Markov Models for Procedural Content Generation

A Thesis
Submitted to the Faculty
of
Drexel University
by
Sam Snodgrass
in partial fulfillment of the
requirements for the degree
of
Doctor of Philosophy
March 2018
Acknowledgments

I would like to thank my advisor, Santiago Ontañón, for the immense amount of support and guidance he gave me while pursuing my Ph.D. Without him, none of this would have been possible. I would also like to thank the rest of my committee members for their thoughtful feedback and the fruitful discussions that arose. I want to thank my collaborators for joining me in interesting pursuits.

I am grateful to my lab mates for all their help during my time as a student. Specifically, I want to thank Josep Valls who helped me stay sane during our long nights writing in the lab. I would like to thank all of my fellow researchers and particularly the members of the game AI research community that I’ve had discussions with at various conferences over the years. Those discussions led to interesting ideas to explore, and I always left conferences with a longer list of research questions than when I arrived. I would also like to thank the mentors I’ve had when attending doctoral consortiums, namely, Adam Smith and Sven Koenig who provided invaluable feedback and guidance. I want to thank the researchers I’ve worked with during internships, David Aha and Venkat Dasari, for allowing me to work on new and interesting projects.

I want to thank my family. Specifically, I want to thank my mother and siblings for their encouragement over the years. Lastly, I want to thank my wife, Jocelyn, for being incredibly supportive, helping me through long nights of writing and research, and generally being amazing.
# Table of Contents

**LIST OF TABLES** ................................................................. vi

**LIST OF FIGURES** ............................................................. viii

**ABSTRACT** ........................................................................... xiv

1. **INTRODUCTION** .............................................................. 1
   1.1 Contributions ................................................................. 4
   1.2 Organization ................................................................. 6

2. **RELATED WORK** ............................................................... 8
   2.1 Search-based Approaches ................................................ 8
   2.2 Fractal and Noise-based Approaches ................................. 12
   2.3 Grammar-based Approaches ............................................ 15
   2.4 Constructive Approaches ............................................... 18
   2.5 Constraint Satisfaction Approaches ................................. 20
   2.6 Machine Learning-based Approaches .............................. 21
   2.7 Evaluation .................................................................. 22
   2.8 Technical Artists ......................................................... 25

3. **MULTI-DIMENSIONAL MARKOV CHAIN LEVEL GENERATION** ...... 31
   3.1 Multi-dimensional Markov Chains .................................... 32
   3.2 Methods .................................................................. 33
   3.3 Experiments ................................................................ 38
   3.4 Conclusions ................................................................ 48

4. **HIERARCHICAL MDMC LEVEL GENERATION** ......................... 49
   4.1 Introduction ................................................................ 49
   4.2 Hierarchical Multi-dimensional Markov Chains ................ 50
   4.3 Methods .................................................................. 51
9.1 Contributions .............................................................. 184
9.2 Future Work ............................................................. 186

BIBLIOGRAPHY ................................................................ 190

APPENDIX A: VIDEO GAME LEVEL DOMAINS ......................... 204
A.1 Super Mario Bros. ....................................................... 204
A.2 Kid Icarus ................................................................. 207
A.3 Lode Runner .............................................................. 209
A.4 Kid Kool ................................................................. 211

APPENDIX B: LIST OF PUBLICATIONS ............................... 213
B.1 Journal Articles ......................................................... 213
B.2 Invited Talks ............................................................. 213
B.3 Conference Papers ................................................... 213
B.4 Workshop Papers ..................................................... 214
B.5 Other Publications ................................................... 215

VITA ................................................................................. 216
### List of Tables

3.1 Multi-dimensional Markov Chain Results: This table shows the results of generating 1000 levels in *Super Mario Bros.*, *Lode Runner*, and *Kid Icarus*. This table reports the percentage of playable levels generated in each domain with each model configuration as well as the average log-likelihood of those levels, and the average number of unseen states in each level. ........... 42

3.2 MdMC *Lode Runner* Limited Training Data Results: This table shows the results of generating 1000 levels in *Lode Runner* after training an MdMC on only the first ten training levels. This table reports the percentage of playable levels generated as well as the average log-likelihood of those levels, and the average number of unseen states in each level. ............... 44

4.1 Manual Hierarchical Multi-dimensional Markov Chain Results: This table shows the results of generating 1000 levels in *Super Mario Bros.* using the manual setting of the hierarchical MdMC approach. This table reports the percentage of playable levels generated with each model configuration as well as the average log-likelihood of those levels, and the average number of unseen states in each level. ................................. 59

4.2 Clustering-based Hierarchical Multi-dimensional Markov Chain Results: This table shows the results of generating 1000 levels in *Super Mario Bros.*, *Lode Runner*, and *Kid Icarus* using the clustering setting of the hierarchical MdMC approach. This table reports the percentage of playable levels generated in each domain with each model configuration as well as the average log-likelihood of those levels, and the average number of unseen states in each level. ........... 60

5.1 Markov Random Field Playability: This table shows the results of generating 1000 levels in *Super Mario Bros.*, *Lode Runner*, and *Kid Icarus* using the Markov random field approach. This table reports the percentage of playable levels generated in each domain with each model configuration as well as the average log-likelihood of those levels, and the average number of unseen states in each level. ................................. 70

6.1 Model Comparison: This table shows a comparison of several machine learning-based level generation approaches represented in our framework. ......................... 99

7.1 Comparison of Constrained Sampling Algorithms: This table shows the results of generating 100 levels in *Super Mario Bros.* and *Kid Icarus*. This table reports percentage of levels generated with each sampling algorithm that were able to satisfy each set of constraints, and the average number of sections resampled for each level. ................................. 117

7.2 Likelihood of Paths through Maps: This table shows the results of generating 100 levels in *Super Mario Bros.* using the constrained sampling approach paired with a path likelihood constraint. This table reports the average likelihood of the paths through the generated levels. .................. 126

7.3 Multi-layer Level Evaluation (*Super Mario Bros.*): This table shows the results of generating 1000 levels in *Super Mario Bros.* with the single layer approach and 250 with each of the multi-layer models. This table reports the average linearity and leniency of the levels, as well as the Fréchet distance between the provided paths and the actual paths through the levels (given by an A* agent) and the average number of springs placed in the levels and the percentage of those springs that are required to complete the level. ................................. 141
7.4 Multi-layer Level Evaluation (*Lode Runner*): This table shows the results of generating 500 levels in *Lode Runner* with the single layer approach and 50 with each of the multi-layer models paired with the constrained sampling approach. This table reports the average and standard deviation of the number of sections resampled and the percentage levels that go unfinished (based on a cut off during sampling).

7.5 Multi-layer vs. Single-layer Standard Sampling Comparison: This table shows the results of generating 500 levels in *Lode Runner* with the single layer approach and 50 with each of the multi-layer models using the standard sampling approach. This table reports the percentage of playable levels generated using the single-layer and multi-layer models.

7.6 Multi-layer vs. Single-layer Gold on Path: This table shows the results of generating 500 levels in *Lode Runner* with the single layer approach and 50 with each of the multi-layer models. This table reports the percentage of gold pieces (i.e., objects that need to be collected to complete the level) are placed in positions that fall on the provided path with each of our models.

8.1 Tile Mappings: This table shows the different tile mappings found by our various approaches.

8.2 Comparison of Mappings. *One* corresponds to one map used for training. *All* corresponds to all the original maps in the *Super Mario Bros.* data set, and *BL* is the baseline of training an MdMC with a single training level. This table reports the average log-likelihood, linearity, and leniency for each of the models we train with the different sets of translated training levels.
List of Figures

2.1 The left shows the general progression of an evolutionary algorithm. The right shows an example chess puzzle generated using an evolutionary approach (reproduced from Ashlock1), where the fitness function was guiding towards puzzles requiring 15 moves. The grey spaces are impassable by the player. The starting position is marked with an E, and the goal is marked with an X. For this puzzle the player uses a rook. ................................................. 11

2.2 Sierpinski triangle at 0, 1, 2, and 3 iterations (from left to right). Shaded regions are the triangles, while non-shaded regions are empty space. To construct the Sierpinski triangle, at each iteration, for each triangle find the midpoints and connect them, and remove the center triangle. ............ 12

2.3 Four height maps generated using noise. Black corresponds to the minimum height, and white corresponds to the maximum height. The left-most image shows a height map filled completely by random values. The next image to the right shows a height map with a lattice of one-tenth the detail used to fill key points, an the rest of the points bilinearly interpolated. The third image shows the same as the second, but using bicubic interpolation. The right-most image shows a height map filled using slopes instead of heights directly with a lattice of one-tenth the detail. .. 14

2.4 Four iterations of a plant-like structure generated using bracketed L-systems. Reproduced from Togelius et. al.2 ................................................................. 17

2.5 An example of a simple graph grammar (left) and an application of the rule (right). Reproduced from Dormans3 ................................................................. 17

2.6 A simple example of a shape grammar alphabet (a), the grammar’s rules (b), and a sample production of the grammar (c). Reproduced from Dormans3 ................................................................. 17

2.7 An illustration of the order in which elements are added by the multi-pass generator. Reproduced from Shaker et al.4 ................................................................. 19

2.8 An example of how Dahlskog et al. represent training levels by treating each column as an element (left), and an example of a level generated using their n-gram approach with n = 3. Reproduced from Dahlskog et al.5 ................................................................. 22

2.9 A heat map showing the expressive range of the LaunchPad generator using linearity and leniency as the axes. (Reproduced from Shaker et al.6) ................................................................. 24

3.1 A third-order multi-dimensional Markov chain (three incoming dependences for each node). .. 33

3.2 Example MdMC network structures. ................................................................. 33

3.3 A section of Super Mario Bros. level we use for training (left) and how we represent that level as a w × h array (right). Notice, we add sentinel tiles to denote the boundaries of the level. ....... 34

3.4 This figure shows example Super Mario Bros. levels sampled using the basic MdMC method (top), the baseline MdMC method with s = 4 and l = 2 (middle), and the MdMC method that achieved the most playable levels with s = 4 and l = 3 (bottom). ................................................................. 43
3.5 This figure shows example sections of *Kid Icarus* levels sampled using the basic method (left), the baseline configuration with $s = 16$ and $l = 2$ (center), and the configuration that was able to sample the most playable levels with $s = 5$ and $l = 2$ (right). .................................................. 44

3.6 This figure shows example *Lode Runner* levels sampled using the basic MdMC method (top-left), the baseline MdMC method with $s = 4$ and $l = 2$ (top-right), the best performing MdMC method with $s = 4$ and $l = 0$ (bottom-left), and the MdMC with the baseline configuration trained on only 10 selected levels (bottom-right). ............................................................... 46

3.7 This figure shows heat maps of the *Super Mario Bros.* levels used for training (left) and those generated with the best performing configuration of the MdMC approach (right). The $x$-axis is leniency and the $y$-axis is linearity. The top right corner of the heat map represents highly-nonlinear and very difficulty levels (high leniency values), and the lower left corner represents the most linear and easy (low leniency) levels. Note, all linearity and leniency values are normalized over all levels generated by all models (including models described in future chapters) to allow for uniformity in comparisons. .......................................................... 46

4.1 This figure shows a section of a *Super Mario Bros.* level (left), that same section represented using low-level tiles (center), and an example high-level representation of that section (right). Notice that each high-level tile is composed of a $4 \times 4$ section of low-level tiles, and each high-level tile type represents a larger pattern occurring in the level. ......................................................... 50

4.2 This figure shows heat maps of the *Super Mario Bros.* levels used for training (top-left) and those generated with the best performing configuration of the manual HMdMC approach (bottom-left), the best configuration of the clustering-based HMdMC (bottom-right), and the best performing MdMC configuration for comparison (top-right). The $x$-axis is leniency and the $y$-axis is linearity. The top right corner of the heat map represents highly-nonlinear and difficult levels (high leniency values), and the lower left corner represents the most linear and easy (low leniency) levels. Note, all linearity and leniency values are normalized over all levels generated by all models (including models described in future chapters) to allow for uniformity in comparisons. .................................................. 57

4.3 This figure shows example *Super Mario Bros.* levels sampled using using the manual HMdMC approach with $z = 4$ and $l_h = 2$ (top), and a level sampled using the clustering-based HMdMC approach with $z = 4$, $l_h = 2$, $k = 16$, and $d =$Histogram (bottom). ......................................................... 58

4.4 This figure shows a set of high-level tile types defined using the clustering method with the Histogram distance metric. ......................................................... 60

4.5 This figure shows an example *Lode Runner* level sampled using the baseline HMdMC configuration with $z = 4$, $l_h = 2$, $k = 16$, $d =$Direct (left), and with the best performing configuration with $z = 8$, $l_h = 2$, $k = 16$, and $d =$Direct (right). ......................................................... 62

4.6 This figure shows a section of a *Kid Icarus* level sampled using the baseline clustering-based HMdMC approach with $z = 4$, $l_h = 2$, $k = 16$, and $d =$Direct (left) and a section of a level sampled using the best performing configuration with $z = 16$, $l_h = 3$, $k = 8$, and $d =$Direct (right). ......................................................... 63

5.1 A Markov random field where each state depends on four surrounding states. The colored portion highlights the dependence of a node (red) on its surrounding nodes (i.e., Markov blanket, blue). .......................... 66

5.2 Log-likelihood of a *Kid Icarus* map after each iteration of the MRF sampling algorithm. Notice how the log-likelihood increases quickly in the beginning and then stabilizes after about 400,000 iterations. The horizontal lines are the log-likelihoods of the six training maps. ......................................................... 69
5.3 This figure shows heat maps of the Super Mario Bros. levels used for training (left) and those generated with the MRF approach (right). The x-axis is leniency and the y-axis is linearity. The top right corner of the heat map represents highly-nonlinear and difficult levels (high leniency values), and the lower left corner represents the most linear and easy (low leniency) levels. Note, all linearity and leniency values are normalized over all levels generated by all models to allow for uniformity in comparisons. ........................................... 70

5.4 This figure shows a section of a Super Mario Bros. level sampled using using the MRF approach with \( b = 400,000 \). .......................................................... 70

5.5 This figure shows a Lode Runner level sampled using the MRF approach with \( b = 50,000 \). .... 71

5.6 This figure shows a section of a Kid Icarus level sampled using the MRF approach with \( b = 400,000 \). .......................................................... 72

6.1 A section of a Super Mario Bros. level (left), our original tile representation (center), and a graph representation (right). ......................................................... 76

6.2 Example MdMC network structures. .............................................................. 81

6.3 This figure shows an example of an initial level graph (1) used to start sampling with the MdMC approach, and subsequent vertices and labels sampled (2-5) for a level with \( 2 \times 2 \) tiles corresponding to positions in the output level. ........................................... 84

6.4 A section of a Super Mario Bros. level (left), the column representation used by the n-gram model, and a graph representation, where each vertex represents a column from the training map for a bigram model (top-right) and a trigram model (bottom-right). ......................................... 85

6.5 A section of a Super Mario Bros. level (left), and our graph representation using the “snaking” encoding of the map proposed by Summerville and Mateas\(^7\), where each vertex represents a \( 16 \times 16 \) pixel section from the training map corresponding to a tile type (right). ........................................... 87

6.6 This figure shows a the set of sprites used in Super Mario Bros. from which the set of tile types, \( T \), can be derived. Notice, that this set of sprites is only for illustrative purposes, and is incomplete as it does not include enemy or pipe sprites. ........................................... 89

6.7 This figure illustrates the level representation of Guzdial and Riedl’s approach\(^9\). Specifically, it shows a section of a Super Mario Bros. level (first), that same section with the \( g \) nodes annotated (i.e., the sprite groupings) as well as the connections for one of those nodes \( g_4 \) (second), a graph representation using the \( g \) nodes as vertices and the connections (or \( Ds \)) as edges (third), and a graph representation using the tile types found via clustering the \( (g, D) \) pairs to label the vertices and the \( D \) nodes for the edges again. Notice that both graph representations are fully connected. This is because each of the \( g \) nodes (and by extension the tile types, \( S_i \)) depend on each of the other nodes or objects in the level. Furthermore, for clarity, we do not display the edges going in both directions, and instead display a single edge. ........................................... 90
6.8 This figure shows an example level graph representation for the wave function collapse approach. Specifically, it shows the a section of a Super Mario Bros. level (A); the atomic elements of that level section, that is, the observable tile types (C); the $3 \times 3$ observable tile type patterns extracted from the level section, note that these are the tile types for the hidden vertices (B); the hidden layer of the level graph which represents the patterns from the level section (D); and the portion of the observable layer of the level graph for one such hidden vertex (E). Notice, that when representing a training level, each hidden vertex only has one possible pattern type, but during sampling each hidden vertex can have labels corresponding to different sets of possible patterns. Furthermore, the pattern and atomic object sizes chosen for this figure are merely illustrative, and other sizes and dimensions are possible.

7.1 This figure shows a representation of the generate and test algorithm in Algorithm 9.

7.2 This figure shows a representation of the incremental sampling algorithm in Algorithm 10.

7.3 This figure shows a representation of the violation location resampling algorithm in Algorithm 11.

7.4 A portion of a Super Mario Bros. level sampled using the Violation Location Resampling algorithm while enforcing the $VLR_h$ constraints. Notice the number of enemies and the drastically varying heights within the level.

7.5 A portion of a Kid Icarus level sampled using the Generate and Test algorithm that was unable to satisfy the $GT_e$ constraints where an unplayable location is circled (left), and a portion sampled using the Violation Location Resampling algorithm successfully enforcing the $VLR_e$ constraints (right). Note that in this game the player can pass through one side of the screen and emerge on the other.

7.6 A section of a frame taken from a gameplay video of Super Mario Bros. (right) and the tile representation (left). Notice that though the player-character does not completely occupy one tile position, its tile representation is placed into the most fully occupied position (denoted by “X”).

7.7 2-D projection of the 4 training levels, the levels sampled with my approach, and levels sampled with the SGMR approach. Circled levels appear in Figure 7.9.

7.8 This figure shows a probabilistic finite state machine where each node represents an action type. This represents the Action Only model. “R” is moving right, “L” is moving left, “U” is moving up, “D” is moving down, “N” is no movement, and the combinations are diagonal movements. For clarity, we only include each node’s two most probable transitions.

7.9 This figure shows example levels sampled using the $VLR_{Likelihood}$ approach (top two), $VLR_{Playability}$ (third and fourth), and an example level for the SGMR approach with each of the play traces ($A - D$) and All (bottom five, in order).

7.10 This figure shows a section of a Super Mario Bros. level (top-left) represented using a structural layer (top-right), a player path layer (bottom-left), and a section layer (bottom-right). Color is added for clarity.

7.11 This figure shows the network structures used by the single-layer MdMC (left), and a network structure used by the multi-layer MdMC. Note, that each $ns_l$ follows the same pattern as the corresponding $ns$, but the multi-layer network structures retain the dependencies from the other layers in each network structure.
7.12 This figure shows a section of a *Lode Runner* level (top-left) represented using a structural layer (top-right), a player path layer (bottom-left), and a section layer (bottom-right). Color is added for clarity. .................................................. 137

7.13 This figure shows the expressive range of the different models. The y-axis is the linearity of the sampled levels and the x-axis is the leniency of the sampled levels; all values are normalized to between 0 and 1. *Standard* refers to the single-layer model, while *Path 1 - Path 4* refer to the multi-layer models using the different paths. .................................................. 142

7.14 This figure shows a section of level sampled using the multi-layer P1 MdMC model where a spring is need to complete the level (left), and a section of a level sampled using the single-layer MdMC model where a spring is placed, but not needed to complete the level (right). ................................. 143

7.15 This figure shows a section of a level sampled with the multi-layer P2 MdMC that achieved a low Fréchet distance (4.123), along with the path provided to the sampler for that level (blue), the actual path taken by the $A^*$ agent through the completed level (red), and the overlap of the two paths (purple). Notice, there is a fair amount of overlap in the paths, and even when not overlapping, the paths do not stray far from each other. ................................. 144

7.16 This figure shows the two-dimensional projection of the training levels and sampled levels. We highlight the first two training levels because the paths through these levels were used as the provided player path layers for the first two multi-layer models. Notice, the two selected training levels are occluded by generated levels. .................................................. 145

7.17 This figure shows two levels sampled with each of the models: single-layer MdMC (top two), and the multi-layer MdMC using each of the four provided paths (from top to bottom). .................................................. 146

7.18 This figure shows the player path layer used with the levels generated with “Multi P6.” ................................. 147

7.19 This figure shows a level sampled using the multi-layer approach (top) and a level sampled using the single-layer approach (bottom), both with the VLR algorithm. The levels have been annotated with solutions, where red X’s indicate ground that must be destroyed. .................................................. 148

7.20 Example multi-layer MdMC network structure. .................................................. 151

7.21 A section of a *Super Mario Bros.* level (top-left), three layers represented using tiles (structural, player path and height), and the three layers represented using our graph representation (right). The edges are given by the network structure defined in Figure 7.20. .................................................. 152

7.22 This figure shows an example of an initial level graph (1) used to start sampling with the multi-layer MdMC approach with 3 total layers, and subsequent vertices and labels sampled (2-5) for a level with $2 \times 2$ tiles corresponding to positions in the output level. Notice that many other initial level graphs can be used for a given level domain and model, but we use this initial level graph for ease of visualization and explanation. .................................................. 153

8.1 Graphical depiction of an LSTM block (reproduced with permission from the authors 7). .................................................. 161

8.2 Kernel density estimate for Linearity and Leniency for the MdMC Most-to-least 29 Level Generator. The contour lines represent increasing density, and the dots represent the generated levels. .................................................. 164
8.3 The expressive volume for the 2 different techniques (MdMCs [triangles] and LSTMs [dots]) and 2 different progressions (most-to-least entropy [red] and least-to-most [blue] entropy). We see that the LSTMs in general have higher expressive volume, due to larger variability in the Leniency. The expressive volume of the original levels is shown for reference. ............................................... 164

8.4 Plots of the plagiarism metric for both the MdMC and LSTM models with increasing amounts of training data, as well as a measure of plagiarism between training levels. This shows that the LSTM tends to sample levels with higher percentages of plagiarized columns. However, the MdMC and LSTM both plagiarize similarly sized sections, when trained with more levels. Notice, that the MdMC plagiarizes slightly less than the training levels plagiarize from each other in terms of section size and percentage, while the LSTM plagiarizes at slightly higher percentages, and around the same section sizes. ................................................................. 166

8.5 The three stage filtering process that we employ to find appropriate tile mappings. We start with the full set of tile mappings, $M_0$, and reduce this set to the final set of mappings, $M_3$, by applying three consecutive filters. ................................................................. 171

8.6 This figure shows a section of a Kid Kool level. Notice, that levels in this domain are much taller than those in Super Mario Bros., and that they are in fact broken into several height-based sections in game (denoted by the black horizontal line). ................................................................. 173

8.7 2-D projection of the original Super Mario Bros. levels (green), levels sampled using Kid Kool to supplement the training set (yellow and red), and the levels sampled using Kid Icarus to supplement the training set (blue). ................................................................. 177

8.8 An example level sampled with each mapping method (i.e., manual, random, proposed domain transfer approach). From the top, the mappings are Kid Icarus: $R_2^1$, $R_3^2$, $B_2$, $M_1$; Kid Kool: $R_1^1$, $R_2^2$, $B_3$, $M_3$. ................................................................. 178

A.1 Four example levels from the Super Mario Bros. domain. ................................................................. 204

A.2 Four example levels from the Kid Icarus domain. ................................................................. 208

A.3 Four example levels from the Lode Runner domain. ................................................................. 210

A.4 Four example levels from the Kid Kool domain. ................................................................. 211
Procedural content generation (PCG) is a growing area of research focused on leveraging artificial intelligence in the design and creation of content (e.g., levels, environments, stories, etc.) oftentimes for video games. However, most current PCG approaches are domain specific or require a substantial amount of domain knowledge to be used across multiple domains. We want to determine whether more general approaches to PCG are possible (i.e., approaches that can be applied across large classes of domains without customization or domain knowledge). The first key contribution of this dissertation is to show that machine learning approaches, specifically Markov models, can be used to model and generate levels across multiple domains by replacing domain knowledge with training data, while still being able to capture much of the domain information, such as structural level information and player interactions. The second key contribution of our work is a new theoretical framework to understand PCG approaches based on machine learning, and provide a unifying view of this new class of approaches, highlighting similarities, differences, and providing insights into future avenues of research. Our third main contribution is the development of extensions to these machine learning-based approaches that allow for more control over the generated content and more accurate modeling of the given domain.
Chapter 1: Introduction

Procedural content generation (PCG) is an area of research that focuses on leveraging artificial intelligence in the design and creation of content, oftentimes for video games. PCG is a subfield of computational creativity, which as a field explores the ideation and development of creative systems. The field of procedural content generation aims to explore this creativity by developing artificially intelligent algorithms capable of exploring design spaces, aiding human designers in mixed initiative or collaborative systems, or most commonly designing and creating novel content. PCG systems and approaches have been used to create a wide range of content, such as levels, in-game objects and items, terrain, quests, and even full games. However, PCG has not only been used in games. There has also been work in applying procedural techniques to training and education. Specifically, PCG has been used in emergency response training and military scenario training, and Niehaus et al. have worked on automatically adapting tutoring scenarios to individuals.

Many of the systems mentioned above and, in fact, many procedural content generation systems in general rely heavily on domain knowledge provided by the user of those systems. In this context, domain knowledge is expert knowledge about a given subject that aids the user in applying or adapting a given PCG system. We note that there are two distinct forms that this domain knowledge takes. The first form is knowledge about the tuning and set up of algorithms and parameters. This is the knowledge needed to better fit specific approaches or algorithms to specific domains. For example, if using an evolutionary algorithm for content generation, then this type of knowledge would be used to inform what the mutation rate for members of the population should be or what the size of the populations should be. We will refer to this type of domain knowledge as technical domain knowledge. The second form is knowledge relating to the specific application domain. This form of domain knowledge relates to how a particular domain (e.g., game, level, story, in-game object, etc.) should be represented or encoded, as well as how the quality of generated content can be evaluated. For example, going back to the evolutionary algorithm example, this form of knowledge is used to decide how each member of the population should be represented (e.g., as a histogram of in-game objects, as an
array of object types, etc.). We call this form of domain knowledge *design domain knowledge*. Notice that the fitness function for an evolutionary algorithm can be seen as requiring both of these forms of knowledge; determining what the fitness function should be optimizing requires design domain knowledge and encoding that desire into a function for use in an evolutionary search process requires technical domain knowledge. The approaches mentioned earlier, and most of the categories of approaches discussed in Chapter 2 typically require large amounts of both technical domain knowledge and design domain knowledge.

Notice, the requirement of domain knowledge (both technical and design) places a substantial burden on the user of these PCG techniques. That is, applying the required technical and design knowledge in a given domain for a given approach takes a lot of effort on the part of the user. We are interested in reducing the effort required to use PCG approaches and to apply them to new domains in order to increase the accessibility and broad applicability of procedural content generation techniques. However, the effort required to apply a technique to a given domain is difficult to quantify and may in fact vary by person. Therefore, we aim to decrease the amount of domain knowledge (both design and technical) required for PCG techniques as a proxy for reducing the effort required. Specifically, in this dissertation we aim to answer the question of how we can develop domain independent PCG methods that require little design and technical domain knowledge.

We explore and develop machine learning-based procedural content generation approaches as a means of reducing the amount of design and technical domain knowledge required from the user, and which can thus be more easily applied across multiple domains. More specifically, machine learning-based level generation approaches aim to reduce the amount of design domain knowledge needed from user, by automatically building a model of the given domain by extracting important design information from training data. Notice, that many of these machine learning-based approaches still require some design domain knowledge in order to represent the target domain in a useful way. However, the data representation used by these machine learning-based approaches are often simpler than those of other PCG approaches, and thus still require less domain knowledge. For example, many machine learning-based level generation approaches represent levels as arrays of tile types, where the tile types correspond to different level elements, whereas grammar-based level generation approaches require an encoding of different structures and patterns in the levels set up in such a way that they can be recombined easily. Furthermore, it may be possible to automatically extract level
representations using unsupervised learning (as we show in Chapter 4). Additionally, in Chapters 3-4 we see that our machine learning-based approaches are able to achieve similar results with a variety of parameter setting. This shows that our approaches are robust when using sub-optimal parameter configurations, which means that users without as much technical domain knowledge regarding algorithmic parameter tuning can still employ our methods with minimal negative effects.

At a high level, the class of machine learning-based PCG approaches we propose in this dissertation work in the following way: first, a machine learning approach is chosen and training data is gathered for that approach; second, the chosen approach is trained, extracting important domain and design information from the provided training data; finally, new content is generated by sampling from the trained model. Building a domain model by extracting important design information from data automatically decreases the amount of required design domain knowledge required from the user for the generation process, and therefore increases the generality of the approaches.

We investigated methods based on Markov models for the task of level generation in order to showcase this generality. Markov models are able to model transitions between states. For level generation and modeling, we use Markov models to capture the relationships between objects at nearby positions within a level. We show that Markov models are a viable approach for procedural content generation, and more specifically, we show that our models are able to generate usable levels for multiple video game domains while requiring little domain specific information from the user. We test our approaches in a collection of classic platforming games: Super Mario Bros., Lode Runner, Kid Icarus, and Kid Kool. Unlike previous PCG approaches which have typically been tested in one domain due to the amount of domain knowledge or customization needed to change domains, we are focused on developing general level generation approaches, and thus test our approaches in multiple domains.

The generality of machine learning-based level generation techniques can lead to more comparisons between techniques, but when comparing techniques it is important to compare not only the quality of the output of the techniques, but the techniques themselves in order to determine the limitations and advantages of the competing systems. In order to more deeply understand machine learning-based procedural content generation approaches, we develop and present a theoretical framework which aims to provide a unifying view of
the field of machine learning-based procedural level generation approaches. This unifying view allows for more uniform and direct comparisons between different machine learning-based level generation techniques. The theoretical framework also helps to highlight similarities and differences between the various machine learning-based level generation approaches, while providing insights into future avenues of research.

1.1 Contributions

This section briefly outlines the contributions of this dissertation. It starts with our key contributions: showing that it is possible to define domain independent PCG methods by leveraging machine learning (demonstrated through Markov model-based approaches), unifying this new field of machine learning-based level generation approaches through the development of a theoretical framework, and three approaches for allowing more accurate modeling of domains and more control over the generated content. This section then moves onto our supporting contributions of experimenting with the effects of training data on our models and using domain transfer to supplement limited training data.

1. Markov Models for Level Generation: Our goal is to show that machine learning methods can be used to create domain independent procedural level generation approaches. To demonstrate this, we developed a variety of Markov model-based level generation approaches, which all learn the probability of a tile type or in-game object given a surrounding neighborhood around that tile or object. The first we developed is one of the earliest machine learning-based level generation approaches, and it leverages multi-dimensional Markov chains (MdMCs) to capture the probability of tile-to-tile transitions from training levels, and then uses those trained probabilities to sample new levels. Next, we explored hierarchical extensions to this approach which use clustering in order to identify high-level structures within the training data, and then sample new maps at two levels of abstraction. Lastly, we developed a Markov random field (MRF) approach which captures the probability of in-game objects and tiles given a neighborhood surrounding that position without the inherent directionality of the MdMC approaches.

2. A Theoretical Framework to Describe Machine Learning-based PCG Approaches: We have developed a theoretical framework for the procedural generation of levels via machine learning. This framework provides a unifying lens through which to view the level representation needed by machine
learning-based approaches, how such approaches are trained and what they capture, and how new content is sampled with this class of approaches. This framework allows for the understanding of different models in a uniform way, and also provides insights on future research directions by helping us see what has and has not been done.

3. **Constrained Sampling Approach for Controllable Level Generation**: Machine learning-based level generation approaches are domain independent, but the user often does not have much direct control over the output of these systems. In order to improve the controllability of these approaches, we developed a general constrained sampling approach that enforces provided constraints in a sampled level by resampling problematic sections of the level. This constrained sampling approach gives the user control over the type of levels that are sampled, and is a step towards making machine learning-based level generation more usable in practice.

4. **An Approach to Tailored Level Generation using Player Modeling**: Many machine learning-based level generation approaches focus on the structural information in the levels, but often disregard player behaviors. To address this shortcoming, we developed an augmentation to the constrained sampling approach that allows for the inclusion of a probabilistic player behavior model. This enables the sampling of levels with paths that are more likely given the provided player behavior model.

5. **Multi-layer Level Representations for PCG**: Continuing from the previous point of incorporating more than structural information into a machine learning-based approach in order to more accurately represent a given domain, we developed a general multi-layer level representation that allows for the representation of level information outside of the standard structural information. This representation builds upon our standard representation by allowing for additional layers of representation which capture, for instance, player path information, level sections, or high-level structures.

6. **Insights on the Effects of Training Data on Machine Learning-based PCG Approaches**: Machine learning-based PCG work has typically used all the training data that is available in a given domain. However, little analysis has been performed on how much training data is actually needed by a model in a given domain, or what effects different amounts and qualities of training data may have on an
approach. To address this, we have explored (with others) the effects of varying the size and quality of
the training data set on my MdMC approach as well as a more complex LSTM approach.

7. **Domain Transfer between Games**: Following the previous point, little work has been performed in
exploring what options exist for when a domain has insufficient amounts of training data. To address
this scenario, we developed an approach for leveraging training levels from other games as training
data for a target domain. We accomplished this by developing a procedure for mapping the tile types
from one domain’s representation to another’s using the likelihood of the tile types in the training levels
and a trained conditional probability distributions for each domain.

### 1.2 Organization

The remainder of this document is organized as follows:

First, in Chapter 2 we provide background on the different categories of procedural content generation
approaches in order to situate and motivate the contributions presented afterwards. This chapter also
introduces our target audience.

Next, we show the viability of machine learning-based level generation approaches by introducing
our Markov model-based level generation approaches. Namely, in Chapter 3 we introduce our multi-
dimensional Markov chain (MdMC) approach. In Chapter 4 we introduce a hierarchical extension to
the MdMC approach, which is able to more accurately capture high-level structures and longer range
dependencies. Then, we introduce our Markov random field (MRF) approach in Chapter 5.

In Chapter 6 we introduce our second key contribution, our theoretical framework for machine learning-
based level generation approaches.

We then begin discussing approaches for increasing the controllability and depth of representation of
machine learning-based approaches in Chapter 7, starting with a constrained sampling approach, then
moving on to an approach for creating tailored level content using player models, and closing with a
multi-layer level representation approach.
Next, in Chapter 8 we explore several experiments pertaining to training data. Specifically, we present experiments exploring the effects of varying amounts and quality of training data on machine learning-based level generation approaches, and then discuss a domain transfer approach for translating training data between domains.

We close by drawing our conclusions and suggesting avenues of future work in Chapter 9.
Chapter 2: Related Work

In this chapter we give an overview of the most prominent families of procedural content generation techniques in order to position the new class of machine learning-based approaches within the context of other categories, and to motivate the need for such approaches. We start with search-based approaches, as they are the very commonly used. We then discuss fractal and noise approaches, followed by grammar-based approaches. Next, we discuss constructive, constraint-based, ad hoc approaches, and briefly introduce machine learning-based approaches. We then discuss evaluation techniques. We close the chapter with a discussion of our target audience for machine learning-based PCG approaches.

2.1 Search-based Approaches

Search-based procedural content generation (SBPCG) refers to the use of search algorithms, such as simulated annealing or evolutionary algorithms, to generate content automatically. In order to apply search methods to a PCG problem, three pieces need to be defined:

- **Search Space**: In order to use a search algorithm, it is necessary to define what a solution can look like by formally defining the space of all possible solutions, commonly referred to as the solution space or the search space. We will use the terms interchangeably. For example, when generating a two-dimensional game level with a SBPCG technique, a solution can be defined as a simple two-dimensional grid, where each cell in the grid represents a particular game element, such as a rock, a tree, a platform, etc. In this case, the search space would be the set of all two-dimensional grids of a desired size.

- **Search Strategy**: Search algorithms explore the search space in a particular order, or using some particular strategy. The search strategy can be exhaustive meaning that the search algorithm will systematically explore every element in the search space, if necessary, until a solution is found. When the search space is infinite or too large for exhaustive search to be feasible, local search strategies are
employed. Local search strategies start by selecting one or more elements from the search space, and explore by moving to neighboring elements. Local search methods do not guarantee an optimal solution, but can explore efficiently over very large search spaces where exhaustive search methods cannot. 

*Hill climbing* and *evolutionary algorithms* are examples of local search strategies.

- **Evaluation Function**: Typically SBPCG approaches use search algorithms to find a solution that maximizes a given criteria. An *evaluation function* is a function that captures the criteria defined by the designer, and can assign a value to an element in the search space according to how well that element satisfies the designer’s criteria.

The most common approaches in the literature of SBPCG are evolutionary algorithms. Therefore, in the remainder of this section, we will first introduce evolutionary algorithms, and then provide examples in the context of PCG.

### 2.1.1 Evolutionary Algorithms

Evolutionary algorithms are a family of local search techniques inspired by evolution in nature. Specifically, these algorithms use the concepts of natural selection and genetics in order to solve optimization problems; the best of the current solutions are used to produce a new generation of solutions, which improve upon the quality of the previous generation. Evolutionary algorithms, like all search algorithms, need to define the search space, a way to evaluate potential solutions, and a way of exploring the search space. We describe those below.

