author these constraints, as in Sturgeon (Cooper 2022). However, frequent re-authoring of these constraints may be burdensome, and designers may struggle to express their design goals in the language of constraints. Similarly, SBPCG methods utilize functions which are faced with the same issue of requiring re-authoring, as evidenced by the lack of mainstream games using SBPCG in their development.

In this paper, we introduce a new PCG method based on Monte Carlo Tree Search (MCTS) and Space Partitioning that can generate a suite of levels from as little as a single human-authored example. On a high level, our approach works by first extracting information from the search tree of one or more MCTS playthroughs of a source level. Using this information, we then reconstruct an intermediary binary representation of the level approximating the source geometry. Finally, we utilize both the source level and the MCTS search tree to perform a binary space partition and probabilistic threat placement to complete the level. We dub this new method **Tree-based Reconstructive Partitioning** (**TRP**) after these three high-level steps. Besides the single human-authored level example, TRP requires access to a forward model of the game and a small amount of domain-specific knowledge provided by a human designer. Notably,

TRP's domain-specific knowledge is not a value or heuristic function, in which a human designer must determine how to evaluate the goodness of a level, but instead a set of affordances and ranges which constrain generation. As such, TRP could feasibly be used in scenarios including early development when training data is scarce.

The contributions of this work to the game AI community are as follows:

- Tree-based reconstructive partitioning (TRP), a novel PCG method that generates valid, unique levels from a single example.
- An empirical evaluation of TRP's performance in comparison to six baseline PCG approaches across two disparate game domains.
- A brief discussion of the potential extensions and applications of TRP to a variety of problem domains.

2 Related Work

In this section, we overview prior work on PCG in the context of level generation, and provide a background primer on MCTS and its prior use in other PCG approaches.

2.1 PCG for Level Generation

PCG approaches can be grouped into categories with similar methodologies. One such category is classical PCG, approaches that do not use machine learning, including constructive, search-based, and constraint-based PCG (Charity et al. 2020; Horswill 2021). These methods require a human-authored notion of goodness to guide generation, such as rules or components for constructive PCG, functions for SBPCG, and constraints for constraint-based PCG. While some of these approaches are commonly used within completed games running PCG live, these requirements make them difficult to develop from scratch during early game development (Schreier 2017).

Procedural Content Generation via Machine Learning (PCGML) refers to the family of PCG approaches that utilize machine learning techniques (Summerville et al. 2018). While the advent of deep learning has brought about many powerful PCGML approaches, we focus specifically on approaches that require a small amount of data due to our specific use-case. One such approach is Wave Function Collapse (WFC), which can use patterns from a single example level to constrain generation (Karth and Smith 2017). WFC, as previously stated, can struggle to adhere to global constraints such as playability without potentially intensive human-authored assistance (Cooper 2022). Other PCGML approaches can run with minimal training data such as Markov Chains, Long Short-Term Memory deep neural networks (LSTM), and Generative Adversarial Networks (GAN) (Snodgrass and Ontanón 2013; Summerville et al. 2018; Awiszus, Schubert, and Rosenhahn 2020). However, prior work shows that these methods can heavily plagiarize when run with few examples, with the expressive volume of the models limited by the sample size (Snodgrass, Summerville, and Ontañón 2017).

Procedural Content Generation via Reinforcement Learning (PCGRL) is an emerging group of approaches that utilize reinforcement learning to generate game content (Khalifa et al. 2020). These approaches work by formulating level generation as a Markov Decision Process (MDP), making incremental changes to a piece of content, and have been successfully applied to multiple domains (Earle et al. 2021). However, similarly to search-based methods, PCGRL approaches require a reward function, with the same limitations as SBPCG heuristics. In addition, small changes to the game may require a complete reformulation of the MDP, which requires a large amount of expertise in reinforcement learning that game designers may lack (OpenAI et al. 2019). While MCTS is an example of model-based RL, we don't categorize our approach under PCGRL, as MCTS doesn't directly alter level structure.

2.2 Paths and Trees in PCG

Prior PCG approaches have made use of paths and trees for data augmentation and as a part of the level generation process. Exhaustive PCG exhaustively explores a search tree of possible designs for a level (Guzdial, Sturtevant, and Yang 2021). This is distinct from our approach, which uses a search tree from a playthrough of the level as input rather than as a part of the generation process. Existing PCG approaches use paths for a variety of tasks, including learning level topology and performing level blending across game domains (Summerville et al. 2015; Sarkar et al. 2020). Our approach differs in that we utilize an entire search-tree, including branches which ended in failure states.

2.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a stochastic tree search algorithm that simulates future actions to find the best next choice (Browne et al. 2012). MCTS has proven to be a strong approach for automated game playing, both alone and embedded within larger systems (Chaslot et al. 2008; Fu 2016). In the context of game playing, MCTS works by building a tree of possible future game states, with nodes represents game states and edges representing actions taken by the player. Our approach uses this tree as input for the generative process. Over each iteration, MCTS incrementally evaluates the value of each node representing a future state, which in turn can be used to infer a good next action for the current state. This process can go on for as many or as few iterations as desired, allowing for a balance between performance and computational requirements. Each iteration of MCTS is made up of the following four steps.

Selection From the root node, we successively select next nodes. We make this selection based on a tree policy function, which guides where next to explore. We stop when we reach a node with an unexpanded child. This tree policy guides how MCTS iteratively builds the tree from the root, balancing exploration of new states versus exploitation of known good states.

Expansion We randomly select an unexpanded child of the selected node from the Selection step.