An evolutionary algorithm defines its search space in two layers: *genotypes* and the *phenotypes*. The genotype is a representation of the content (e.g., a game level) being generated at some level of abstraction (e.g., as a string of bits), that is, the set of possible genotypes defines the actual search space. A phenotype is the instantiation of the content associated with a given genotype. Intuitively, the genotype can be likened to a recipe, whereas the phenotype is the actual meal. Therefore, an evolutionary algorithm searches over the space of genotypes. A given genotype is converted to a phenotype in order to get scored by the evaluation function. The mapping from a genotype to a phenotype is domain dependent. Using level generation for *Super Mario Bros.* as an example, we can represent a level (phenotype) using a two-dimensional array.
(genotype), where each cell corresponds to a portion of the level. This would be a very simple genotype to phenotype mapping, as the phenotype is represented directly. Alternatively, we could represent the level as an array of numbers representing the number of instances of specific structures in the level (e.g., pipes, gaps, etc.). This would be a more complex mapping as it would not represent the phenotype directly, but instead give a description from which the phenotype could be derived.

There are several factors to consider when choosing a genotype representation. The size of the space of genotypes may become difficult to search if representing the phenotypes directly. However, if we use too abstract of a representation, we may not be able to distinguish between all phenotypes. Additionally, we want small changes in the genotype to have small effects on the phenotype, and by extension the score given to that phenotype\textsuperscript{26}, otherwise, the local search strategy employed by an evolutionary algorithm would not be effective.

Evolutionary algorithms start with an initial population of genotypes selected randomly from the solution space. Next, they evaluate the phenotypes corresponding to the genotypes in the population using a fitness (or evaluation) function, and choose members of the population using different means. For example, roulette wheel selection gives higher scoring members of the population a larger chance to be selected than lower scoring members, but all members still have a small chance of being selected\textsuperscript{30}. The chosen members are then added to the next generation of the population through crossover and mutation.

- **Crossover**: Crossover is a process used to generate new members of a population by combining the genotypes of two existing members. The combination is done by splitting the genotypes of the existing members and then combining pieces from each genotype to generate the new genotype, or child. The crossover point is the point where the genotype will be split, which dictates what data will go to which child. Where the crossover point occurs, how it is chosen, and how it is represented are dependent on the representation of the genotype. For example, if the genotype is simply a string of binary digits, then the crossover point could be a position in that string. Once the crossover point is chosen, the children are generated by taking the designated portions of the genotype from the members and combining them into new members. Now suppose we have two binary string genotypes, 1111 and 0000, and the crossover point for each of them is in the center. Then the new members generated would be 1100 and
Figure 2.1: The left shows the general progression of an evolutionary algorithm. The right shows an example chess puzzle generated using an evolutionary approach (reproduced from Ashlock[^1]), where the fitness function was guiding towards puzzles requiring 15 moves. The grey spaces are impassable by the player. The starting position is marked with an \( E \), and the goal is marked with an \( X \). For this puzzle the player uses a rook.

The members generated in this way are then entered into the population.

- **Mutations**: After new members are generated, there is a small chance that a small change in their genotypes may be forced by the algorithm. A common method for implementing mutations (when the genotype is a string) is by using a per digit probability of \( \frac{1}{l} \) where \( l \) is the length of the string, achieving an average of one mutation per member. Taking the new members from the binary example above, a small mutation could change \( 1100 \) into \( 1101 \) for example. These small changes force exploration of nearby neighbors.

The process of choosing the highest scoring members followed by crossover and mutation is repeated a specified number of times. Continually replacing the lowest scoring members with new members created from high scoring members results in a hill-climbing process that gradually increases the average fitness of the population, which leads to a local or global maximum.

There have been many procedural content generation approaches that employ evolutionary algorithms[^26]. One such approach is Ashlock’s[^1] approach for generating chess puzzles. His approach generates chess puzzles by representing the board and piece positions using a list of piece positions. Children are generated by choosing two members of the population and randomly populating a new list with a set number of piece
positions from each parent. Mutations are induced that move a single piece’s position. Figure 2.1 shows the general process of evolutionary algorithms (left) and a puzzle generated using Ashlock’s approach (right).

Additionally, evolutionary algorithms have been used for generating levels\textsuperscript{31–33}, maps for massively multi-player online games\textsuperscript{34}, physics-based puzzles\textsuperscript{35,36}, racing tracks\textsuperscript{37}, weapons\textsuperscript{16}, and even game rules\textsuperscript{38,39}. A recent approach also used an evolutionary approach to search the space of different constructive level generation model specifications\textsuperscript{40}.

### 2.1.2 Limitations

The main limitation of using a search-based procedural content generation approach is the amount of domain knowledge required from the user both for content representation and guidance during generation. Specifically, search-based approaches require the user to define the search space (i.e., an abstract representation of the types of content that are possible). For evolutionary approaches this can be particularly taxing as careful consideration must be put into how abstract of a representation is appropriate. The user must also define a fitness or evaluation function to guide the search. This function requires deep knowledge about what makes levels in the given domain desirable.

### 2.2 Fractal and Noise-based Approaches

In this section we give an introduction to fractals and noise as they apply to procedural content generation.

#### 2.2.1 Fractals

Fractals\textsuperscript{41} are special geometric objects that have several properties relevant to PCG:
• **Self-similarity**: A fractal has a repeated structure at different levels of magnification. For example, Figure 2.2 shows the Sierpinski triangle, a well known fractal. The repeated structure can clearly be seen by looking at different portions of the triangle, then looking at the triangle as a whole.

• **Iterative Generatability**: Fractals are produced by applying an operation, function, or modification to a set of numbers or to an object, and then each resulting shape or set of numbers ad infinitum. The Sierpinski triangle, in Figure 2.2 is created by starting with an equilateral triangle, then drawing a line from each of the midpoints of the lines making up the triangle to each other midpoint, and removing the triangle in the middle. This yields three new triangles. The process is then repeated for each triangle.

Figure 2.2 shows three iterations of this operation, from left to right.

Fractals have many other properties not directly related to PCG, such as non-integer dimensionality. For a more mathematically rigorous explanation of fractals, the reader is referred to Falconer. In addition to mathematics, many naturally occurring structures have fractal qualities. For example, coastlines, plants, and mountains, can be seen as resembling fractals of different types.

Fractals are well suited for use in procedural content generation, because they are naturally occurring, and can be produced iteratively. By observing where fractals occur in nature, designers can see how to apply fractals in digital mediums. Furthermore, iterative and recursive processes are easily translated into functions or programs. Some areas where fractals have been used extensively for PCG include terrain generation and plant generation. Common fractals include the Mandelbrot set, Julia set, and Sierpinski triangle, though these are not typically used for PCG.

### 2.2.2 Noise

Noise is a random perturbation. In most domains noise is unwanted; for example in photography, noise refers to changes in pixel values where there should be none, and in statistics, noise is meaningless or ill-fitting data. In procedural content generation, however, pseudo-random noise can be quite useful. The randomness, space efficiency, and time efficiency of noise generation make it convenient for adding diversity to otherwise repetitive textures and surfaces. For example, one can generate interesting textures using noise to fill a two-dimensional grid with pseudo-random numbers. This grid can then be applied to a grid representation.
Figure 2.3: Four height maps generated using noise. Black corresponds to the minimum height, and white corresponds to the maximum height. The left-most image shows a height map filled completely by random values. The next image to the right shows a height map with a lattice of one-tenth the detail used to fill key points, and the rest of the points bilinearly interpolated. The third image shows the same as the second, but using bicubic interpolation. The right-most image shows a height map filled using slopes instead of heights directly with a lattice of one-tenth the detail.

of a texture by using the values in the noise grid to change color intensities or darken the texture at specific points.

Noise has been used extensively for PCG. Perlin noise has been used to generate textures as well as realistic clouds. Pseudo-random noise generation techniques have been applied to generating textures and terrain. There has also been work in generating sounds using noise. This involves mapping the sound to a surface, generating the surface using a noise technique, then translating the surface back into a sound. For an in depth review of different noise techniques, the reader is referred to Lagae et al. Figure 2.3 shows some examples of different ways of generating noise. The noise generated in the figure can be used to create height-maps for terrain, where lighter portions correlate to higher points and darker portions are lower points as seen in.

2.2.3 Limitations

Fractals and noise techniques are simple approaches for adding diversity to content or generating natural terrain and objects such as mountains and plants. However, fractals have not been used to generate more complex content which have semantics as well as structural information, such as platforming game levels or puzzles. Additionally, the more complex the type of content to be created, the more rules the fractal techniques would require in order to create that content.
2.3 Grammar-based Approaches

Formal grammars\textsuperscript{2} are used to describe languages. The symbols of these languages can be used to represent portions of a piece of content. Additionally, the languages define rules for combining their symbols. These representations and rules can then be used to model domains and generate new content adhering to the defined rules, using the provided symbols. More formally, grammars are defined by sets of:

- **Non-terminal Symbols**: These symbols represent the portions of a string that can be modified or rewritten using the production rules. By convention, these symbols are typically capital letters.

- **Terminal Symbols**: These symbols represent the portions of a string that cannot be modified or rewritten by the production rules. By convention, these symbols are typically lower case letters.

- **Production Rules**: These rules are used to generate strings in the language of the grammar, by rewriting a previous string. Production rules take a string of non-terminal symbols and transform it into a new string of non-terminal and terminal symbols, (i.e., [Non-terminal symbol(s)] → [Terminal and/or non-terminal symbol(s)]). Production rules are often numbered for ease of discussion.

When grammars describe languages where the symbols are characters, as described above, the grammar is called a *string grammars*. A grammar whose language consists of nodes and transitions is a *graph grammar*\textsuperscript{52}, and a grammar whose language consists of geometric shapes is a *shape grammar*\textsuperscript{53}. Note, each of these grammar types have the same requirements as above, but simply use different symbols. A simple string grammar is defined below\textsuperscript{2}:

1. $A \rightarrow AB$

2. $B \rightarrow b$

where \{A, B\} is the set of non-terminal symbols, \{b\} is the set of terminal symbols, and there are two production rules, as stated above.

In addition to the definition of the grammar itself, we also need to consider how to apply the production rules. We can apply the rules *sequentially*, that is, apply the production rules to the current string moving
from left to right. For example, if we start with the string $A$, then applying the rules sequentially would produce $Ab$, by applying the first rule to $A$, and getting $AB$, then applying the second rule to $B$, and getting $Ab (A \rightarrow AB \rightarrow Ab)$. Alternatively, we can apply the rules in parallel, that is, apply all the rules at the same time and store the result as a new string. For example, if we start with the same string, $A$, then applying the rules in parallel would produce $AB$, by finding all applicable rules, which is only the first rule, and applying it ($A \rightarrow AB$). Note that applying rules in parallel can lead to fractal structures. Further, notice that how we apply the rules affects which strings are produced.

The grammar above only has one rule that is applicable to each non-terminal symbol, making this grammar deterministic. However, we can also have non-deterministic grammars, or grammars that have multiple rules for a single non-terminal symbol. For instance, if we add the rule:

3. $A \rightarrow BB$

then we now have two rules that can be applied to the non-terminal symbol, $A$. One way to handle non-deterministic grammars is to assign a probability to each rule that is applicable to the same symbol, then choose which rule to apply according to those probabilities. Another option, is to maintain a set of conditions for each rule, such that those sets of conditions can be used to deterministically decide which rule should be applied for a given symbol.

In the context of PCG, grammars are used to describe languages and generate strings within that language. These languages can be used to represent instructions for creating many things, such as vegetation, buildings, quests, puzzle levels, and even histories of game worlds. In particular, Togelius et. al. developed a system that models and generates vegetation using bracketed $L$-systems where different symbols correspond to drawing lines (“F”) or rotating the direction of drawing (“+” and “-”) and a stack is used to allow for grouping of different symbols (or draw instructions). Figure 2.4 shows a plant generated using this approach. Additionally, Dormans developed a method for generating missions (i.e., a series of tasks) and mission spaces (i.e., a place that allows the performance of those tasks) using a graph grammar and shape grammar, respectively. Figure 2.5 shows an example graph grammar (left), and an application of its rule (right). Figure 2.6 shows a shape grammar alphabet (a), the grammar’s rules (b), and an example production of the grammar (c).
2.3.1 Limitations

Much like search-based approaches, grammar-based approaches require a significant amount of domain knowledge from the user both for content representation and during generation. The user must represent the domain using a set of symbols, which typically abstract some of the details from the complete domain. The user must also define the production rules, which specify how the symbols can be recombined and replaced in order to create new content. Both of these requirements assume significant knowledge of the
chosen domain.

2.4 Constructive Approaches

Constructive approaches\textsuperscript{59} refer to a set of techniques, typically used for level generation, that have the following features: an abstract representation that is used to describe the structure of a level, a way to generate that representation, and a way to realize that representation into a playable level. However, these features are too broad and are applicable to many content generation techniques. To remedy this problem, we require a technique to have an additional feature before it is classified as constructive. Constructive approaches rely on guidance from a designer in the form of parameters, pre-authored content, etc. in order to ensure high quality content is generated.

Fractal and noise techniques can be described by the first three features, but introducing the fourth feature excludes them. Noise and fractal methods rely on randomness or repeated structures, respectively, and can possibly take simple parameters from the user, such as the number of iterations to run the fractal generator or the size of lattice for noise interpolation techniques. However, constructive methods require more guidance in the form of more informative input parameters or pre-authored content. Multiple techniques can be applied to each of these features. For example, in the Binary Space Partition algorithm for dungeon generation\textsuperscript{59}, the abstract representation is a tree where each leaf represents a room in the dungeon. The way to generate that tree is by recursively splitting the entire level area into two parts, until an end condition is met (e.g., the pieces are a desired size). The tree is then adapted into an actual level by filling those rooms with enemies, treasure, traps, etc. Guidance is given in the form of how the splitting performed as well as the end condition. Additionally, how the spaces are filled can be thought of as guidance.

Another constructive approach for generating levels for a platform game is The Multi-Pass Generator\textsuperscript{4}. The idea behind the multi-pass generator is to generate a level iteratively, adding a new set of elements each iteration, or pass. Which elements should be added in each pass, the order of the passes, and the frequency and likelihood of each element being placed, as well as the playability constraints all rely on the domain knowledge of the designer. This multi-pass generator works in six passes. First, the ground height is modified, and gaps are placed. Next, background platforms are placed at various heights. Then, pipes are
Figure 2.7: An illustration of the order in which elements are added by the multi-pass generator. Reproduced from Shaker et al.4.

positioned throughout the level. Next, enemies are placed. Afterwards, different block types are added (e.g., breakable, item, coin, etc.). Finally, coins are placed. This process can be seen in Figure 2.7.

One of the earliest examples of procedural level generation in a game comes from Rogue, a dungeon exploration game released in 1980. The levels were generated by partitioning sections of a map into solid or empty spaces, then populating the empty spaces with game elements, such as enemies, treasure, and traps60. There is now an entire genre of games called Rogue-like games, which use constructive methods to create dungeon levels60. An example of a recent Rogue-like game is Rogue Legacy161. This game uses a randomized method of choosing pre-authored level sections and stitching them together. Playability of the generated levels is ensured either through constraining the combinations of sections allowed, or by building the sections in such a way that any combination is valid.

2.4.1 Limitations

These approaches are by their definition limited by the amount of domain knowledge the user has. Either the user needs to be able to create hand-authored content pieces to be recombined or be able to define functions and parameters that can represent the given domain.

1http://www.cellardoorgames.com/roguelegacy
2.5 Constraint Satisfaction Approaches

Constraint satisfaction problems (CPSs) are a group of problems in which a set of variables must satisfy some set of constraints (or limitations on their values). The problem is formulated as a finite collection of variables, each with a finite set of values they can assume. A set of constraints that restrict the allowed values of the variables is then defined. Formally, CSPs are defined by a finite set of variables, \( V = \{ v_1, v_2, ..., v_t \} \), a finite domain for each of those variables, \( D = \{ d_1, d_2, ..., d_t \} \), and a set of constraints, \( C = \{ c_1, c_2, ..., c_s \} \). A solution to a CSP is an instantiation of each \( v_i \in V \) with values from the corresponding \( d_i \in D \) that satisfies the limitations imposed by each \( c_j \in C \). Therefore, the problem is finding such an instantiation.

Some notable constraint satisfaction approaches include the Tanagra level generator, a mixed initiative approach that allows the user to place level geometry while it fills in the rest, satisfying playability among other constraints. Butler et al. propose another mixed-initiative game design tool that aids the user in enforcing constraints between levels, with regards to progression. The system employs several stages of user input and editing each followed by procedural generation. Additionally, Smith et al. developed a level generation system that is able to take progression plans as input and generate a series of levels constrained to that progression plan using answer set programming (a combination of CSPs and logic programming). Recently, Karth and Smith formulated the WaveFunctionCollapse algorithm as a constraint satisfaction approach to content generation. In a later chapter we show that this approach can also be thought of as a machine learning-based approach.

Additionally, Horswil and Foged proposed an approach for populating dungeon maps with items, enemies, and rewards. Their approach starts by assuming all variables (rooms) could take on all values (contents of the room, or type of room). They then narrow the values one of the variables could assume, in order to get closer to a solution satisfying the constraints. This may effect other variables, and if so the necessary changes are propagated. The narrowing process is repeated until a solution is found; backtracking may be used in order to find multiple solutions.
2.5.1 Limitations

The limitations of constraint satisfaction problems again come in the form of domain knowledge. Specifically, defining the constraints over the variables and values representing the given domain requires knowledge not only of the domain, but of what makes content within that domain desirable.

2.6 Machine Learning-based Approaches

In this dissertation we present a new class of PCG approaches based on machine learning. Machine learning-based approaches aim to overcome the limitations of the previously discussed categories of approaches by extracting pertinent information from existing data (e.g., levels, gameplay videos, example quests, etc.) to replace user domain knowledge. These are the class of approaches the remainder of the dissertation discusses, but we provide a brief introduction here.

Machine learning-based approaches have three requirements:

1. **Training Data**: Machine learning-based approaches require a corpus of data to be trained on. The training data needs to be represented in a way readable by the training algorithm. For level generation approaches, oftentimes the data will be images of levels or representations of those images.

2. **Training Algorithm**: A training algorithm is used to extract information from the training corpus and build a model based on the extracted information. Many machine learning algorithms are applicable here, provided the training data is represented appropriately. Graphical models are often used for training in level generation because they are able to model relations between objects in levels (e.g., tiles or structures).

3. **Sampling Algorithm**: A sampling algorithm uses the trained model and produces new content. This can be as straightforward as probabilistically sampling from the learned distribution until a complete piece of content (e.g., a level or quest) is sampled in the same representation as the training corpus. However, there are many ways to extend this simple sampling approach. We will discuss several such extensions in Chapter 7.
A simple example of a learning-based approach is that of Dahlskog et al.\textsuperscript{5}. They explored an $n$-gram approach for modeling and sampling *Super Mario Bros.* levels. Their approach uses images of the levels and treat each 16 pixel wide column as an element (as seen in Figure 2.8, left). Next, the training algorithm learns the probability for each column following the previous $n-1$ columns, based on the observed frequency in the training levels. The sampling algorithm generates new levels by probabilistically choosing columns to place following the previous $n-1$ columns based on the probability distribution learned by the $n$-gram. Figure 2.8 (right) shows an example of a section of level sampled with this method.

### 2.7 Evaluation

Procedural content generators are able to produce vast amounts of content. However, content alone is not useful unless we are able to determine the quality of the content. In addition to determining the quality of the content, evaluation of generated content allows us to make observations and possibly guarantees about a generator as a whole\textsuperscript{6}. Reliable evaluation methods also allow us to measure the progress of a system, both against itself and against other generators. This comparison further allows a designer to iterate upon previous work, and track the results\textsuperscript{6}.

We can classify work on procedural content evaluation into two categories: human-based and metric-based\textsuperscript{6,70}. In the following sections, we discuss each of these approaches to procedural content evaluation. We will discuss the advantages and disadvantages of each, as well as prominent examples.

---

**Figure 2.8**: An example of how Dahlskog et al. represent training levels by treating each column as an element (left), and an example of a level generated using their $n$-gram approach with $n = 3$. Reproduced from Dahlskog et al.\textsuperscript{5}
2.7.1 Human-based Evaluation

Human-based content evaluation approaches, as the name suggests, rely on humans to determine the quality of the generated content. This is often accomplished either with A/B testing, where the user is presented with multiple pieces of content, and is made to choose the best, or rank the options; or with Likert scales, where the user is asked to score the content on a scale. All of the content presented to the tester need not be generated content. Presenting the tester with known quality content and unevaluated generated content can give insight into the quality of the generated content relative to established examples. There are also methods for extracting information from the testers through their game play data. That is, while the tester is playing, a system can record information about how the user plays, and, potentially, how the user is physically responding to the game.

The main advantage of human-based evaluation is that humans are intuitively good judges of quality. That is, since the content is being generated with players in mind, it follows that the players’ feedback would be the most valuable. Unfortunately, user testing requires special care to be taken to avoid biasing the results, which can become time consuming and expensive. Additionally, some generators, such as generators using evolutionary algorithms, require some form of evaluation or fitness function in order to generate the content. These generators could still benefit from user testing after generation is complete, but that does not alleviate the need of an automatic evaluation method. Thus, while human-based evaluation could provide valuable insight, it may not be feasible in every situation.

2.7.2 Metric-based Evaluation

Metric-based evaluation approaches try to determine how well the generated content matches the goals of the content generator by extracting information from the content itself. There are both online (performed during generation) and offline (performed after generation) metric-based evaluation approaches. Online techniques are typically used to guide the content generator towards better solutions. The fitness functions employed by the evolutionary algorithms described in Section 2.1 are examples of online metric-based evaluation approaches. Alternatively, offline techniques are used to evaluate the quality of the items generated, the quality of the generator as a whole, and also for comparing generators. Most metric-based evaluation tech-
niques are specifically designed for a single generator, or a small group of generators, but there has been some work in developing more general content evaluators and more broad and informative evaluation metrics.

One interesting online evaluation approach is the critic system used by Smith et al., which involves a set of critics, each of which is trying to maximize some quality of the generated levels. Different weights for the critics and different objectives lead to different types of levels being generated.

One offline approach to evaluating a content generator is measuring its expressive range. The expressive range of a content generator refers to the space of content that the generator is able to create, as well as whatever biases the generator may have. The expressive range of a generator can be measured by choosing two metrics to evaluate the generated content, and then plotting many generated pieces of content using the two metrics as the axes. Figure 2.9 shows an expressive range heat map generated for LauchPad using linearity and leniency as the axes (reproduced from Shaker et al.). In a recent collaboration, we expanded this concept to more than 2 dimensions in order to determine the expressive volume of a generator. Knowing the biases of your generator allows you to investigate and question why those biases exist, whether they are wanted, and, if necessary, how to remove them.

Recently, Marino et al. attempted to bridge the gap between human-based and metric-based evaluation. Using Super Mario Bros. as their domain, they generated levels with a variety of generators and had humans
evaluate the difficulty, fun, and aesthetics of each level using a Likert scale. Next, the authors chose several common computational metrics to evaluate the levels. Finally, they checked for correlations between the computational metrics and the human evaluations. They were able to find weak correlations between the computational metrics and the human evaluated difficulty and aesthetics. More recently, in collaboration with these authors (and others) we expanded on this work\textsuperscript{85}, in order to broaden the range of automatic metrics evaluated, and we were able to find more correlations between the proposed metrics and the human evaluations.

### 2.8 Technical Artists

In this dissertation we are interested in leveraging machine learning techniques for procedural content generation in order to reduce the amount of technical and design domain knowledge required from the user. In this section we introduce a target audience for our machine learning-based approaches, and how they may benefit from such approaches. Specifically, the audience we will be addressing is that of Technical Artists. Technical artists bridge the gap between artists and programmers. They are responsible for ensuring that the artists creative vision is conveyed while working within the technical limitations of the platform\textsuperscript{86,87}.

As a target audience for our machine learning-based PCG approaches, it is important to delineate what forms of domain knowledge technical artists can easily provide, and which forms of domain knowledge may be more difficult for them to provide. In general, we believe that technical artists are able to provide design domain knowledge more easily than technical domain knowledge. That is, technical artists are generally more familiar with the intricacies and semantics of specific target domains or games than they are with the tuning of algorithm parameters.

Therefore, the most straightforward form of design domain knowledge they could provide to machine learning-based PCG approaches is training data. That is, it is reasonable to expect technical artists to be able to acquire example levels which incorporate designs and patterns they deem important or desirable to be used as training data with a machine learning-based PCG approach. Another type of design domain knowledge they can provide is the evaluation of either training data or generated content. More specifically, using their understanding and knowledge of a specific target domain, they can tag portions of the content that they
consider to be of high-quality or that contain desirable elements and tag undesirable or low-quality portions another way. For example, in the domain of platformer game levels, given a generated or training level, a technical artist should be able to easily paint or highlight sections of the level that are desirable in one color and highlight low-quality undesirable sections in another. These labels can be used for more targeted training of the machine learning-based PCG models.

Where technical artist may struggle is with providing more technical domain knowledge regarding the tuning of algorithms to a specific domains. A technical artist may be able to modify certain intuitive model parameters, such as network structure or model dependencies. However, it is unlikely that they will be willing to fully tune a model’s parameters; this is particularly true of more opaque models, such as neural networks, which have many parameters that may not have obvious effects on the model’s output.

In the remainder of this section we will explore several projects developed and used by individuals in the technical artist role in order to highlight the use of domain knowledge within those projects and discuss how machine learning-based approaches can fit into and benefit those projects.

2.8.1 Infinite Mario Bros.

_Infinite Mario Bros._ is a platforming game developed by Markus Persson which uses graphics and mechanics from the _Super Mario Bros._ game. The key difference between the games is that instead of having discrete, predefined levels like _Super Mario Bros._, _Infinite Mario Bros._ continuously creates a single unending level for a player character’s lifespan using a procedural generation technique.

A level in _Infinite Mario Bros._ is generated in sections with each section extending the level to allow the player to continue playing. Persson defined several section types according to his desired patterns and designs, such as simple flat sections, sections with hills, sections with pipes, cannon sections, and sections with gaps. Each of these section types was assigned a probability of being generated according to predefined baseline values and the desired difficulty provided to the system (e.g., a higher desired difficulty made it more likely that a section with cannons would be generated); the type of the next section to be generated was chosen according to the given probabilities. Once the type of the section is chosen, it is populated with different objects and enemies according to the rules Persson defined for that section. For example, in the
“cannon section,” a random position is chosen for the first cannon to be placed, and then the next cannon is placed in a random position following the previous cannon. This process is repeated until the end of the section is reached. The other section types follow similar strategies of randomly populating the space according to some rules over the desired objects for the given section type.

The domain and design knowledge required in this approach are encoded in the form of the defined section types, the probability of each of those section types occurring, and to a lesser extent the methods for populating each of those section types with game objects. Notice, defining section types requires design domain knowledge, because it depends on knowledge of the intricacies of the given domain and how that domain functions, whereas manually defining the probability distribution over the occurrences of section types is technical domain knowledge, because it relies heavily on information about how the algorithm functions. Recall that we want to limit the both types of domain knowledge required, but it is easier for our target audience to provide design domain knowledge. To limit the required technical domain knowledge, we can easily supplement or replace that required knowledge by estimating the probability distribution from data. Specifically, instead of directly providing the section probabilities, we can instead tag sections of example input levels with the provided section types, and then estimate the section types’ probabilities of occurring from their occurrences in those example levels. Additionally, we can also limit the required design domain knowledge by performing a clustering over the sections from a set of example levels (using simple information like object counts within each section) in order to automatically determine the section types. By incorporating these simple machine learning augmentations into this approach, we can lessen the authorial burden on the user of such a system.

In the following section we discuss a game that was developed by leveraging an existing machine learning-based PCG approach for its level generation.

### 2.8.2 Proc Skater 2016

*Proc Skater 2016* is a skateboarding game with procedurally generated skateparks created by Joseph Parker and Ryan Jones for ProcJam 2016 (the Procedural Generation Jam). Unlike the creator of *Infinite Mario Bros.*, the creators of this game leveraged an existing procedural content generation approach in order to create the
game’s levels. More specifically, this game uses the WaveFunctionCollapse (WFC) algorithm to generate the skateparks for players in the game.

At a high level, WFC is given a set of patterns of objects that are possible in the given domain, and then generates new content by placing objects while enforcing local consistency according to the provided patterns. Notice, the WFC algorithm has two requirements. First, it needs a set of objects or building blocks that compose the content to be generated. In the case of Proc Skater 2016 this is the set of objects that make up the level geometry of the skate park. Next, it needs the set of patterns of objects that are possible. In the case of Proc Skater 2016 this is how the different level geometry objects can fit together. We describe the WFC algorithm in more detail and in terms of our theoretical framework for machine learning-based PCG approaches in Section 6.6.

This project shows that there are machine learning-based PCG techniques, such as the WaveFunctionCollapse algorithm, that can easily be used to generate content while requiring little technical domain knowledge from the user. That is, use of this algorithm is an example of a machine learning-based PCG approach that does not require a deep understanding of how the algorithm functions in order to be used effectively. There are two ways to provide the needed information to the algorithm that do not require technical domain knowledge (i.e., knowledge of the algorithm), but only design domain knowledge (i.e., knowledge of the structure and design of the given domain). The first way is to create example content (in this case skateparks) and indicate what the atomic building blocks are within those examples (e.g., sections of the skatepark or specific objects in the parks). By doing this, the allowed combinations and patterns can be automatically extracted from the example content. The second way is to create the atomic building blocks separate from a complete piece of content, and then designate which building blocks can connect to each other in which ways. This second method may requires more forethought on the part of the user, but still does not require technical domain knowledge.

Next, we discuss a tool that easily allows users to create bots by leveraging grammars, as well as a machine learning extension for the tool which extracts such grammars from a corpus of text.
2.8.3 Cheap Bots, Done Quick

Cheap Bots, Done Quick (CBDQ) is an online tool developed by George Buckenham that allows users to easily create Twitter bots (i.e., Twitter accounts that will automatically create posts). CBDQ leverages generative grammars to automatically generate text for the bots, and then posts the generated text with a provided Twitter account. More specifically, this tool uses Tracery, a generative grammar system designed by Kate Compton with the goal of making text generation more accessible to a casual user.

Cheap Bots, Done Quick and Tracery made it easier to create Twitter bots and simple text generation bots, and allowed for the creation of interesting themed Twitter bots, such as Kate Compton’s @lostTesla. It is important to note, however, that crafting grammars can be time consuming and requires technical domain knowledge about the functionality of the grammars and the desired structure of the generated text, as well as design domain knowledge about the goals of the designed grammar or bot.

Cheap Markovs, Traced Quick (CMTQ) is an extension to Cheap Bots, Done Quick developed by Serin Delaunay that aims to reduce the amount of required technical domain knowledge by leveraging a simple machine learning technique. That is, the creator of CMTQ noticed the connection between Tracery grammars and Markov models (i.e., they both model transitions from one state to the next dependent on some number of previous states), and developed a tool to automatically extract a Tracery grammar from a corpus of text using Markov chains. This tool greatly reduces the amount of technical domain knowledge required from the user by allowing them to replace the technical knowledge with example text and a few simple parameters. Specifically, instead of defining a grammar manually, with CMTQ the user only needs to provide

Training Text: This is the corpus of text from which the Tracery grammar will be extracted. This will define the themes and patterns from which the generative model will draw.

Line Delimiters: These are a set of individual characters that signify a line break in the text. Notice, this can be specified to be actual new line characters or end of sentence characters or any other character the user desires. This gives the user control over how long they want their lines to be as well as some control over the structure of the output.

https://cheapbotsdonequick.com/
https://twitter.com/losttesla
https://serin-delaunay.github.io/cheapmarkovstracedquick/
Tokenization Mode: This specifies whether an atomic element in the Markov chain is an individual character or an entire word. This lets the user decide whether they want their model to generate text at the character by character level or at the word by word level.

Dependency Structure: This specifies how many previous tokens (characters or words) affect the value of the current token. This gives the user control of the coherency of the output text versus the expressivity of the model.

Notice that none of the above parameters require much technical domain knowledge. Furthermore, this tool allows for interesting behavior, such as the blending of multiple text sources into a single model by simply providing text from more than one domain as the training text. This tool is a great example of how we can leverage machine learning as way of reducing the amount of technical domain knowledge required from users, which is the goal of our research into machine learning-based level generation approaches.

In this section we discussed several tools and approaches developed by and for technical artists. We first showed an example of a level generation tool and explored ways in which it could be made more accessible to others through the use of machine learning. We then explored the application of a machine learning-based level generation approach in which deep knowledge of the approach itself was not required in order for it to be used effectively. Lastly, we discussed a text generation tool for which a machine learning extension was developed in order to ease its use for more casual users. Each of these examples shows ways in which machine learning can be used by technical artists.
Chapter 3: Multi-dimensional Markov Chain Level Generation

In the previous chapter, we discussed many approaches to procedural content generation. Most of those approaches require a significant amount of technical or design domain knowledge in order to be used in a specific domain. For example, evolutionary approaches require a meaningful objective function that is often specific to the domain in order to guide the evolution of content; the constructive approaches discussed required sections of pre-authored content that could be stitched together; the grammar-based approaches require sets of rules for how the symbols can be combined; and constraint satisfaction approaches require the definition of constraints related to the given domain and content type. Each of these requires the user of the approach to have significant knowledge of the chosen domain, and the ability to translate that knowledge into the form required of the chosen approach. We are interested in developing techniques for level generation that eschew the large amount of domain knowledge required, and that are applicable across domains.

In order to develop more general PCG techniques that require less domain knowledge from the user, we explore machine learning-based level generation approaches. There has been some recent work in machine learning-based level generation approaches (as seen in Section 2.6), and they have been shown to produce high-quality content while requiring little domain knowledge. These approaches are able to replace much of the domain knowledge required from the user with training data (i.e., examples of the type of content the user wants to generate). At a high level, machine learning-based level generation approaches work by (1) being provided a set of training examples represented in a way that is easily parseable by the chosen machine learning method; (2) using the chosen machine learning method to build a model of the target domain from the training data, often by learning the probabilities of various objects and structures appearing within the levels; and (3) generating a new level using the trained model.

In this chapter we describe one of the first such machine learning-based approaches, multi-dimensional Markov chains (MdMCs). This approach leverages a tile-based level representation (i.e., each training level is discretized into a grid, and each position in the grid is assigned a tile type from a finite set of types representing structures, items, enemies, and objects in the level domain). The MdMC model is trained on a
set of tile levels, and learns the distribution of tile types given the surrounding tile types (i.e., probability of a structure/enemy/object/etc. given what preceded). Finally, sampling from the MdMC’s trained distribution, a new level is generated tile by tile until complete.

The remainder of this chapter is organized as follows: first, we describe multi-dimensional Markov chains and how they are derived in Section 3.1; next, we discuss the methods of the MdMC approach in Section 3.2, namely, the level representation, the training procedure, and how a new level is sampled; in Section 3.3 we describe the experimental setup and the results; lastly, we draw our conclusions about the approach in Section 3.4.

3.1 Multi-dimensional Markov Chains

Markov models capture probabilistic relations between variables, be them previous variables, as in Markov chains, or surrounding variables, as in Markov random fields. Markov models have been used for music modeling and composition, speech recognition, and texture synthesis and analysis in computer graphics. In this section we introduce the multi-dimensional Markov model we use for modeling and generating game levels.

Markov chains model stochastic transitions between states over time. A Markov chain is defined as a set of states $S = \{s_1, s_2, ..., s_n\}$ and the conditional probability distribution (CPD) $P(S_t|S_{t-1})$, representing the probability of transitioning to a state $S_t \in S$ given that the previous state was $S_{t-1} \in S$. The configuration of previous states upon which the current state is conditioned is also referred to as the network structure of the model. An example application of Markov chains can be seen through modeling sentences; a chain could capture the probability a certain word occurs, given the previous word.

Markov chains restrict the probability distribution to only depend on the previous state. Higher-order Markov chains relax this condition by allowing the network structure to include $d$ previous states, where $d$ is a natural number. The CPD defining a Markov chain of order $d$ can be written as: $P(S_t|S_{t-1}, ..., S_{t-d})$. That is, $P$ is the conditional probability of transitioning to a state, $S_t$, given the past $d$ states of the Markov chain. Following the sentence modeling example from before, this model could capture the probability of a certain word occurring given the previous $d$ words.
Multi-dimensional Markov chains (MdMCs) are an extension of high-order Markov chains that expand the model by relaxing the network structure even further, allowing surrounding states in a multi-dimensional graph to be included. For example, the CPD defining the MdMC in Figure 3.1 can be written as $P(S_{t,r} | S_{t-1,r}, S_{t,r-1}, S_{t-1,r-1})$. Note that this is only one network structure for a third-order MdMC; there are other combinations of previous states that satisfy the definition of a third-order MdMC. By allowing the previous states of the chain to come from states in a multi-dimensional graph instead of a one dimensional sequence, the model is able to more easily capture relations from two-dimensional training data, such as graphical textures or video game levels.

3.2 Methods

In this section we discuss the techniques employed by our multi-dimensional Markov chain approach. We start by explaining how we represent our training data. We then discuss how we train and sample from our model.
3.2.1 Map Representation

We represent a level as a $w \times h$ two-dimensional array $m$, where each cell, $m[x][y]$, takes a value from a finite set of tiles, $T$. The tiles correspond to different elements of the training level (e.g., a tile representing empty space or the ground). Figure 3.3 shows a section of a Super Mario Bros. level we use for training (1) and how we represent that level section as a $w \times h$ array (2). Notice, we add sentinel tiles to mark the boundaries of the level.

3.2.2 Training

In order to train an MdMC, we first need to specify the network structure, $ns$, or the set of previous states upon which the current state depends (e.g., a chain where each tile depends only on the previous horizontal tile (a standard Markov chain), a chain where each tile depends on the previous horizontal tile as well as the previous vertical tile, or further complicated structures). Figure 3.2 shows the different network structures that we used in our experiments. Though more complicated structures could be devised, increasing the complexity of the network structure exponentially increases the number of possible tile configurations which corresponds to the size of the probability table that needs to be estimated from the data. This results in a need for more training data to properly estimate the parameters of the MdMC.

Algorithm 1 shows how we construct a probability table $P_{ns}$ for a specific network structure $ns$ for an
MdMC from the frequency of occurrences in the training levels. The algorithm has three parameters: \( ns \) is the network structure of the Markov chain, \( M \) is the set of training levels (each \( m \in M \) corresponds to a single level from the set), and \( T \) is the set of tiles. We train our chain in two stages: absolute counts and probability estimation. In the first stage, we pass through every position in each training level, \( m \), where \( \text{positions}(m) \) is the sequence of all the positions in \( m \) (lines 1-2). At each position the algorithm determine the current tile type, \( t \) (line 3), and configuration of previous tiles, \( c \), corresponding to the network structure, \( ns \), at the current position, \( \text{pos} \) (line 4). We then increment the number of times \( t \) has been encountered following \( c \) (line 5). The tile counts are stored in an array, \( S \), whose dimensions are the number of possible tile configurations by the number of possible tiles. In the second stage, we determine the probability of each tile type, \( t \), following a given configuration of previous tiles, \( c \), by dividing the number of times we have observed \( t \) following \( c \) by the total number of times we have observed \( c \) (lines 8-12), and store the result in \( P_{ns} \). Notice, we simply use division here to estimate the probabilities, but other methods, such as smoothing can be employed here as well. Finally, we return the estimated probability table, \( P_{ns} \) (line 13). This algorithm trains a single CPD for a specified network structure. When the chain is in a position in the level where a previous tile would fall outside of the level, that tile is replaced with a special sentinel tile. Configurations with sentinel tiles are included in the absolute counts and CPD. The sampling algorithm requires a set of CPDs corresponding to a set of network structures, which requires this training algorithm to be run several times, one for each of the desired network structures.

In our MdMC experiments we also explored the effects of splitting a training level into several horizontal slices (where \( s \) is the number of sections), and training a separate model for each slice. By splitting a level and training a separate model for each section, we aim to more accurately model each section. For example, in Super Mario Bros. levels, the ground and pipe structures are typically at the bottom of the level, whereas platforms of bricks are typically in the middle or top of the level. By splitting a level and training in this way we hope to reduce the possibility of misplacing structures and increase the confidence we have in the model for each section. We discuss this more in Section 3.3.
Algorithm 1 train-MdMC(ns, M, T)

1: for \( m \in M \) do
2: \hspace{1em} for \( \text{pos} \in \text{positions}(m) \) do
3: \hspace{2em} \( t = m[\text{pos}] \)
4: \hspace{2em} \( c = \text{config}(m, \text{pos}, ns) \)
5: \hspace{2em} \( S[c][t] = S[c][t] + 1 \)
6: for each \( c \in C \) do
7: \hspace{1em} for each \( t \in T \) do
8: \hspace{2em} \( P_{ns}(t|c) = S[c][t] / \sum_{v \in T} S[c][v] \)
9: return \( P_{ns} \)

3.2.3 Sampling

Sampling a new level using an MdMC means filling a matrix with tiles one position at a time by sampling from the trained model. Occasionally, sampling a new level this way will lead to a configuration of tiles for which we do not have data, an unseen state. Notice, that while sampling with a one-dimensional Markov chain, an unobserved configuration of tiles can never occur by construction. This, however, is not the case for MdMCs. An unseen state is formally defined as a configuration of tiles, \( c \), where \( \forall t \in T, S[c][t] = 0 \), where \( S \) is the counts array used in Algorithm 1. We assume unseen states are undesirable, because we have no training data for them and therefore would only be randomly choosing a tile. In practice, we have observed that allowing unseen states results in the sampler randomly generating tiles for a large subsequent section (since it has moved away from the space of tile configurations observed in the training data). To combat this issue, our algorithm uses a look-ahead procedure that allows it to sample several tiles ahead to ensure that no unseen states are reached. If an unseen state is unavoidable with the current network structure, the algorithm falls back to a simpler structure. Intuitively, it is less likely to encounter an unseen state when using a simpler network structure, since the CPD in a simpler network depends on fewer past states.

Algorithm 2 shows how we sample a new level. Our sampling algorithm consists of two functions, one which tracks the current position in the generated level and the current network structure being used, and one which tries to choose a tile type for that position while performing the look-ahead. The first function, sampleLevel (lines 1-11) has four parameters: \( l \) is the length of the look-ahead, or how many tiles after the current tile the algorithm will sample and check for unseen states; \( NS \) is an ordered list of network structures
used for sampling, where the first is the most complex, and the subsequent ones are the structures used for fallback; we define $P$ to be the set of CPDs trained for each network structure (i.e., $P = \{P_{ns} : ns \in NS\}$), and allow elements of $P$ to be accessed by indexing (e.g., $P[ns] = P_{ns}$); and $T$ is the set of possible tiles. Algorithm 1 produces a single element of the set $P$, and therefore Algorithm 1 must be called several times, and the outputs combined into $P$ before being passed to Algorithm 2. 

sampleLevel starts by initializing an empty level (line 2). It then tracks where in the level the algorithm is currently sampling (line 3), and which network structure, $ns$, is being used (line 4). $\text{positions}(m)$ is defined as a sequence of positions in the generated level ordered such that the first element is the first position to be sampled, the second element is the second position to be sampled, and so on. This information is passed to the second function $\text{sampleTile}$ (lines 12-34), which samples a tile for the given position. $\text{sampleTile}$ has six parameters: $m$ is the level currently being sampled, $\text{pos}$ gives the position in $m$ to be sampled, $l$ is the length of the look-ahead, $ns$ is the current network structure, $P_{ns}$ is the probability table to be used, and $T$ is the set of tiles. Intuitively, $\text{sampleTile}$ recursively explores the tree of possible tile sequences for the next $l$ level positions until all $l + 1$ tiles are chosen without an unseen state being reached, in which case it fills the tile in the level and returns true. Otherwise, if all possibilities have been explored and all lead to an unseen state, the function fails, returning false. In more detail, $\text{sampleTile}$ can be explained in two stages:

- **Current Tile** (lines 13-23): This stage samples a tile at the current position. $\text{sampleTile}$ determines if the look-ahead has been satisfied (lines 13-15). If it hasn’t, then $\text{sampleTile}$ gets the set of possible tiles (line 16), determines the configuration of the surrounding tiles (line 17), and checks if the current configuration is an unseen state (line 18-20). If the configuration is not an unseen state, then $\text{sampleTile}$ samples a tile according to the CPD of the model (lines 21-23). If the configuration is an unseen state, then the function fails (line 19).

- **Look-ahead** (lines 24-34): This stage implements the look-ahead (recursive) portion of the sampling algorithm. First, $\text{sampleTile}$ is called at the next position in the map (line 24). Each time this call to $\text{sampleTile}$ fails to sample a tile (an unseen state is reached at a further look-ahead), the tile that was chosen at this position is removed from the set of possible tiles (line 25), and a new tile is sampled in its place (lines 29-31). If this call to $\text{sampleTile}$ is successful, then the loop ends and $\text{sampleTile}$ returns
Algorithm 2 sample-MdMC

1: function sampleLevel(l, NS, P, T)
2:     \( m = \text{Empty level} \)
3:     for \( \text{pos} \in \text{positions}(m) \) do
4:         for each \( ns \in NS \) do
5:             if sampleTile(m, pos, l, ns, P[ns], T) then
6:                 break
7:     return \( m \)
8: function sampleTile(m, pos, l, ns, P_{ns}, T)
9:     if \( l < 0 \) \lor \text{outsideLevel}(pos, m) then
10:        return \( \text{true} \)
11:    \( T^* = T \)
12:    \( c = \text{config}(m, pos, ns) \)
13:    if \( c \) is an unseen state then
14:        return \( \text{false} \)
15:    else
16:        \( t \) sampled according to \( P_{ns}(T^*|c) \)
17:        \( m[pos] = t \)
18:    while \( \neg \)sampleTile(m, pos + 1, l - 1, ns, P_{ns}, T^*) do
19:        \( T^* = T^* \setminus t \)
20:    if \( T^* = \emptyset \) then
21:        return \( \text{false} \)
22:    else
23:        \( t \) sampled according to \( P_{ns}(T^*|c) \)
24:        \( m[pos] = t \)
25:    return \( \text{true} \)

true (line 33). If during any recursive call, the set of possible tiles is empty, that call returns false (lines 26-27). If sampleTile returns false at the top level, then a simpler network structure is chosen to sample the tile (line 4).

3.3 Experiments

3.3.1 Domains

We chose to use the platforming games, Super Mario Bros. and Kid Icarus, and the puzzle-platforming game, Lode Runner, as our target domains. All games feature two-dimensional tile-based maps. Super Mario Bros. and Kid Icarus maps are linear (as defined by Dahlskog et. al.\(^5\)): Super Mario Bros. maps are horizontally linear, while Kid Icarus maps are vertically linear. However, Lode Runner maps are maze-like and non-linear. We gather training levels by taking a set of map images from a given game, deciding on a set of low-level
tile types corresponding to elements of those maps, and then translating the images into training levels. All training levels are available in the video game level corpus (VGLC)\(^1\).

**Super Mario Bros.**

*Super Mario Bros.* is a platforming game where the player must perform timed jumps in order to pass obstacles and complete the level. We trained our models using 29 outdoor maps from the original *Super Mario Bros.* and from *Super Mario Bros.: The Lost Levels*. *Super Mario Bros.* is a linear, platforming game with simple structures that should be captured using our models. We represent each level using a set of 36 tile types corresponding to objects, structures, and enemy types in the game. More detailed information on the tile types used to represent levels in this domain can be found in Appendix A.1. We experiment with sampling levels that are 12 tiles tall and 210 tiles wide.

**Kid Icarus**

*Kid Icarus* is a platforming game, much like *Super Mario Bros.*, where the player must perform well controlled jumps in order to reach the end of the level. Unlike *Super Mario Bros.*, *Kid Icarus* levels are vertically oriented, and include many more platforms, increasing the amount of long-range dependencies and making these levels more difficult to accurately model. In our experiments, we used six vertical levels from the original *Kid Icarus*. We represent the training levels with a set of seven tile types corresponding to structures and in-game objects. More detailed information on the tile types used to represent levels in this domain can be found in Appendix A.2. We experiment with sampling levels that are 160 tiles tall and 16 tiles wide.

**Lode Runner**

*Lode Runner* is a puzzle-platforming game that requires the player to collect treasure while avoiding guards. The player is able to dig holes to trap guards and to reach new areas. Our training set consists of 150 levels from the original *Lode Runner*. *Lode Runner* levels are represented by a set of eight tile types. More detailed information on the tile types used to represent levels in this domain can be found in Appendix A.3. We experiment with sampling levels that are 16 tiles tall and 32 tiles wide.

\(^1\)https://github.com/TheVGLC/TheVGLC
3.3.2 Experimental Set-up

Parameters

We applied our MdMC approach to the above domains varying the following parameters.

- **Row Splits** ($s$): In order to train more accurate models, we split the training levels into sections during training using horizontal slices, and train a separate model for each section. That is, we split the levels into groups of complete rows, and train a model for each group of rows. During sampling, each model is used to sample its respective level section. We experimented with $s \in \{1, 2, 3, 4, 6, 12\}$ for *Super Mario Bros.*, $s \in \{1, 2, 4, 8, 16\}$ for *Lode Runner*, and $s \in \{1, 2, 4, 5, 8, 10, 16, 20, 32, 40, 80, 160\}$ for *Kid Icarus*. We chose the possible values of $s$ according to the factors of the height in tiles of the levels to be generated.

- **Look-ahead** ($l$): As part of the fallback procedure we use during sampling, we employ a look-ahead in order to determine if an *unseen state* (i.e., configuration of tile types that was not observed during training) is reached. If the look-ahead finds an unseen state, the tiles are resampled. If no combination from sampling avoids the unseen state, it falls back to a simpler model to sample. We experimented with $l \in \{0, 1, 2, 3, 5\}$ when sampling using our MdMC models.

Figure 3.2 shows the network structures used in our MdMC experiments. For the MdMC model, we used the network structure of $ns = ns_3$, which falls back to $ns_2$, which falls back to $ns_1$, which falls back to $ns_0$. We chose a baseline configuration for each domain and varied the $s$ and $l$ values individually from those baselines.

Evaluation Metrics

We evaluate the levels generated by the different model configurations using the following metrics.

- **Playability** (*Play*): We tested 1000 sampled levels to determine the percentage of them which are playable for a given configuration. For *Super Mario Bros.* we use Adam Summerville’s A* agent. For *Lode Runner*, we checked for the existence of a path between all treasures, as collecting all the
treasures in a level completes it. For Kid Icarus we check for the existence of a path between the start and end of the level. We recorded the percentage of playable levels for each configuration for each domain. Notice, even with a very low percentage of playable levels, we can generate some desired number of playable levels given enough time. Therefore, this playability metric can be seen as a metric of how well the model captures the features that allow for playability and as an efficiency measure for the algorithms.

- **Log-likelihood (LL):** We compute the average log-likelihood of the 1000 sampled levels for each domain and configuration by training an MdMC with $ns = n_{S}$, and taking the sum of the log-likelihoods of each tile in a level, given the trained model. We ignore unseen states while calculating this sum.

- **Unseen States (US):** We determine the average number of unseen states over the 1000 sampled levels in each domain and configuration by training an MdMC as with the log-likelihood metric and recording the number of unseen states in each level, then averaging over the number of levels.

- **Expressive Range:** We determined the expressive range of the configuration which produced the most playable Super Mario Bros. levels, using the levels sampled with that model configuration. The expressive range of a model refers to the variety of levels the model can sample. We measured the expressive range of a model by plotting a two-dimensional heat map using the linearity and leniency of each level sampled by that model configuration:

  - **Linearity:** This refers to the trend of the vertical positions of platforms and ground. We measure linearity by treating each platform or ground section as a point and using linear regression to find the best-fit line for those points. We normalize the sum of the distances of the point from the best-fit line into $[0, 1]$. The normalized linearity value is inversely proportional to how linear the level is.

  - **Leniency:** This is used to measure how forgiving a level is, or how likely a player is to be harmed in that level. We measure the leniency of a level by summing the number of gaps, weighted by their length, and the number of enemies weighted by 0.5, and normalizing that value into $[0, 1]$. 

---

**Chapter 3: Multi-dimensional Markov Chain Level Generation**

3.3 Experiments
Table 3.1: Multi-dimensional Markov Chain Results: This table shows the results of generating 1000 levels in *Super Mario Bros.*, *Lode Runner*, and *Kid Icarus*. This table reports the percentage of playable levels generated in each domain with each model configuration as well as the average log-likelihood of those levels, and the average number of unseen states in each level.

<table>
<thead>
<tr>
<th></th>
<th><strong>Super Mario Bros.</strong></th>
<th></th>
<th><strong>Lode Runner</strong></th>
<th></th>
<th><strong>Kid Icarus</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Play</td>
<td>LL</td>
<td>US</td>
<td>Play</td>
<td>LL</td>
</tr>
<tr>
<td>Basic</td>
<td>26.6%</td>
<td>-187.457</td>
<td>6.62</td>
<td>1.0%</td>
<td>-126.95</td>
</tr>
<tr>
<td><strong>s</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>29.4%</td>
<td>-181.75</td>
<td>1.51</td>
<td>1.0%</td>
<td>-127.594</td>
</tr>
<tr>
<td>2</td>
<td>32.5%</td>
<td>-181.15</td>
<td><strong>0.27</strong></td>
<td>1.3%</td>
<td>-128.427</td>
</tr>
<tr>
<td>3</td>
<td>30.9%</td>
<td>-181.83</td>
<td>0.29</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4^SL</td>
<td>34.6%</td>
<td>-180.56</td>
<td>0.81</td>
<td>1.3%</td>
<td>-127.701</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>34.5%</td>
<td>-183.81</td>
<td>0.49</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.6%</td>
<td>-124.813</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>35.4%</td>
<td>-182.94</td>
<td>0.28</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16^K</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.1%</td>
<td>-346.55</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.7%</td>
<td>-125.934</td>
</tr>
<tr>
<td>32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>40</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>80</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>160</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>l</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>35.6%</td>
<td>-186.49</td>
<td>9.02</td>
<td>2.6%</td>
<td>-128.668</td>
</tr>
<tr>
<td>1</td>
<td>34.5%</td>
<td>-181.03</td>
<td>0.91</td>
<td>1.7%</td>
<td>-128.148</td>
</tr>
<tr>
<td>2^SL^K</td>
<td>34.6%</td>
<td>-180.56</td>
<td>0.81</td>
<td>1.3%</td>
<td>-127.701</td>
</tr>
<tr>
<td>3</td>
<td><strong>36.4%</strong></td>
<td>-179.372</td>
<td>0.69</td>
<td>0.9%</td>
<td>-127.99</td>
</tr>
<tr>
<td>5</td>
<td>34.2%</td>
<td>-181.634</td>
<td>0.87</td>
<td>1.6%</td>
<td>-128.362</td>
</tr>
</tbody>
</table>

### 3.3.3 Results

We chose the baseline configuration of $s = 4$ and $l = 2$ for *Super Mario Bros.* and *Lode Runner*, and a baseline configuration of $s = 16$ and $l = 2$ for *Kid Icarus*. Using that baseline as a starting point, we report experiments varying only one of those variables at a time, allowing us to explore the effects of each of the individual variables on the model. Additionally, we compare against the basic MdMC approach with $l = 0$, $s = 1$, and no fallback. In each domain, we trained each model using all the training maps for that domain, and sampled 1000 levels with each configuration to evaluate playability, log-likelihood, and number of unseen states. For *Super Mario Bros.* we then used the levels sampled with the configuration that yielded the most playable maps to evaluate the expressive range of the model.
Table 3.1 shows the results of our MdMC experiments where we varied the number of row splits and the length of look-ahead from the baseline configurations described above. Note, the baseline configurations are denoted in the table with \( S \) for *Super Mario Bros.*, \( L \) for *Lode Runner*, and \( K \) for *Kid Icarus*. In our experiments not all configurations were applicable in each domain. We denote a configuration that was not tested with a dash (−) in our tables. From our results, we can see that for *Super Mario Bros.*, more than 3 row splits resulted in more playable levels (around 35% instead of around 30%). This is likely because the structures in this domain tend to be around 4 tiles large, which is also how high the player can jump. Therefore, splitting the level into sections of that size or smaller allows for the structures to be more easily modeled and reproduced at passable sizes. We can also see that the length of the look-ahead does not greatly impact the playability of the sampled levels, but using a non-zero look-ahead value does substantially reduce the average number of unseen states and slightly impacts the likelihood of the levels. This is to be expected, as the look-ahead is exactly meant to avoid unseen states while sampling. Figure 3.4 shows a level sampled using the basic MdMC approach (top, \( l = 0, s = 1 \)), the baseline approach (middle, \( l = 2, s = 4 \)), and the best performing approach (bottom, \( l = 3, s = 4 \)).

For *Kid Icarus* we see that some of the configurations succeed in generating very few playable levels. This is likely because of the levels’ vertical orientation and sparsity of structures, which require careful platform placement in order for them to be playable. Splitting the levels’ in sections with the row split does not
help in this domain, as the levels are fairly homogeneous through all height. However, we are able to see that using a non-zero look-ahead value drastically reduces the average number of unseen states in generated levels. Figure 3.5 shows several sections of generated Kid Icarus levels: a section of a level sampled with the basic MdMC approach (left), a section of a level sampled using the baseline MdMC approach (center), and a section of a level sampled using the configuration that sampled the most playable levels (right). Notice the poorly replicated “door” structures in the center level and the incredibly long gaps in the right-most level. In later chapters we explore methods for better capturing these types of levels.

Table 3.2: MdMC Lode Runner Limited Training Data Results: This table shows the results of generating 1000 levels in Lode Runner after training an MdMC on only the first ten training levels. This table reports the percentage of playable levels generated as well as the average log-likelihood of those levels, and the average number of unseen states in each level.

<table>
<thead>
<tr>
<th>Play</th>
<th>LL</th>
<th>US</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.7%</td>
<td>-79.71</td>
<td>0.61</td>
</tr>
</tbody>
</table>
For *Lode Runner*, we can see that the MdMC approach is able to sample more playable levels than in the *Kid Icarus* domain, but still many fewer than for *Super Mario Bros*. This is likely due to the complex paths required in order for a level in this domain to playable. Additionally, there is a large number of training levels in this domain (compared to our other domains), and some of those training levels contain special puzzles or unusual level set-ups (such as only containing ladders, or requiring a unique path by falling from a series of platforms) which may be causing our approach to learn spurious relationships. In order to test this we trained an MdMC using the baseline configuration on 10 levels selected from the training set, which after manual inspection appeared to not contain any unusual set-ups. Table 3.2 shows the results of this experiment. Notice, the playability is much higher than the best performing MdMC approach using all the training levels (17.7% as compared to 2.6%). This suggests that carefully selecting training data can greatly benefit machine learning-based PCG approaches. We explore this possibility more in Section 8.1. Figure 3.6 shows several generated *Lode Runner* levels: a level sampled using the basic MdMC configuration (top-left), a level sampled using the baseline configuration (top-right), a level sampled using the configuration that sampled the most playable levels (bottom-left), and a level sampled using the MdMC trained on only 10 levels selected from the training set. Notice that the level sampled by the MdMC trained on only 10 levels (bottom-right) has simpler and more cohesive structures than the other configurations.

Figure 3.7 shows the expressive ranges of the training levels (A) and the expressive range of our MdMC model with $s = 4$ and $l = 3$ (B), both for *Super Mario Bros*. The expressive range is found by computing the leniency ($x$-axis) and the linearity ($y$-axis) of each level, normalizing the values across all the models (including models introduced later in the dissertation to allow for uniform comparisons), and then plotting a heat map of the levels. The top-right corresponds to non-linear and difficult levels, whereas the bottom-left corresponds to linear and easy levels. Notice that the expressive range of our MdMC model closely follows the distribution of the training maps, further emphasizing that this approach is able to accurately model the target domain.
Figure 3.6: This figure shows example *Lode Runner* levels sampled using the basic MdMC method (top-left), the baseline MdMC method with $s = 4$ and $l = 2$ (top-right), the best performing MdMC method with $s = 4$ and $l = 0$ (bottom-left), and the MdMC with the baseline configuration trained on only 10 selected levels (bottom-right).

Figure 3.7: This figure shows heat maps of the *Super Mario Bros.* levels used for training (left) and those generated with the best performing configuration of the MdMC approach (right). The $x$-axis is leniency and the $y$-axis is linearity. The top right corner of the heat map represents highly-nonlinear and very difficulty levels (high leniency values), and the lower left corner represents the most linear and easy (low leniency) levels. Note, all linearity and leniency values are normalized over all levels generated by all models (including models described in future chapters) to allow for uniformity in comparisons.

Models as Distributions

Above we describe our multi-dimensional Markov chain approach as a probability distribution, specifically, a conditional distribution of tiles from which we sample individual portions of an output level. However, it is important to notice that this model (and other PCG approaches) can be described as defining a distribution over designs. That is, these models can be described as a probability distribution over the set of possible output levels. Specifically, given the desired dimensions of an output level, and sampling using our standard
MdMC approach with no look-ahead and no row splits, we can easily compute the probability of an entire level. We do so by taking the product of the probability of the tile type at each position in a generated level:

$$P(m|M) = \prod_{pos \in \text{positions}(m)} P_M(m[pos]\mid\text{config}(m[pos])), \text{ where } m \text{ is the generated level, } M \text{ represents the trained model used to generate the level (in this case using our MdMC approach), } pos \text{ is a position in the level, positions}(m) \text{ gives the set of all positions in the level, and } \text{config}(m[pos]) \text{ gives the configuration of surrounding tiles for the given position in the level, all as described in Sections 3.2.2 and 3.2.3. Note that } P_M(m[pos]\mid\text{config}(m[pos])) \text{ is defined when the particular configuration was observed in the training data. In order to be able to evaluate any arbitrary level, we define any unobserved configurations (i.e., unseen states) using a uniform distribution. The product of the probabilities at each position in the level, gives the probability of the entire level, given the trained model. This is touched on briefly in our results, when we discuss the log-likelihood of an output artifact given the trained model. Notice, if a look-ahead is used and if the row split techniques are used, then computing this probability becomes more complicated. Specifically, the look-ahead technique implements sampling of levels that have no unseen states: } P(m|M, \text{UnseenStates}_M(m) = 0) \text{. We show the simplest version above for clarity. Furthermore, for other models where the levels are not created at the tile level (e.g., Guzdial and Riedl’s}^8 \text{ graphical approach), the probability of a level would need to be defined differently, but the model itself could still be defined as a distribution over possible levels.}

Describing a model in this way can provide some insight into how likely given outputs may be. This information can be useful when analyzing a model’s expressive range or exploring methods for expanding a model’s expressivity. However, for our models, this description does not provide many other affordances. Furthermore, many PCG approaches (not only machine learning-based PCG approaches) can be described as distributions over designs. This requires determining the output space of the approach, and the probability of arriving at specific elements using that approach. Performing such an analysis may be more straightforward when using statistical approaches, but it does not preclude other approaches. While this representation of an approach is interesting, it may obfuscate the workings of our approaches. Furthermore, because of its applicability across many PCG approaches, it does not provide us with a useful generalization across specifically
machine learning-based approaches. Therefore, we will not use this representation further in this document.

### 3.4 Conclusions

In this chapter we presented our first machine learning-based level generation approach, multi-dimensional Markov chains (MdMCs), which is meant to be a general level generation approach, applicable across multiple domains, and alleviate the amount of domain knowledge required from the user. This approach learns a distribution of tile types and relationships between those tile types for a domain by observing occurrences in a set of training levels represented with those tile types. The MdMC can then be used to generate levels by sampling a new level tile by tile using the learned distribution and the previously sampled tiles in that level.

We tested this approach in 3 domains, *Super Mario Bros.*, *Lode Runner*, and *Kid Icarus*.

We found that the MdMC approach is able to reliably generate usable levels for *Super Mario Bros.*, but struggled with the other two domains. We believe it struggled in *Lode Runner* because of potentially spurious training data. We tested this by more carefully selecting the levels used during training, and found that the result improved dramatically, supporting our belief. For *Kid Icarus*, we believe that the MdMC approach struggled because the levels are vertically oriented and sparse. This means that levels need to have carefully placed platforms at proper distances to allow the player to complete the level. The MdMC models local dependencies, and thus struggled to capture the proper platform placement.

In the following chapters we introduce two more machine learning-based approaches in order to explore which patterns and regularities can be captured by different models. In particular, the next chapter presents a hierarchical MdMC approach which aims to capture longer range dependencies and higher level structures in the training levels by representing levels at multiple layers of abstraction.
Chapter 4: Hierarchical MdMC Level Generation

4.1 Introduction

In Chapter 3 we introduced a multi-dimensional Markov chain (MdMC) approach to level generation. We tested that approach through experimentation in several distinct domains: *Super Mario Bros.*, *Lode Runner*, and *Kid Icarus*. Through that experimentation we found that the MdMC approach is able to reliably model and create playable levels for *Super Mario Bros.* (36.4%), which is a fairly simple, linear platforming game. We also saw that the MdMC approach was able to occasionally generate playable levels for *Lode Runner*, a more complex puzzle platforming game that requires paths through the level connecting various collectible objects (17.7% with carefully selected training data, and 2.6% when using all available training data). We believe the MdMC approach struggled with capturing the complex paths and connected structures required for playable *Lode Runner* levels. Lastly, we saw that the MdMC approach was able to generate very few playable levels for *Kid Icarus* (0.6%), which is a simple platforming game, but is vertically oriented with more sparse structures. The MdMC was unable to capture the more long-range dependencies between the platforms and structures in this domain, and thus struggled to create playable levels.

In order to create more general machine learning-based approaches, we explore a hierarchical approach that aims to model a given domain at multiple levels of abstraction. Hierarchical models have been used to increase performance in a variety of machine learning tasks, such as action recognition\textsuperscript{98}, dialogue generation\textsuperscript{99}, and image segmentation\textsuperscript{100}. By representing the training data at multiple levels of abstraction, we aim to more accurately capture and reproduce the structures and intricacies of a given domain while also capturing longer-range dependencies between structures. We experiment with an unsupervised approach to generating the additional abstraction layers so as not to increase the amount of domain knowledge required from the user. However, there are several challenges inherent in this approach. Namely, the challenges are defining a useful abstraction, defining a representation for that abstraction, developing models that can leverage these abstractions and representations, and determining whether these models actually succeed in capturing longer-range
dependencies.

The remainder of this chapter is organized as follows: first, in Section 4.2 we introduce hierarchical multi-dimensional Markov chains (HMdMCs); next, we discuss our hierarchical level representation, and how we train and sample using multiple levels of representation in Section 4.3; we then describe our experiments and results in Section 4.4; and close with our conclusions in Section 4.5.

4.2 Hierarchical Multi-dimensional Markov Chains

We test our HMdMC model using two levels of abstraction (though it could be extended to more). We start with the standard level representation used by the MdMC approach. Then, each additional level of abstraction is meant to capture higher-level structures and patterns, and longer range dependencies. The two levels of abstraction we use are defined below.

- **High-Level**: This level trains and samples using a set of high-level tiles that represent common structures (e.g., slopes, platforms, pipes) throughout the training maps. High-level tiles are created by grouping sections of low-level tiles together in order to represent patterns in the level. Once a map is represented at a high-level, an MdMC is trained on that high-level representation normally. Figure 4.1 shows a section of a *Super Mario Bros.* level (left), that same section represented using low-level tiles (center), and an example high-level representation of that section (right). Notice that each high-level tile is composed of a $4 \times 4$ section of low-level tiles, and each high-level tile type represents a larger pattern occurring in the level.

![Figure 4.1](image.png)
• **Low-Level**: This level trains a separate model for each high-level tile, ensuring that each high-level tile accurately captures the structure it is meant to represent.

### 4.3 Methods

In this section we discuss the techniques employed by our hierarchical MdMC approach. We start with an explanation of our hierarchical level representation. We then explain the techniques used for training and sampling from this model.

#### 4.3.1 High-level Map Representation

This section addresses the challenge of representing the hierarchical abstraction by introducing our high-level level representation. Our hierarchical multi-dimensional Markov chain approach trains and samples at two levels of abstraction, requiring multiple representations of each map. Our HMdMC approach trains using the map representation described in Section 3.2 as the low-level map, and an additional high-level map. A high-level map is represented as a $w/z \times h/z$ two-dimensional array, where $z$ is the height and width of a high-level tile, and $w$ and $h$ are the width and height of the low-level map, respectively. Figure 4.1 shows a section of an input map (1), the low-level representation (2), and a possible high-level representation (3). The high-level map is constructed by mapping each $z \times z$ section from the low-level map to one of a set of high-level tile types, $T_h$, corresponding to structures within the maps (e.g., platforms, hills, or gaps). The following section gives more details on how we define $T_h$ and how we convert a low-level map into a high-level map.

#### 4.3.2 Defining High-level Abstractions

This section addresses the challenge of defining a useful abstraction for our hierarchical approach. Low-level training maps can be created by translating readily available map images from various games using a set of tiles corresponding to low-level elements in that game. High-level maps, however, are not as easily attainable. It is not clear what the set of high-level abstractions should be, or even what the size of a high-level abstraction should be. We propose a method which defines a vocabulary of high-level tiles, and then automatically translates training maps using the defined vocabulary. These high-level tiles are meant to
capture structures and patterns which occur in the given domain. In order to define the high-level tiles we experimented with a manual approach and an unsupervised machine learning approach. We can then translate a map into a high-level map using these high-level tile types by matching the defined high-level tile types with groups of low-level tiles in the standard maps.

In more detail, we generate high-level maps by processing low-level maps using a set of high-level tiles, $T_h$, and a distance function, $d$, that can compare sections of low-level tiles against the tiles in $T_h$. We experimented with two approaches for defining $T_h$:

- **Manual**: For this approach, we observed common structures in the training levels and manually constructed a set $T_h$ of high-level tiles. Specifically, we constructed functions to recognize a set of high-level structures and patterns in a low-level map.

- **Clustering-based**: For this approach, we clustered all of the $z \times z$ sections of low-level tiles from the low-level maps using $k$-medoids. After clustering, $T_h$ is defined as the set of $k$ centroids returned by $k$-medoids.

For the clustering-based approach, we experimented with the following distance functions with $k$-medoids:

- **Direct**: Compares two sections tile by tile, and counts the number of tiles that are different.

- **Histogram**: Converts both section into histograms of low-level tile types. The distance is computed as the Manhattan distance between the resulting histograms.

- **Markov**: Trains a tile distribution for each section using a standard Markov chain. The distance is computed as the Manhattan distance between the resulting distributions.

- **Shape**: Slides one section over the other, and applies the Direct metric on the overlapping area of the two sections, looking for the position in which the Direct distance is minimized after normalization by the overlapping area.

Algorithm 3 converts a low-level map into a high-level representation (using $T_h$) as follows: iterate over the low-level map in increments of $z$ (lines 1-2). At each iteration, define the $z \times z$ section of tiles in the low-level map with its bottom-left corner at $m_{i}[x][y]$ as the current high-level tile to be classified (line 3).
Algorithm 3 High-Level Map Conversion($m_l, d, z, T_h$)

1: for $y \in 0, \ldots, \text{height}(m_l) - 1$, in increments of $z$ do
2:   for $x \in 0, \ldots, \text{width}(m_l) - 1$, in increments of $z$ do
3:      $s = z \times z$ section starting at $m_l[x][y]$
4:      $t^* = \text{argmin}_{t \in T_h} d(t, s)$
5:      $m_h[x/z][y/z] = t^*$
6: return $m_h$

Next, determine which high-level tile in $T_h$ the section is most similar to, using the chosen distance function, $d$ (line 4), and fill the corresponding position in the high-level output map, $m_h$, with the most similar tile from $T_h$ (line 5). Note, for the manual approach, the distance function used is the manually encoded method for each manually defined high-level tile.

4.3.3 Training

Algorithm 4 shows how we train an HMdMC. In order to train, this algorithm requires a set of high-level maps ($M_h$) and their corresponding low-level maps ($M_l$); it requires the network structures to be used by the MdMCs for both the high and low-level models ($ns_h$ and $ns_l$), respectively; the algorithm requires the set of high-level tile types representing the high-level maps ($T_h$) and the set of tile types representing the low-level maps ($T_l$); and finally, it requires the size of the high-level tiles, $z$ (i.e., each high-level tile represents a section of $z \times z$ low-level tiles in the low-level maps).

Intuitively, this algorithm works by training a single high-level model on the high-level map, and separate low-level models for each of the high-level tile types. This ensures that the structures and patterns encoded by each high-level tile type are captured accurately by the overall model. Algorithm 4 returns the high-level conditional probability distribution (CPD), $P_{ns_h}$ trained using the specified network structure, $ns_h$, and the set of low-level CPDs, $P^{\text{lh}}_{ns_l}$, trained on the low-level tiles within a specified high-level tile, $t_h$, using a specified network structure, $ns_l$. The returned set is denoted by $P_{ns_l}$.

The algorithm first trains an MdMC with Algorithm 1 using the high-level training maps. Next, it trains a separate MdMC for each high-level tile type. That is, it passes through the low-level training maps maintaining separate absolute counts for each high-level tile, and only updating the counts for the high-level tile that
Algorithm 4 train-HMdMC$(n_{sh}, n_{sl}, M_h, M_l, T_h, T_l, z)$

1: $\mathcal{P}_{n_{sl}} = \text{train-MdMC}(n_{sh}, M_h, T_h)$
2: for each $m_l \in M_l$ and corresponding $m_h \in M_h$ do
3:   for pos$_l \in \text{positions}(m_l)$ and corresponding pos$_h \in \text{positions}(m_h)$ do
4:     $t_l = m_l[pos_l]$  \hspace{1cm} \triangleright \text{Low-level tile}
5:     $c = \text{config}(m_l, pos_l, n_{sl})$
6:     $t_h = m_h[pos_h]$  \hspace{1cm} \triangleright \text{Tile configuration}
7:     $S[t_h][c][t_l] = S[t_h][c][t_l] + 1$
8:     $\mathcal{P}_{n_{sl}} = \emptyset$
9: for each $t_h \in T_h$ do
10:   for $c \in C_l$ do
11:     for each $t_l \in T_l$ do
12:       $P_{n_{sl}}^h(t_l|c) = S[t_h][c][t_l]/\sum_{v \in T_l} S[t_h][c][v]$
13:     $\mathcal{P}_{n_{sl}} = \mathcal{P}_{n_{sl}} \cup \{P_{n_{sl}}^h \}$
14: return $\mathcal{P}_{n_{sh}}, \mathcal{P}_{n_{sl}}$

the current section of the low-level map corresponds to. Note, no position in a low-level map may correspond to multiple high-level tile types. Therefore, each high-level tile has a distinct probability distribution.