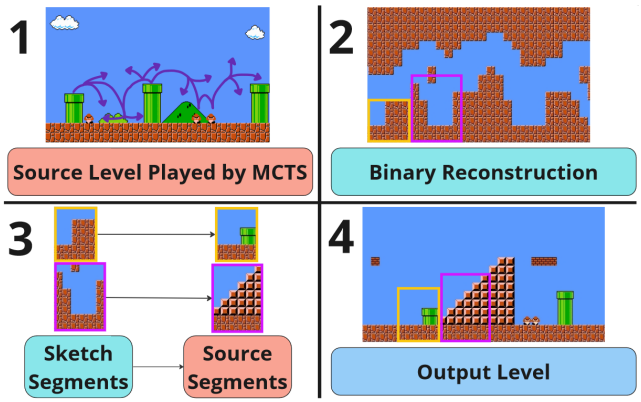


Figure 1: Visualization of the TRP generation pipeline. With MCTS search trees and designer knowledge kit as input, TRP constructs an intermediate binary representation. TRP then uses the input data to perform a binary space partition and probablistic threat placement, producing the final level.

Simulation The purpose of the simulation step is to simulate the long-term result of the game from the state of the unexpanded child. In our case, we simulate a playthrough beginning from this unexpanded child, taking random actions in every step. This playthrough continues until either a pre-defined number of actions are taken, or we reach a terminal state, such as a win, loss, or draw.

Backpropagation We evaluate the value of the final state we reach in the random playout, then backpropagate this value back up the tree to each parent until we reach the root.

MCTS in PCG While MCTS has been incorporated into some prior PCG systems, it has mostly been used to directly guide the generation process, such as in Summerville et al.'s MCMCTS where MCTS places columns of existing levels to create new levels (Summerville, Philip, and Mateas 2015; Graves et al. 2016). MCTS has also been used as part of a heuristic in SBPCG approaches (Sorochan and Guzdial 2022). This is in contrast to our approach, which uses MCTS to play through a source level to derive a tree that serves as input to a generation process.

3 Tree-based Reconstructive Partitioning

In this section we discuss the implementation of our approach, Tree-based Reconstructive Partitioning (TRP), in terms of a generic, token-based game domain. Figure 1 depicts our level generation pipeline from the original source level to our generated level as output. As input, TRP takes in the source level and a "knowledge kit" of domain-specific human-authored information. This information, described in full below, takes the form of affordances, ranges, and sets of tokens, such as a group of tokens that should be considered "threats." First, we perform one or more playthroughs of the source level with a tree search algorithm, saving the search trees from these playthroughs. In this paper, we use MCTS. We then convert the output search trees to an intermediary binary representation. Finally, we perform a binary space partition using this intermediary representation to generate new level structure, and use the position of deaths in the search tree to populate our new level with threats.

3.1 Application Requirements

TRP assumes a discrete token-based representation of the game domain's levels, as a discrete representation is required to perform binary space partition. Further, TRP requires access to a forward model of the game and a game-playing heuristic in order to collect the search tree information. While this is a somewhat burdensome requirement, we speculate that the use of an engine or plugin that automatically sets up a forward model for the developers could alleviate the authoring work required to add this functionality.

3.2 Tree-based Reconstruction

As input, TRP requires one or more search trees representing potential paths through a game level. Thus, we begin by conducting one or more playthroughs through the source level using MCTS to guide the player. We chose MCTS due to its "just-in-time" nature which allows the algorithm to perform well on a variety of hardware strengths. This is a relevant concern for developers who may not have access to a large amount of computing power. Further, the stochastic nature of MCTS allows us to explore different possibilities throughout different playthroughs, i.e., the player character taking a different choice in a branching path. However, any tree search algorithm could be used for this stage, as long as it outputs the required search tree information.

For both of the domains in this paper, we used the UCT selection policy, where the value of the i-th node is given by

$$v_i + c\sqrt{\frac{\ln N_i}{n_i}} \tag{1}$$

where v_i is the average reward value, c is a tunable constant, and N_i and n_i are the number of visits to the node's parent and itself respectively. The reward value v_i of a given state is domain-specific, as it is dependent on the designer's description of goal states, as discussed in subsection 3.4.

Throughout the playthroughs, we retain two key pieces of information: the position of the player at each timestep, and the position and cause of any failure states (i.e. death) within any of the rollouts. We utilize the player positions across all the playthroughs to construct an intermediary binary representation of tokens that represent whether there is structure at a position or not. Figure 1 contains examples of this intermediary representation of the combined paths of each search tree through a source level. The number of playthroughs performed through the level is another parameter users can change to affect the generated artifacts, as it impacts this intermediary representation. The intention behind using the player's positions to generate a level is to roughly approximate the geometry of the target level while finding the *most important* required elements to progress. If we can generate a level with a similar path, we hypothesize that the design intention that afforded that path in the original level may be captured in the new level. This intermediary representation is used in the following steps of generation alongside the source level in order to create the final output.

3.3 Binary Space Partitioning

We fill in the intermediary binary reconstruction of the source level in an attempt to aesthetically match the content of the source level while retaining the extracted structure from the trees' paths. To do this, we use an example-driven binary space partition adapted from Snodgrass' implementation (Snodgrass 2019). Binary space partition is an algorithm for constructing levels by filling in a binary "sketch" of a level with corresponding pieces of a source level. For our use-case, we use the intermediary reconstruction as the binary sketch, and the original level stripped of all threats as the source level. We chose binary space partition for this task due to its ability to retain structural information from the reconstruction while replicating patterns from the original source level, such as complete pipes in Super Mario Bros.

Binary space partitioning works by first dividing the binary sketch into segments of size $w \times h$. w and h vary in size, but are bounded by a parameter s that limits their maximum value. Next, each of these segments are "matched up" with corresponding segments from the source level by a binary similarity function which counts the number of similarities between the two. This function is given by

$$\sum_{(i,j) \in |S_{n \times m}|} 1 - \text{sign}^2(S[i,j] - L[i,j]) \tag{2}$$

where S and L are $n \times m$ segments of level, with $S[i,j] = 0$ if the token at the (i,j)-th position of S is an empty token and $S[i,j] = 1$ otherwise. Sign is the sign function, which outputs 1 and -1 for positive and negative numbers respectively, and 0 otherwise. We use a sliding window across the source level to find all of the closest matching segments of the same size, then randomly select one of the resulting matches with uniform probability. These matches "fill in" the details of the sketch one at a time until all of the matches are processed. As an example using $s = 5$, if a binary sketch was divided into 4x3 and 2x5 segments, the algorithm would first examine all 4x3 segments of the source level, matching the closest one via Equation 2. Then, the algorithm would repeat this process for the 2x5 segments.