An element of $\mathcal{P}_{n_{sl}}$ can be accessed by indexing with the high-level tile type. That is, $\mathcal{P}_{n_{sl}}[t_h] = P_{n_{sl}}^h$. For indexing, we assume that each low-level CPD in the set $\mathcal{P}_{n_{sl}}$ is trained using the same network structure. The sampling algorithm described in the next section requires a set, $\mathcal{P}$, composed of the low-level CPDs corresponding to all the high-level tiles for all the different network structures, (i.e., $\mathcal{P} = \{P_{n_{sl}} : n_{sl} \in NS_l\}$), and a set, $\mathcal{P}$, of high-level CPDs trained using various network structures. $\mathcal{P}$ is defined the same way as in the MdMC sampling section; the only difference is the set of training maps. Elements of $\mathcal{P}$ can be accessed through indexing as follows: $\mathcal{P}[n_{sl}][t_h] = P_{n_{sl}}^h$ (i.e., the network structure, $n_{sl}$, is used to index into the proper subset, $\mathcal{P}_{n_{sl}}$, while the high-level tile type, $t_h$, is used to index into the proper CPD within that subset). $\mathcal{P}$ is indexed the same way as in Section 3.2.3.

4.3.4 Sampling

Algorithm 5 shows the process of sampling a new level using an HMdMC. Though sampling a level using an HMdMC is similar to sampling using an MdMC, there are two key differences. The first difference is that to sample a level, the hierarchical model first samples a high-level map (line 2), $m_h$, using Algorithm 2. The
Algorithm 5 sample-HMdMC

```plaintext
1: function sampleMapH(lh, ll, NSh, NHL, P, Th, Tl)
2:    mh = sampleMap(lh, NSh, P, Th)
3:    ml = Empty map
4:    for pl \in positions(ml) and corresponding
5:      ph \in positions(mh) do
6:        if sampleTileH(mh, ml, pl, ph, lh, ls, PS, Tl) then
7:            break
8:    return n
9: function sampleTileH(mh, ml, pl, ph, lh, ls, PS, T)
10:   if l < 0 \lor outsideMap(pos, ml) then
11:      return true
12:   T* = T
13:   c = config(ml, pl, ls)
14:   th = mh[ph]
15:   if c is an unseen state then
16:      return false
17:   else
18:      t sampled according to PS[ls][th](T*|c)
19:      ml[pl] = t
20:     while ¬sampleTileH(mh, ml, pl+1, ph, lh-1, ls, PS, T*) do
21:      T* = T* \setminus t
22:      if T* = \emptyset then
23:         return false
24:      else
25:         t sampled according to PS[ls][th](T*|c)
26:         ml[pl] = t
27:     return true
```

other difference occurs when sampling a tile in the low-level map, ml. When sampling a low-level tile, the position in the high-level map must be considered; which high-level tile the current low-level tile’s position falls within (line 4) determines which CPD will be used to sample the low-level tile (lines 23, 31). Other than these two differences, Algorithm 5 follows the same procedures of Algorithm 2.

### 4.4 Experiments

In this section we first discuss our domains and how we set up our experiments to evaluate our HMdMC model. We then discuss the results of those experiments, including how the manual hierarchical approach compares to the clustering-based approach.
4.4.1 Experimental Set-up

Domains

For our experiments, we use the same 3 domains as the previous chapter. Namely, we use *Super Mario Bros.*, a simple, linear platforming game; *Kid Icarus* a simple, linear platforming game with vertically oriented and more sparse levels with longer-range dependencies between structures; and *Lode Runner*, a more complex puzzle platforming games that requires intricate paths through levels. More information on how we represent these domains can be found in Appendix A.

We applied our HMdMC approaches to our domains varying the following parameters:

- **High-level look-ahead** ($l_h$): This determines the depth of the tree when checking for unseen states after sampling each tile in the high-level map. We experimented with $l_h \in \{0, 1, 2, 3, 5\}$ when sampling using our HMdMC models. Note, we use a constant low-level look-ahead of 3 for all domains with the hierarchical approaches based on our previous results.

- **Tile Size** ($z$): This determines the height and width of the high-level tiles in low-level tiles used by our HMdMC models. The possible values for $z$ are determined by the chosen map size in each domain. That is, we want to use a $z$ that can evenly divide the maps into high-level tiles. Therefore, we experimented with $z \in \{2, 3, 4, 6, 12\}$ for *Super Mario Bros.*, $z \in \{2, 4, 8, 16\}$ for *Lode Runner*, and $z \in \{2, 4, 8, 16\}$ for *Kid Icarus*.

- **Number of Clusters** ($k$): This determines the number of clusters for our clustering-based HMdMC approach (i.e., the number of high-level tile types). We experimented with $k \in \{8, 12, 16, 20, 24, 28, 32\}$ for all domains.

- **Distance Metric** ($d$): This is the metric used by $k$-medoids to form the clusters and extract the high-level tiles for our clustering-based HMdMC models. It is also used to convert low-level maps into high-level representations. We experimented with $d \in \{\text{Direct, Histogram, Markov, Shape}\}$, as defined in Section 4.3.2. Notice, for our manual HMdMC approach we use manually defined functions to identify each manually defined tile type.
Figure 4.2: This figure shows heat maps of the Super Mario Bros. levels used for training (top-left) and those generated with the best performing configuration of the manual HMdMC approach (bottom-left), the best configuration of the clustering-based HMdMC (bottom-right), and the best performing MdMC configuration for comparison (top-right). The $x$-axis is leniency and the $y$-axis is linearity. The top right corner of the heat map represents highly-nonlinear and difficult levels (high leniency values), and the lower left corner represents the most linear and easy (low leniency) levels. Note, all linearity and leniency values are normalized over all levels generated by all models (including models described in future chapters) to allow for uniformity in comparisons.

For the high-level training and sampling of our HMdMC models, we use the network structure $ns_2$, which falls back to $ns_1$, which falls back to $ns_0$, each of which can be seen in Figure 3.2. For the low-level training and sampling of our HMdMC models, we use $ns_3$, which falls back to $ns_2$, which falls back as above. Additionally, we chose baseline configurations for each of our domains and varied the above parameters individually from the baselines. We evaluate the configurations using the same metrics as with the MdMC approach described in Section 3.3.2.

4.4.2 Manual Hierarchical Multi-dimensional Markov Chains Results

For our manual HMdMC experiments, we vary $l_h$ and $z$ as we use only the manually-defined high-level tile types. Specifically, we define a set of 8 tile types: a predominantly empty space tile, a pipe structure tile, a platform structure tile, a pillar tile (i.e., tall thin structure), a plateau structure tile (i.e., shorter wide structure),
an uphill structure tile, a downhill structure tile, and a default tile to capture any structures not categorized by the previous tile types. We chose a baseline of $l_h = 2$ and $z = 4$ based on preliminary experiments. For these experiments, we converted our training maps into high-level representations using our manually-defined tile types, trained our model as in Algorithm 4, and sampled 1000 levels per configurations using Algorithm 5. We only investigate this approach in the domain of *Super Mario Bros.*, as it requires a large amount of domain knowledge in order to define the various high-level tile types. Tables 4.1 show the percentage of playable maps sampled with the various configurations, the average log-likelihood for the maps generated with those configurations, and the average number of unseen states observed in those generated levels. The baseline configuration is denoted with an $S$ in the table. Note that we compute the log-likelihood and the number of unseen states using a standard MdMC with network structure $nS_3$ from Figure 3.2. This table shows that the manual hierarchical MdMC approach is able to achieve a higher percentage of playable levels (49.5% as compared to 36.4% with the MdMC approach). This is promising as it shows that hierarchical modeling approaches can offer improvements over standard non-hierarchical approaches. Notice that the percentage of playable levels sampled is heavily influenced by the size of the high-level tiles. Specifically, this approach performs best when $z = 3$ and $z = 4$. This is likely due to the size of the structures that appear in the training levels, which tend to be around 3 to 4 tiles tall. This suggests that the ideal size of the high-level tiles is likely to vary depending on the domain. However, the likelihood increases and number of unseen states decreases as the size of the tiles increases. This is because the larger the high-level tiles, the closer the low-level distributions for each high-level tile gets to the distribution of the standard MdMC trained distribution.

From the table we can also see that the high-level look-ahead value has very little effect on the playability
Table 4.1: Manual Hierarchical Multi-dimensional Markov Chain Results: This table shows the results of generating 1000 levels in *Super Mario Bros.* using the manual setting of the hierarchical MdMC approach. This table reports the percentage of playable levels generated with each model configuration as well as the average log-likelihood of those levels, and the average number of unseen states in each level.

<table>
<thead>
<tr>
<th>z</th>
<th>Play</th>
<th>LL</th>
<th>US</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>37.7%</td>
<td>-213.41</td>
<td>9.87</td>
</tr>
<tr>
<td>3</td>
<td>43.4%</td>
<td>-186.27</td>
<td>5.60</td>
</tr>
<tr>
<td>$4^S$</td>
<td>49.5%</td>
<td>-183.17</td>
<td>2.69</td>
</tr>
<tr>
<td>6</td>
<td>34.6%</td>
<td>-178.60</td>
<td>1.58</td>
</tr>
<tr>
<td>12</td>
<td>27.2%</td>
<td>-176.82</td>
<td>0.13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>lh</th>
<th>Play</th>
<th>LL</th>
<th>US</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>47.1%</td>
<td>-184.179</td>
<td>2.84</td>
</tr>
<tr>
<td>1</td>
<td>49.1%</td>
<td>-181.614</td>
<td>2.78</td>
</tr>
<tr>
<td>$2^S$</td>
<td>49.5%</td>
<td>-183.17</td>
<td>2.69</td>
</tr>
<tr>
<td>3</td>
<td>49.3%</td>
<td>-183.65</td>
<td>2.68</td>
</tr>
<tr>
<td>5</td>
<td>49.0%</td>
<td>-183.06</td>
<td>2.66</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Play</th>
<th>LL</th>
<th>US</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>MdMC</em></td>
<td>36.4%</td>
<td>-179.372</td>
<td>0.69</td>
</tr>
</tbody>
</table>

of the levels or on the likelihood and unseen states. This is not surprising as the high-level look-ahead only ensures that the high-level tiles are placed in patterns that have been observed before, but does not affect the placement of the low-level tiles or the sampling of those structures.

Figure 4.2 shows the expressive range of our manual and clustering-based HMdMC approaches (bottom-left and bottom-right, respectively). Notice that the hierarchical approaches have similar expressive ranges to the standard MdMC approach. However, the manual hierarchical approach spreads more over the linearity axis ($y$) and less over the leniency axis ($x$). This indicates that the levels sampled with the manual hierarchical approach are slightly easier (in terms of enemies and gaps), but that the placement of the structures is more varied. Figure 4.3 (top) shows a level sampled using the manual hierarchical approach.

### 4.4.3 Clustering-based Hierarchical Multi-dimensional Markov Chains Results

The manual HMdMC approach offered improved results over the standard MdMC approach at the cost of requiring a large amount of domain knowledge. With our clustering-based HMdMC approach, we seek to remove the need for domain knowledge while retaining the benefits of the hierarchical approach. For our
Table 4.2: Clustering-based Hierarchical Multi-dimensional Markov Chain Results: This table shows the results of generating 1000 levels in Super Mario Bros., Lode Runner, and Kid Icarus using the clustering setting of the hierarchical MdMC approach. This table reports the percentage of playable levels generated in each domain with each model configuration as well as the average log-likelihood of those levels, and the average number of unseen states in each level.

<table>
<thead>
<tr>
<th>k</th>
<th>Super Mario Bros.</th>
<th>Lode Runner</th>
<th>Kid Icarus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Play</td>
<td>LL</td>
<td>US</td>
</tr>
<tr>
<td>8</td>
<td>37.8%</td>
<td>−182.63</td>
<td>0.76</td>
</tr>
<tr>
<td>12</td>
<td>37.6%</td>
<td>−185.18</td>
<td>1.02</td>
</tr>
<tr>
<td>16</td>
<td>37.6%</td>
<td>−180.79</td>
<td>0.90</td>
</tr>
<tr>
<td>20</td>
<td>34.6%</td>
<td>−176.63</td>
<td>1.53</td>
</tr>
<tr>
<td>24</td>
<td>35.7%</td>
<td>−179.17</td>
<td>1.54</td>
</tr>
<tr>
<td>28</td>
<td>35.3%</td>
<td>−176.91</td>
<td>2.70</td>
</tr>
<tr>
<td>32</td>
<td>48.0%</td>
<td>−177.34</td>
<td>1.45</td>
</tr>
<tr>
<td></td>
<td>z</td>
<td>Play</td>
<td>LL</td>
</tr>
<tr>
<td>2</td>
<td>46.4%</td>
<td>−224.49</td>
<td>21.74</td>
</tr>
<tr>
<td>3</td>
<td>35.8%</td>
<td>−187.42</td>
<td>3.53</td>
</tr>
<tr>
<td>6</td>
<td>37.6%</td>
<td>−180.79</td>
<td>0.90</td>
</tr>
<tr>
<td>8</td>
<td>31.0%</td>
<td>−183.42</td>
<td>0.99</td>
</tr>
<tr>
<td>12</td>
<td>27.1%</td>
<td><strong>175.35</strong></td>
<td>0.43</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>l_n</td>
<td>Play</td>
<td>LL</td>
</tr>
<tr>
<td>0</td>
<td>36.9%</td>
<td>−179.88</td>
<td>0.97</td>
</tr>
<tr>
<td>1</td>
<td>32.6%</td>
<td>−179.77</td>
<td>0.92</td>
</tr>
<tr>
<td>2</td>
<td>37.6%</td>
<td>−180.79</td>
<td>0.90</td>
</tr>
<tr>
<td>3</td>
<td>34.1%</td>
<td>−180.62</td>
<td>0.82</td>
</tr>
<tr>
<td>5</td>
<td>33.8%</td>
<td>−179.45</td>
<td>0.94</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>Play</td>
<td>LL</td>
</tr>
<tr>
<td>Direct</td>
<td>37.6%</td>
<td>−180.79</td>
<td>0.90</td>
</tr>
<tr>
<td>Histogram</td>
<td><strong>50.8%</strong></td>
<td>−184.15</td>
<td>3.92</td>
</tr>
<tr>
<td>Markov</td>
<td>32.6%</td>
<td>−185.13</td>
<td>1.73</td>
</tr>
<tr>
<td>Shape</td>
<td>48.8%</td>
<td>−180.17</td>
<td>3.61</td>
</tr>
</tbody>
</table>

Figure 4.4: This figure shows a set of high-level tile types defined using the clustering method with the Histogram distance metric.
clustering-based HMdMC experiments, we chose the baseline configurations of $l_h = 2$, $z = 4$, $k = 16$, and $d = \text{Direct}$ for each of the domains based on preliminary experiments. Recall, we use a constant low-level look-ahead of 3 for each domain.

For each configuration of parameters in each domain, we performed $k$-medoids clustering to define the sets of high-level tile types. We then trained our HMdMC model, as described in Section 4.3.3, using the converted high-level maps and the low-level maps, and sampled 1000 levels for each configuration. For Super Mario Bros., we then used the maps sampled using the configurations that yielded the highest percentage of playable levels to create a heat map showing the expressive range of the model. Figure 4.4 shows a set of high-level tile types for Super Mario Bros. found with the best performing configuration, which uses the Histogram distance metric. Notice, several of the defined high-level tile types match with our manually-defined tile types. For example, both sets have an uphill structure, both have pipe structures (although the clustering approach differentiates several different pipe structures), and both have platform structures (again the clustering approach differentiates several). However, the clustering approach also finds several other high-level tile types not identified in the manual approach, such as the coins tile, the cannon tile, and the tile containing the flying enemy. This shows that our unsupervised method is able to extract meaningful high-level tile types from the levels (i.e., the tile types that match the manual specifications), but is also able to expand upon the known structures with additional interesting patterns.

Table 4.2 shows the results of our clustering-based HMdMC experiments. Notice that the baseline configuration settings are denoted by the superscript baseline in the table. For Super Mario Bros., we found that the choice of distance metric has the biggest impact on the playability of the sampled levels. Specifically, we see that the Histogram and Shape distance metrics allow for the highest percentage of playable levels. However, the levels sampled with these metrics have many more unseen states than the other configuration (excluding the configuration with $z = 2$). We can also see that using larger sets of high-level tiles ($k = 32$) can improve the playability of the generated levels, while keeping the number of unseen states low. This is likely due to the large number of low-level tiles used to represent the training levels, which may require a larger set of high-level tiles in order to capture the intricacies of the domain. Notice, the clustering-based HMdMC performs similarly to the manual HMdMC. This is an important result, because it shows that the
Figure 4.5: This figure shows an example *Lode Runner* level sampled using the baseline HMdMC configuration with $z = 4$, $l_h = 2$, $k = 16$, $d = {\text{Direct}}$ (left), and with the best performing configuration with $z = 8$, $l_h = 2$, $k = 16$, and $d = {\text{Direct}}$ (right).

benefits of the hierarchical model are not tied to the manual definition of high-level tile types. Figure 4.2 (bottom-left and bottom-right) shows the expressive ranges of the hierarchical approaches. Notice, that the clustering-based approach spreads along the linearity axis ($y$), more than the manual approach. This shows that with the clustering hierarchical approach the levels sampled have more varied linearity values. Figure 4.3 (bottom) shows a map sampled using the configuration that yielded the most playable maps.

For *Lode Runner*, we found that the hierarchical approach samples around the same number of playable levels as the standard MdMC model. This may be due to the configurations we explored being sub-optimal for this domain. For example, we see the best results with a large tile size, $z = 8$, and with few high-level tile types ($k = 8$), suggesting that more informative distance metrics may be helpful in this domain. Figure 4.5 shows a *Lode Runner* level sampled using the baseline configuration (left) and using the configuration that yielded the most playable levels (right).

For *Kid Icarus*, we were able to sample the same percentage of playable levels as the standard MdMC approach (0.6%). The best performance occurs when $z = 16$, which is close to simply using the standard MdMC approach. This suggests that the low level MdMC models are doing most of the work. As with *Lode Runner*, more informative distance metrics may be beneficial here. Figure 4.6 shows a level sampled using the baseline configuration (left) and a level sampled using the configuration that produced the most playable levels (right). Notice that in both levels there are portions that are playable, but there are also portions that contain platforms that are too far apart to be reached, which render the levels unplayable.
4.5 Conclusions

In this chapter we introduced a hierarchical multi-dimensional Markov chain (HMdMC) approach to level generation. The goal was to develop a model that would be able to capture longer-range tile dependencies and model high-level structures from the training levels by representing levels at multiple levels of abstraction.

We tested this approach in three domains, *Super Mario Bros.*, *Lode Runner*, and *Kid Icarus*.

We found that the HMdMC approach was able to improve upon the MdMC results in *Super Mario Bros.* Furthermore, we saw that automatically defining the high-level tile types through clustering performed similarly to using manually defined high-level tile types. This is promising, as it suggests that the HMdMC approach can be used without requiring additional domain knowledge. For the other domains, we saw that the HMdMC approach performed similarly to the MdMC approach. We believe this is due to the models not having an understanding of what makes a level playable. In *Super Mario Bros.* playability is tied directly to...
the height of structures and width of gaps, which can be modeled locally. However, in *Lode Runner*, playability is more global; a path must exist between all the treasure in a level, and a path can be non-existent for a number of reasons. Additionally, in *Kid Icarus*, playability is tied to the structures, but the placement of those structures is the most important feature, not necessarily their size. Incorporating player information to improve the quality of generated levels is a concept we explore in later chapters (namely Sections 7.2 and 7.3).

The MdMC and HMdMC approaches we have discussed thus far model levels sequentially, and therefore assume a sequential dependence between structures and tiles. However, this may not be the case in all domains. For instance, in *Kid Icarus* the player must move vertically through the level, but while doing so moves left and right across the various platforms. In the next section we introduce a Markov random field approach that tries to capture local tile dependencies without assuming sequential dependence between them.
Chapter 5: Markov Random Field Map Generation

5.1 Introduction

In the previous chapters we introduced two machine learning-based level generation approaches, multi-dimensional Markov chains (MdMCs) and hierarchical multi-dimensional Markov chains (HMdMCs). These approaches assume sequential dependencies between the structures and objects in the levels, because the models are built from Markov chains where the tile type at each position depends on the previous. These approaches were successful in the horizontally linear domain of Super Mario Bros., where the assumption of sequential dependence makes sense. However, the MdMC and HMdMC approaches struggled in Kid Icarus where the player needs to travel back and forth to ascend through the level. In order to more accurately model domains where the sequential dependence assumption doesn’t hold, we need a different approach.

In this chapter we introduce a Markov random field (MRF) approach to procedural level generation. MRFs have previously been used extensively in the field of computer vision and image analysis/recognition. We aim to leverage MRFs to model and generate levels which have more semantic information and requirements than images. Markov random fields model the relationships between a state and its neighbors in all directions, which eschews the assumption of directionality in a domain. By using MRFs we aim to more accurately model and generate levels for domains in which the levels do not have strong sequential dependencies.

The remainder of this chapter is organized as follows: first, in Section 5.2 we introduce and define Markov random fields; next, we describe our MRF training and sampling procedures in Section 5.3; we then describe our experiments and discuss the results in Section 5.4; lastly, in Section 5.5 we draw our conclusions.

5.2 Markov Random Fields

Recall that in Markov chains the probability of a state taking a particular value is completely dependent upon the previous state. In a Markov random field (MRF), however, there are no “previous states,” as MRFs...
model probabilistic relations between states in an undirected graph, not directed graphs or sequences as Markov chains do. Instead the value of a state is dependent upon the states composing the Markov blanket of the current state. The Markov blanket of a state is the set of that state’s parent, child, and co-parent nodes. In an undirected graph, the Markov blanket is exactly the neighbors of the current state, which we refer to as the surrounding states. Figure 5.1 shows an example Markov blanket for a node with four neighbors. Because Markov random fields do not model directed graphs, it is difficult for them to capture sequential dependencies (which are common in Super Mario Bros.). Additionally, their lack of directionality precludes them from being paired with the sampling approach described in Chapter 3 which samples a new level tile-by-tile in a given direction or order. Therefore, the challenges introduced by using an MRF approach are: devising a sampling approach that does not assume directionality, determining if the MRF approach can accurately model domains without strong sequential dependencies, and determining if the MRF approach struggles in domains that do have strong sequential dependencies.

In this chapter we assume that each state in the level is connected to the four surrounding states (i.e., the state above, the state below, the state to the right, and the state to the left), however, different network structures (both more complex and simpler) could be used with an MRF. Previously, Markov random fields have been used to reason about and synthesize textures, which can be likened to simplified two-dimensional video game levels. In the following section I describe how we train our MRF model and the sampling procedure we use for generating new levels.

Figure 5.1: A Markov random field where each state depends on four surrounding states. The colored portion highlights the dependence of a node (red) on its surrounding nodes (i.e., Markov blanket, blue).
5.3 Methods

In this section we discuss how we train an MRF on a set of training levels and how we sample new levels.

5.3.1 Training

At a high level, the training procedure for the MRF approach is very similar to that of the multi-dimensional Markov chain (MdMC) approach. That is, both approaches construct a conditional probability distribution by visiting each position in the training levels and counting how many times each tile type occurs with a given configuration of surrounding (or previous) states. The only difference comes from the definition of their neighbors or dependencies. Notice, that the lack of directionality in MRFs does not have a large impact on the training procedure because in the training levels all of the positions are already present and labeled with tile types, and therefore there is no possibility of reaching a position which does not have the required neighbors labeled. Below we describe our training algorithm in more detail.

We use a histogram-based non-parametric probability estimation approach to train our Markov random fields (Korc et. al.\textsuperscript{105} employ a similar training approach when modeling medical data). Our MRF training algorithm returns a CPD, $P_{ns}$, and has three parameters: $ns$ is the network structure, $M$ is a set of training levels, and $T$ is a set of tile types. Algorithm 6 trains our model in two stages: Absolute Counts and Probability Estimation. The first stage counts the number of times each tile type occurs with each surrounding configuration given the network structure, $ns$. For each position in each level (lines 1-2), the algorithm counts how many times each tile type, $t$, is surrounded by each tile configuration, $c$ (lines 3-5). Next, for each combination of tile configuration, $c$, and tile type, $t$, (lines 8-9) the algorithm sets the probability of $t$ when surrounded by $c$ according to the absolute counts (line 10). Finally, the conditional probability distribution is returned, $P_{ns}$ (line 13).

5.3.2 Sampling

Sampling a new level using a Markov random field requires a vastly different approach than our previous sampling algorithms, because MRFs are inherently not sequential. Our sampler employs the Metropolis-Hasting
Algorithm 6 train-MRF($ns, M, T$)

1: for each $m \in M$ do
2:   for pos $\in$ positions($m$) do
3:     $t = m[pos]$
4:     $c = \text{config}(m, pos, ns)$
5:     $S[ns][c][t] = S[ns][c][t] + 1$
6: for each $c \in S[ns]_{size}$ do
7:   for each $t \in T$ do
8:     $P_{ns}(t|c) = S[ns][c][t] / \sum_{t \in T} S[ns][c][t]$
9: return $P_{ns}$

Algorithm 7 sample-MRF-Metropolis-Hastings($ns, P, T, b$)

1: $m = \text{Empty map}$
2: for pos $\in$ positions($m$) do
3:   $m[pos]$ sampled according to $P_0(T)$
4: for $j \leq b$ do
5:   while $\exists 2$ unselected $pos \in$ positions($m$) do
6:     randomly choose $pos_1, pos_2 \in$ positions($m$)
7:     $L_{pre} = \log{\text{likelihood}(m|P)}$
8:     swap $m[pos_1]$ and $m[pos_2]$
9:     $L_{post} = \log{\text{likelihood}(m|P)}$
10: $m$ accepted with probability $\min(1, e^{L_{post} - L_{pre}})$
11: if $m$ not accepted then
12:     swap $m[pos_1]$ and $m[pos_2]$
13: $j = j + 1$
14: return $m$

algorithm\textsuperscript{94}. For completeness, Algorithm 7 shows the algorithm applied to level generation. The algorithm starts by initializing a level with the tile type at each position chosen using the single tile distribution observed in the training levels ($P_0$), corresponding to a network structure with no dependencies. Modifications are made to the level by choosing two positions in the level randomly, and swapping the tile types located there. The tile swap is accepted with probability equal to $\min(1, e^{\log(P_{post}) - \log(P_{pre})})$. The swapping procedure is repeated until each location in the level has been chosen $b$ times, after which the level is returned. Unlike the previous models, our MRFs do not sample sequentially, and, thus, do not employ a look-ahead nor fallback. However, we weight the likelihood of unseen states negatively in order to make it difficult for the model to swap a tile into an unseen state.
5.4 Experiments

In this section we will discuss how we conducted our experiments to evaluate our Markov Random Field (MRF) approach. We then discuss the results of those experiments.

5.4.1 Experimental Set-up

In our Markov Random Field approach the only parameter we need to set is $b$, the number of iterations to perform. We set $b$ by inspecting a preliminary graph of the log-likelihood of a sampled map over the number of iterations. We calculate the the log-likelihood of a map by using the trained MRF model to compute the log-likelihood of each tile in the map, and summing those values. We assign the probability of 0.0001 to unseen states in order to make them likely to be swapped out of. The preliminary log-likelihood graph for Kid Icarus can be seen in Figure 5.2. Notice that there is a sharp increase in the log-likelihood in the beginning, until around 400,000 iterations, and then the log-likelihood stabilizes. We choose the value of $b$ for each domain according to when this stabilization occurs in the preliminary graphs inspected for each domain. We set $b = 400,000$ for Super Mario Bros. and Kid Icarus, and $b = 50,000$ for Lode Runner.
Table 5.1: Markov Random Field Playability: This table shows the results of generating 1000 levels in *Super Mario Bros.*, *Lode Runner*, and *Kid Icarus* using the Markov random field approach. This table reports the percentage of playable levels generated in each domain with each model configuration as well as the average log-likelihood of those levels, and the average number of unseen states in each level.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Playability</th>
<th>Log-Likelihood</th>
<th>Unseen States</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Super Mario Bros.</em></td>
<td>21.9%</td>
<td>−367.53</td>
<td>20.37</td>
</tr>
<tr>
<td><em>Lode Runner</em></td>
<td>0.0%</td>
<td>−288.61</td>
<td>1.55</td>
</tr>
<tr>
<td><em>Kid Icarus</em></td>
<td>3.2%</td>
<td>−602.37</td>
<td>9.74</td>
</tr>
</tbody>
</table>

Figure 5.3: This figure shows heat maps of the *Super Mario Bros.* levels used for training (left) and those generated with the MRF approach (right). The $x$-axis is leniency and the $y$-axis is linearity. The top right corner of the heat map represents highly-nonlinear and difficult levels (high leniency values), and the lower left corner represents the most linear and easy (low leniency) levels. Note, all linearity and leniency values are normalized over all levels generated by all models to allow for uniformity in comparisons.

5.4.2 Results

Table 5.1 shows the results of our MRF experiments with the chosen $b$ values. We report the average log-likelihood and average number of unseen states as well as the playability. The log-likelihood and unseen states we compute using the MdMC model with $ns_3$, for uniformity with the other evaluations. It is important to note that though we compute the likelihood and unseen states with a different model, the computed values still correlate with those of the MRF model, and thus are only listed for completeness, as we use the MRF
Figure 5.5: This figure shows a Lode Runner level sampled using the MRF approach with $b = 50,000$.

log-likelihood as an indication of when to stop the sampling process.

First, notice that this model produces many fewer playable levels for Super Mario Bros. (21.9%) than the previous approaches (maximum of 50.8%). We found that the sampled maps tended to have many fewer solid tiles in the bottom row (ground) of the level. This resulted in levels that required constant jumping from platform to platform, where oftentimes the platforms were too far to reach. We believe that even though the likelihood of these maps stabilized, it is due to the amount of empty space that is present in the training levels. After the level is initialized, there will be solid tiles scattered throughout the level, but after 400,000 iterations, there are groups of these solid tiles, which increase the likelihood, but do not produce playable levels. The expressive range for this approach is very interesting (seen in Figure 5.3) as it does not mirror the training data at all. This model produces levels that are highly non-linear and have a very high leniency. This can obviously be seen in Figure 5.4, which shows a Super Mario Bros. level that is mostly platforms, with very little solid ground for the player to stand on. Notice that even though solid tiles are likely to occur near the bottom of the level, in order to place a solid tile there a solid tile elsewhere and a position near the bottom must both be selected during swapping. Additionally, because the MRF only captures local dependencies, a solid tile not near the bottom, but surrounded by other solid tiles, may be more likely than a solid tile alone near the bottom, preventing the swap from occurring.

For Kid Icarus, this model sampled the largest number of playable maps (albeit only 3.2%, as compared to 0.6% for the previous approaches). Figure 5.6 shows an example Kid Icarus level sampled with this approach. Similar to the sampled Super Mario Bros. level, the sampled Kid Icarus level contained more many platforms than levels sampled with the other models, giving the levels more of an opportunity for reachable platforms. However, this model still has difficulty capturing the proper distances between platforms, which often leads
Figure 5.6: This figure shows a section of a *Kid Icarus* level sampled using the MRF approach with $b = 400,000$.

...to platforms that are unreachable, rendering a level unplayable.

Lastly, this model was unable to produce any playable *Lode Runner* levels. Much like it could not capture the proper jumping distance due to its limited scope, it was also unable to capture proper paths. This is because the paths through the levels are inherently global to the level, while the MRF only focuses on local coherency. Figure 5.5 shows a *Lode Runner* level sampled with this model. Notice that each of the small sections of the level are cohesive, and there are paths between several of the treasures, but that there is not a path that connects all of the treasures.

5.5 Conclusions

In this chapter we introduced a Markov random field (MRF) level generation approach. MRFs model dependencies between a state and its neighbors in all directions. With this approach our goal was to more
accurately model domains which do not have strong sequential dependencies between the states. We tested this approach in three domains, *Super Mario Bros.*, *Lode Runner*, and *Kid Icarus*.

We found that our MRF level generation model is able to sample more playable levels for *Kid Icarus* than the previous models (0.6% playable with the HMdMC and MdMC models compared to 3.2% playable with the MRF model). However, it performs significantly worse in the other domains, producing many fewer playable levels for *Super Mario Bros.* and no playable levels for *Lode Runner*. This model encounters a problem similar to the MdMC model, which is that it cannot capture long-range dependencies, resulting in structures that render a level unplayable (e.g., gaps that are too long, platforms that are unreachable, blocked paths). Furthermore, the sampling approach was vastly different from the other models, and did not easily allow for a look-ahead, which resulted in many malformed structures and strange looking levels in all domains.

Having presented our first key contribution (i.e., showing that machine learning-based level generation approaches can be used across domains with minimal domain knowledge from the user via Markov models), we present our second key contribution in the following chapter, a unifying theoretical framework for machine learning-based level generation approaches. We then present our other contributions categorized into either extensions to machine learning-based level generation approaches or experiments in supplementing and choosing training data for machine learning-based level generation approaches.
Chapter 6: Theoretical Framework

Recently, several machine learning approaches in addition to the ones we introduced have begun to be explored in the context of procedural content creation\textsuperscript{5,8,68}. Due to its recent emergence, the category of machine learning-based PCG approaches is not as well understood as more established categories, such as search-based approaches. Therefore, it is necessary to begin investigating the commonalities and differences between the various recent techniques in order to gain an understanding of their strengths and weaknesses, their limitations, and potential interesting lines of future work. The theoretical framework presented in this chapter is our attempt at a unified theory of machine learning-based level generation techniques, which tries to capture as much of the current work as possible.

With this framework we aim to provide a deeper understanding of this new class of PCG approaches, highlight similarities and differences between approaches, and provide insight into future avenues of research based on observed gaps in the research. Our framework is meant to represent these techniques in a general, uniform way to allow for deeper analysis of the techniques, and allow for more direct comparisons between techniques. We define our framework by investigating multiple machine learning-based level generation approaches and finding core similarities between them and using those core similarities as the basis of the framework. Specifically, we present a framework that represents machine learning-based approaches in terms of their level representation (i.e., how they represent their training and output data), how the model is trained (i.e., how the model is represented, what the model captures, and how the model is estimated from the training data), and how new levels are sampled (i.e., how the trained model can be used to generate new levels in the desired representation).

In the remainder of this chapter, we first describe the general form of our theoretical framework in Section 6.1, in the subsequent sections we describe individual approaches in the terms of the framework, starting with our own multi-dimensional Markov chain approach in Section 6.2 before moving onto other approaches. We then perform a comparison of the methods described by our framework in Section 6.7. We close with a discussion of the framework in Section 6.8 in terms of two questions that arise.
6.1 General Theoretical Framework

A machine learning-based PCG approach can be completely specified by defining three main components:

- **Data Representation**: Machine learning approaches rely on a set of training data from which a model is estimated. The data representation is the way the training data and output content is represented.

- **Training Algorithm**: Given a set of training data, the training algorithm is the method for estimating a model from the training data. These methods can range in complexity from simple probability estimations to neural networks.

- **Sampling Algorithm**: Once the model is trained, the sampling algorithm is the method for generating new content.

In the following sections we discuss each of the components of the framework in more detail, starting with the data representation.

6.1.1 Data Representation

In this section we describe the representation defined by the framework for the training data as well as the generated content. For our Markov model-based level generation approaches described in Sections 3 - 5 we represent a level as a matrix of tile types corresponding to elements in the level. However, other approaches have represented levels differently, such as Guzdial and Riedl’s\(^8\) approach which represents levels as graphs of level structures and Dahlskog et. al’s\(^5\) approach which represents levels using sequences of columns from the levels. At the core of each of these level representations we have objects corresponding to level elements, and a way of connecting or associating these objects with each other. Notice, this information can be captured by a labeled graph, where the level elements correspond to the vertices’ labels and the edges (defined by a neighbor function specific to the model or domain) dictate the associations and relationships between those elements. Furthermore, representing levels as graphs does not lose any generality from the approaches’ original representations, and does not complicate the algorithms’ details.

More formally, we represent a level in this graph format as a triple, \(\langle T, L, l \rangle \):
Figure 6.1: A section of a Super Mario Bros. level (left), our original tile representation (center), and a graph representation (right).

- $T = \{t_1, t_2, ..., t_k\}$ is the set of tile types representing the building blocks of the domain. For example, Figure 6.1 shows a section of a Super Mario Bros. level (left) and that section decomposed using its tile types (center). We can see in this example the tile types correspond to objects in the level as well as enemies, however, the tile types can be made to represent less structural elements as well, such as possible paths through the level or other meta-information like current height in the level.

- $L = \langle V, E \rangle$, where $V$ is the set of vertices, $E = \{e_i : (v_a, v_b)\}$ is the set of directed edges, and each $v_i \in V$ is labeled with a tile type in $T$. Furthermore, each edge represents the dependence of one vertex on another, encoding the network structure of a model. Specifically, an edge from one vertex to another, $v_a$ to $v_b$, denotes that the tile type of $v_b$ depends on the tile type of $v_a$. A vertex may have many incoming and outgoing edges. Notice, we reduce all models to directed graphical models because there are often instances and models where dependence only flows in one direction (e.g., $n$-grams, MdMCs, etc.), which are easily captured with directed edges, but more difficult to capture with undirected edges. Alternatively, undirected models (e.g. Bayes Nets) can easily be represented with directed edges by simply having edges connecting in both directions. $L$ is the core of the level representation as it captures the tile representation of the level as well as the network structure.

- $l : V \rightarrow T$ is a labeling function that maps every $v_i \in V$ to a $t_j \in T$.

Additionally, given the graph $L$, we define $\text{neighbors}(v) = \{v' \in V | (v', v) \in E\}$ (i.e., the set of vertices with outgoing edges connected to $v_i$).
6.1.2 Model Training

In this section we describe how machine learning-based level generation approaches model a domain, and how that model is estimated from the training data. From the recent approaches, our Markov model approaches\textsuperscript{106} and Dahlskog et. al’s\textsuperscript{5} approach model a domain by capturing the probability of a tile type (or column) at a position given the surrounding tile types. Summerville and Matteas’\textsuperscript{7} approach models a domain by training an LSTM on the level represented as a sequence of tile types, and applying a soft-max layer to the output of the trained network. Guzdial and Riedl’s\textsuperscript{8} use a probabilistic graphical model to capture the probability of a level element given other elements in the level section and other information derived from the levels. At the core of each of these models is a conditional probability distribution which captures the relationship between the various level elements. Therefore, without loss of generality, the models learned by these machine learning-based level generation approaches can all be seen as a conditional probability distribution.

Specifically, we define a model as

\[ P(v_i = t_j | C). \]

That is, a model is defined as the probability of a vertex, \( v_i \), taking a tile type, \( t_j \), given some set of conditions, \( C \). The definition of \( C \) depends on the training algorithm chosen. For example, if our basic multi-dimensional Markov chain approach is used then \( C \) is defined as the neighbors of the given vertex. Note, some models may require multiple distributions, for example, when our MdMC approach employs a fallback procedure (as described in Section 3.2), in these cases \( P \) is the union of those distributions, and which distribution to use is another element of \( C \). Lastly, it is important to point out that our framework assumes a discriminative model is being used to model the domain, that is, our framework assumes a conditional probability distribution. However, note that generative models can easily be transformed into discriminative models\textsuperscript{107}, so this is not a limitation.
6.1.3 Sampling

In this section we define the sampling method of the theoretical framework. Recall, our multi-dimensional Markov chain approaches sample a new level one tile at a time, probabilistically choosing the next tile in a given ordering using the trained probability distribution and the previously sampled tiles. Similarly, Dahlskog et. al’s\(^5\) and Summerville and Matteas’\(^7\) approaches probabilistically choose the next tile in a sequence given the previously sampled tiles and the trained distributions. Notice, however, that Guzdial and Riedl’s\(^8\) sampling approach does not rely on a predefined ordering to define where in the level to sample next, and instead probabilistically chooses the position as well as the object.

Notice, we can view all of the above sampling procedures as starting with an initial (possibly empty) level graph, and then alternating between adding a vertex and assigning a label (a tile type) to the newly added vertex. This process is continued until a predefined stopping criterion is met. Below we define the initial level graph and other input parameters to the general sampling procedure used by most machine learning-based level generation algorithms, as well as the stopping criteria, and the vertex and tile sampling functions.

- \(L_{\text{initial}}\): This is the initial level graph which acts as the starting point to which additional vertices and labels will be added. This initial graph is composed of a (potentially empty) set of edges and labeled vertices. Notice, there can be many valid definitions of \(L_{\text{initial}}\) for a given domain and model, and the valid initial level graphs may vary based on a given domain, model, or model set-up.

- \(P\): This is the probability distribution which defines the model. \(P\) is used to determine the label of the sampled vertices of the graphs.

- \(T\): This is the set of tile types that define the domain and are used to represent levels within that domain.

- \(\text{StoppingCriteria}(L)\): This is a boolean function that is used to determine if the sampling process should stop (i.e., the sampled level is complete). For example, this function could be used to check if the dimensions of the graph are of the desired size, or if the desired number of each tile type has been sampled in the output level, etc.

- \(\text{SampleVertex}(L)\): This function determines where the next vertex should be placed in order to grow
Algorithm 8 SequentialLevelSampling($L$, $P$, $T$, StoppingCriteria, SampleVertex, SampleTile)

1: 
   while ¬ StoppingCriteria($L$) do
2:     SampleVertex($L$)
3:     SampleTile($P$, $T$, $L$)

the level graph. This can be determined, for example, by a simple ordering desired for sampling, as in
the MdMC approaches, or probabilistically as in Guzdial and Riedl’s approach.\textsuperscript{\ref{footnote:sequential}}.

- \textit{SampleTile($P$, $T$, $L$)}: This function probabilistically determines the tile type of any unlabeled vertices
  in $L$ using the trained probability distribution, $P$. Essentially, this function incrementally constructs $l$,
  the labeling function for the level being sampled, $L$.

Once these input parameters and functions are defined, a level is sampled using Algorithm 8. It starts with
the provided starting level graph $L_{\text{initial}}$, and then until the stopping criteria are met (line 1) it incrementally
grows the initial graph by sampling a new vertex using the \textit{SampleVertex} function (line 2), and then choosing
the label from the set of tile types, $T$, for that vertex based on $P$ (line 3).

It is important to note that while most approaches sample levels sequentially (as we describe here), there
are approaches in the literature that do not use a sequential sampling approach. For example, our Markov
random field approach discussed in Section 5 instead employs the Metropolis-Hastings algorithm\textsuperscript{\ref{footnote:concurrent}} which is
a particular case of what we will refer to as “concurrent sampling.” In concurrent sampling approaches, the
entire (unlabeled) level graph is known prior to assigning the tile types to the vertices. However, a concurrent
sampling approach can be defined in terms of the sequential sampling approach by carefully defining the
helper functions and parameters. We discuss the effects of the different sampling approaches in Section
6.7.3.

In this section we claimed that a machine learning-based level generation approach can be specified by
defining three components: a data representation, a training algorithm and model, and a sampling algorithm.
We then described each of those components and their requirements in more detail. To further support this
claim, in the following sections we present existing machine learning-based level generation approaches in
these terms, starting with our multi-dimensional Markov chain approach.
6.2 Multi-dimensional Markov Chains

In this section we discuss how to redefine our multi-dimensional Markov chain (MdMC) level generation approach using the proposed framework above. Recall that our framework defines a machine learning-based level generation approach as requiring three components to be completely defined: a data representation, a training algorithm, and a sampling algorithm. Therefore, for our MdMC approach and for all the approaches described in future sections, we define each of these three components.

6.2.1 Data Representation

In order to translate the level representation used by our MdMC approach to the proposed framework’s representation, we need to define \( \langle T, L, l \rangle \).

- \( T \): This approach uses a set of tile types corresponding to game objects from the domain. For example, in *Super Mario Bros.* the tile types represent the various enemies, solid ground, pipe pieces, etc. Figure 6.1 (center) shows a section of *Super Mario Bros.* level represented using the chosen tile types. Appendix A.1 has a more detailed description of the tile types used to represent our domains.

- \( L = \langle V, E \rangle \): In Section 3 we describe our MdMC approach as representing levels as a matrix of tile types. To describe these levels in our framework, we use a set of vertices, \( V \), to represent each of the cells in the matrix. Thus, in this model, each \( v_i \in V \) represents a discretized position in the game level. Furthermore, recall that the chosen network structure of the MdMC defines which cells are “previous states,” and therefore, should be connected. \( E \) is defined by this network structure. That is, \( L \) implicitly encodes the network structure into the graph structure of \( L \) by creating an edge from vertex \( v_a \) to vertex \( v_b \) if \( v_a \) is a “previous state” of \( v_b \). Figure 6.2 shows several example network structures. Figure 6.1 (right) shows the graph representation of a level section using \( ns_2 \) to define the edges.

- \( l \): For this model, \( l \) assigns the tile type of each vertex according to the object in that position in the input level image.
6.2.2 Model Definition

To represent the MdMC method in terms of our new framework, we need to define the probability distribution, \( P \), of the model. For our standard MdMC method, \( P \) is defined as a conditional probability distribution (CPD), where the tile type of the current position is dependent upon the tile types of surrounding previous positions. The conditional, \( C \), for the probability distribution is the set of neighbors. Therefore, the CPD can be defined as

\[
P(v_i = t_j | \text{neighbors}(v_i)).
\]

Recall that \( \text{neighbors}(v_i) \) is defined as all vertices connected to \( v_i \) by an incoming edge. Additionally, for this model, the distribution is estimated from the observed occurrences from the training levels, as described in Chapter 3.

In addition to our standard MdMC, we also have previously defined an MdMC that uses a fallback or back-off model when sampling. These fallback models are equivalent to defining several models with decreasingly complex network structures (i.e., a less connected graph, \( L \)), and combining their probability distributions into a piecewise distribution. Specifically, we need to redefine the level representations used by the various fallback models

- \( L \): Several \( L_i \)'s must be defined with decreasing complexity. Each of these graphs still uses the base definition described for the MdMC level representation. Each \( L_i \) is defined as the graph where each vertex has \( i \) incoming edges.

Now, we can define the probability distribution for the MdMC employing a fallback procedure. \( P \) is
defined as:

\[
P = \begin{cases} 
    P_n(v_i = t_j|\text{neighbors}_n(v_i)) & \text{if count(\text{neighbors}_n(v_i))} \geq k \\
    P_{n-1}(v_i = t_j|\text{neighbors}_{n-1}(v_i)) & \text{otherwise, if count(\text{neighbors}_{n-1}(v_i))} \geq k \\
    \vdots \\
    P_0(v_i = t_j|\text{neighbors}_0(v_i)) & \text{otherwise}
\end{cases}
\]

For the above probability distribution, \(\text{count(\text{neighbors}(v_i))}\) is defined as the number of times the given configuration of neighboring vertices of the current vertex \(v_i\) with their specific values has been observed in the training data. Therefore, what the above piecewise conditional distribution says, is that if the current configuration of neighbors and values has not been observed enough times in the training data \((< k)\), use the conditional distribution of a model with a less connected graph with neighbor configurations that have been observed enough times.

Similarly, we can define a CPD that allows for interpolating over several models’ distributions instead of using a back-off approach. Below we define \(P\) for the MdMC approach using a simple linear interpolation over the above distributions, instead of a back-off model.

\[
P(v_i = t_j) = \begin{cases} 
    \lambda_n \cdot P_n(v_i = t_j|\text{neighbors}_n(v_i)) + \\
    \lambda_{n-1} \cdot P_{n-1}(v_i = t_j|\text{neighbors}_{n-1}(v_i)) + \\
    \vdots \\
    \lambda_0 \cdot P_0(v_i = t_j|\text{neighbors}_0(v_i))
\end{cases}
\]

Notice that these extensions are not exclusive to the MdMC method. The back-off and interpolation extensions can be used with any models able to be defined in our framework.

### 6.2.3 Sampling

We now need to define how the MdMC approach generates new content. We do this by defining the \(\text{SampleLevel}\) function along with its parameters and helper functions. We define the parameters as follows:

- \(L_{\text{initial}}\): We define an initial level graph with 2 vertices labeled with the special sentinel tile type
used to denote the boundaries of a level. Specifically, because of the inherent ordering in the MdMC approach, we only define a vertex for two of the borders of the level. We only use these sentinel vertices where they will have outgoing edges connecting to other level vertices, and not where they would have incoming edges from other level vertices. For example, previously our graphical level representations all had edges pointing up or to the right (or diagonally up and right); in this case, we only place a vertex to the left and one below the rest of the level vertices, and not to the right or above the level vertices. Figure 6.3 (1) shows this initial level graph. Notice, that many other initial level graphs can be used for this approach (and it may vary by domain), but we chose this level graph for simplicity and ease of visualization.

- **StoppingCriteria**$(L)$: The stopping criterion for this approach is whether the level graph is of the desired size. Specifically, $StoppingCriteria(L)$ compares the number of vertices in $L$ to the desired dimensions of the generated level and returns true (stop) if the number of vertices labeled with non-sentinel tile types in $L$ is equal to the desired height$\times$width.

- **SampleVertex**$(L)$: This function determines where the next vertex should be placed in order to grow the level graph. For this approach the next vertex to be sampled must be placed in such a way that it has the necessary neighbors as defined by the network structure. For this approach, we typically sample row by row from the bottom up. However, vertices can be added in any order provided they satisfy the network structure. Figure 6.3 (2-5) illustrates one way that vertices could be added to the initial level graph during sampling.

- **SampleTile**$(P, T, L)$: This function probabilistically determines the tile type of any unlabeled vertices in $L$ using the trained probability distribution, $P$. For this approach, the tile type is chosen probabilistically based on the neighbors of the vertex being labeled.

Once a new vertex, $v$, is added to the level graph, the tile type of $v$ is chosen according to the trained probability distribution, $P$ and the tile types of the neighbors of $v$. Recall, that this is incrementally defining the labeling function, $l$, of the level by determining which tile type is associated with each vertex. Figure 6.3 shows the sampling process from the initial level graph (1) to the completed (albeit small) level (5).
Figure 6.3: This figure shows an example of an initial level graph (1) used to start sampling with the MdMC approach, and subsequent vertices and labels sampled (2-5) for a level with $2 \times 2$ tiles corresponding to positions in the output level.

6.3 $n$-grams

In this section we discuss Dahlskog et al.’s $n$-gram level generation approach. This model treats the training data as one dimensional by using each distinct column in the training levels as a tile type. This has the benefit of simplifying the domain, but limits the expressiveness of the model. An $n$-gram model is then trained on the series of columns from the training levels to build the probability distribution, and new levels are generated by sampling a new series of columns from that distribution.

6.3.1 Data Representation

To represent Dahlskog et al.’s $n$-gram model we need to define its data representation, namely, $(T, L, l)$.

- $T$: The set of tiles for this approach is the set of unique columns from the training map images. Note, this could be modified depending on the domain. For example, in Super Mario Bros. using columns is intuitive, however, in Kid Icarus where the player traverses the level vertically, using rows as the tile types may make more sense. Figure 6.4 (center) shows columns taken from a section of a Super Mario Bros. level (left) and labeled accordingly. Notice, duplicate columns are marked with the same character representation.

- $L = (V, E)$: Each $v_i \in V$ represents a column in the level. $E$ is derived from the value of $n$. That is, the value of $n$ determines how many previous vertices the tile type of the current vertex depends on, and thus how many incoming edges it has. For example, when $n = 3$ the tile type of $v_i$ depends on the tile types of $v_{i-1}$ and $v_{i-2}$. 
Figure 6.4: A section of a Super Mario Bros. level (left), the column representation used by the n-gram model, and a graph representation, where each vertex represents a column from the training map for a bigram model (top-right) and a trigram model (bottom-right).

- \( l \): For this model, \( l \) assigns the tile type of each vertex according to which column from the training data it is associated with.

### 6.3.2 Model Definition

We now need to define the probability distribution, \( P \), for the model. In the original definition of their model, Dahlskog et al. define the probability distribution as the value of the current state depending on the values of the previous \( n-1 \) states, where each state is a column in the map. Notice, we can encode this \( n \) into the map representation, and therefore we can rewrite the distribution as

\[
P(v_i = t_j|\text{neighbors}(v_i)).
\]

These probabilities are approximated from the observed instances that occur in the training levels, similar to how the MdMC’s probability distribution is estimated. The interpolation and back-off models can also be applied here to encode backing off to an \((n-1)\)-gram, and so on.

### 6.3.3 Sampling

We now define the parameters of the SequentialLevelSampling function. \( P \) and \( T \) are defined the same as for the model definition.

- \( L_{\text{initial}} \): The \( n \)-gram approach does not require any vertices in its initial level graph in order to begin sampling. Therefore, for this model \( L_{\text{initial}} \) is the empty graph consisting of no vertices or edges.
• **StoppingCriteria**(\(L\))**: The stopping criterion for this approach is whether the level graph is of the desired size. Specifically, \(\text{StoppingCriteria}(L)\) compares the number of vertices in \(L\) to the desired dimensions of the generated level and returns true (stop) if the number of labeled vertices in \(L\) is equal to the desired size.

• **SampleVertex**(\(L\))**: This function determines where the next vertex should be placed in order to grow the level graph. For this approach the next vertex sampled must be placed in such a way that it has the necessary neighbors as defined by \(n\). Specifically, when a new vertex is sampled incoming edges are added from the \(n - 1\) previous vertices. The first \(n - 1\) vertices are special cases where the added vertex gets an incoming edge from each previous vertex.

• **SampleTile**(\(P,T,L\))**: This function probabilistically determines the tile type of any unlabeled vertices in \(L\) using the trained probability distribution, \(P\). For this approach, the tile type is chosen probabilistically based on the neighbors of the vertex being labeled. Notice, that for the first \(n - 1\) vertices in \(L\), a simplified \((n - m)\)-gram probability distribution must be used as there are not enough neighbors.

### 6.4 LSTM RNNs

In this section we discuss Summerville and Mateas’s 2016 long short-term memory recurrent neural network (LSTM RNN) method. This model treats the input levels as strings, essentially compressing multi-dimensional input data into one-dimensional data. The model then learns the probability of one tile type (or character in the string) following another. The interesting thing about this model is that it is able to remember many characters in the past while it is training its distribution and while it is sampling new levels. Additionally, this model was the first to leverage training levels annotated with a tile type indicating a possible player path, as given by an \(A^*\) agent. This allows the model to learn both structural information about the level as well as how those structures affect the player’s movements. This approach of annotation has been incorporated into our later MdMC approaches as well.

#### 6.4.1 Data Representation

We now need to define the data representation of the LSTM approach using \(\langle T, L, l \rangle\).
Figure 6.5: A section of a Super Mario Bros. level (left), and our graph representation using the “snaking” encoding of the map proposed by Summerville and Mateas, where each vertex represents a 16 × 16 pixel section from the training map corresponding to a tile type (right).

- **T**: This approach uses a set of tile types corresponding to game objects in the current domain and the A* agent’s path through the level. For example, in Super Mario Bros, Summerville and Mateas use tiles corresponding to ?-blocks, enemies, solid blocks, agent positions determined by an A* agent, coins, etc.

- **L = (V, E)**: For this approach, each $v_i \in V$ corresponds to a discretized position in the game level. $E$ is defined according to the provided ordering of the vertices. In their work, Summerville and Mateas explore several different orderings. One such ordering is the “snaking” ordering which traverses the level from the top of one column to the bottom before moving to the next column and going from the bottom of that column to the top, etc. However, because the LSTM has a memory parameter $n$, each vertex $v_i$ is connected to the previous $n - 1$ vertices. Figure 6.5 shows a section of a Super Mario Bros. level (left) and the “snaking” ordering in a graph representation (right). Note, that we only show a single edge between the previous and current vertex for clarity, but in fact there are edges coming from each vertex to the next $n$ vertices, in Summerville and Mateas’s case $n = 200$.

- **l**: For this model, $l$ assigns the tile type of each vertex according to the object in that position in the input level image or a special tile if the A* agent passed through that position during its traversal of the level.
6.4.2 Model Definition

We now define the probability distribution, $P$, of the LSTM approach. In this case, $P$ is defined by the trained LSTM RNN. Note, they apply a SoftMax layer to the final weights of the network in order to transform the output weights into a probability distribution. This allows levels to be sampled tile-by-tile by passing the previously sampled sequence of tiles through the network.

6.4.3 Sampling

To represent the LSTM’s sampling procedure, we need to define the parameters of the SequentialLevelSampling function. $P$ and $T$ are defined the same as for the model definition.

- $L_{\text{initial}}$: This approach requires a predefined section of the level to be available to build on during sampling. Specifically, this approach starts with an $h \times w$ seed as the initial level graph, where $h$ is the desired height of the level and $w$ is the desired number of starting columns. In their experiments, Summerville and Matteas use $w = 3$ for their initial level graphs. These seeds correspond to typical beginning sections of levels in the domain.

- $\text{StoppingCriteria}(L)$: The stopping criterion for this approach returns true (stop) if a vertex is labeled with a special end tile that signifies the end of the level. The placement of this tile is learned as part of $P$.

- $\text{SampleVertex}(L)$: This function determines where the next vertex should be placed in order to grow the level graph. For this approach the next vertex sampled must be placed in such a way that it has the necessary neighbors as defined by the network of the LSTM. In Summerville and Matteas’ experiments, they use a neighborhood of 200. Notice, the first 200 vertices sampled are special cases where all the previous vertices are neighbors.

- $\text{SampleTile}(P, T, L)$: This function probabilistically determines the tile type of any unlabeled vertices in $L$ using the trained probability distribution, $P$, by feeding the neighbors through the LSTM in order and then probabilistically choosing the next tile in the sequence.
Figure 6.6: This figure shows the set of sprites used in Super Mario Bros. from which the set of tile types, $T$, can be derived. Notice, that this set of sprites is only for illustrative purposes, and is incomplete as it does not include enemy or pipe sprites.

6.5 Level Generation from Gameplay Videos

In this section we discuss Guzdial and Riedl’s approach to level generation using clustering and a Bayesian model. In their approach they convert gameplay videos to sections of levels. They then use a set of predefined atomic sprite types to identify representative sprite shape styles in the level sections based on groupings of atomic sprites and their relative locations within the level section. These sprite shape styles are then used to re-represent and reason about the geometry of the level sections. Specifically, their model learns a probability distribution for the sprite shape styles’ occurrences and positions within level sections. To generate a new level section they place sprite shape styles in the level section while greedily maximizing the average probability of the level section based on the shapes’ relative positions.

6.5.1 Data Representation

We will now define the level representation of this model, defining $(T, L, l)$.

- $T$: This approach uses a set of tile types that represent structures within the levels. At a high level, the authors identify their set of tile types by performing clustering on sets of objects and their relative positions within the level sections. In more detail, the authors define their tile types in several stages.
Figure 6.7: This figure illustrates the level representation of Guzdial and Riedl’s approach. Specifically, it shows a section of a *Super Mario Bros.* level (first), that same section with the \( g \) nodes annotated (i.e., the sprite groupings) as well as the connections for one of those nodes \( g_4 \) (second), a graph representation using the \( g \) nodes as vertices and the connections (or \( D \)s) as edges (third), and a graph representation using the tile types found via clustering the \( (g,D) \) pairs to label the vertices and the \( D \) nodes for the edges again. Notice that both graph representations are fully connected. This is because each of the \( g \) nodes (and by extension the tile types, \( S_i \)) depend on each of the other nodes or objects in the level. Furthermore, for clarity, we do not display the edges going in both directions, and instead display a single edge.

First, they convert a gameplay video into a set of level sections based on frame similarity. Next, they define the set of objects, \( G \), in the level section. Intuitively, each element of \( G \) corresponds to an object (composed of one atomic sprite type), its position within the level section, and its position relative to other objects. Formally, each \( g_i \in G \) is defined by the tuple \( \langle M, (min_x, min_y), t \rangle \), where \( M \) is a binary matrix with 1s indicating positions of atomic sprites comprising the object, \( (min_x, min_y) \) indicates the minimum coordinates of the object within the level section, and \( t \) indicates the atomic sprite type of which the object is comprised. It is important to notice that each \( g_i \) is composed of all adjacent atomic sprites of the same type.

The authors then define a set of connections \( D_i \) for each \( g_i \in G \). Intuitively, each \( D_i \) can be thought of as containing the relationships between the given \( g_i \) and all other \( g_j \)'s in the level section. Formally, \( D_i \) is composed of a list of \( d_j \), and each \( d_j \in D_i \) is defined by a tuple

\[
\langle \text{connectionPoint}_{g_i}, \text{connectionPoint}_{g_k}, \text{dist}, v \rangle,
\]

where the *connectionPoints* indicate the points on the \( g_i \)s that are closest to each other in the level section, *dist* is the distance between the two connection points, and *v* is the normalized vector between
the two connection points.

Once $G$ and the corresponding $D_i$’s are defined, the authors use clustering on $(g_i, D_i)$ pairs. Specifically, they employ $k$–means clustering where $k$ is estimated using the distortion ratio. To compute the distances between each $(g_i, D_i)$ pair, they equally weight the results of matrix subtraction between the $g_i$’s $M$’s and the sum of differences between the corresponding $D_i$’s $v$’s. The resulting clusters of $(g_i, D_i)$ pairs are called $S$, where an $S_k \in S$ corresponds to the $k$th cluster. Thus, each $t_i \in T$ corresponds to an $S_k \in S$. These tile types are more abstract than those used in previous approaches, because each tile type may correspond to multiple instantiations of objects within the given domain. Notice, however, that these tile types can be likened to using a general “enemy” tile type in a domain where multiple enemy types exist. The level representation in both cases abstracts away some details which need to be added back in when the levels are reconstructed or generated.

- $L = \langle V, E \rangle$: For this approach, each $v_i \in V$ corresponds to an abstract structure representation given by the clusters defined above. Specifically, each $v_i$ corresponds to a set of similar structures found in the training levels, represented by a single abstract value. In this approach, each of the vertices depends on the values of the other vertices, therefore each level section is represented with a fully connected graph. That is, $E = \{(v_i, v_j) \forall v_i, v_j \in V$ such that $v_i \neq v_j\}$.

- $l$: In this approach, $l$ assigns the tile type of each vertex according to which cluster, $S_k$, the shape represented by that vertex belongs to.

### 6.5.2 Model Definition

We next need to define the probability distribution, $P$, for this approach. At a high level the probability distribution for this approach captures the probability of objects occurring together in the section, or the joint probability of the tile types. Specifically, the probability distribution is given by

$$P(v_i = t_k | t_j, d),$$
where $d$ is the connection between $t_k$ and $t_j$. $P$ says that the value of $t_k$ depends on the value of $t_j$ and the connection between the two, which includes: the connection points, the discretized distance between them, and the normalized directional vector. Notice, that this gives the probability of a tile type given one of the other vertices in graph. Since each vertex is dependent on each other vertex, the final probability distribution used when sampling is a combination of the probabilities of the given vertex and each other vertex. Each $P(v_i = t_k | t_j, d)$ is built up from the observed occurrences of the $g_i$’s in the training level sections. That is, the probabilities of the $t_k$’s co-occurring are estimated from $P(g_a | g_b, d_c)$, the probabilities of the $g$’s co-occurring in the training levels along with knowledge of which $S_c$ cluster (corresponding to $t_c$) each $g$ belongs to.

### 6.5.3 Sampling

We will now define the parameters and helper functions of the \textit{SequentialLevelSampling} function, in order to specify the sampling procedure of this approach. $P$ and $T$ are defined the same as in the model definition.

- $L_{\text{initial}}$: This approach does not require any vertices in order to begin sampling. Therefore, the initial level graph $L_{\text{initial}}$ is an empty graph with no vertices or edges to begin with.

- $\text{StoppingCriteria}(L)$: The stopping criterion for this approach returns true (stop) once the level contains the desired number of game objects that make up the $g$ nodes in the $S$ clusters.

- $\text{SampleVertex}(L)$: This function determines where the next vertex should be placed in order to grow the level graph. For this approach, the next vertex to be placed is based on the connections of the $S$ nodes to the other $S$ nodes, and how well the connections align.

- $\text{SampleTile}(P, T, L)$: This function probabilistically determines the tile type of any unlabeled vertices in $L$ using the trained probability distribution, $P$.

### 6.6 WaveFunctionCollapse

WaveFunctionCollapse (WFC) is a content generation approach developed by Maxim Gumin\textsuperscript{65,109} which has been used for level generation in several games\textsuperscript{89,110}. At a high level, this approach is given a set of patterns
(composed of atomic level elements) that can connect in different ways, and uses the knowledge of those allowed and disallowed patterns (extracted from input examples) to enforce local consistency. A new level is generated by defining a grid of patterns and a grid of tiles, and collapsing the states at each position in the grid of patterns by choosing the position in the grid that has the fewest allowed pattern options (according to the allowed patterns) and choosing one of the options based on the frequency of the patterns’ occurrences in the provided examples. This continues until the level is complete or until a contradiction is reached in the local consistency propagation, in which case sampling restarts. This is a simple approach that requires very little training data, but is able to produce remarkable results.

6.6.1 Data Representation

This approach needs to be able to represent both the patterns that encode the possible combinations of objects, and the objects themselves. Notice, the objects in this case correspond to the level elements (e.g., in Super Mario Bros. a brick, an enemy, a ?-block, etc.). Therefore, this approach models levels using two layers of representation: a hidden layer in which the nodes track which patterns are possible at a given set of positions, and an observed layer in which the nodes are labeled with the corresponding object types. We now define the level representation, $\langle T, L, l \rangle$, used by this approach.

- $T$: In order to represent a level domain in this approach we need two tile sets. The first is for the observed nodes, which correspond to the objects in the level domain. These are similar to the tile types used by the MdMC and LSTM approaches (i.e., they correspond to simple level elements from the game domain such as enemies, bricks, empty space, etc.). We call these tile types $T_{\text{observed}}$. The second set of tiles are used to label the hidden nodes. These tile types represent the sets of patterns possible at a given position in the level graph, and are given by the power set of possible pattern types. Notice, this is required because it is possible for each hidden node to be able to take the value of any of the pattern types. The set of possible patterns for each hidden node reduces as more observed nodes are labeled. Therefore, if the set of patterns is $T_{\text{patterns}}$, then the set of tile types for the hidden nodes is $T_{\text{hidden}} = 2^{T_{\text{patterns}}}$. Thus, the full set of tile types for this approach is the union of the two sets, or $T = 2^{T_{\text{patterns}}} \cup T_{\text{observed}}$. 

• $L = \langle V, E \rangle$: As mentioned previously, this approach represents levels at two layers of abstraction: a hidden layer for the patterns and an observed layer for the level elements. Therefore, this approach uses two sets of vertices for the level graph. The first set of vertices are the observed vertices, $V_{\text{observed}}$. These nodes correspond to the elements of the level domain and their discrete positions in the levels. These observed vertices are not connected to one another by edges, because their labels are only influenced by the accompanying hidden vertex label, not the surrounding observed vertices. The second set of vertices are the hidden vertices, $V_{\text{hidden}}$. These correspond to patterns in the levels and each hidden vertex represents a discrete section of the level. Notice, the hidden vertices represent larger sections of the level than the observed vertices, and in fact the hidden vertices correspond to patterns of observed vertices. The number of edges each hidden vertex has depends on the size of the pattern it represents, and the dimensions of that pattern. That is, each hidden vertex will have an edge from itself to each observed vertex that is a part of the pattern the given hidden vertex represents, and each hidden vertex will have an edge from itself to each adjacent hidden vertex. For example, if the hidden vertex represents a $2 \times 2$ rectangular pattern, then each hidden vertex will have 4 edges from itself to 4 observable vertices, and 4 edges from itself to the 4 adjacent hidden vertices (1 for each side of the pattern shape). Notice, $V$ contains both sets of vertices. That is, $V = V_{\text{observed}} \cup V_{\text{hidden}}$.

• $l$: For this approach, the labeling function $l$ assigns tile types to each $v \in V_{\text{observed}}$ according to the level element at that location in the level, and assigns tile types to each $v \in V_{\text{hidden}}$ according to the possible patterns corresponding to the given patch in the level.

### 6.6.2 Model Definition

We now need to define the probability distribution of the model. This model’s distribution captures the probability of a pattern given the patterns (or possible patterns) of the adjacent hidden vertices, $P_{\text{hidden}}(v_i \in V_{\text{hidden}} = t_j \in T_{\text{hidden}} | \text{neighbors}(v_i))$. This distribution captures whether a combination of patterns is possible, and if so with what probability should a given pattern occur. Whether a combination of patterns is possible is determined by whether the given combination was observed in the training data, and the probability of a possible pattern occurring is given by the distribution of such patterns in the training data.
Figure 6.8: This figure shows an example level graph representation for the wave function collapse approach. Specifically, it shows a section of a *Super Mario Bros.* level (A); the atomic elements of that level section, that is, the observable tile types (C); the $3 \times 3$ observable tile type patterns extracted from the level section, note that these are the tile types for the hidden vertices (B); the hidden layer of the level graph which represents the patterns from the level section (D); and the portion of the observable layer of the level graph for one such hidden vertex (E). Notice, that when representing a training level, each hidden vertex only has one possible pattern type, but during sampling each hidden vertex can have labels corresponding to different sets of possible patterns. Furthermore, the pattern and atomic object sizes chosen for this figure are merely illustrative, and other sizes and dimensions are possible.

6.6.3 Sampling

Lastly, we need to define the parameters and helper functions of the *SequentialLevelSampling* function in order to define the sampling procedure of the WaveFunctionCollapse approach. $P_{\text{hidden}}$, $P_{\text{observed}}$, and $T$ are defined the same as in the model definition.

- $L_{\text{initial}}$: For the initial level graph, we need $n$ hidden vertices, each with $m$ edges to adjacent vertices forming a grid-like graph structure, where $n$ is the number of patches corresponding to patterns that make up the desired level (i.e., the desired size of the level) and $m$ is the number of adjacent patches to each patch (given by the shape of the patterns). Each of these hidden vertices will be labeled with the hidden tile type representing that all patterns are possible for the given position. Notice, this may not be the case if there are special restrictions on the types of patterns that may occur around the boundaries of the levels.

- StoppingCriteria($L$): The stopping criteria for this approach depends on the labels for both the hidden and observed vertices. Specifically, if the algorithm reaches a point where there is not a legal hidden
tile type for a given hidden vertex (i.e., the pattern sampling reaches a contradiction), then the stopping criteria is triggered, signifying a failure. Otherwise, if all the hidden vertices are labeled with tile types corresponding to a single pattern and all the observed vertices are sampled and labeled, then the stopping criteria triggers, signifying success.

- **SampleVertex(L):** This function creates the observed vertices, \( V_{\text{observed}} \). However, unlike the previous approaches we discussed, this function does not sample any observed vertices until all the hidden vertices have been assigned tile types corresponding to singular patterns. That is, the observed portion of level graph is not grown until each of the hidden vertices have been collapsed into a singular tile type. Once each of the hidden vertices is assigned a tile type corresponding to a single pattern, then this function creates the required number of observed vertices and edges for the given hidden vertices.

- **SampleTile(P, T, L):** This function is more accurately described by two separate functions:

  \[ \text{SampleTile}_{\text{hidden}}(P, T, L) \] \text{ and } \[ \text{SampleTile}_{\text{observed}}(P, T, L) \].

\( \text{SampleTile}_{\text{hidden}} \) operates on the set of hidden vertices in \( L \). This function first determines which hidden vertex has the lowest non-zero entropy. Notice, the entropy of a vertex is dependent upon the number of possible patterns that given vertex is able to take (i.e., more possible patterns leads to higher entropy). Once the lowest entropy vertex is found (with ties being broken randomly), this function probabilistically assigns a tile type corresponding to a single pattern to the chosen vertex based on \( P_{\text{hidden}} \). After choosing that vertex’s tile type, each of the other hidden vertices’ tile types are updated to reflect which patterns are still possible at that vertex. This process is repeated until all the hidden vertices are labeled with tile types corresponding to individual patterns, or until there is a contradiction (i.e., a hidden vertex has no possible patterns).

\( \text{SampleTile}_{\text{observed}} \) is only called after all of the hidden vertices have been assigned tile types corresponding to individual patterns. This function assigns the tile types of the sampled observed vertices. Notice, that each observed vertex will be assigned a specific tile type with probability 1.0 because of the influence of the hidden vertices’ tile types.
6.7 Comparison of Methods

This section compares the approaches described in this chapter in order to find gaps in the research, motivate future work, and determine limitations of the framework. The section starts with the level representation, then discusses the model definitions (i.e., the trained probability distribution), and finally the sampling approaches.

6.7.1 Level Representation

First, it is important to note that by defining the input of approaches as “level graphs” instead of the various representations the previously discussed models employed, we can see that even though most of the approaches have only been tested in 2-D games, and often only 2-D platformer games, they should in principle be applicable to any domain that can be represented by a “level graph.” That is, though all of the approaches above, aside from WaveFunctionCollapse\textsuperscript{65}, have primarily been applied to two-dimensional levels, the level graph representation easily allows for the extension to 3-D domains. In our level graph representation, this is as straightforward as adding more vertices and edges to the level graph and modifying what the tile types represent (i.e., 3-D objects and shapes instead of 2-D objects and shapes).

Additionally, the approaches we discussed mostly rely on grid-style levels to train their models. That is, the vertices correspond to discrete positions or areas within the levels, and the tile types easily correspond to objects and shapes to fit those areas. However, there are other types of levels. For example, levels in the first-person shooter game \textit{Doom} are commonly represented with connected line segments and objects. Representing these vector-based levels with the proposed framework is not as straightforward, but is possible. Continuing with the example of \textit{Doom} levels, we have 2 options:

1. Discretize the level to force it into a grid-like structure. Then this grid can be converted into a level graph using any of the previously discussed approaches. For example, each unique object can be given a corresponding tile type, or each unique column in the grid can be treated as a tile type, or clustering can be performed on groups of objects in order to extract tile types. In fact the Video-game Level Corpus\textsuperscript{96} contains a discretized level representation of \textit{Doom} levels in addition to the vector representations.

2. Treat each unique object and unique line segments of varying lengths and orientations as tile types.
Then each object and line segment in the level is represented by a vertex in the level graph. Edges can be added to the graph in several ways. For example, the \( n \) nearest vertices of each vertex can be connected, or all vertices within a given distance can be connected. Depending on the complexity of the level, this may not be desirable.

These approaches extend to 3-D continuous level domains such as terrain generation as well. Discretizing the terrain and identifying relevant or unique features is immediately applicable. For the second approach, moving to a 3-D domain may require some modification (e.g., treating terrain faces instead of lines as vertices).