After completing the partitioning, our final step is to repopulate our output level with threats. Our threat repopulation methodology leverages the information from the collected search trees to identify the most relevant threats in the level, then places those threats in order from most to least relevant to reach a designer-set desired difficulty threshold. Relevance is determined through the formula $\frac{d_e}{d_t}$, where d_e represents the number of deaths caused by a threat e in a given position with respect to all rollouts, and d_t is the number of total deaths across all nodes.

The identity of a threat is domain-specific and the difficulty threshold is an adjustable parameter, as explained in subsection 3.4 below. Threats are placed directly within the corresponding position of the failure state in the search tree. We chose this methodology in order to maximize the impact and intentionality of the placement of threats by the system in an attempt to replicate the patterns of threat placements within the original level. By placing only the most relevant threats for a given level, we hope to mimic the designer's original intentions with threat placement and create levels which provide the desired level of challenge without overpopulating threats throughout the level.

3.4 Knowledge Kit

TRP's required human-authoring takes the form of a set of human-authored affordances and ranges. This information is used to guide the tree search to play through the game, and help with the categorization and repopulation of threats. These parameters include three affordances regarding the game's states and entities, two numerical parameters for MCTS, and three numerical parameters for TRP which affect generation.

The three human-authored affordances are the definition of a sequence of goal states $G = \{S_{g_1}, \ldots, S_{g_k}\}$, a sequence of failure states $F = \{S_{f_1}, \ldots, S_{f_k}\}$ and a list of tokens which represent threats within the game domain. The first of these guides the tree search agent through the level playthroughs, as the value of states relies on their proximity to the current goal. Designers can provide an arbitrary number of ordered goal definitions, such as first collecting a key and then reaching a locked door, which would be encoded as a sequence of two goal states $\{S_{g_k}, S_{g_d}\}$. In both domains, we computed reward value based on the Manhattan distance from the closest current goal state to improve the MCTS agent's performance. Similarly, the failure states corresponding to the current goal state determines negative reward signal for the MCTS agent. The list of threat tokens is used to identify the causes of failure states during the threat repopulation phase based on token proximity. Notably, these requirements are fundamentally different from the requirements of SBPCG systems, as we require designers to provide information about game states and entities instead of level quality.

There are two numerical parameters for MCTS that designers can adjust from their default values to improve both computational performance and the strength of the agent as a player. These are the tuneable exploration constant c defined in subsection 3.2, and the maximum depth for rollouts in the Simulation step. The theoretically optimal value for c is $\sqrt{2}$ (Kocsis and Szepesvári 2006). However, adjusting it may improve performance in some domains, allowing the agent to play more complex or challenging levels.

By default, the maximum depth for rollouts is unbounded (i.e. a rollout will continue until a terminal state is reached), but this may not be appropriate for more complicated domains where it may take hundreds or thousands of steps to reach a terminal state. Instead, by bounding the depth of rollouts, we can force the MCTS agent to optimize more locally, which can improve performance in some domains (Zook, Harrison, and Riedl 2015). While these parameters do require tuning in order to maximize the potential of the MCTS agent, we found the set of reasonable parameters to be quite similar across both our evaluation domains. Further, in a potential designer-focused TRP tool, these parameters could be automatically adjusted and/or discovered via a parameter optimization approach such as a grid search.

TRP takes three numerical parameters which allow human designers to control the generation process. The first,

$t \in \mathbb{N}$, denotes the number of playthroughs TRP will perform of the source level. This count affects the openness of the intermediary level, with more playthroughs opening up more options and identifying more paths in the source level. The second, $s \in \mathbb{N}$, denotes the maximum allowed size of any segment taken from the source level during the binary space partition, affecting the amount of direct plagiarism tolerated in the output level. The third, $e \in [0.0, 1.0]$ denotes the weighted percentage of threats to take from the search tree as explained above. This affects the resulting threat density of the output level, with 0.0 representing no threats, and 1.0 representing the maximum density of threats.

While TRP is usable with fixed parameters, having these parameters vary over a range dramatically increases the diversity of the generated levels. As such, for each domain, we present results with both a fixed set of parameters and a predefined range, with the knowledge kits for both of our evaluation domains provided in the following section. For both of our domains, we determined the parameter values by our own human judgement of the generated levels. Due to the speed of generation, it was trivial to determine reasonable values for both domains, taking minutes of experimentation.

4 Domains

In this section, we discuss our implementation of TRP within two game domains: Super Mario Bros. and GVGAI Zelda (Khalifa 2022; Perez-Liebana et al. 2016). We chose Super Mario Bros. as a domain as it is a ubiquitous standard for PCG approaches (Summerville et al. 2018). We chose GVGAI Zelda as it directly contrasted Super Mario Bros. in gameplay and generation requirements, and represented a domain in which levels are encoded in very few tokens. In the following subsections, we discuss the implementation details of each domain and provide the knowledge kit of human-authored information used to generate levels for them. We provide the source code for each of these implementations[1], including scripts for generating new levels from prior MCTS playthroughs.

4.1 Super Mario Bros.

Super Mario Bros. is a platformer game originally released for the NES in 1983 (Miyamoto, Yamauchi, and Tezuka 1985). The player starts on the far left of each level, and the goal is to navigate to a flagpole at the end of a stage using jumps, movement, and various powerups. An example of a Super Mario Bros. level is depicted in Figure 2. Our implementation uses version 0.80 of the updated Mario AI Framework, a Java reimplementation of the original Super Mario Bros. (Khalifa 2022). Each level is encoded as an $n \times 16$ matrix of tokens within a text file, with each token representing a singular tile or block in the game. The framework includes a forward model, which we used to implement the MCTS agent required by our approach.

Super Mario Bros. Knowledge Kit We defined the goal state in Super Mario Bros. to be when Mario's x-coordinate is the same as the flagpole's. Notably, this goal state may correspond to several possible world states. We defined the failure states as states in which Mario is in contact with an enemy or enemy projectile, or is one tile below the screen, which represents Mario falling in a pit. Using these two sets and the previously defined scheme of measuring the Manhattan distance from the goal, the value function guiding MCTS scores nodes based on how far to the right Mario can reach. This is similar to Jacobsen et al.'s prior work on implementing MCTS for Super Mario Bros. (Jacobsen, Greve, and Togelius 2014). The set of threat tokens were simply any token that is associated with an enemy, such as Goombas, as well as any air tile which is at the very bottom of the screen, which represents the bottom of a pit. Notably, we were able to perform this encoding by swapping out the bottom tile for another special token in a pre-processing step. This gave us a total of 17 threat tokens in this domain.

For MCTS parameters, we used an exploration constant value of $c = 0.25$, and a maximum rollout depth of 12. These values are based on both prior work on MCTS controllers for Super Mario Bros., and our own empirical evaluation (Jacobsen, Greve, and Togelius 2014). With larger rollout depths, Mario becomes noticeably "cowardly," avoiding all threats, and as such we keep the depth relatively low.

The parameters for the fixed and variable versions of TRP are as follows. The fixed version uses parameter values

$$(t = 2, s = 9, e = 0.67) \tag{3}$$

and the variable version uses parameter ranges

$$t \in \{1, 2, 3, 4, 5\} \tag{4}$$

$$s \in \{8, 10, 12, 14, 16\} \tag{5}$$

$$e \in \{0, 0.33, 0.67, 1\} \tag{6}$$

with t, s, e defined as above. We chose 1 through 5 as a range for playthroughs (t), as we found including more than 5 playthroughs began to show diminishing returns in terms of exploration of new paths. We chose the values of s based on the width of a 16×16 chunk of screen, with 8 representing half a row or column, and 16 representing a full row or column. We selected the values of e to create levels of varying difficulty.

4.2 GVGAI Zelda

The General Video Game AI (GVGAI) framework is a research framework mainly used for games research competitions (Perez-Liebana et al. 2016). GVGAI is made up of a corpus of over 100 single-player games, including GVGAI Zelda. GVGAI Zelda is a top down puzzle game where the player navigates a tile-based environment full of enemies with orthogonal movement. The game's goal is to collect a key then reach a door in the level. Examples of GVGAI Zelda levels are depicted in Figure 3. We utilize the OpenAI Gym implementation of GVGAI which acts as a Python wrapper for the Java implementation of GVGAI (Torrado et al. 2018). Each level is encoded as an 13×9 matrix of tokens within a text-file, with each token representing a singular tile or block in the game. As the Gym implementation did not contain a forward model by default, we implemented our own which can be found in the provided source code.

[1] https://github.com/emily-halina/TRP/

GVGAI Zelda Knowledge Kit We defined the sequence of goal states in GVGAI Zelda as $\{S_k, S_g\}$, which encodes the required steps of first obtaining a key, and then reaching the door in order to complete a level. We defined the failure state as any state in which the player has the same x and y position as an enemy. The set of tokens representing threats consists of the three enemy types within the game, which differ by moving randomly at different frequencies and intervals. This gave us a total of 3 threat tokens in this domain.

For MCTS paramaters, we used an exploration constant $c = 0.25$, and a maximum rollout depth of 3. These choices were informed by our own empirical evaluation of the performance of the agent. While 3 may seem like a relatively small depth for exploration, the randomized movement of enemies forces the agent to make decisions local to its immediate surroundings.

The parameters used for the fixed and variable versions of TRP are as follows. The fixed parameter values

$$(t = 1, s = 2, e = 0.25) \tag{7}$$

and the variable version uses parameter ranges

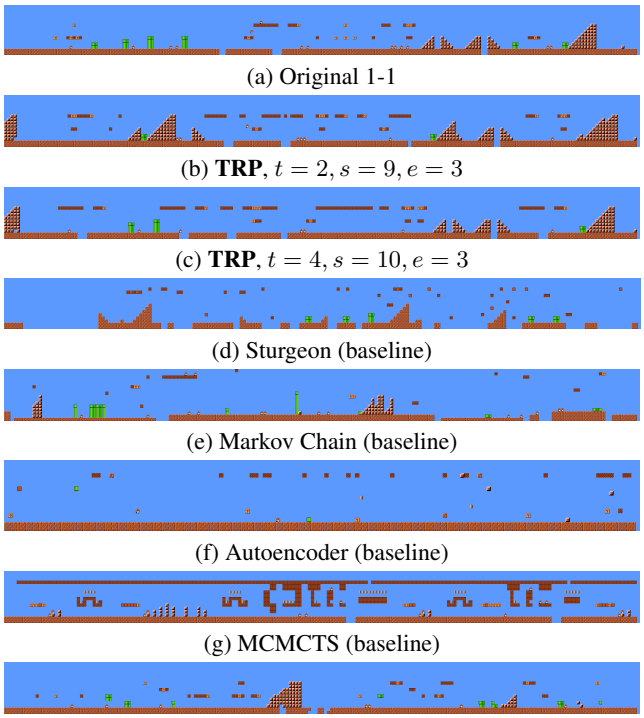
$$t \in \{1, 2, 3\} \tag{8}$$
$$s \in \{1, 2, 3, 4, 5\} \tag{9}$$
$$e \in \{0, 0.25, 0.375, 0.5\} \tag{10}$$

with t, s, e defined as above. As before, we chose the range of playthroughs (t) based on observed diminishing returns from including more playthroughs. For this domain, we found that smaller values for s performed best, with larger values than 5 resulting in complete plagiarism of the source level. The set of values for e is lower in comparison to Super Mario Bros., as the density of deaths is much higher in this domain.