### 6.7.2 Model Definition

Table 6.1 shows a summarized visualization of the different approaches described above. From the table we can see that there are several different methods for estimating the probability distribution of the tile types for the level graph vertices. The approaches employ different neighborhood functions (and therefore level graph network structures) and different estimation techniques. For example, several approaches estimate their distributions from the discrete counts of the occurrences in the training levels, while the LSTM approach employs a neural network to model a given domain. Notice, that we could in principle use any machine learning approach from the literature in order to estimate the probability distribution. However, it is important to note that different probability estimation methods have different tradeoffs.

One important tradeoff between different probability estimation methods is the amount of training data needed for the various approaches. For example, in Section 8.1 we show that the MdMC approach requires very little training data in order to get an accurate estimate on the probability distribution (sometimes as little at 1 level), while the LSTM approach may require more data (several more levels in general). In the extreme example, WaveFunctionCollapse only requires a single example of each desired tile type combination in order to generate quality content, while a neural network with many layers and connections typically requires a large amount of training data.

It is also important to consider that the type of neighborhood function possible may depend on which machine learning approach is desired for estimating the probability distribution. That is, not all neighborhood functions are supported by all machine learning approaches. For example, Markov chain-based approaches...
do not allow for cycles in the level graph, whereas Guzdial and Riedl’s approach requires a fully connected level graph.

**Table 6.1: Model Comparison**

This table shows a comparison of several machine learning-based level generation approaches represented in our framework.

<table>
<thead>
<tr>
<th>Method</th>
<th>T</th>
<th>Neighbors</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>MdMC $ns = n_{s3}$, no fallback</td>
<td>The set of tiles representing game objects (e.g., enemies, empty space, $?\text{-blocks, etc.})</td>
<td>$P$ estimated from counts of tile types given the neighbors.</td>
<td></td>
</tr>
<tr>
<td>$n$-grams, $n = 2$</td>
<td>The set of uniques columns appearing in the training data</td>
<td>$P$ estimated from counts of tile types given the neighbors.</td>
<td></td>
</tr>
<tr>
<td>LSTM</td>
<td>The set of tiles representing game objects (e.g., enemies, empty space, $?\text{-blocks, etc.})</td>
<td>$P$ estimated by applying a Soft-Max layer to the output layer of the LSTM.</td>
<td></td>
</tr>
<tr>
<td>Probabilistic Graph</td>
<td>The resulting clusters from groupings of game objects (e.g., enemies, empty space, $?\text{-blocks) and those groupings’ positions in the level.}</td>
<td>$P$ is estimated from the combined probabilities of each of the $g$ nodes within each cluster.</td>
<td></td>
</tr>
<tr>
<td>Wave Function Collapse</td>
<td>Patches of uniform size and shape that represent unique sections or objects in a level; Set of tiles representing game objects.</td>
<td>$P$ is the distribution over the pattern tile types that have been observed in the training data.</td>
<td></td>
</tr>
</tbody>
</table>

### 6.7.3 Sampling

In Section 6.1.3 we state that there are two types of sampling: **sequential** and **concurrent**. The sequential sampling approach alternates between sampling vertices in the level graph and then labeling those vertices. The concurrent sampling approach starts with the level graph completely sampled, and then assigns the labels to the vertices. The Metropolis-Hasting Markov random field sampling approach is an example of the concurrent approach, where the level graph is defined, each vertex is assigned a label, and then labels are swapped between the vertices based on a trained probability distribution.

The choice of sampling approach can have a potentially large effect on the type of levels that can be generated. With the sequential sampling approach, levels of various sizes and shapes can be generated without needing to define the dimensions before sampling; whereas with the concurrent sampling approaches, the dimensions of the level graph must be known prior to generation. Notice, some of the sequential sampling approaches have used stopping criteria in order to enforce particular level graph dimensions (e.g., the MdMC approaches), but these dimensional constraints can easily be replaced with other stopping criteria, such as tile counts or level likelihood, which were used by Guzdial and Riedl’s model.

---

**Chapter 6: Theoretical Framework**

6.7 Comparison of Methods
In this chapter we only discussed methods that employ the sequential sampling approach. However, it is straightforward to represent concurrent sampling approaches. In fact, the `ConcurrentLevelSampling` function can be seen as a special case of the `SequentialLevelSampling` function, where the initial level graph, \( L \), already contains all the necessary vertices and edges, and the vertex sampling function, `SampleVertex`, is a null function which does nothing. Then the `SampleTile` function would simply apply labels to all of the vertices. Notice, the `StoppingCriteria` for concurrent sampling approaches will need to depend on the vertex labels or other constraints, and not the dimensions of the level graph.

Lastly, though we only described approaches that employ sequential sampling, notice that the different approaches used a wide range of sampling techniques. For example, several different vertex sampling techniques were explored (e.g., sampling a vertex based on the immediately previously sampled vertices (MdMC) and sampling a vertex based on location in the level and connection probabilities (Guzdial)).

### 6.8 Discussion

Our theoretical framework outlined above has helped us to see that at the core of machine learning-based level generation models is a trained tile probability distribution based on a neighborhood function. This observation gives rise to two important questions:

1. **To what extent can video game levels be represented by a graph?**

2. **What relationships can be captured using neighborhood functions?**

In the following sections we discuss these questions.

#### 6.8.1 To what extent can video game levels be represented by a graph?

We touched on this first question when comparing the level representations of the various approaches. We saw that vertices can be used to represent objects of varying size and shape, as well as player path information. However, in order to more fully capture a domain, there are many more concepts that need to be represented.

First, notice that all the domains and models we explored all used regular patterns in the level representation (i.e., uses the same network structures throughout the level representation). However, there may be level
domains which have different structures or require different dependencies through the level. For example, *Mega Man* levels often have interleaved sections of vertical and horizontal progressions. Notice, that this can easily be captured by our level representation by modifying the connections between the nodes during these different sections. Specifically, the horizontal sections can be represented the same as, for example, *Super Mario Bros.* levels and the vertical sections can be represented the same as, for example, *Kid Icarus* levels. Then, where these sections connect or transition, the dependencies will need to be considered more carefully. Notice, that using irregular level structures may effect how training and sampling needs to occur. Specifically, we may need to train separate models for the vertical and horizontal sections, and the sampling algorithm (specifically the *SampleVertex* function) would need to account for the possibility of changing orientations. This same concept applies for other differences in the network structures through the level graph, such as different vertices having different numbers of dependencies (incoming edges). That is, the training and sampling procedures may need to be modified to account for different numbers of incoming edges depending on the model used.

Within gameplay, a player path has been captured as a tile type and used by several approaches, but this path only captures the discretized positions of the player-character as she moves through the level geometry. A more complete representation would need to include the actual actions the player took in order to reach each of those positions (e.g., button presses, in-game actions, etc.). More fully modeling gameplay may also need to account for how the enemies or other in-game objects behave (e.g., enemy paths, object state-machines). It may be possible to capture these concepts as tile types. For example, in the case of player actions or button presses, each action or press can be given a tile type, and an additional vertex labeled with an action or button press pointing to each player position vertex can be added to the level graph at each player position vertex. Notice, however, that thus far we have only discussed platformer or action/adventure games, where a player path can easily be extracted. There are games that do not have a concept of a player-character, and as such do not have a player path (i.e., some puzzle games require the player to directly move objects around without controlling a player character, or narrative-based games may have the player controlling a character, but only have them respond to plot points and events and not actually moving through a level space). In games such as these, capturing the gameplay in a tile type may prove much more difficult, as even capturing
the button presses may not give enough information, and capturing abstract actions (e.g., player moved object A to position \((x, y)\)) cannot easily be captured by tile types and level graphs. Additionally, other aspects of gameplay, such as the narrative/mission/story for a given level, may not be easily integrated into our level representation, because of their continuous and complex structure.

A concept that has not been explored much in level generation is aesthetics/cosmetics. Aesthetics/cosmetics are essentially the decoration within levels that do not impact gameplay or level structure (e.g., different sprite pallets, background, etc.). There has been very little level generation via machine learning work that has captured these ideas explicitly, but many aspects should be straightforward to model. For example, decorative objects and background can be captured with additional tile types corresponding to different images or objects, the same as with the structural components of the level. Different sprite pallets can be captured by creating separate tile types for the same object with different pallets, or by capturing the sprite pallet as a separate vertex indicating which style should be used for a given object connected to each vertex labelled with structural information.

In addition to the gameplay, aesthetic, and the standard structural information, there is meta-information about levels that may be beneficial to capture. Some examples of important meta-information include difficulty, the level’s position in a level progression, level archetype (different from aesthetic pallets), purpose of a level (teach player a new skill, introduce a new enemy, etc.). Some of these concepts are more easily modeled by tile types than others. For instance, difficulty and the difficulty curve of a level can be modeled with an additional set of vertices and tile types corresponding to different discretized difficulty scores. Similarly, the level archetypes can be captured using different tile types, and separate models can be trained for each archetype. However, the purpose of a level is a very complex concept, which may be very difficult to capture as a set of tile types.

Lastly, though all the approaches we discussed in the framework were easily represented by vertices and tile types, there are continuous and more complex domains which are more difficult to model. As we discussed previously, continuous levels from games such as *Doom* can be represented with level graphs by employing discretization or other techniques. However, this was a thought experiment and we do not know whether the results of applying the above level generation techniques to a transformed continuous domain
would yield desirable results. Similarly, any of the above approaches can be extended to 3-D domains by modifying what the vertices represent.

We have shown that a graph-based level representation can capture many aspects of level design. The places where this representation breaks down is when dealing with more abstract or complex concepts, such as narrative or level motivation. A potential avenue for future work would be to find a way to represent these more complex concepts with the theoretical framework. Another area of future work is applying the above (and other) approaches in more complex domains, such as 3-D levels or continuous levels, and determining whether the suggested level representation approaches hold well in those domains.

6.8.2 What relationships can be captured using neighborhood functions?

Our theoretical framework uses a neighborhood function which is defined as the incoming neighbors of a given vertex, and is used to determine which vertices the value of a given vertex depends on. The neighborhood function is another way of representing the network structure of the level graph. Using a neighborhood function allows us to capture dependencies between vertices in the level graph.

Edges between vertices in the level graph represent dependencies between those vertices, but the edges typically arise from the vertices proximity in the input level (i.e., two objects that are close together are more dependent upon one another than two objects that are far apart). These neighborhood functions allow for the easy modeling of local dependencies and relationships, but do not as easily allow for global coherency. An exception is Guzdial and Matteas’ approach\(^8\), where the graph is fully connected, and therefore captures global coherency. However, this approach focuses on level sections, so all of the objects in the level are within a bounded distance from one another. Notice, that in general the more connected and complex the network structure of the level graph, the more complex the probability distribution, \( P \), will be, and the more difficult \( P \) will be to estimate from training data. Therefore, the balance between connectedness and model complexity must be considered carefully. Because of the model complexity increase from adding many connections to the level graph, other ways of ensuring global coherence may be preferable (e.g., adding several vertices that encode high-level information about the level and influence the other vertices in the level graph).

It is important to notice that the edges in the level graphs in our theoretical framework are unweighted.
edges. This means that all of the vertices in a neighborhood exert the same amount of influence over the vertex they are connected to. However, there may be situations in which varying degrees of influence would be preferable. For instance, an approach that was aimed at capturing long-range dependencies as well as local dependencies may benefit from longer edges having less influence (lower weight) than shorter edges. Additionally, if a model is incorporating aesthetic information such as background objects into the level graph, then these aesthetic elements should likely have less influence on the structural components of the level than other structural components.

In this chapter we discussed several machine learning-based level generation approaches, and extracted a general representation for them. We then explored limitations and challenges for machine learning-based level generation approaches. In the following chapter we introduce extensions that could be applied to machine learning-based level generation approaches in the hopes of addressing some of the presented limitations and suggestions of improvements.
Chapter 7: Model Improvements

In the previous chapters we explored the use of machine learning-based level generation approaches using tile-based level representations. We first explored our own multi-dimensional Markov chain (MdMC) approach, our hierarchical multi-dimensional Markov chain (HMdMC) approach, and our Markov random field (MRF) approach, and then viewed these and several other machine learning-based level generation approaches in a uniform way through the lens of our theoretical framework. In those chapters we saw the limitations of our approaches as they were applied across several different domains. Specifically, we saw that in each domain our models had some difficulty inferring what the player is able to do (e.g., jump distance), or what the player may enjoy doing. Another drawback of the machine learning-based approaches discussed thus far are that while they do not require much domain knowledge from the user, they also do not give the user much control over the features of the generated levels. Lastly, most machine learning-based level generation approaches only attempt to capture structural information, but much more design information goes into level design than simply structural information. A notable exception is Summerville and Matteas’ approach which captures player paths as tile types. However, giving these models access to additional information may allow them to generate more interesting and usable levels. In order to address these issues, we must develop extensions applicable to machine learning-based level generation approaches.

In this chapter we introduce 3 such extensions:

1. **Constrained Sampling**: To address the issues of controllability and quality, we introduce several constrained sampling approaches that extend and incorporate the SequentialLevelSampling algorithm of the theoretical framework. We test these constrained approaches with our MdMC approach, but they are applicable to any approach that uses the SequentialLevelSampling algorithm. These constrained sampling approaches allow the user to provide a set of constraints that must be satisfied in the generated levels. Some examples of constraints include playability (i.e., it is possible to complete the level) and a limit on the number of enemies or on any specific object. These constrained sampling approaches give the user much more control over the output, but require more domain information as well.
2. **Player Modeling**: To address the issue of not accounting for player behavior or preference, we develop an extension that uses added information from a player model in order to guide the level generator towards what the player model believes to be desirable content. This again gives the user more control over the output of the models, and provides some design information to the models in the form of player interactions. Additionally, because the player model guidance is encoded as a constraint for the above constrained sampling approaches, this approach can be used with any models that employ the `SequentialLevelSampling` algorithm.

3. **Multi-layer Representations**: To further address the issue of providing additional design information to the models, we develop a multi-layer level representation. This multi-layer representation can be used to capture many different aspects of level design, and in our experiments we test a player path layer and a section layer. However, other layers such as a difficulty curve layer or an enemy path layer can easily be added. With these extensions we provide the user with more control over the output, and create more representative models of the domains.

The remainder of this chapter is organized as follows: first, in Section 7.1 we introduce our constrained sampling extensions and the results of experimenting with them; next, in Section 7.2 we describe our player modeling extension and how we use it to guide our level generation models towards generating more desirable levels; we then discuss our multi-layer level representation and experiment with it in two domains in Section 7.3; finally, we discuss the effects of our extensions in Section 7.4.

### 7.1 Constrained Sampling

Machine learning-based level generation approaches are able to capture models of a domain and generate levels for that domain, but they offer very little control to the user over the qualities of the generated levels. The only methods of control over the output available to the user are either manipulating the training data or modifying the model parameters. In Chapter 3 we show that carefully selecting the training data for certain domains can improve the quality of the generated levels, and in previous work\textsuperscript{106} we show that you can influence the expressive range of a generator by selecting certain training levels. However, selecting the proper training examples requires some knowledge of the domain, as well as knowledge of how the
model handles training data, and therefore how different training data may influence the model. Additionally, optimizing model parameters may require significant domain knowledge and model knowledge. Thus, we explore other methods for controlling the output of the models. Introducing this controllability to these machine learning-based approaches is important and desirable as it will allow the user to ensure that playable and usable levels are created, and it can increase the diversity of the generated levels by allowing levels to be generated that would be unlikely when using only the base model.

To address this problem, we turned to level generation approaches that allow the user a large amount of control over the generated content, constraint satisfaction-based approaches. In Section 2.5 we discuss constraint satisfaction approaches to PCG. At a high level, constraint satisfaction approaches take a set of constraints (i.e., properties the generated content must satisfy), and explore the space of possible solutions looking for content that satisfies the provided constraints. This is often performed using search and constraint propagation. To provide more control over the levels generated by machine learning-based approaches, we leverage the ideas from constraint satisfaction level generation approaches and develop several constrained sampling approaches. These constrained sampling approaches enforce user provided constraints in the generated levels through several different methods of resampling.

The remainder of the section is organized as follows: first, we introduce the types of constraints we use with our constrained sampling approaches and describe three constrained sampling approaches in Section 7.1.1; next, we discuss our experiments, including the set-up and the results in Section 7.1.2; and lastly, we draw conclusions about the constrained sampling approaches and their applicability in Section 7.1.3.

7.1.1 Methods

As discussed above, our goal is to augment the SequentialLevelSampling approach of our theoretical framework with a constrained sampling extension that gives the user more control over the types of levels generated. It is important to note while we test these constrained sampling approaches with the MdMC model, these extensions are applicable to any model that uses the SequentialLevelSampling algorithm to generate levels. We use the same level representation and training methods as described in the theoretical framework. Namely, a level is still represented by graph where each vertex is labeled with a tile type, and an edge indicates the
dependence between two nodes as described in Section 6.2.1; and the probability distribution for the model is still computed from the observed occurrences of the tile types in the training levels as described in Sections 3.2 and 6.2.2. The only change is in the sampling method. In order to change the sampling method, we first need to define the types of constraints that can be used, and then the different constrained sampling approaches we propose to enforce those constraints.

**Constraint Types**

We represent our constraints as cost functions that assign a numeric cost to a given level or section, assuming a higher cost signifies that the constraint is further from satisfied. We propose two types of constraints: the first simply returns a numeric cost associated with a section of level, where a section of a level is a collection of adjacent vertices and connecting edges delimiting a discrete portion of a level, and can include up to the entire level; the second returns a set of sections and total cost of the level. These constraint types are defined more formally below.

- **Simple Constraints**: These constraints take a section of a level (or an entire level) as input, and return a cost representing the degree to which the provided section violates the constraint. The cost is 0 if the constraint is satisfied, and some positive value otherwise. Formally, \( c : M \rightarrow [0, \infty) \), where \( c \) is the constraint function, and \( M \) is the section of the level. Specifically, \( M \) is a subgraph \( \langle V_M, E_M \rangle \) where \( V_M \subseteq V \) and \( E_M \subseteq E \), where \( L = \langle V, E \rangle \) is the level we are currently generating.

- **Location-aware Constraints**: These constraints take the entire level as input. They return a set of sections of the level that violate the constraint, as well as the total cost of \( c \) over \( L \) in the range of \( [0, \infty) \), where a cost of 0 means the level satisfies \( c \). Formally, \( c : L \rightarrow 2^S \times [0, \infty) \), where \( L \) is the generated level graph, \( S \) is the set of level sections in \( L \), and \( 2^S \) is the set of all subsets of level sections. We use \( c(L).cost \) and \( c(L).sections \) to represent the cost and sections returned by constraint \( c \) in map \( L \). Notice, location-aware constraints could be used as simple constraints by ignoring the returned level sections.

An example of a simple constraint in the domain of *Super Mario Bros.* is one that is satisfied (i.e., returns cost 0) if a path exists from the beginning to the end of the level, whereas an example of a similar location-aware
Figure 7.1: This figure shows a representation of the generate and test algorithm in Algorithm 9.

cost is one that is satisfied if a path exists from the beginning to the end of the level, but otherwise returns the section of the level in which the path ends and a nonzero cost.

The description of the specific simple and location-aware constraints we use in the experiments can be found in Section 7.1.2. Next, we describe three different algorithms for sampling a level while enforcing a set of provided constraints.

**Generate and Test Sampling**

This algorithm can be seen in Algorithm 9; a visual representation can be seen in Figure 7.1. This algorithm is the simplest of the three; it follows a standard generate and test methodology. That is, it takes a set of simple constraints, $C$, and starts by sampling a new level using the standard *SequentialLevelSampling* algorithm using the parameters described in Section 6.2.3 (line 3). It then determines the cost of the level using the given constraints (line 4). If the total cost of the level over all the provided constraints is greater than 0, then the loop repeats (line 2), and a new level is sampled (line 3). Notice that the initial level graph, $L_{initial}$, passed to the algorithm varies depending on the model and domain. Some examples of initial level graphs can be seen in Chapter 6.

```
Algorithm 9 GenerateAndTest($C, L_{initial}$)
1: totalCost = 1
2: while totalCost > 0 do
3:   $L_{new} = SequentialLevelSampling(L_{initial}, P, T, StoppingCriteria, SampleVertex, SampleTile)$
4:   totalCost = $\sum_{c \in C} c(L_{new})$
return $L_{new}$
```

**Incremental Sampling**

This algorithm can be seen in Algorithm 10; a visual representation can be seen in Figure 7.2. This approach samples a new level one section ($M$) of the level at a time, resampling the current section as needed until it
Figure 7.2: This figure shows a representation of the incremental sampling algorithm in Algorithm 10.

This sampling approach starts with an initial level graph (line 1), which as described in Chapter 6.1 varies by domain and approach. For the MdMC approach, the initial graph starts with a series of vertices labeled with sentinel tile types to signify the boundary of the level. Then, for each section to be sampled (line 2), it first extracts a new initial level graph from the current level \( L_{new} \) using the \texttt{getNewInitialGraph} function (line 5). This function takes a set of vertices from the level graph to which the next section will be appended. For the MdMC approach this method will give the column of rightmost vertices to indicate the current end of the level, and a vertex labeled with the sentinel tile to indicate the boundary. It then samples a section using the standard \texttt{SequentialLevelSampling} algorithm with the parameters described in Section 6.2.3 (line 6). Notice that the \texttt{StoppingCriteria} in this case indicates the size of the section to be sampled, and when to stop sampling new vertices. It then computes the cost of the sampled section given the constraints (line

---

Algorithm 10 \texttt{IncrementalSampling}(C, n, append, L\textsubscript{initial})

1: \( L_{new} = L_{initial} \)
2: \textbf{for} i = 1 \rightarrow n \textbf{do}
3: \hspace{1em} sectionCost = 1
4: \hspace{1em} \textbf{while} sectionCost > 0 \textbf{do}
5: \hspace{2em} M = L_{new}.\texttt{getNewInitialGraph}
6: \hspace{2em} M = \texttt{SequentialLevelSampling}(M, P, T, StoppingCriteria, SampleVertex, SampleTile)
7: \hspace{2em} sectionCost = \sum_{c \in C} c(M)
8: \hspace{1em} L_{new}.\texttt{append} \text{\textbf{sec}}
\textbf{return} Map

---

Chapter 7: Model Improvements 7.1 Constrained Sampling
7). If the cost of the section is positive (i.e., it does not satisfy the constraints), it is resampled. Otherwise the section, \( M \), is appended to the end of the current level by replacing the additional vertices added by the \textit{getNewInitialGraph} function with the original vertices they represent in \( L_{\text{new}} \).

![Figure 7.3: This figure shows a representation of the violation location resampling algorithm in Algorithm 11.]

**Violation Location Resampling**

This algorithm can be seen in Algorithm 11; a visual representation of the approach can be seen in Figure 7.3. At a high level, this algorithm works by sampling a new level and determining which sections of the level should be resampled according to the constraints. Next, it resamples each of those sections until the cost of that section is reduced with regards to the current constraint (without raising the cost of any other constraints). Afterwards, if any constraints are not satisfied, the process of finding and resampling sections is repeated.

In more detail, this algorithm takes a set of location-aware constraints, \( C \), and returns a level satisfying those constraints. The algorithm first samples a new level, \( L_{\text{new}} \) using the \textit{SequentialLevelSampling} algorithm (line 1). Next, if any constraints are unsatisfied (line 2), then for each constraint, \( c \in C \) (line 3), it iterates over the sections which violate constraint \( c \) (line 4). It then records the cost of the current section according to each constraint (lines 5-6). When checking the cost of a section, we use the location-aware constraint as a simple constraint. The algorithm then samples a new section, \( s_{\text{new}} \) (line 9). Notice, it samples a new section by taking current section, \( s_{\text{current}} \) and reducing it to an initial level graph by removing all the vertices defining it and only leaving the boundary vertices and the connecting vertices from the previous section. This allows the resampling to automatically replace the old section in the level graph. If the cost of \( s_{\text{new}} \) is greater than the previous cost for any other constraint (lines 10-12), then \( s_{\text{new}} \) is resampled, and cost checking is repeated. If the cost with regards to the other constraints is not raised, then \( s_{\text{new}} \) is only accepted if the cost with regards to the current constraint, \( c \), is lowered (line 13). This process of finding violated sections and
improving their costs is repeated until all constraints return cost 0 (line 2).

Algorithm 11 ViolationLocationResampling($C$, $L_{initial}$)

1: $L_{new} = \text{SequentialLevelSampling}(L_{initial}, P, T, \text{StoppingCriteria}, \text{SampleVertex}, \text{SampleTile})$
2: while ($\sum_{c \in C} c(L_{new}).\text{cost} > 0$) do
3:     for all $c \in C$ do
4:         for $s_{current} \in c(L_{new}).\text{sections}$ do
5:             for all $c_i \in C$ do
6:                 $\text{cost}_{c_i} = c_i(s_{current}).\text{cost}$
7:                 repeat
8:                     $s_{initial} = s_{current}.\text{getInitialLevelGraph}$
9:                     $s_{new} = \text{SequentialLevelSampling}(s_{initial}, P, T, \text{StoppingCriteria}, \text{SampleVertex}, \text{SampleTile})$
10:                for all $c_j \in C \setminus c$ do
11:                    if $\text{cost}_{c_j} > c_j(s_{new}).\text{cost}$ then
12:                       GoTo line 8
13:                until $c(s_{new}).\text{cost} < \text{cost}_c$
14:         return $L_{new}$

7.1.2 Experiments

We test these algorithms by sampling levels for two classic video games: Super Mario Bros. and Kid Icarus. We chose Kid Icarus to showcase the power of the constrained sampling approach. Recall, when using the standard MdMC, the hierarchical MdMC, and the MRF approaches, the best performing model was only able to sample a playable level 3.2% of the time. In the remainder of this section we introduce our domains in more depth, discuss the constraints to be used in each of the domains, explain the experimental set-up, and finally discuss the results.

Domains

Super Mario Bros. We experiment with sampling levels that are 12 tiles tall and 210 tiles wide, using sections that are 12 tiles tall by 10 tiles wide. Appendix A.1 has more information on the level representation.

Kid Icarus We experiment with sampling levels that are 160 tiles tall and 16 tiles wide, using sections that are 10 tiles tall by 16 tiles wide. Appendix A.2 has more information on the level representation.
Constraint Definitions

We defined constraints for each domain ranging from aesthetic constraints to playability constraints. Below we outline each constraint for each domain. Constraints listed with a • are simple constraints, and those listed with a − are location-aware constraints. Recall that location-aware constraints can be used as simple constraints, but simple constraints cannot be used as location-aware constraints.

Each constraint that enforces an interval has two settings: an easy setting (denoted below by \( E \)) that is tuned to the average value over the training levels plus a standard deviation for the maximum value and minus a standard deviation for the minimum value, and a hard setting (denoted below by \( H \)) that sets the minimum value to the average and the maximum value to the average plus two standard deviations. This allows us to test whether these constrained approaches are able to sample levels that are different from the training levels.

For Super Mario Bros. we used the following constraints:

- **Number-of-Pipes** \((\min, \max)\): To satisfy this constraint, a level must contain a number of pipes falling within \([\min, \max]\). If not satisfied, sections in the level where pipes can be added or removed are returned. \( E = \text{Number-of-Pipes}(1, 13), H = \text{Number-of-Pipes}(7, 19) \).

- **Number-of-Enemies** \((\min, \max)\): To satisfy this constraint, a levels must contain a number of enemies falling within \([\min, \max]\). If not satisfied, sections in the level where enemies can be added or removed are returned. \( E = \text{Number-of-Enemies}(11, 31), H = \text{Number-of-Enemies}(21, 41) \).

- **Number-of-Gaps** \((\min, \max)\): To satisfy this constraint, a level must contain a number of gaps falling within \([\min, \max]\). If not satisfied, sections in the level where gaps can be added or removed are returned. \( E = \text{Number-of-Gaps}(4, 12), H = \text{Number-of-Gaps}(8, 16) \).

- **Longest-Gap** \((\min, \max)\): To satisfy this constraint, the length of the longest gap must fall within \([\min, \max]\). If not satisfied, sections in the level where gaps can be modified are returned. \( E = \text{Longest-Gap}(4, 8), H = \text{Longest-Gap}(6, 10) \).

- **No-Malformed-Pipes**(): To satisfy this constraint, a level must not contain any malformed pipes. A malformed pipe is any pipe not consisting of both left and right pipe tiles or any upward facing pipe
without solid tiles beneath it (pipes that extend to the bottom of the map are not considered malformed). Sections in the level containing malformed pipes are returned for resampling.

- **Playability**(): To satisfy this constraint, a path must exist from the beginning to the end of the level. This is tested with an augmented version of Summerville’s A* agent\(^{97}\) that is able to account for springs, a special in game structure that allows the player to jump longer and higher than usual. Unplayable sections are returned for resampling.

- **Linearity**\((\text{min}, \text{max})\): To satisfy this constraint, the linearity of the level must fall within \([\text{min}, \text{max}]\). Linearity is the sum of distances from a best-fit line, where solid tiles (i.e., any tile that the player cannot pass through) are treated as points\(^{83}\). \(E = \text{Linearity}(280, 566), H = \text{Linearity}(423, 709)\)

- **Leniency**\((\text{min}, \text{max})\): To satisfy this constraint, the leniency of the level must fall within \([\text{min}, \text{max}]\). Leniency is the weighted sum of the number of enemies and gaps weighted by length\(^{83}\). \(E = \text{Leniency}(14, 46), H = \text{Leniency}(30, 62)\)

Notice, linearity and leniency only make sense when applied to an entire level, because having individual sections that are all highly linear does not necessarily mean the entire level will be linear. Similarly, requiring a high leniency value does not necessarily mean that we want each section to have high leniency values. For these reasons, the linearity and leniency constraints are only checked on an entire map, not sections.

For *Kid Icarus* we used the following constraints:

- **Number-of-Hazards**\((\text{min}, \text{max})\): To satisfy this constraint, a level must contain a number of hazards falling within \([\text{min}, \text{max}]\). If not satisfied, sections in the level where hazards can be added or removed are returned. \(E = \text{Number-of-Hazards}(0, 27), H = \text{Number-of-Hazards}(13, 41)\)

- **Longest-Gap**\((\text{min}, \text{max})\): To satisfy this constraint, the length of the longest gap must fall within \([\text{min}, \text{max}]\). If not satisfied, sections in the level where gaps can be modified are returned. \(E = \text{Longest-Gap}(10, 12), H = \text{Longest-Gap}(11, 13)\)

- **Average-Platform-Length**\((\text{min}, \text{max})\): To satisfy this constraint, the average length of the platforms in a level must fall within \([\text{min}, \text{max}]\). If not satisfied, sections in the level where platforms can be modified are returned. \(E = \text{Average-Platform-Length}(3, 4), H = \text{Average-Platform-Length}(3.5, 4.5)\)
– **Playability**: To satisfy this constraint, a path must exist from the beginning to the end of the level. This is tested by checking the distances between nearby platforms. Unplayable sections are returned for resampling.

The section sizes used are explained in the following section.

**Experimental Set-up**

We tested the algorithms by training MdMCs (as described in Section 3) on the training levels. The employed MdMC approach allows configuring parameters to better suit certain domains. In these experiments we set the parameters to the following values: for *Super Mario Bros.* we configure an MdMC with rowsplits = 12 and look-ahead = 3. For *Kid Icarus* we configure an MdMC with rowsplits = 10 and look-ahead = 3. We use these MdMCs to sample new levels with the constraint enforcement algorithms. However, not all algorithms support all the constraints defined. Specifically, the *VLR* algorithm cannot use simple constraints, because they do not provide the sections needing to be resampled. The incremental sampling approach cannot use constraints that limit the number of a specific element or that specify maximum or minimum average structure sizes, because the algorithm cannot reason globally about these constraints or identify problem sections; it can only modify the current section, and therefore may get stuck regenerating a single section in order to try to satisfy global properties that are unsatisfiable at that point. Thus, we experimented with three subsets of constraints:

- **IS**: For *Super Mario Bros.*, $IS = \{\text{Playability, No-Malformed-Pipes}\}$. For *Kid Icarus*, $IS = \{\text{Playability}\}$. These sets pair with all three of the algorithms.

- **VLR**: For *Super Mario Bros.*, $VLR = IS \cup \{\text{Number-of-Pipes, Number-of-Enemies, Number-of-Gaps, Longest-Gap}\}$. For *Kid Icarus*, $VLR = IS \cup \{\text{Number-of-Hazards, Longest-Gap, Average-Platform-Length}\}$. These sets pair with the *Violation Location Resampling* and *Generate and Test* algorithms.

- **GT**: For *Super Mario Bros.*, $GT = VLR \cup \{\text{Linearity, Leniency}\}$. For *Kid Icarus* $GT = VLR$. These sets pair with the *Generate and Test* algorithm.
We use the subindex $e$ to represent easy constraint value settings and $h$ to represent hard constraint value settings. For example $GT_e$ would be the setting of the $GT$ constraints using the easy setting of constraint values.

Location-aware constraints experiments return sections of size $10 \times 12$ for Super Mario Bros. and $16 \times 10$ for Kid Icarus (width by height). The constraints select the sections to return via a sliding window approach. Each of the windows tested where a violation was found is returned by the constraints; overlapping windows are combined. The Incremental Sampling algorithm uses the same section sizes as above. With these section sizes, Super Mario Bros. maps are one section tall, and Kid Icarus maps are one section wide. We sampled levels that are 21 sections long for Super Mario Bros. and 17 section tall for Kid Icarus.

During preliminary experiments we found that occasionally an algorithm would get stuck and be unable to satisfy the constraints (or take an excessive amount of time to do so). Thus, in these experiments, we limit the number of times a level could be resampled. That is, given a map of $N$ sections, we limit the number of total sections sampled to $50N$, after which we consider the execution to have failed.

In the preliminary experiments, we tested the sampling algorithms with a uniform distribution MdMC (i.e., an MdMC trained using no previous states, only learning the raw distribution of tile types through the level). We do this to determine whether the algorithms were overwriting the distribution by enforcing the constraints. We found that none of the algorithms paired with the uniform distribution MdMC could satisfy even the simplest constraints, showing the trained MdMC probability distribution remains an important part of sampling.

To evaluate the algorithms, we sampled 100 levels with each viable combination of algorithm and set of constraints. We recorded the percentage of levels sampled that satisfied the provided constraints ($Satisfied$) and the average number of sections sampled per satisfied level ($Attempts$). Note, $Attempts$ includes the initial sections sampled as well as the sections resampled. Lastly, we compare the algorithms against a baseline, where we sampled 100 maps using an MdMC with the same configuration as the constrained algorithms, but without enforcing any constraints. We then compute how many of those maps satisfy the various sets of constraints.
Table 7.1: Comparison of Constrained Sampling Algorithms: This table shows the results of generating 100 levels in Super Mario Bros. and Kid Icarus. This table reports percentage of levels generated with each sampling algorithm that were able to satisfy each set of constraints, and the average number of sections resampled for each level.

<table>
<thead>
<tr>
<th>Alg.</th>
<th>C</th>
<th>Super Mario Bros. Satisfied</th>
<th>Super Mario Bros. Attempts</th>
<th>KidIcarus Satisfied</th>
<th>KidIcarus Attempts</th>
</tr>
</thead>
<tbody>
<tr>
<td>MdMC</td>
<td>$GT_e$</td>
<td>42%</td>
<td>21.00</td>
<td>0%</td>
<td>NA</td>
</tr>
<tr>
<td>MdMC</td>
<td>$GT_h$</td>
<td>1%</td>
<td>21.00</td>
<td>0%</td>
<td>NA</td>
</tr>
<tr>
<td>MdMC</td>
<td>VLR$_e$</td>
<td>50%</td>
<td>21.00</td>
<td>0%</td>
<td>NA</td>
</tr>
<tr>
<td>MdMC</td>
<td>VLR$_h$</td>
<td>3%</td>
<td>21.00</td>
<td>0%</td>
<td>NA</td>
</tr>
<tr>
<td>MdMC</td>
<td>IS</td>
<td>72%</td>
<td>21.00</td>
<td>0%</td>
<td>NA</td>
</tr>
<tr>
<td>G+T</td>
<td>$GT_e$</td>
<td>100%</td>
<td>75.18</td>
<td>0%</td>
<td>NA</td>
</tr>
<tr>
<td>G+T</td>
<td>$GT_h$</td>
<td>59%</td>
<td>419.29</td>
<td>0%</td>
<td>NA</td>
</tr>
<tr>
<td>G+T</td>
<td>VLR$_e$</td>
<td>100%</td>
<td>72.66</td>
<td>0%</td>
<td>NA</td>
</tr>
<tr>
<td>G+T</td>
<td>VLR$_h$</td>
<td>75%</td>
<td>435.96</td>
<td>0%</td>
<td>NA</td>
</tr>
<tr>
<td>G+T</td>
<td>IS</td>
<td>100%</td>
<td>54.81</td>
<td>0%</td>
<td>NA</td>
</tr>
<tr>
<td>VLR</td>
<td>VLR$_e$</td>
<td>100%</td>
<td>23.01</td>
<td>94%</td>
<td>275.04</td>
</tr>
<tr>
<td>VLR</td>
<td>VLR$_h$</td>
<td>75%</td>
<td>176.43</td>
<td>0%</td>
<td>NA</td>
</tr>
<tr>
<td>VLR</td>
<td>IS</td>
<td>100%</td>
<td>21.32</td>
<td>100%</td>
<td>98.36</td>
</tr>
<tr>
<td>IS</td>
<td>IS</td>
<td>100%</td>
<td>21.02</td>
<td>100%</td>
<td>210.76</td>
</tr>
</tbody>
</table>

Figure 7.4: A portion of a Super Mario Bros. level sampled using the Violation Location Resampling algorithm while enforcing the $VLR_h$ constraints. Notice the number of enemies and the drastically varying heights within the level.