5 Evaluation

In this section, we discuss the evaluation of our approach in the previously defined game domains, overviewing the baselines and evaluation metrics. Recall that our goal is to address the problem of producing high-quality content with minimal human-authoring. We could have evaluated our approach through a human subject study with game designers either directly using TRP in practice or evaluating generated level content. However, as this paper is an initial foray into TRP as an approach, our goal is to perform an exhaustive analysis of the approach that would not be appropriate for a human subject study.

Instead, to evaluate our approach we compared levels generated by TRP to levels generated by a number of baseline approaches across both game domains. These baselines were Markov Chains, Markov Chain Monte Carlo Tree Search (MCMCTS), WFC, Sturgeon, an autoencoder-based PCGML approach, and TOAD-GAN. We had two levels from each domain as human-authored training data, namely Level 1-1 and 1-2 from Super Mario Bros. and Level 1 and 2 from GVGAI Zelda. Notably, we do not include a SBPCG or PCGRL baseline, as the required authoring of level quality functions makes it difficult to compare these approaches to those reliant on only a handful of human-authored examples.



(a) Original 1-1

(b) **TRP**, $t = 2, s = 9, e = 3$

(c) **TRP**, $t = 4, s = 10, e = 3$

(d) Sturgeon (baseline)

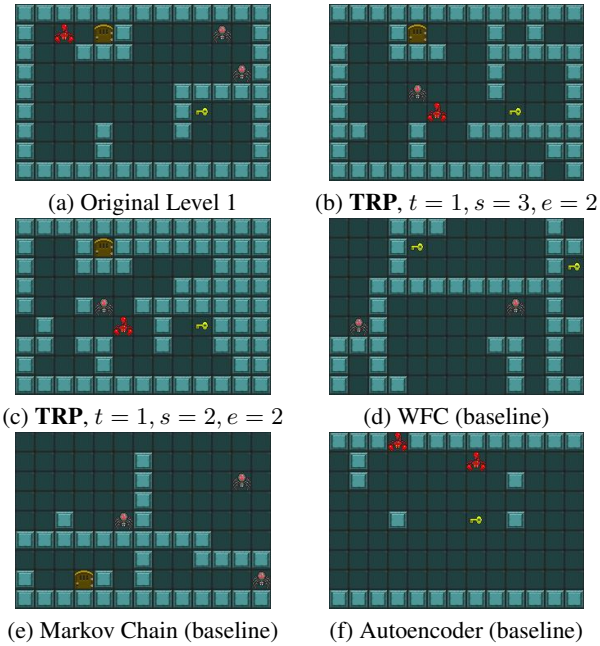
(e) Markov Chain (baseline)

(f) Autoencoder (baseline)

(g) MCMCTS (baseline)

(h) TOAD-GAN (baseline)

Figure 2: Randomly selected levels generated by TRP and baselines for Super Mario Bros. using Level 1-1 as training data, except Autoencoder and MCMCTS which use 1-1 and 1-2. For more examples, see the Github repository.



(a) Original Level 1

(b) **TRP**, $t = 1, s = 3, e = 2$

(c) **TRP**, $t = 1, s = 2, e = 2$

(d) WFC (baseline)

(e) Markov Chain (baseline)

(f) Autoencoder (baseline)

Figure 3: Randomly selected levels generated by TRP and baselines for GVGAI Zelda using Level 1 as training data, except Autoencoder which uses Levels 1 and 2.

The implementation details for each baseline are outlined in the Baselines subsection.

For comparative metrics, we employed playability, plagiarism of the original source levels, and self-similarity between the population of generated levels. These metrics were chosen as stand-ins for a human evaluation of quality due to their approximation of the validity, originality, and variety of generated levels respectively. The details of each metric are outlined in the Metrics subsection.

5.1 Baselines

Markov Chain (MC) Markov chains are a classic PCGML approach that involves using local context in order to probablistically determine which tile to place in a given location. The probability distribution of tiles based on their immediate surroundings is learned from training examples, which in our case are the human-authored levels. We chose to use Markov chains as a baseline due to their prominence in early PCGML level generation work, and their success in domains such as Super Mario Bros. (Summerville et al. 2018). For both of our domains, our implementation uses a 2×2 context window where the value of the top-right tile is based off the remaining three tiles surrounding it in an "L" shape with equal weighting. This window size was chosen to maximize the amount of training examples given the small sample size, as larger window sizes resulted in many "unseen states" during generation. In our implementation, we selected the empty tile for any unseen state.

MCMCTS MCMCTS is an extension of Markov chains introduced by Summerville et al. with the intention of increasing the controllability of generation via MCTS (Summerville, Philip, and Mateas 2015). It learns the probability distributions for entire columns instead of singular tiles, and then generate columns via an MCTS-driven controller guided by a heuristic that aims to balance playability and the contents of the level. The chosen heuristic for our implementation measured the difference in the number of pits, enemies, and rewards from the target level, which is loosely based off the described heuristics in the original MCMCTS paper (Summerville, Philip, and Mateas 2015). We chose MCMCTS as a baseline since it represents the only other prominent use of MCTS within level generation. Notably, MCMCTS was unable to generate unique levels for the GVGAI Zelda domain, as the levels proved too small to collect a sufficient variety of column-based probabilities to avoid total plagiarism. Similarly, MCMCTS was unable to generate levels based on 1-1 and 1-2 individually, and as such we present results after training MCMCTS on both levels.

WFC & Sturgeon WFC is a constraint-based PCGML approach which learns constraints for tile placement from example levels, then applies those constraints to greedily generate new levels. Constraints are learned using a sliding window as in Markov chains, and generation works by repeatedly choosing a random option out of the possibilities for the most constrained tile, then "collapsing" any tiles constrained to a single possibility by that choice. We chose to compare against WFC due to its ability to generate content from a single example.

However, for Super Mario Bros., we found the base implementation of WFC was too slow to generate entire levels at a reasonable rate. As such, we used Sturgeon, a similar constraint-based PCGML system in WFC's place for the Super Mario Bros. domain (Cooper 2022). Sturgeon works similarly to WFC, using a constraint-based pattern matching window while also matching the tile distribution of the source level. Notably, we chose to avoid the additional global constraint features of Sturgeon such as playability due to the additional human-authoring they represent, and to keep the baseline as close to WFC as possible. For Super Mario Bros., we utilized a "ring" shaped window, which considers each of the 8 adjacent tiles in learning constraints. For GVGAI Zelda, we utilized a 2×2 window due to the relatively smaller size of the levels.

Autoencoder Autoencoders are a type of neural network which are designed to first "encode" input data into a compressed representation, and then "decode" this data with the goal of reconstructing the input as closely as possible. In the context of level generation, learning this compressed representation allows us to feed noise into the decoder segment of the autoencoder to generate novel levels. We chose autoencoders as a baseline due to their success in prior work within Super Mario Bros., along with their ability to train with relatively little data compared to other deep learning models (Jain et al. 2016). Our baseline implementation is based off of the autoencoder architecture from Jain et al.'s paper, with minor, domain-dependent changes to the dimensionality. This architecture is made up of fully-connected layers only, with one layer for the encoder and decoder. All layers use ReLU activation, except the output layer of softmax activation to determine the output tokens. During generation, we input the existing levels to the model with a noise variance of 0.01 added to each variable. This scheme, as described in prior work, allows for the generation of levels consistent with the source without direct plagiarism (Jain et al. 2016). Further, we used probabilistic decoding rather than a greedy decoding to maximize the variety of output levels.

For Super Mario Bros., we used 16×16 level chunks as input. The encoder and decoder layers each had 512 nodes, and the latent layer consisted of 8 nodes. For GVGAI Zelda, we used 1×9 chunks of level as input, which represent single columns of a level. The encoder and decoder layers each had 256 nodes, and the latent layer consisted of 4 nodes. These differences in dimensionality and input size are due to the differences in size between these two domains, as GVGAI Zelda has fewer training examples and potential input states than Super Mario Bros.

Notably, this approach was unable to produce non-empty levels using training data from only a single example in both domains. We hypothesize that this is due to the model getting trapped in a local minima due to the lack of training examples. As such, we present results for autoencoders trained on both levels for each domain.

TOAD-GAN TOAD-GAN is a level generation model for token-based game levels based on Generative Adversarial Networks (GANs) (Awiszus, Schubert, and Rosenhahn 2020). TOAD-GAN can train on a single input example by

	Playable	Plagiarism	Self Sim
TRP-Fixed	95%	91.21 ± 0.7	94.4 ± 0.4
TRP-Variety	85%	87.79 ± 10.5	83.3 ± 8.8
Sturgeon	3%	88.18 ± 2.6	90.1 ± 3.9
MC	47%	81.11 ± 14.7	79.2 ± 13.1
TOAD-GAN	94%	90.03 ± 1.0	91.3 ± 1.1

Table 1: Results for Super Mario Bros. on Level 1-1.

	Playable	Plagiarism	Self Sim
TRP-Fixed	86%	79.64 ± 1.5	84.9 ± 0.9
TRP-Variety	90%	78.42 ± 8.3	75.9 ± 7.1
Sturgeon	51%	68.48 ± 2.1	69.9 ± 1.0
MC	14%	67.63 ± 3.0	66.6 ± 2.1
TOAD-GAN	55%	82.20 ± 1.5	83.8 ± 1.3

Table 2: Results for Super Mario Bros. on Level 1-2.

	Playable	Plagiarism	Self Sim
TRP-Fixed	88%	74.51 ± 2.0	78.9 ± 0.8
TRP-Variety	88%	72.90 ± 7.4	71.4 ± 7.3
MCMCTS	100%	81.23 ± 1.0	85.2 ± 1.0
Autoencoder	100%	87.80 ± 0.2	95.6 ± 0.3

Table 3: Results for Super Mario Bros. on both levels.

downsampling the input level to multiple different scales during training. We chose to compare against TOAD-GAN due to its ability to generate high-quality levels from only a single example. In generation, we used pre-trained, open source models provided by the TOAD-GAN authors.

Notably, we do not provide results for TOAD-GAN on the Zelda GVGAI domain, as reimplementation and retraining of the approach would require significant design decisions. While we would ideally be able to compare with this baseline across both domains, we cannot make these design decisions in an unbiased way.

5.2 Metrics

To evaluate the ability of each approach to generate a variety of valid, unique levels, we chose playability, plagiarism, and self-similarity.

We chose playability as a metric as it determines the validity of a generated level. By validity, we refer to a level being consistent with the global constraints of the game as well as having a solution. In order to measure playability for Super Mario Bros., we used an existing text-based A* playability tester to determine if the ending of the level was reachable from a starting position in a level (Summerville et al. 2016). This starting position was the first available position that was not blocked by solid objects in the third column of the level. For GVGAI Zelda, we first ensured that the required game elements were present (exactly one player, one or more doors, and one or more keys), and then performed a flood-fill from the player's position to determine if both a key and goal were reachable.

We chose plagiarism as a metric to measure the uniqueness of the levels generated by each approach with respect to

	Playable	Plagiarism	Self Sim
TRP-Fixed	100%	90.87 ± 3.4	90.81 ± 1.6
TRP-Variety	84%	87.11 ± 7.1	84.46 ± 3.3
WFC	0%	45.83 ± 5.9	50.99 ± 2.6
MC	4%	56.25 ± 5.7	63.61 ± 3.4

Table 4: Results for Zelda GVGAI on Level 1.

	Playable	Plagiarism	Self Sim
TRP-Fixed	96%	88.26 ± 1.6	95.28 ± 1.1
TRP-Variety	72%	90.44 ± 4.2	89.29 ± 2.1
WFC	0%	44.99 ± 5.2	48.08 ± 1.4
MC	6%	50.61 ± 5.5	55.08 ± 1.8

Table 5: Results for Zelda GVGAI on Level 2.