Results

Table 7.1 shows the results of the experiments. Rows labeled with “MdMC” are the baseline model, as in Chapter 3 (without considering any constraints), where the number of levels satisfying each set of constraints is computed afterwards.

As expected, all of the algorithms (excluding the Generate and Test algorithm when used with Kid Icarus) were able to produce a higher percentage of levels satisfying the provided constraints than the MdMC alone was able to. This suggests that by providing constraints, the user is able to guide the sampler towards more desirable maps.

An interesting trend, when looking at the Super Mario Bros. results, is that when comparing the three algorithms using the $VLR_e$, $VLR_h$, and IS constraints the three algorithms achieve the same satisfaction.
Figure 7.5: A portion of a Kid Icarus level sampled using the Generate and Test algorithm that was unable to satisfy the $GT_c$ constraints where an unplayable location is circled (left), and a portion sampled using the Violation Location Resampling algorithm successfully enforcing the $VLR_c$ constraints (right). Note that in this game the player can pass through one side of the screen and emerge on the other.

percentage for each set of constraints, but the Violation Location Resampling (VLR) and the Incremental Sampling (IS) algorithms outperform the Generate and Test (G+T) algorithm with respect to the number of sections sampled. This is because $G+T$ must resample all sections if a constraint is violated anywhere, whereas VLR and IS selectively resample particular sections in violation of the constraints. However, in Kid Icarus, notice that VLR outperforms IS in terms of sections sampled (VLR, sampling fewer than half as many sections as IS). This may stem from the dependence a section has on the previous section in Super Mario Bros., where each row continues from the row of the previous section, whereas in Kid Icarus a section is far less dependent on the previous section, due to the vertical orientation of the map, which can lead to more diversity in the sections being sampled, and therefore may require more resampling to produce a satisfactory
Furthermore, in Kid Icarus, the VLR and IS algorithms both outperform the G+T algorithm, achieving the same percentage of satisfied maps, where the G+T algorithm and the baseline were unable to produce any. This is a remarkable result as it indicates that with the proper constraints, an MdMC can sample usable maps for domains which may be beyond the reach of current machine learning-based PCG methods.

Figure 7.4 shows a portion of a level sampled using the VLR algorithm enforcing the $VLR_h$ constraints, as evidenced by the drastically varying heights and large number of enemies. Figure 7.5 (left) shows a portion of an unsatisfied Kid Icarus level sampled using the G+T algorithm enforcing the $GT_e$ constraints, where a gap that is too long to jump over is circled. Figure 7.5 (right) shows a portion of a level sampled using the VLR algorithm enforcing the $VLR_e$ constraints.

### 7.1.3 Conclusions

In this section we presented extensions to the SequentialLevelSampling approach of the theoretical framework. The aim of these extensions was to provide more control over the type of levels generated by a machine learning-based level generation approach by allowing the user to provide the sampling algorithm with constraints on the level. The presented approaches are three ways of enforcing the user provided constraints while sampling new levels: a generate and test approach, an incremental sampling approach, and a violation location resampling approach. We tested the algorithms in two domains, Super Mario Bros. and Kid Icarus; the first of which the standard MdMC approach has been shown to be able to sample usable levels for, and the second of which the MdMC approach has been shown to have a very difficult time generating usable levels for.

The experiments showed that the constrained sampling algorithms do provide more control over the generated levels than the standard MdMC sampling approach allows. Specifically, in Super Mario Bros. the constrained approaches allow the user to control the specific qualities of the sampled levels (e.g., number of enemies, lengths of gaps, linearity, etc.). The algorithms even succeed in adhering to constraint settings that are outside of what is typical in training levels (e.g., large number of enemies). Alternatively, for Kid Icarus the constrained sampling approaches allow for the reliable sampling of playable content. Notice, that
while the harder constraint settings were not satisfied in this domain, being able to generate playable content 94 – 100% of the time is a big step forward in applying machine learning-based level generation approaches to more complex domains.
7.2 Incorporating Player Movement Models

Player modeling has been used in procedural content generation as a way of guiding generators towards content tailored to a specific player or player type\(^{112,113}\). Pairing a player model with a level generation approach allows for the creation of levels that support the kinds of movements and behaviors captured by the player model. One approach to using player models to guide content generation is called experience-driven PCG\(^{114}\). In experience-driven PCG, the player models are most commonly paired with search-based approaches as the means of generating content, but we are interested in defining both the player model and a model of the levels via machine learning.

Many machine learning-based level generation approaches thus far only focus on learning a model of the structural features of a given level domain\(^{5,68}\), without taking into account how a human player may interact with those structures and levels. Some notable exceptions include Guzdial and Riedl’s\(^{108}\) level graph model which attributes more importance to structures that the player interacts with more; and our multi-layer approach\(^{115}\) discussed in Section 7.3 which incorporates player path information into our MdMC model. Recently, Summerville et al.\(^{116}\) developed an approach that incorporates player behavior into the training levels in the form of a path, then trains an LSTM neural network on those annotated maps. This allows them to capture an implicit player model within their level model, which they then use to generate new levels. We will refer to this approach as the SGMR approach (after the authors’ names). However, all of these approaches use implicit player models, or player models that are subsumed by and learned in tandem as part of the level model. We are interested in separating the player model from the structural level model. Doing so allows the player model to be investigated separately from the level model, to be used to evaluate potential paths through the generated levels, and to be used more explicitly to guide the generator towards desirable content.

In this section we present an approach to learning a player movement model, and using that learned model to guide our level generation algorithm. Our player movement model is meant to capture the probability of the player’s movements through the level (e.g., moving left, moving down, etc.). We present several variations of movement models, where we learn the probabilities of movements based on previous movements, based on the surroundings of the player, and based on both.
The remainder of this section is organized as follows: first, in Section 7.2.1 we describe the player movement models we will be using to guide our constrained MdMC approach, including how we extract player traces from videos and how to train the movement model from the extracted traces; next, we discuss the chosen domain, the experimental setup, and the results of the experiments in Section 7.2.3; then in Section 7.2.4 we draw our conclusions.

### 7.2.1 Player Movement Model

Our player movement model captures the probability of the player-character performing a specific action. Formally, we define a player movement model as a finite set of actions $A = \{a_1, a_2, ..., a_n\}$, and a conditional probability distribution (CPD) conditioned on either the previous action, the current surroundings of the player-character, or both. We define an *action* as the operation performed by the player-character in a given time-step (e.g., movement, interacting with an objects, etc.). We define a *surrounding* as the area in the map around the player-character. Specifically, when using tile maps, a surrounding, $s$, is an $x \times y$ window of tiles with the player-character at the center. Given that we use a finite tile set to define our maps and the levels are of finite size, there are a finite set of surroundings possible, denoted $S$. Therefore, we can define the CPD of the player movement model as $P(A_t|A_{t-1})$, when conditioned on the previous action; $P(A_t|S_t)$, when conditioned on the current surroundings of the player-character; or $P(A_t|A_{t-1}, S_t)$, when conditioned on both, where $t$ indicates the current time-step.

To build the above conditional probability distributions, we extract sequences of actions and accompanying surroundings from gameplay videos. Below we explain how to extract the actions and surroundings from a gameplay video, and then discuss how to train the player movement models.

### 7.2.2 Play Trace Extraction from Video

Experience-driven procedural content generation\textsuperscript{114} leverages player models in order to guide generators towards desirable content. We are interested in automatically learning one such player movement model from gameplay videos. We start with explaining how to extract the sequence of actions from the videos and then how to extract the surroundings.
Figure 7.6: A section of a frame taken from a gameplay video of Super Mario Bros. (right) and the tile representation (left). Notice that though the player-character does not completely occupy one tile position, its tile representation is placed into the most fully occupied position (denoted by “X”).

We start, similarly to the SGMR approach, by converting the gameplay video into individual frames. We then represent each frame as a $w \times h$ tile map, with a special tile type representing the player-character’s position in the map. Note that while the position of the player-character in the tile map must be discrete, the player-character’s movement in the video may be continuous, resulting in the player-character not falling exactly into one position in the tile representation. To remedy this, we determine which tile position contains the most of the player-character, and place the player-character’s tile representation in that position in the tile map. An illustration of this process can be seen in Figure 7.6. This process is also used to place other moving elements, such as enemies.

Once we convert all the frames into their tile representations, we can extract the action and surrounding sequences. To extract the action sequence, we compare sequential pairs of frames. First, we align them according to their level geometry, and then determine the difference between the positions of the player-character in each frame. This gives the action taken between the two frames (e.g., moving right, jumping up, standing still). We repeat this process for each sequential pair of frames to get the sequence of performed actions.

To extract the sequence of surroundings, we examine each frame’s tile representation individually. We locate the position of the player-character’s tile and extract an $x \times y$ tile window centered at that position. If the window extends beyond the edges of the tile frame, we fill those positions with sentinel tiles. Repeating this for each frame gives a sequence of surroundings. Because there are many possible surroundings, we perform
2.1 Movement Model Training

Given action and surrounding sequences we can now train the player movement models. We propose three models:

- **Actions Only**: This model learns the probability of performing an action given the previous action. That is, it learns \( P(a_t|a_{t-1}) \), where \( a_t, a_{t-1} \in A \), and \( A \) is the finite set of all observed actions.

- **Surroundings Only**: This model learns the probability of performing an action given the surroundings of the player (i.e., the map geometry). That is, it learns \( P(a_t|s_t) \), where \( a_t \in A \), as above, and \( s_t \in S \), and \( S \) is the finite set of exemplar surroundings.

- **Actions and Surroundings**: This model learns the probability of performing an action given the previous action and the current surroundings. That is, it learns \( P(a_t|a_{t-1}, s_t) \), where \( a_t, a_{t-1} \in A \) and \( s_t \in S \).

To train these models, we estimate the conditional probability distribution for the model according to the frequency of occurrences in the gameplay video via the extracted action and surrounding sequences. That is, we count how many times each action follows each condition (action, surrounding, or pair of action and surrounding), and then set the probability of each action occurring following each condition according to the observed counts.

### 7.2.3 Experiments

We test our approach by sampling maps for the classic video game, *Super Mario Bros*. The remainder of this section describes the chosen domain, elaborates on the experimental set-up, and reports the obtained results.
Domain

We use *Super Mario Bros.* as our test domain. We extracted play traces for 4 levels using the method outlined in Section 7.2.1, for a total of 2,685 frames. We extracted the play traces from a gameplay video posted online\(^1\) of a single human player playing through the game. A more detailed description of the domain can be found in Appendix A.1.

Experimental Setup

We tested the approach by training an MdMC (as explained in Chapter 3) on the 4 maps for which we had play traces. The MdMC approach allows for the configuration of several parameters to improve performance depending on the domain. For these experiments, we set the parameters as follows: \(\text{rowsplits} = 14\), the height of the levels; \(\text{look-ahead} = 3\); and using the network structure \(n_3\), seen in Figure 3.2, and falling back to \(n_2, n_1,\) and \(n_0\) as needed. We use the trained MdMCs paired with the violation location resampling (VLR described in Section 7.1) algorithm in order to enforce 2 constraints:

- **Playability()**: To satisfy this constraint, a path must exist from the beginning to the end of the map. This is tested with Summerville et al.’s A* agent\(^97\). Unplayable sections are returned for resampling.

- **Likelihood(\(\text{min}\))**: To satisfy this constraint, the path through the level found by the A* agent must have a likelihood above \(\text{min}\), as evaluated by a specified player movement model defined in Section 7.2.1. The lowest likelihood sections are returned for resampling.

In order to simplify the *Surroundings Only* and *Actions and Surroundings* player movement models, we performed \(k\)-medoids clustering (with \(k = 20\)) using the \(5 \times 5\) windows surrounding the player in each frame as the objects to cluster. We found in preliminary experiments that 20 clusters were enough to capture most of the various structures found in the training maps, and that \(5 \times 5\) windows captured enough of immediate surrounding information.

For these experiments, we use the *Actions and Surroundings* player model to evaluate the likelihood of the agent’s path during sampling. We chose the minimum value of 0.15 based on preliminary experimental

---

\(^1\)youtube.com/watch?v=bNNwIFuZCMo
Table 7.2: Likelihood of Paths through Maps: This table shows the results of generating 100 levels in *Super Mario Bros.* using the constrained sampling approach paired with a path likelihood constraint. This table reports the average likelihood of the paths through the generated levels.

<table>
<thead>
<tr>
<th></th>
<th>L_A</th>
<th>L_S</th>
<th>L_AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$TD_{Player}$</td>
<td>0.22592</td>
<td>0.14577</td>
<td>0.23890</td>
</tr>
<tr>
<td>$TD_{Agent}$</td>
<td>0.12872</td>
<td>0.14512</td>
<td>0.11846</td>
</tr>
<tr>
<td>$VLR_{Playability}$</td>
<td>0.12687</td>
<td>0.15300</td>
<td>0.11894</td>
</tr>
<tr>
<td>$VLR_{Likelihood}$</td>
<td><strong>0.16079</strong></td>
<td><strong>0.19128</strong></td>
<td><strong>0.15204</strong></td>
</tr>
<tr>
<td>$SGMR_{A}$</td>
<td>0.13388</td>
<td>0.15630</td>
<td>0.12134</td>
</tr>
<tr>
<td>$SGMR_{B}$</td>
<td>0.13133</td>
<td>0.14751</td>
<td>0.11740</td>
</tr>
<tr>
<td>$SGMR_{C}$</td>
<td>0.13265</td>
<td>0.15402</td>
<td>0.12043</td>
</tr>
<tr>
<td>$SGMR_{D}$</td>
<td>0.13352</td>
<td>0.15893</td>
<td>0.11801</td>
</tr>
<tr>
<td>$SGMR_{Avg}$</td>
<td>0.13284</td>
<td>0.15419</td>
<td>0.11929</td>
</tr>
<tr>
<td>$SGMR_{All}$</td>
<td>0.13750</td>
<td>0.15817</td>
<td>0.12261</td>
</tr>
</tbody>
</table>

Results

We then sampled 100 levels using the VLR algorithm paired with only the playability constraint and 100 levels with the VLR algorithm paired with both the playability constraint and the likelihood constraint. We compare the likelihood of the A* agent’s path through the sampled levels against the observed player’s path through the training maps, the A* agent’s path through the training maps, as well as the A* agent’s path through 100 levels sampled by the SGMR approach using various play traces.

Results

Table 7.2 shows the results of the experiments. $L_A$ refers to the average likelihood of the A* agent’s path through the levels evaluated by the *Actions Only* player movement model, $L_S$ refers to the same using the *Surroundings Only* model, and $L_{AS}$ refers to the same using the *Actions and Surroundings* model. $TD_{agent}$ and $TD_{player}$ refer to the training levels’ paths for both agent and player, respectively; $VLR_{Playability}$ and $VLR_{Likelihood}$ refer to the sampled levels’ paths, using only a playability constraint and with both playability and likelihood constraints, respectively; and $SGMR$ refers to the paths of levels sampled by the SGMR approach using each of their play traces for training (A-D), the combined values for the levels sampled using each of those videos (Avg.), and the levels sampled after training on all of the videos (All). Note that we randomly selected levels from sets of levels generated by the SGMR approach with each of the play traces until we had 100 levels that were able to be completed by the A* agent for each of the configurations. We used the A* agent’s paths in these levels instead of the paths generated by their method in order to ensure
uniformity throughout our evaluation.

First, it is important to note that, in general, the paths obtained from the A* agent are much less likely than the player’s path (when comparing the paths in the training maps). This is to be expected, as the evaluation models were trained on the observed player’s path, and an A* agent is unlikely to behave very similarly to a human player. However, when only accounting for the current surroundings of the player, the likelihoods of the player and agent paths are similar (again for the training maps’ paths). This is because when considering only the surroundings, there are often situations with one obvious solution. For example, if approaching a pit with no other obstacles, it is likely for the player or agent to jump; similarly, if on a flat surface with no other obstacles, it is likely for the player or agent to move forward. This movement model is too simplified though, as it only captures what the player might do given the current surroundings without accounting for what may have happened immediately previous. For example, if the player-character is in the air, then only accounting for the surroundings it is difficult to predict if she is moving up (jumping) or down (falling). With knowledge of the previous action, this prediction is much easier, as it is more clear in which direction she was moving previously.

Second, notice that the likelihood of the agent’s path is fairly uniform across all of the maps, except for the...
Figure 7.8: This figure shows a probabilistic finite state machine where each node represents an action type. This represents the *Action Only* model. “R” is moving right, “L” is moving left, “U” is moving up, “D” is moving down, “N” is no movement, and the combinations are diagonal movements. For clarity, we only include each node’s two most probable transitions.

Figure 7.9: This figure shows example levels sampled using the *VLR*\textsubscript{Labelhood} approach (top two), *VLR*\textsubscript{Playability} (third and fourth), and an example level for the *SGMR* approach with each of the play traces (*A – D*) and *All* (bottom five, in order).
maps sampled by the VLR algorithm enforcing a playability and likelihood constraint. This is to be expected, as this approach will only sample levels that have a path likelihood above a threshold. However, while our method guarantees a certain level of path likelihood, we are also interested in investigating how the structures of the various levels relate.

Figure 7.7 shows the sampled levels projected in a two-dimensional space based on a measure of distance between them using the t-SNE visualization algorithm. To determine the distance between two levels, we represent them as a histogram of high-level tiles and compute the Euclidean distance between these histograms. High-level tiles were found by clustering $4 \times 4$ tile sections using $k$-medoids ($k = 30$) with the four training levels and one level from each of the sampled sets. The projection shows that levels sampled with our approach and the other sets of sampled levels are quite distinct from one another, and that our sampled levels lie closer to the training levels, while the other levels are separated from them. Further, the levels sampled using the likelihood constraint lie closer to one particular training level, while the levels sampled using only the playability constraint lie closer to the other three training levels. This is likely due to the structure of the training levels. The one training level is flat with very few other structures implying that to create levels with high-likelihood paths, the model generated levels with stretches of flat space (as moving forward on a flat space is very likely); this is reflected in Figure 7.9 (top two). Alternatively, the levels sampled without the likelihood constraint have more obstacles, as seen in Figure 7.9 (third and fourth).

Lastly, notice that the levels sampled by $SGMR_D$ are distanced even from the other $SGMR$ levels. In the authors note that the player in Video D attempted to collect all the coins and took long paths through the levels. This resulted in levels with many more structures and platforms than the other sampled levels, and could explain the distance from the other sampled levels.

An additional benefit of separating the player movement model from the level model is the ability to examine the movement model independently. Figure 7.8 shows a visualization of the Actions Only player movement model (showing only the two most probable transitions from each action, for clarity). This allows the investigation of what types of behaviors are common or uncommon given the trained model. For example, the figure shows that many of the actions have a high probability of transitioning to the “None” action (not moving). This indicates that it is likely for the player to change between different actions by first stopping.
(e.g., if the player is moving left (“L”) and wants to move right (“R”), it is more likely for the player to stop moving, and then begin moving right than it is to immediately start moving right).

Figure 7.9 shows randomly selected sampled levels from each of the approaches (also highlighted in Figure 7.7). As expected of the $VLR_{Likelihood}$ maps (top 2), there are long stretches of flat terrain, whereas the $VLR_{Playability}$ maps (third and fourth) have many more mountainous structures and pipes.

The most important result is that these approaches are able to sample levels that force the A* agent to take higher likelihood paths through the levels. This shows that by guiding the sampling algorithm with a player movement model it can sample levels that afford paths resembling those taken by the human players from whom the player movement model was learned.

### 7.2.4 Conclusions

In this section we present an approach for using player models to guide a machine learning-based level generator. The goal of this approach was to develop an explicit player model separate from the level model, in order to allow the player model to be investigated independently of the level model, and to allow the player model to be used to explicitly guide the level generator. We tested this approach using a few simple player movement models paired with our constrained MdMC approach, and compared against another approach that uses a combined player and level model approach.

The experiments show that explicitly guiding the generator allows the level generator to create levels with likely paths according to the player model. Additionally, the experiments show that the levels generated with our paired approach are more structurally similar to the training levels that those generated by the other approach. Lastly, we were able to gain deeper understanding of the player model by being able to investigate it directly, and exploring the likely transitions between the different player actions.

This work shows that using an explicit player model to guide machine learning-based level generators (much like how player models are used to guide search-based generators in experience-driven PCG) is a viable approach. However, this basic approach can be extended in several ways. For example, the levels generated when forcing likely paths tended towards flatness, due to the use of an A* agent as the means of generating the new path. Leveraging a more human-like agent during testing can alleviate this issue.
Furthermore, the player movement model we proposed and tested is a fairly simple model that only accounts for player movements. This can be extended to account for additional actions (e.g., shooting fireballs in Super Mario Bros.), or more high-level actions (such as jump or kill enemy). Lastly, we only explored this work in the context of Super Mario Bros., a platforming game; it may be interesting to explore these approaches in the context of more complex games.
7.3 Multi-layer Level Representations

Multiple layers of representation have been used in machine learning, most commonly with neural networks which often learn multiple layers of representation from the input\textsuperscript{118}, in order to better represent training data, and develop more robust models. For example, Oquab et al.\textsuperscript{119} present an approach for learning a mid-level abstraction of the training data with a convolutional neural network, and then using that learned representation along with data from another related domain to help with classification of images. Additionally, neural network approaches have been used to generate interesting images\textsuperscript{120} and models\textsuperscript{121}. However, a key difference between images/models and levels are that levels have functional requirements as well as aesthetic requirements (i.e., the levels need to be playable and well-formed in order to be usable, whereas an interesting image does not to allow for interaction). Neural networks also typically require a large amount of training data in order to train, but training data is often scarce for machine learning-based level generation approaches. Therefore, machine learning techniques that can leverage smaller datasets are required.

The machine learning-based level generation techniques discussed thus far have all focused on learning the structural information of a given level domain. However, there is much more to level design than simply placing objects in reasonable positions. For example, a level may be designed with a specific player path in mind, or it might be designed in order to introduce new concepts, or to increase the difficulty as a part of a series of levels. There have been approaches that incorporate player interaction\textsuperscript{108} and player/agent paths through levels\textsuperscript{116} into their models, and the approach discussed in Section 7.2 builds a separate player model in order to guide the generator. However, incorporating player interactions and paths or player models in these ways is only meant to generate more usable levels with more human-like paths, but does not allow the user of the generators to specify a path for the generator to allow in the generated levels. Additionally, the above approaches are only able to leverage player and path information, and are not easily extensible to other level design information.

To address this gap, we develop a general multi-layer level representation that allows for the inclusion of arbitrary layers of representation of a given level. This multi-layer representation is usable with any machine learning-based level generation technique that uses the level graph representation defined in our framework.
We test this multi-layer approach with a modified version of our multi-dimensional Markov chain (MdMC) approach paired with the violation location resampling algorithm (from Section 7.1 by generating levels in *Super Mario Bros.* and *Lode Runner*).

The remainder of this section is organized as follows: first, we introduce the multi-layer level representation and the augmentations to the MdMC and VLR approaches needed for it to function in Section 7.3.1; then, we discuss the chosen domains, the layers we use to represent those domains, and the experiments and results in Section 7.3.2; we close the section by reframing this multi-layer approach through the lens of our theoretical framework in Section 7.3.3 and by drawing our conclusions in Section 7.3.4.

### 7.3.1 Methods

In this section we first introduce the multi-layer level representation, and then discuss how we modify the multi-dimensional Markov chain level generation approach to work with multi-layered levels.

#### Multi-layer Level Representation

We represent a level using a set of layers, $Y = \{y_1, y_2, \ldots, y_n\}$, where each $y_i$ is a two-dimensional matrix of tiles with dimensions $h \times w$ (height and width). Each layer has a separate set of tile types, $T_i$, where the meaning of the tiles varies by layer. For example, one layer may have tile types to represent the structures and objects in the level, while another layer may have tiles to represent paths through the level, and yet another layer could have tiles to represent meta-data about the level, such as sections.

Figure 7.10 shows a section of a *Super Mario Bros.* level represented using three layers: a *structural* layer (top-right), representing the placement of objects in the level; a *player path* layer (bottom-left), representing the path a player may take through the level; and a *section* layer (bottom-right) representing the section for each position in the level to help the model learn that patterns appearing at lower sections might be different from those at higher sections (Section 7.3.2 describes these layers in more detail). We refer to the structural layer as the *main* layer, because it is the layer that provides the tile types used during level sampling.

Though this is a straightforward concept, it opens up the possibility for level representations which are able to represent more information about a level, such as player path information, in addition to the infor
Training a single-layer MdMC requires two things: 1) the network structure and 2) training levels. The network structure specifies which of the surrounding states the value of the current state depends upon. Using
the network structure and training levels, the conditional probability distribution, $P$, of each tile given each set of surrounding tiles is calculated according to the frequencies observed in the training data. A more detailed description of this can be found in Chapter 3.

Training a multi-layer MdMC requires a network structure, and training levels represented in multiple layers. The conditional probability distribution, $P_m$, is computed much the same way as for the single-layer approach. The main difference between training a single-layer and multi-layer MdMC is that the network structure of the multi-layer MdMC may contain states from other layers, allowing $P_m$ to learn dependences across multiple layers of representation. Figure 7.11 shows example network structures that can be used to train a single-layer MdMC (left) and a multi-layer MdMC (right).

Multi-layer Sampling

We sample new levels using our MdMC model paired with the violation location resampling (VLR) algorithm in order to ensure playable and well-formed levels. Sampling a new level using a single-layer MdMC paired with VLR algorithm requires desired level dimensions, $h \times w$, and the conditional probability distribution, $P$, and a set of constraints, $C$. A new level is then sampled in its entirety one tile at a time starting, for example, in the bottom left corner and completing an entire row before moving onto the next row. For each position in the level, a tile is sampled according to $P$ and the previous configuration. Once the complete level is sampled, each constraint $c \in C$ is checked against the level, and sections of the level violating those constraints are resampled. When sampling a new tile, the most complex model is used first, and if it cannot generate a tile...
satisfying the look-ahead, then it falls back to a simpler one. For the single layer model, network structure $ns_3$ falls back to $ns_2$, then to $ns_1$, and finally $ns_0$, which is the raw distribution of tiles observed during training. More information on the look-ahead and fallback procedures as they apply to MdMCs can be found in Section 3.2.3 and more information on the VLR algorithm can be found in Section 7.1.

Sampling a level using a multi-layer MdMC paired with the VLR algorithm functions the same as the single-layer method, but with one adjustment: because the trained distribution, $P_m$, only models the probability of tiles in the main layer, and previous configurations contain states from the other layers, the other layers need to be defined before sampling or computed during. In these experiments we define the player path and section layers prior to sampling. Our goal is to guide the sampler towards creating levels that allow for the provided path. Notice, Figure 7.11 (right) only shows $nsl_3$ for the multi-layer model. However, the multi-layer network structures follow the same pattern as the single-layer network structures, but retain the dependencies from the other layers. For example, $nsl_2$ would depend on the tiles to the left and below in the structural layer (as in $ns_2$), as well as on the tiles at the current position in the height and player path layers. The fallback order for the multi-layer models is the same as in the single-layer approach.

7.3.2 Experiments

We test this approach by generating levels for the classic games Super Mario Bros. and Lode Runner. For Super Mario Bros. we experiment with generating levels that are $210 \times 14$ tiles, and for Lode Runner we generate levels that are $28 \times 16$ tiles. The goal is to determine whether the multi-layer approach is able to accurately model the domains including the complexity of the paths through the levels.

Level Representation

In this section we describe the layers used to represent the training levels in the two chosen domains. Each of the layers is represented by a grid, where each cell in the grid takes a value from a set of tile types meant to represent aspects from that layer.

**Structural Layer:** The structural layer is the main layer of the representation, and captures the placement of objects and enemies through the level. In Super Mario Bros. each cell takes the value from
a set of 36 tile types corresponding to the elements in the level (e.g., pipes, enemies, blocks, springs, etc.). Similarly, in Lode Runner each cell takes a value from a set of 10 tile types corresponding to blocks, ladders, ropes, enemies, and collectibles. Recall, this is the layer that will be sampled during generation. More information on these representations can be found in Appendix A.

**Section Layer:** The section layer is one of the additional layers, and captures the various sections of the level. In Super Mario Bros. this layer essentially splits the level into multiple sections by grouping rows of the level together to allow for more focused training to occur within each section. This works similarly to the row split parameter used by the single-layer MdMC approach. The Super Mario Bros. section layer uses a set of 6 tile types, where 4 tile types are each used for 3 consecutive rows, one is used for the final two rows, and the final tile type is a sentinel tile that signifies the boundaries of the layer. Figure 7.10 (bottom-right) shows a section of a level represented with this section layer.

In Lode Runner this layer splits the level into multiple rectangular sections to allow for more focused training within each of the sections. The Lode Runner section layers use a set of 21 tiles to represent the section layer. It splits each level into collections of $5 \times 4$ and $6 \times 4$ (width $\times$ height) tile sections, depending on the positioning within the level. We chose these sizes based on preliminary experiments.
Figure 7.12 (bottom-right) shows the section layer of a *Lode Runner* level.

**Player Path Layer:** The player path layer is the final layer. This layer captures a possible path through a level. In *Super Mario Bros.*, this layer is represented using 3 tile types: “X” represents a position in the level that the path passes through, “-” denotes positions not on the path, and “S” is a sentinel tile. During training, we use a modified version of Summerville et al.’s *A*\(^*\) agent\(^97\) to create the player path layer for the training levels. During sampling, we experiment with providing path layers of the *A*\(^*\) agent traversing a training level as well as hand authored path layers; these paths are discussed more in the following section. Figure 7.10 (bottom-left) shows a section of a level representing with the player path layer.

In *Lode Runner* this layer is represented with 5 tile types: “-” represents a position the player did not pass through; “d” represents a position where the player performed the “dig left” action, temporarily destroying a piece of breakable ground to the left and below the player; “D” represents a position where the player performed the “dig right” action; “X” represents a spot the player moved through without taking either dig action; and “S” is a sentinel tile. These paths are extracted from a video of a player completing the levels\(^2\). Figure 7.12 (bottom-left) shows the player path layer of a *Lode Runner* level.

**Experimental Set-up**

We tested our multi-layer approach by first training a single-layer MdMC (as described in Chapter 3) for *Super Mario Bros.* on the 29 training levels and for *Lode Runner* on 10 training levels. Recall that the training procedure does not change when using the VLR algorithm during sampling. The single-layer and multi-layer MdMCs require several parameters to be set before they can be trained and used to sample new levels.

- **Single-layer MdMC:** This model requires setting a look-ahead, a row split, and a set of network structures to be used during training and sampling. In our experiments for *Super Mario Bros.*, we use a look-ahead of 3 and row split of 5. For *Lode Runner*, we use a look-ahead of 2 and row split of 1.

For both domains we use \(n_{S3}\) as the main network structure. During sampling, \(n_{S3}\) will fallback to \(n_{S2}\)

---

\(^2\)www.youtube.com/watch?v=VwxLCxHl8WA
which falls back to \( ns_1 \), which finally falls back to \( ns_0 \), the raw distribution of tiles in the structural layer.

- **Multi-layer MdMC**: This requires a look-ahead value, a set of network structures to be used during training and as fallbacks during sampling, a player path layer to be used during sampling, and a section layer to be used during sampling. In our experiments, we use a look-ahead of 3. We use \( nsl_3 \) as our main network structure (as seen in Figure 7.11, right). For the fallback network structures, we use \( nsl_2 \), which falls back to \( nsl_1 \), which falls back to \( nsl_0 \). Recall, the \( nsl_i \) network structures are defined the same way as the \( ns_i \) network structures, with the addition of dependencies on the section and player path layer. During sampling, we provide the section layer shown in Figure 7.10 (bottom-right), and various player path layers for *Super Mario Bros*. For *Lode Runner* we use the same network structures, and provide the section layer in Figure 7.12 (bottom-right), and various player paths. A description of the player path layers we use during sampling can be found later in this section.

We use the trained model paired with the violation location resampling (VLR) algorithm described in Section 7.1 the following constraints (denoted \( M \) for *Super Mario Bros*. constraints and \( L \) for *Lode Runner* constraints):

**M Playability()**: To satisfy this constraint, a path must exist from the beginning to the end of the level. We test this with a version of Summerville et al.’s \( A^* \) agent augmented to account for springboards, which change the movements possible for the agent.

**M Wellformedness()**: To satisfy this constraint, each pipe, cannon, and springboard must be well formed in the level. That is, pipes must have a width of 2, with pipe tops placed accordingly; cannons must have bullet-bill shooters on top; and springboards must have both the bottom and top portion of the springboard. Additionally, all of these must be placed correctly on solid tiles.

**L Playability()**: To satisfy this constraint, a path must exist that connects all the treasure tiles in the level. We test this using a specialized method that is able to find the connected components of a simplified graph representation of the level, and determine if all the treasures are reachable using that simplified graph.
For *Super Mario Bros.* we sampled 1000 levels with the single-layer MdMC approach, and 1000 levels with the multi-layered MdMC approach using 4 different player path layers (i.e., 250 levels sampled for each player path layer). For the player path layers, we used 2 paths given by the $A^*$ agent: the first for *Super Mario Bros.* level 1-1, the first level; and the second for *Super Mario Bros.* level 2-1, which includes a springboard. Additionally, we use 2 hand-crafted player path layers: the first, is a simple plateau-shaped path, requiring a single jump and fall; the second, is a more complex path including many jumps of various height and distances.

For *Super Mario Bros.*, we are interested in evaluating how well our multi-layer approach is able to model interesting interactions from the tile representation we are using in our structural layer. To evaluate this, we use “spring boards” (an infrequent tile type that allows the main character to perform very high jumps), and calculate the average number of springs per sampled level and the percentage of those springs that are required to complete the given level (i.e., we want to see whether those spring boards were included because they were necessary for the specified player path, or if they were inserted by chance). We are also interested in evaluating how well our multi-layer model, given a player-path layer, is able to sample a level allowing for that path. Therefore, we compute the discrete Fréchet\(^{122}\) distance between the provided path and the actual path taken through the sampled level using an $A^*$ agent. Finally, we would like to know how well our approach models the training data. To accomplish this we show a t-SNE\(^{117}\) visualization of the training levels and sampled levels, and also compare the linearity and leniency\(^{83}\) of the sampled levels against the training levels. These metrics are described more fully below:

- **Linearity**: This measures how well the platforms in the level can be approximated with a best fit line. It returns the sum of distances of each solid tile type from the best-fit line, normalized by the level length.

- **Leniency**: This approximates the difficulty of the level by summing the gaps (weighted by length) and enemies (weighted by 0.5), and normalizing by the level length.

- **Fréchet\(^{122}\)**: This measures the distance between two paths. Intuitively it can be thought of as the minimum length of a rope needed to connect two people walking on two separate paths over the course
Table 7.3: Multi-layer Level Evaluation (Super Mario Bros.): This table shows the results of generating 1000 levels in Super Mario Bros. with the single layer approach and 250 with each of the multi-layer models. This table reports the average linearity and leniency of the levels, as well as the Fréchet distance between the provided paths and the actual paths through the levels (given by an A* agent) and the average number of springs placed in the levels and the percentage of those springs that are required to complete the level.

<table>
<thead>
<tr>
<th></th>
<th>Linearity</th>
<th>Leniency</th>
<th>Fréchet</th>
<th>Springs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
<td>2.309</td>
<td>0.151</td>
<td>–</td>
<td>0.440/27.3%</td>
</tr>
<tr>
<td>Single</td>
<td>2.213</td>
<td>0.160</td>
<td>8.456 (P1)</td>
<td>0.350/0.9%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12.54 (P2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8.696 (P3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30.58 (P4)</td>
<td></td>
</tr>
<tr>
<td>Multi P1</td>
<td>2.168</td>
<td>0.161</td>
<td>6.222</td>
<td>0.144/5.6%</td>
</tr>
<tr>
<td>Multi P2</td>
<td>2.180</td>
<td>0.168</td>
<td>8.144</td>
<td>0.272/1.5%</td>
</tr>
<tr>
<td>Multi P3</td>
<td>2.461</td>
<td>0.150</td>
<td>6.219</td>
<td>0.212/5.7%</td>
</tr>
<tr>
<td>Multi P4</td>
<td>2.251</td>
<td>0.166</td>
<td>6.794</td>
<td>0.220/5.5%</td>
</tr>
</tbody>
</table>

of the paths.

In these experiments, we want our sampled levels’ linearity and leniency to closely match the training levels’ linearity and leniency. Alternatively, we want the Fréchet distance between the provided path and the actual path to be minimized.

For Lode Runner, we sampled 500 levels with the single-layer and multi-layer model paired with each sampling approach (VLR and standard sampling). For this domain, we are interested in determining whether the multi-layer approach allows us to more easily sample usable levels for complex domains. Therefore, we record the playability of all levels sampled using the standard sampling approach. Additionally, we compare how easily the multi-layered approach generated playable levels as compared to the single-layer approach when using the VLR algorithm. We determine this by setting a limit on the number of sections that can be resampled (100 sections), and recording how many levels are unfinished as well as the average number of sections resampled per finished level for each model. Lastly, we are interested in whether the multi-layer approach gives the user more control over the sampled levels. This is determined by computing the percentage of gold pieces placed in the level that appear on the provided player path. We compare this percentage for the multi-layer approach against the average over all player paths for the single-layer approach as a baseline.