	Playable	Plagiarism	Self Sim
TRP-Fixed	100%	80.26 ± 1.9	80.30 ± 1.4
TRP-Variety	74%	78.38 ± 4.6	76.58 ± 2.4
Autoencoder	18%	64.32 ± 1.9	83.58 ± 1.9

Table 6: Results for Zelda GVGAI on both levels.

the original source levels. While a certain amount of plagiarism may reflect a generator learning the structure of a level, we intuit that a generator which overtly copies source levels may not be useful to designers. Further, this metric serves to measure the effect of overfitting on the training data for the machine learning-based baselines. We used edit-distance to measure the amount of plagiarism between the generated output and source levels, which here is the percentage of tile positions which directly match between the output and source level. This metric is the same across both domains.

We chose self-similarity as a metric in order to measure the variety of the levels generated by each approach. As with plagiarism, we measure self-similarity as an average of the edit-distance of each combination of levels within a generated population. Self-similarity can be viewed as a measure of the consistency of the output levels from a given generator. As such, high self-similarity is not necessarily a negative trait, but rather a value can be used to contextualize the performance of a generator.

Ideally, a generator would be able to generate a population of levels that were all playable, which minimally plagiarized the source levels, and with a low amount of self-similarity. However, while it is always strictly better for all levels to be playable, the other two metrics do not indicate quality on their own. For example, random noise would score low on plagiarism and self-similarity, but would likely be useless to a human designer. As such, it is important to consider all three metrics in concert.

6 Results

In this section, we compare the results of TRP and the baseline approaches across both game domains. For Super Mario Bros., we generated 100 levels per approach for comparison, and 50 levels for GVGAI Zelda. We empirically determined

these level counts by observing the highest count that could be reached before our baselines began repeating levels verbatim. Tables 1, 2 and 3 contain the results of our evaluation for Super Mario Bros., and Tables 4, 5 and 6 contain the results for GVGAI Zelda. In addition, Figures 2 and 3 depict original and generated levels from both domains.

From our results across both domains, we identified three major takeaways. **(i)** On average, TRP generated more playable levels without a hard coded playability constraint, **(ii)** TRP generated levels with lower source plagiarism and self-similarity than baselines with comparable playability, and **(iii)** varying TRP's parameters lessens self-similarity and plagiarism at the expense of playability.

Across both domains, the vast majority of generated levels by TRP were playable. Notably, playability is not guaranteed by default with TRP due to the binary space partition potentially blocking the paths of the input search trees. In the Super Mario Bros. domain, TRP outperformed Sturgeon, Markov Chains (MC), and TOAD-GAN overall in playability, consistently generating playable levels regardless of training data. We hypothesize the changes in playability between 1-1 and 1-2 are due to the more constrained geometry of 1-2, which helps approaches like Sturgeon and hinders approaches like Markov Chains. Notably, both MCMCTS and the autoencoder approach scored 100% playability when trained on both levels for Super Mario Bros. However, similarly to the levels depicted in Figure 2, the autoencoder approach generated largely empty levels, while MCMCTS simply permuted columns from 1-2 until the desired length of level was reached, leading to higher values of plagiarism and self-similarity. In the GVGAI Zelda domain, TRP vastly outperformed the baseline approaches, which struggled to generate playable levels. This is likely due to the hyper-local nature of WFC and MC. We hypothesize TRP's performance is due to its use of the knowledge kit of goals, which help in ensuring the correct number of required game objects are placed in a generated level. These results are promising overall, as they show TRP was able to generalize to generate playable levels in both domains.

Across both domains, TRP was able to achieve lower plagiarism and self-similarity scores than approaches with similar playability. In particular, TRP-Variety performed well on these metrics, as the variance in parameters allows for a larger variety of generatable levels. In the Super Mario Bros. domain, TRP scored substantially lower in plagiarism and self-similarity than TOAD-GAN, MCMCTS, and the autoencoder, which had similar playability to TRP. These results indicate TRP was able to generate roughly the same amount of playable levels as these approaches while plagiarizing from both the source and itself less. While Sturgeon and Markov Chains outperformed TRP according to these metrics in the Super Mario Bros. domain, and overall within the GVGAI Zelda domain, we intuit that this is due to noise based on the unplayable nature of the majority of the levels generated by these approaches.

Across both domains, it is clear that the variance of TRP's parameters results in lower plagiarism and self-similarity in comparison to fixed parameters, regardless of the amount of training data. However, this may result in a drop-off in the playability of generated levels, as seen in the Zelda GVGAI results. We hypothesize that this is due to the increased variety in the binary space partition step, which has the highest chance of rendering a level unplayable. More investigation is required to find the best range of values to optimize for playability and against plagiarism for a given domain.

7 Discussion

7.1 Limitations

A major limitation of TRP is the requirement of a forward model for generation. While this is a non-trivial ask for game developers, we hypothesize this issue could be solved if the forward model was already built into a game engine via a plugin. Another limitation of TRP is that playability of generated levels is not strictly guaranteed. This could potentially be addressed by the addition of a post-processing step in which the existing MCTS agent plays through the final level to ensure playability. While TRP was able to perform well with both fixed and varied parameters, fixing the parameters resulted in more plagiarism and less variety in generation. We found varying the parameters alleviates this within our evaluation domains, but this can negatively affect playability of generated levels. Determining this balance automatically without requiring additional designer input could be a possible solution, which we leave to future work.

7.2 Future Work

An immediate next step would be to utilize TRP in a human-subject study in order to allow designers to develop games alongside this approach. However, there exist several potential applications of TRP to tasks outside of early game development level generation. An example is level blending, which could be achieved by mixing search trees from multiple source levels during the reconstruction step. Another is the generation of auxiliary environments for reinforcement learning in order to help with generalizability of agents. For example, this could entail using TRP to generate levels based on a single source example in order to prepare an agent to generalize across future unseen levels.

8 Conclusions

In this paper, we introduced Tree-based Reconstructive Partitioning (TRP), a novel PCGML approach that can generate levels based on only a single example. We found TRP outperformed other low data PCG approaches across two game domains with respect to playability, plagiarism, and self-similarity. We consider TRP to be a promising new approach, and hope it can support developers within the early stages of game development without requiring intensive hand-authoring.

Acknowledgements

This work was funded by the Canada CIFAR AI Chairs Program. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

Awiszus, M.; Schubert, F.; and Rosenhahn, B. 2020. Toadgan: coherent style level generation from a single example. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, 10–16.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1): 1–43.

Charity, M.; Green, M. C.; Khalifa, A.; and Togelius, J. 2020. Mech-elites: Illuminating the mechanic space of gvg-ai. In *Proceedings of the 15th International Conference on the Foundations of Digital Games*, 1–10.

Chaslot, G.; Bakkes, S.; Szita, I.; and Spronck, P. 2008. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 4, 216–217.

Cooper, S. 2022. Sturgeon: Tile-Based Procedural Level Generation via Learned and Designed Constraints. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 18, 26–36.

Earle, S.; Edwards, M.; Khalifa, A.; Bontrager, P.; and Togelius, J. 2021. Learning Controllable Content Generators. In *2021 IEEE Conference on Games (CoG)*, 1–9.

Fu, M. C. 2016. AlphaGo and Monte Carlo tree search: the simulation optimization perspective. In *2016 Winter Simulation Conference (WSC)*, 659–670. IEEE.

Graves, M.; et al. 2016. Procedural content generation of Angry Birds levels using monte carlo tree search. *University of Texas*.

Guzdial, M.; Sturtevant, N.; and Yang, C. 2021. The impact of visualizing design gradients for human designers. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 17, 18–25.

Horswill, I. 2021. Answer Set Programming for PCG: the Good, the Bad, and the Ugly. In *CEUR Workshop Proceedings*, volume 3217. CEUR-WS.

Interactive Data Visualization, I. 2023. Speedtree. *Interactive Data Visualization, Inc.*

Jacobsen, E. J.; Greve, R.; and Togelius, J. 2014. Monte mario: platforming with mcts. In *Proceedings of the 2014 annual conference on genetic and evolutionary computation*, 293–300.

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*, 1–10.

Khalifa, A. 2022. Mario-AI-Framework: 10th Anniversary Edition. https://github.com/amidos2006/Mario-AI-Framework. Accessed: 2023-08-26.

Khalifa, A.; Bontrager, P.; Earle, S.; and Togelius, J. 2020. Pcgrl: Procedural content generation via reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, 95–101.

Kocsis, L.; and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006: 17th European Conference on Machine Learning Berlin, Germany, September 18-22, 2006 Proceedings 17*, 282–293. Springer.

Miyamoto, S.; Yamauchi, H.; and Tezuka, T. 1985. Super Mario Bros. *Nintendo Entertainment System. Nintendo*.

OpenAI; Berner, C.; Brockman, G.; Chan, B.; Cheung, V.; Debiak, P.; Dennison, C.; Farhi, D.; Fischer, Q.; Hashme, S.; Hesse, C.; Józefowicz, R.; Gray, S.; Olsson, C.; Pachocki, J.; Petrov, M.; de Oliveira Pinto, H. P.; Raiman, J.; Salimans, T.; Schlatter, J.; Schneider, J.; Sidor, S.; Sutskever, I.; Tang, J.; Wolski, F.; and Zhang, S. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. https://arxiv.org/abs/1912.06680. Accessed: 2023-08-26.

Perez-Liebana, D.; Samothrakis, S.; Togelius, J.; Schaul, T.; and Lucas, S. 2016. General video game ai: Competition, challenges and opportunities. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30.

Ramadan, R.; and Widyani, Y. 2013. Game development life cycle guidelines. In *2013 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*, 95–100. IEEE.

Sarkar, A.; Summerville, A.; Snodgrass, S.; Bentley, G.; and Osborn, J. 2020. Exploring level blending across platformers via paths and affordances. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, 280–286.

Schreier, J. 2017. The Story behind Mass Effect: Andromeda's Troubled Five-year Development. https://kotaku.com/the-story-behind-mass-effect-andromedas-troubled-five-1795886428. Accessed: 2023-08-26.

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

Snodgrass, S. 2019. Levels from sketches with example-driven binary space partition. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, 73–79.

Snodgrass, S.; and Ontanón, S. 2013. Generating maps using markov chains. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 9, 25–28.

Snodgrass, S.; Summerville, A.; and Ontañón, S. 2017. Studying the effects of training data on machine learning-based procedural content generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 13, 122–128.

Sorochan, K.; and Guzdial, M. 2022. Generating Real-Time Strategy Game Units Using Search-Based Procedural Content Generation and Monte Carlo Tree Search. *Experimental AI in Games Workshop (EXAG)*.

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.; Behrooz, M.; Mateas, M.; and Jhala, A. 2015. The learning of zelda: Data-driven learning of level topology. In *Proceedings of the FDG workshop on Procedural Content Generation in Games*, 5–12.

Summerville, A. J.; Snodgrass, S.; Mateas, M.; and Ontanón, S. 2016. The vglc: The video game level corpus. In *Proceedings of 1st International Joint Conference of DiGRA and FDG*.

Torrado, R. R.; Bontrager, P.; Togelius, J.; Liu, J.; and Perez-Liebana, D. 2018. Deep Reinforcement Learning for General Video Game AI. In *Computational Intelligence and Games (CIG), 2018 IEEE Conference on*. IEEE.

Van Der Linden, R.; Lopes, R.; and Bidarra, R. 2013. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1): 78–89.

Xbox Game Studios. 2011. Minecraft. *Microsoft*.

Zook, A.; Harrison, B.; and Riedl, M. O. 2015. Monte-carlo tree search for simulation-based strategy analysis. In *Proceedings of the 10th International Conference on the Foundations of Digital Games (FDG 2015)*.