In the following sections we will first discuss the Super Mario Bros. results, and then discuss the Lode Runner results.
Figure 7.13: This figure shows the expressive range of the different models. The y-axis is the linearity of the sampled levels and the x-axis is the leniency of the sampled levels; all values are normalized to between 0 and 1. Standard refers to the single-layer model, while Path 1 - Path 4 refer to the multi-layer models using the different paths.

Super Mario Bros. Results

Table 7.3 shows the results of our experiments. The linearity and leniency columns show the averages of the values computed over all the levels in each set of levels. The Fréchet column shows the average Fréchet distance between the path provided in the player path layer to the multi-layer MdMC and the path taken by the $A^*$ agent through the sampled levels over all the sampled levels. For the single-layer model we compute the Fréchet distance between the four paths used for the multi-layer models and the path found for each of the sampled single-layer levels. The final column shows the average number of springs per level and the percentage of those springs that are required to complete the level. The rows correspond to the training levels, the levels sampled using the single-layer MdMC, and the levels sampled using the multi-layer MdMC model paired with each of the four provided player path layers.

First, notice that the single-layer MdMC and the multi-layer MdMC approaches were able to achieve linearity and leniency values close to the training levels. This is further reflected in Figure 7.13, which
Chapter 7: Model Improvements

7.3 Multi-layer Level Representations

Figure 7.14: This figure shows a section of level sampled using the multi-layer P1 MdMC model where a spring is needed to complete the level (left), and a section of a level sampled using the single-layer MdMC model where a spring is placed, but not needed to complete the level (right).

shows the expressive ranges of the sampled levels with respect to linearity (x-axis) and leniency (y-axis), and shows that all the models achieve a similar expressive range. This indicates that both of the models are able to accurately model the structural elements of the training levels. However, if we look to the Springs column in Table 7.3, we see that the single-layer MdMC has a very difficult time placing springboards in meaningful places. Alternatively, the multi-layer MdMC models place springboards much more reliably (typically around 5% instead of 0.9%). This shows that the multi-layer model is able to capture the nuances about how the springboards function better than the single-layer model. Figure 7.14 shows a section of a level sampled with the standard approach that placed a springboard where it is not needed (right) and a level sampled with the multi-layer approach that placed a necessary springboard (left).

Next, the Fréchet distances for all of the multi-layer models are lower than when comparing the provided paths against the A* paths taken through the single-layer models. This shows that our multi-layer approach is able to reliably generate levels that allow for the path provided to the model. In particular, notice that the Fréchet distance between single-layer model levels’ A* paths and P4 is much larger (30.58) than the other paths. This is likely due to the complexity of P4, which requires many large and small jumps. However, the multi-layer model (Multi P4) is still able to achieve a low Fréchet distance (6.794) for this complex path.

Figure 7.15 illustrates this result with a section of a level sampled using the multi-layer MdMC approach annotated with the provided and actual paths through the level. We see there is a fair amount of overlap in
Figure 7.15: This figure shows a section of a level sampled with the multi-layer P2 MdMC that achieved a low Fréchet distance (4.123), along with the path provided to the sampler for that level (blue), the actual path taken by the $A^*$ agent through the completed level (red), and the overlap of the two paths (purple). Notice, there is a fair amount of overlap in the paths, and even when not overlapping, the paths do not stray far from each other.

Figure 7.16 shows a two-dimensional projection of the evaluated levels. Maps were projected based on a measure of distance between them using the t-SNE visualization algorithm. To determine the distance between two maps, we represented them as a histogram of high-level tiles, and computed the Euclidean distance between these histograms. High-level tiles were found by clustering $4 \times 4$ tile sections using $k$-medoids ($k = 30$) with all the training levels, 20 levels sampled by the single-layer approach, and 5 levels sampled by each of the multi-layer models. For $k$-medoids, we used a distance metric that compares the shape of the structures in the two sections and returns the difference between those shapes scaled by the overlap of the sections (more information about using $k$-medoids for clustering levels sections can be found in Chapter 4). We have highlighted training levels 1 and 2, because Multi-layer P1 and P2 (Multi P1 and Multi P2 in the figure), respectively, used the paths from these training levels in their player path layers. Notice that training level 2 is occluded by a level sampled by Multi-layer P2, and further that those training levels are surrounded by the levels sampled by P1 and P2. This shows that the multi-layer models are able to capture information about the training levels more accurately than the single-layer approach. Furthermore, there are only a few single-layer sampled levels near training levels 1 and 2, which shows that our multi-layer MdMC more accurately models these levels than the single-layer approach. Finally, the levels sampled by Multi-layer P4 (Multi P4 in the figure) are separated from the rest of the sampled levels, but still near several
Figure 7.16: This figure shows the two-dimensional projection of the training levels and sampled levels. We highlight the first two training levels because the paths through these levels were used as the provided player path layers for the first two multi-layer models. Notice, the two selected training levels are occluded by generated levels.

of the training levels indicating that if a new unseen path is provided to the multi-layerMdMC it is able to produce reasonable, but different levels.

Lastly, Figure 7.17 shows 2 randomly selected sampled levels for the single-layer MdMC and each of the multi-layer MdMC models, with the $A^*$ agent’s path annotated. Notice, the levels sampled using the multi-layer MdMC with the plateau-shaped path (Multi P3 in the figure) have very few structures at the beginning and end of the levels, but many in the middle. This allows the agent to follow a path close to the one provided. Similarly, the other levels sampled using the multi-layer MdMC models place structures near where jumps
Our most important result is that our multi-layer MdMC model is able to sample levels according to a provided player path, while still producing levels similar to the training levels. This shows that our multi-layer MdMC approach is a viable way to more accurately and deeply model video-game levels than similar single-layer models, opening the door for other representation layers to be developed and applied.

**Lode Runner Results**

Table 7.4 shows the results of sampling levels with both the multi-layer and single-layer MdMC models paired with the violation location resampling algorithm. Each “Multi” represents the 50 levels sampled using the path layer from one of the training levels. The first column (Average) is the average number of sections resampled counting only the sections resampled in levels that did not hit the limit of resampled sections. The second column (Std. Dev.) is the standard deviation of the number of sections resampled, again only counting...
Table 7.4: Multi-layer Level Evaluation (Lode Runner): This table shows the results of generating 500 levels in Lode Runner with the single layer approach and 50 with each of the multi-layer models paired with the constrained sampling approach. This table reports the average and standard deviation of the number of sections resampled and the percentage levels that go unfinished (based on a cut off during sampling).

<table>
<thead>
<tr>
<th>Model</th>
<th>Sections Resampled</th>
<th>Unfinished</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi P0</td>
<td>14.250</td>
<td>12%</td>
</tr>
<tr>
<td>Multi P1</td>
<td>23.267</td>
<td>10%</td>
</tr>
<tr>
<td>Multi P2</td>
<td>33.028</td>
<td>28%</td>
</tr>
<tr>
<td>Multi P3</td>
<td>20.281</td>
<td>36%</td>
</tr>
<tr>
<td>Multi P4</td>
<td>20.629</td>
<td>30%</td>
</tr>
<tr>
<td>Multi P5</td>
<td>45.677</td>
<td>38%</td>
</tr>
<tr>
<td>Multi P6</td>
<td>35.118</td>
<td>66%</td>
</tr>
<tr>
<td>Multi P7</td>
<td>30.452</td>
<td>38%</td>
</tr>
<tr>
<td>Multi P8</td>
<td>38.882</td>
<td>32%</td>
</tr>
<tr>
<td>Multi P9</td>
<td>13.625</td>
<td>4%</td>
</tr>
<tr>
<td>Multi All</td>
<td>18.334</td>
<td>29.4%</td>
</tr>
<tr>
<td>Single</td>
<td>17.359</td>
<td>27.0%</td>
</tr>
</tbody>
</table>

Figure 7.18: This figure shows the player path layer used with the levels generated with “Multi P6.”

completed levels. The final column (Unfinished) shows the percentage of levels that were abandoned during sampling because they resampled too many sections. Note, the values for “Multi All” are the weighted averages based on the number of completed levels for each path layer.

We see in Table 7.4 that the single and multi-layer approaches are both able to reliably generate playable levels, on average, using the VLR algorithm. In fact, there is no statistically significant difference between the average number of sections resampled (when \( p = 0.1 \), using a T-test), or between the average number of unfinished levels (when \( p = 0.1 \) with a chi-square test). However, despite their similar performance on
Figure 7.19: This figure shows a level sampled using the multi-layer approach (top) and a level sampled using the single-layer approach (bottom), both with the VLR algorithm. The levels have been annotated with solutions, where red X’s indicate ground that must be destroyed.

average, notice that the choice of path can have a large impact on the VLR’s ability to sample levels, as can be seen in the “Multi P6” row of the table. Figure 7.18 shows the “Multi P6” path. Notice it contains more groups of movements and dig actions together in the path than the path shown in Figure 7.12 (bottom-right), which is the “Multi P0” path, which makes it more difficult to replicate.

Figure 7.19 shows a level sampled using the multi-layer approach (top) and with the single-layer approach (bottom), both sampled with the VLR algorithm. We annotated a possible solution to each level. Note that red X’s represent sections of ground that need to be destroyed by the player to complete the level.

Table 7.5 shows the percentage of playable levels sampled with both the multi-layer and single-layer models paired with the standard sampling approach. As above, each “Multi” represents the 50 levels sampled using the path layer from one of the training levels, and “Multi All” represents the percentage playable over all the levels sampled with the multi-layer model. On average, we see the single-layer approach is able to sample significantly more playable levels (with $p = 0.1$, using a chi-square test). However, we see the performance of the multi-layer approach heavily depends upon the player path layer used during sampling. For instance,
Table 7.5: Multi-layer vs. Single-layer Standard Sampling Comparison: This table shows the results of generating 500 levels in *Lode Runner* with the single layer approach and 50 with each of the multi-layer models using the standard sampling approach. This table reports the percentage of playable levels generated using the single-layer and multi-layer models.

<table>
<thead>
<tr>
<th>Model</th>
<th>Percent Playable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi P0</td>
<td>30%</td>
</tr>
<tr>
<td>Multi P1</td>
<td>10%</td>
</tr>
<tr>
<td>Multi P2</td>
<td>18%</td>
</tr>
<tr>
<td>Multi P3</td>
<td>10%</td>
</tr>
<tr>
<td>Multi P4</td>
<td>20%</td>
</tr>
<tr>
<td>Multi P5</td>
<td>4%</td>
</tr>
<tr>
<td>Multi P6</td>
<td>6%</td>
</tr>
<tr>
<td>Multi P7</td>
<td>4%</td>
</tr>
<tr>
<td>Multi P8</td>
<td>6%</td>
</tr>
<tr>
<td>Multi P9</td>
<td>24%</td>
</tr>
<tr>
<td>Multi All</td>
<td>13.2%</td>
</tr>
<tr>
<td>Single</td>
<td>32.6%</td>
</tr>
</tbody>
</table>

Table 7.6: Multi-layer vs. Single-layer Gold on Path: This table shows the results of generating 500 levels in *Lode Runner* with the single layer approach and 50 with each of the multi-layer models. This table reports the percentage of gold pieces (i.e., objects that need to be collected to complete the level) are placed in positions that fall on the provided path with each of our models.

<table>
<thead>
<tr>
<th>Path</th>
<th>Single_{std}</th>
<th>Multi_{std}</th>
<th>Single_{VLR}</th>
<th>Multi_{VLR}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi P0</td>
<td>38.17%</td>
<td>98.15%</td>
<td>40.22%</td>
<td>31.28%</td>
</tr>
<tr>
<td>Multi P1</td>
<td>37.46%</td>
<td>100.00%</td>
<td>38.58%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Multi P2</td>
<td>31.29%</td>
<td>96.16%</td>
<td>33.45%</td>
<td>95.78%</td>
</tr>
<tr>
<td>Multi P3</td>
<td>39.23%</td>
<td>99.67%</td>
<td>39.45%</td>
<td>99.71%</td>
</tr>
<tr>
<td>Multi P4</td>
<td>39.67%</td>
<td>91.81%</td>
<td>37.99%</td>
<td>91.12%</td>
</tr>
<tr>
<td>Multi P5</td>
<td>46.06%</td>
<td>98.49%</td>
<td>46.95%</td>
<td>99.58%</td>
</tr>
<tr>
<td>Multi P6</td>
<td>35.71%</td>
<td>98.16%</td>
<td>35.68%</td>
<td>98.16%</td>
</tr>
<tr>
<td>Multi P7</td>
<td>40.56%</td>
<td>95.68%</td>
<td>41.71%</td>
<td>98.14%</td>
</tr>
<tr>
<td>Multi P8</td>
<td>21.81%</td>
<td>100.00%</td>
<td>23.11%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Multi P9</td>
<td>44.18%</td>
<td>97.43%</td>
<td>44.71%</td>
<td>97.49%</td>
</tr>
<tr>
<td>Avg.</td>
<td>37.42%</td>
<td>97.55%</td>
<td>38.19%</td>
<td>91.13%</td>
</tr>
</tbody>
</table>

when a path includes a lot of digging actions and clumped sections of movements (as in “Multi P6” in Figure 7.18) the model has a more difficult time capturing the structural patterns surrounding the movements, and thus creates fewer playable levels.

Table 7.6 shows the percentage of golds in the sampled levels that are placed on the provided player path. We believe this metric gives an approximation of how well the sampled levels adhere to the provided path, and therefore how much control providing a path gives the user. For the single-layer approaches we computed the average percentage of gold pieces placed on each of the paths to act as a baseline. Notice, that when using the multi-layered approaches an average of over 90% of the gold pieces appear on the provided
path, compared to the maximum of 46.06% with the single-layer baseline. This shows that by providing a player path layer, the multi-layer approach is able to sample a level with a solution similar to that provided path. A notable exception is with the VLR approach paired with P0. It is unclear why this particular path with multi-layer VLR approach performs so poorly, while it performs similar to other paths in the standard sampling approach, and in terms of the other metrics used.

Our results show that while the single-layer and multi-layer approaches perform similarly in terms of sampling time (sections resampled, levels finished) when using the VLR algorithm for generation, and the single layer approach even outperforms the multi-layer approach in terms of playability when using a standard sampling algorithm, the multi-layer approach provides the user with much more control over the type of levels created by the system, as evidenced by the placement of the gold pieces relative to the provided path. This suggests that the multi-layer approach can be of benefit when the user desires specific types of outputs from the system.

### 7.3.3 Multi-layer Level Representations in the Theoretical Framework

The multi-layer level representation we have been discussing in this section allows for the inclusion of additional, non-structural information. By adding this additional information to our level representation, it may appear that the multi-layer approach goes beyond what can be represented by our theoretical framework. However, in this section we show that by treating the positions in the additional layers as a node in the level graph, and connecting those added nodes with edges to the corresponding nodes in the level graph representing the structural information. In the remainder of this section we translate our multi-layer MdMC approach, and specifically the multi-layer level representation, into the terms of the theoretical framework introduced in Chapter 6.

**Data Representation**

We first need to define the multi-layer level representation using our frameworks definitions. That is, we need to define \( \langle T, L, l \rangle \).

- \( T \): The multi-layer MdMC uses a separate set of tile types for each layer of representation. Therefore,
$T = \langle T_1 \cup \ldots \cup T_n \rangle$ (i.e., the union of all tile types across all layers), where $n$ is the number of layers used in the level representation, and $T_i$ is the set of tiles used to represent the $i$th layer. Depending on the layer, the tile types may correspond to very different elements in the level. For example, in a structural layer the tile types can represent the various enemies and objects in the level, but in a player path layer there may be a tile type to indicate a position the player passed through and a tile type to indicate positions the player did not pass through.

- $L = \langle V, E \rangle$: $L$ requires that we define $V$ and $E$. In this model, each $v_i \in V$ represents a discretized position in the game level corresponding to one of the layers of representation. $E$ is defined by the network structure chosen for the multi-layer MdMC. That is, $L$ implicitly encodes the network structure into the graph structure of $L$. Figure 7.20 shows an example network structure for the multi-layer MdMC where there are three total layers: a structural layer, a height layer, and a player path layer. Figure 7.21 (right) shows the graph representation of a level section using the network structure in Figure 7.20 to define the edges.

- $l$: For this model, $l$ assigns the tile type of each vertex according to what occurs at that position in the level and what layer the vertex belongs to.

**Model Definition**

We now define the probability distribution, $P$, of the multi-layer model. Recall, $P$ for our standard MdMC approach is defined as

$$P(v_i = t_j|\text{neighbors}(v_i)),$$
Figure 7.21: A section of a Super Mario Bros. level (top-left), three layers represented using tiles (structural, player path and height), and the three layers represented using our graph representation (right). The edges are given by the network structure defined in Figure 7.20.

where neighbors(v_i) is defined as the set of all vertices connected to v_i by an incoming edge. For the multi-layer MdMC the definition of P remains the same, because of our definition of the level representation graph the tile type of the current vertex still depends on the tile types of the neighboring vertices. The neighboring vertices now include vertices representing positions in the other layers, as can be seen in Figure 7.21 (right).

Sampling

We next define the parameters and helper functions of the SequentialLevelSampling function. P and T are defined the same as for the model definition.

- L_{initial}: The multi-layer MdMC approach requires that all but 1 of the layers of the level representation are defined before sampling. We define the initial level graph as having two nodes labeled with the sentinel tile types to signify the left and bottom border of the level; we also need to define the vertices corresponding to the layers of the level that we are not sampling. Specifically, for each vertex that we plan to sample, we need to define a vertex in the initial level graph for each layer of representation, and label each of these vertices with a tile type corresponding to that layer. Figure 7.22 (1) shows an example initial level graph with 2 x 2 tiles needing to be sampled, and with 3 total layers (2 of which
Figure 7.22: This figure shows an example of an initial level graph (1) used to start sampling with the multi-layer MdMC approach with 3 total layers, and subsequent vertices and labels sampled (2-5) for a level with $2 \times 2$ tiles corresponding to positions in the output level. Notice that many other initial level graphs can be used for a given level domain and model, but we use this initial level graph for ease of visualization and explanation.

do not get sampled). Notice that we use two sentinel tiles for this initial level graph, but that is only for ease of explanation and visualization. The initial level graph can be defined in many different ways depending on the domain, the desired level generation approach, the configuration of the models, and user preference.

- StoppingCriteria($L$): The stopping criterion for this approach checks whether the level graph is of the desired size. Specifically, StoppingCriteria($L$) compares the number of vertices in $L$ to the desired dimensions of the generated level and returns true (stop) if the number of vertices labeled with tile types corresponding to the sampled layer in $L$ is equal to the desired height $\times$ width.

- SampleVertex($L$): This function determines where the next vertex should be placed in order to grow the level graph. For this approach the next vertex to be sampled must be placed in such a way that it has the necessary neighbors as defined by the network structure. For this approach, we typically sample row by row from the bottom up. However, vertices can be added in any order provided they satisfy the network structure. Figure 7.22 (2-5) illustrates one way that vertices could be added to the initial level graph.

- SampleTile($P, T, L$): This function probabilistically determines the tile type of any unlabeled vertices in $L$ using the trained probability distribution, $P$. For this approach, the tile type is chosen probabilistically based on the neighbors of the vertex being labeled.

After representing this multi-layer approach in terms of the theoretical framework, we can see the key difference between the multi-layer approach and the standard MdMC approach. Namely, the definition of
the initial level graph must include nodes corresponding to the components of the other layers. Once the level graphs are updated to include the additional layers’ nodes, the training and sampling procedures function the same as for the standard MdMC approach. Additionally, we can see that because the multi-layer level representation does not affect the training or sampling procedures, it can be incorporated into the other approaches described by our framework (e.g., Summerville and Matteas’s LSTM approach\textsuperscript{7} or Guzdial and Riedl’s probabilistic graph approach\textsuperscript{8}).

7.3.4 Conclusions

In this section we present a multi-layer level representation to be used with machine learning-based procedural level generation systems, and show that this multi-layer level representation can easily be incorporated into the level graph representation defined in our framework. By adding additional layers of representations outside of the standard structural layer, we aim to build more representative and accurate models of the chosen domains. We test this approach by encoding a player path layer and a section layer for the test domains of Super Mario Bros. and Lode Runner.

The experiments show that in simpler domains (Super Mario Bros.), employing a multi-layer representation and learning approach allows for models that are able to capture interesting player interactions and gameplay information. This is shown by the more accurate placement of special objects (springs) that allow for movements required to complete particular sampled levels. In complex (Lode Runner) and simple domains, employing this multi-layer approach and providing a desired path during sampling gives the user more control over the type of sampled levels. This can be seen in the Fréchet distance results and the gold on path results for Lode Runner.

This work shows that leveraging multi-layer representations can allow for the training of more accurate level models. We can extend this approach in several ways. For instance, we can explore additional layers that we believe may provide useful information to the models (such as, difficulty curve layers). We can try applying these approaches in more domains to prove its generality. Lastly, we saw by representing the multi-layer approach in the theoretical framework that it should be applicable across all the models represented in the framework, so it would be interesting to test this representation with other models to see if it offers similar
improvements to other models.

7.4 Discussion

In this chapter we explored several extensions to machine learning-based level generation approaches that address key weaknesses of many such approaches. Specifically, we introduced a constrained sampling approach in order address the problem of controllability in PCG systems; we described a player model-guided level generation approach to address the issue of tailoring content to players and incorporating more than structural information into the models; and we explored a multi-layer level representation to address the problem of providing more design information to the models during training.

We found that our constrained sampling approach gave the user more control over the properties of the generated levels when used in a domain that has previously been successfully modeled, and in a domain where the standard MdMC model struggled the constrained sampling approach allowed it to reliably generate usable levels. We found that our player model-guided generation approach was able to generate levels that allowed for paths that were likely, given the player model. Additionally, by separating our player model from the level model, we were able to investigate the player model more carefully and understand the results more easily.

Lastly, the multi-layer level representation allowed us to incorporate more than just structural information into our level generation models. We found that by providing a player path, we can generate levels that allow for similar paths in multiple domains. We also saw that incorporating path information allowed our models to capture interesting gameplay features (e.g., spring mechanics in Super Mario Bros.).

The extensions presented in this chapter provide more control to the user and more information to the models. Though we have only tested these extensions with our multi-dimensional Markov chain approach, they are generally applicable. For instance, we showed that the multi-layer approach can be used with any training approach represented that is able to be described by our framework. Similarly for the constrained sampling approaches. The player movement model we experimented with was implemented as a constraint given to the constrained sampling approach, so it too is generally applicable. In the future, we would like to explore the application of these extensions to other models. We are also interested in developing further extensions to the class of machine learning-based level generation approaches. For example, we currently
only experiment with 2-D training levels, but nothing in the framework prohibits 3-D levels. Additionally, the constrained sampling approach has only been used with sequentially sampling approaches; developing a constrained approach that can work with non-sequential sampling approaches, such as Markov random fields, would be an interesting future line.
Chapter 8: Experiments with Training Data

Machine learning-based PCG systems all rely on training data in order to define their models. When experimenting with these approaches authors typically use all the training data that is available to them (or as much as they can reasonably create) without consideration of if that amount of training data is appropriate for the given model or domain. Specifically, while there has been considerable work in evaluating generated content (which can be seen in Section 2.7), there has not been work in evaluating the quality of training data provided to machine learning-based PCG approaches or how the quality of the training data may affect the generated content. Additionally, there has not been work in evaluating the effects of varying quality or quantities of training data on these approaches, or solutions for if there is not enough training data for a given model.

In this chapter we aim to address the above issues. In order to determine how the quality of training levels and the amount of available training data affects machine learning-based level generation approaches, we devise training data evaluation metrics, and experiment by training two such models with varying amounts and quality of training levels. We investigate how the expressiveness of the models change with the different sets of training data, and how much training data (and of what quality) is required to reduce the amount each model plagiarizes from the training levels. Afterwards, we explore a possible solution to the scenario where not enough training data is available in the desired domain. We introduce a domain transfer approach that aims to supplement a small set of training levels with training levels from a similar domain. We investigate how different supplemental domains affect the trained models and the output of those models.

The remainder of this chapter is organized as follows: first, we explore training data effects experiments in order to determine how the quantity and quality of training data influences machine learning-based PCG approaches in Section 8.1; we then introduce our domain transfer approach as a means of supplementing a domain with too little training data in Section 8.2; we close in Section 8.3 by discussing our findings.
8.1 Effects of Training Data

Acknowledgements

We would like to note that the work presented in this section was performed jointly with Adam Summerville. Adam Summerville was instrumental in testing this work with his LSTM approach, and in the evaluation in using both approaches.

Machine learning-based level generation approaches all rely on training data in order to define their models. However, there has been considerable work in evaluating level generators and generated content (as seen in Section 2.7), there has been very little work in evaluating the quality of training levels for machine learning-based level generation approaches, and the effects of training with varying amounts and varying quality of training levels. It is important to understand these effects as they can help to delineate models based on how much training data they require and how robust they are in the presence of low quality training data. Furthermore, once these effects are understood for a variety of models, it can help to indicate which models are appropriate given the amount of training data available, or if in a new domain, how much training levels (and of what quality) needs to be gathered for the chosen model to perform well.

In the field of machine learning-based level generation almost no research has explored the effects of training data on the models and output. A minor exception is our brief discussion on trying to guide the MdMC approach towards specific types of output by carefully selecting the training data and in my MdMC experiments in Section 3.3 where we briefly tested using fewer levels when training the model for Lode Runner, but we did not perform an in-depth analysis of the effects of varying amounts and quality of training data. Notice that this was a very minor foray into a topic that deserves much more attention. There has been a lot of work in exploring training data effects on machine learning models in other domains. For more information on the relationship between training data and performance, Dietterich provides a nice summary.

In order to determine the effects of varying amounts and quality of training data on machine learning-
based level generation approaches, we first define a metric for training levels that is meant to approximate the quality of a level in terms of how much information it provides to the model. Next, we evaluate the training data with the defined metric, and define a number of training data sets ranging in size and in quality. We then train two machine learning-based level generation approaches on each of the training data sets, and evaluate the levels generated with each of the trained model with an emphasis on the expressive volume of the generator as well as how much the model plagiarizes from the training data.

In more detail, the remainder of this chapter is organized as follows: first, we describe the two machine learning-based level generation approaches for which we are exploring the training data effects in Section 8.1.1; then, we describe our experimental set-up including our chosen domain, metrics, training sets, as well as our results in Section 8.1.2; finally, in Section 8.1.3 we draw our conclusions.

### 8.1.1 Level Generation Models

In this section we describe the two machine learning-based level generation techniques we use to explore the effects of training data. In addition to our MdMC approach, we chose to include the Long Short-term Memory Recurrent Neural Network approach of Summerville and Matteas in the evaluation. We chose this approach because it uses a similar tile set to our MdMC approach and has been shown to produce high quality content.

**Constrained Multi-dimensional Markov Chains for Level Generation**

For the MdMC, we use the constrained MdMC sampling approach described in Section 7.1. Specifically, we use the Violation Location Resampling algorithm for sampling, and the standard training approach of estimating the distribution of tiles from the observances in the training levels. This sampling algorithm enforces a set of constraints on the generated levels while sampling the level using the trained distribution. For the experiments in this chapter, we provide the sampling algorithm with a playability constraint, which ensures that all the levels generated are playable, and a constraint that disallows any malformed structures (i.e., combinations of tile types that are illegal in the given domain).
Long Short-term Memory Recurrent Neural Networks for Level Generation

Recurrent Neural Networks (RNNs) operate in a manner similar to standard neural networks (i.e., they are trained on data and errors are back-propagated to learn weight vectors). However, in an RNN the edge vectors are not just connected from input to hidden layers to output, they are also connected from a node to itself across time. This means that back-propagation occurs not just between different nodes, but also across time steps.

LSTMs are a neural network topology first proposed by Hochreiter and Shmidhuber\(^\text{124}\) for the purpose of eliminating the vanishing gradient problem found in RNNs. LSTMs mitigate the problem via nodes that act as a memory mechanism, telling the network when to remember and when to forget. The LSTM architecture can be seen in Figure 8.1 (used with the permission of the authors\(^7\)).

Torch\(^7\)\(^\text{125}\) was used to train the networks in the experiments, based on code from Andrej Karpathy\(^\text{126}\) using previously optimized parameters\(^7\). The model was trained on sequences of 200 tiles at a time, in a network with 512 LSTM cells per layer and 3 layers. To fight overfitting, dropout was aggressively used, with 80% of LSTM cells being dropped at each training instance.

Following the work from Summerville and Mateas\(^7\) the LSTM used a “Snaking” path (it starts from the bottom left, goes bottom-to-top, flips directions going top-to-bottom, flips, etc.) and “Depth” information (a special meta-tile is inserted at the top of a column once per each ten columns into the level).

To prime the network to begin generation, an input seed is passed in with 3 empty columns with a single ground tile at the bottom. The generator is then sampled until an end-of-level termination character is found, with each newly sampled tile being used as the input for the next step in the auto-regression process. Note, no constraints are enforced when sampling with the LSTM.

**8.1.2 Experimental Evaluation**

This section first describes the domain used for experimentation, how to assess the quality of training data, the experimental set-up, the evaluation metrics, and finally the results.
Figure 8.1: Graphical depiction of an LSTM block (reproduced with permission from the authors).

Domain

We perform the experiments using *Super Mario Bros.* as the domain. We chose *SMB* because of its wide use in the field of level generation. Its common use allows us to leverage previous work on evaluation metrics, and also makes the effects of the training data easier to understand. More information on how we represent this domain can be found in Appendix A.1.

Training Data Quality

In addition to evaluating the performance of the two chosen methods when using varying amounts of training data, we also evaluate the methods using training data of varying “quality.”

“Quality” is a subjective term, thus, we consider levels that are more uniform in structure to be of “lesser quality” than levels with move variety, as it follows that more uniform levels would provide less new information to a model. We approximate this quality by computing the entropy of the training levels through their high-level structures. That is, we split the training levels into $4 \times 4$ tile sections. We then perform $k$-medoids on those sections with $k = 30$. For the $k$-medoids distance metric, we find the positioning of two sections that yields the most overlap in tile types between the sections, and weight that by the area of the overlapping sections. The idea is that this metric provides a measure of how structurally similar two sections are. Once the
clusters are computed, we represent each level as a histogram containing the number of $4 \times 4$ level sections belonging to each cluster. Finally, we compute the entropy of those histograms$^{127}$, and assume that a higher entropy corresponds to more information in that level, and thus a higher quality training level.

**Training Data Sets**

In order to evaluate the effects of both the quality and quantity of training data on the chosen models, we create several sets of training data. We first order the training levels from most to least entropic. We then train separate models using the first 16 columns of the most entropic level, using the first 32 columns, using the first 64 columns, using the first 128 columns, using the most entropic level in its entirety, using the two most entropic levels, etc. We repeat this process using the least to most entropic ordering of the levels. In total, we train 66 MdMC models and 66 LSTM models.

**Evaluation Metrics**

We evaluate the levels sampled by our systems using both standard level evaluation metrics and metrics that explore the expressiveness of the systems given the training data.

- **Linearity**: This measures how well the platforms in the level are approximated with a best fit line$^{83}$. It returns the sum of distances of each solid tile type (i.e., not empty, not enemies) from the best-fit line, normalized by the level length.

- **Leniency**: This approximates the difficulty of the level by summing the gaps (weighted by length) and enemies (weighted by 0.5), and normalizing by the level length$^{83}$.

- **Enemy Sparsity**: This measures the horizontal spread of enemies through the level by taking the average distance of enemies from the average of enemy $x$ positions in the level$^{85}$. A large Enemy Sparsity value means enemies are scattered around the level, whereas a low value means enemies are grouped together.

- **Kernel Density Estimation**: The expressive range$^{83}$ of a generator has typically been thought of as a visualization of the metric space covered by the generated content. Most commonly, this has been
visualized as a heat map in 2 dimensions (linearity and leniency, classically\(^{128}\)) although some have
done up to 8 dimensions (2 at a time)\(^7\). However, in the literature it is common to refer to the “width” of
the expressive range, but this has yet to be done in anything beyond a qualitative visual assessment\(^{14}\).
From this point on we use \textit{volume} as the measure we care about; width is problematic as it is a linear
dimension. For instance, a generator that always produced perfectly linear levels that had a very wide
range in leniency would still be unlikely to be thought of as very expressive, given that it is completely
lacking in 1 dimension. To this end, we consider the \(n\)−dimensional volume (e.g., area in 2-D, standard
volume in 3-D, etc.) of the generated metric space to be the \textit{size} of the expressive range.
To calculate this volume, we use Kernel Density Estimation (KDE) as calculated by the “ks” R pack-
age\(^{129}\). KDE determines a non-parametric function of the density of a sampled space, similar to the
binning process of a histogram, but typically smooth given the use of a Gaussian kernel. Figure \ref{fig:8.2}
shows the calculated Linearity and Leniency for the 1000 levels generated by MdMC Most-to-least 29
Level Generator as grey circles. The density estimate is visualized by black contour lines. We then
threshold this density estimate for points greater than 0, which we take to be the boundaries of the
expressive range. We then form an \(n\)−dimensional grid, and count the number of bins that lie within
the expressive \(n\)−volume, multiplying the count with the volume of a single bin. For the experiments
we compute the expressive volume of the models using linearity, leniency, and enemy sparsity for the
density estimation.

• \textbf{Plagiarism:} This measures the percentage of an output level that is directly copied from the training
levels. We compute this by first splitting the levels in the training set into overlapping sections of \(n\)
columns and removing any duplicates. We call this set of level sections \(T_n\). We then split an output
level into overlapping sections of \(n\) columns, but do not remove duplicates. We call this series of level
sections \(L_n\). We compute the plagiarism of a level by counting how many \(l \in L_n\) are also in \(T_n\).
We then determine the number of columns from the output level that make up the sections that are
plagiarized (accounting for overlapping columns). The value returned is the percentage of columns
plagiarized in the output level. Notice, we do not simply count how many individual columns are
plagiarized directly because we are interested in seeing how large the sections of plagiarism are as
well as how much of the level is plagiarized. For example, given two levels, one of which has 50% of columns plagiarized with $n = 4$, and the second of which also has 50% of columns plagiarized, but with $n = 20$, then we consider the second to be more plagiarized than the first level, because of the large amount of continuous plagiarism (i.e., a section of 20 columns copied directly from the training data is considered worse than 5 separate 4 column sections).
Results

Figure 8.3 shows the results of the expressive volume calculations. Notice, what we care about here are the ratios between the expressive volumes of the models, as the actual volume scales will vary for different domains. In general, the figure shows that a small amount of data (< 3 levels) results in a very small expressive volume, which is expected given that there isn’t much variation in the supplied data. A notable exception is the MdMC when using the most to least ordering of levels. This model’s expressive volume levels off after only 1 level. Surprisingly, additional data does not increase the expressive volume of any of the models after about 5 training levels. In the range with sufficient data (> 4 levels) the LSTMs generally have a larger expressive volume (27% greater), but all generators have a much smaller expressive volume than that of the original levels, which is 50% larger than the next closest generator (LSTM Most-to-least 8 levels). This larger volume is present in all of the individual metrics, meaning it is not just a failing in any one particular aspect. Furthermore, note that after reaching 5 levels worth of data, the information density of those levels does not effect the expressive volume of the models. Perhaps, this is due to the fact that the variety found in multiple levels, even those of relatively low information content, exceeds that of any one level. Of course, there are certainly degenerate counter-examples (e.g., 5 empty levels would provide no worthwhile information), but in any reasonable practical application it is more important to acquire a sufficient amount of data.

Figure 8.4 shows how much the MdMC and LSTM models trained with various amounts of training data plagiarize from the training levels. We also display the plagiarism results between the training levels. We compute this by treating each level individually as the output level, and treating the remaining 28 training levels as the training data. Then, for each level we compute the plagiarism of that one level against the other 28 levels. We do this for each training level, and average the values.

An interesting result when using the most to least order (top-left and bottom-left) is that the percentage of plagiarism and the size of plagiarized sections increases with the amount of training data. We believe this is due to the fact that as the amount of training data increases, the number of common structures increases, which makes it more likely for something that is sampled to be present in the training data. Alternatively, when using the least to most ordering (top-right and bottom-right), there are more mixed results with fewer
Figure 8.4: Plots of the plagiarism metric for both the MdMC and LSTM models with increasing amounts of training data, as well as a measure of plagiarism between training levels. This shows that the LSTM tends to sample levels with higher percentages of plagiarized columns. However, the MdMC and LSTM both plagiarize similarly sized sections, when trained with more levels. Notice, that the MdMC plagiarizes slightly less than the training levels plagiarize from each other in terms of section size and percentage, while the LSTM plagiarizes at slightly higher percentages, and around the same section sizes.

training levels resulting in a higher percentage of plagiarism. We believe this is due to the simplicity of those few training levels that are being used. That is, because there are so few differing structures in the initial training levels, the models are likely to copy those few simple structures.

When comparing between the MdMC and LSTM approaches, the LSTM tends to plagiarize a higher percentage of columns and larger sections than the MdMC approach. This is to be expected as the LSTM considers a larger context when generating, which leads to learning of larger structures (and thus more plagiarism of such structures). This is exemplified when only one training level is used, and upwards of 150 column sections are plagiarized from the given training level, due to the ability of the LSTM to nearly memorize the entire level. However, when only 5 training levels are used, the plagiarism decreases drastically from the one level model.

Furthermore, the MdMC when trained with all the available training levels (29), plagiarizes a lower percentage of columns from the training data than the training levels plagiarize from each other, while the
LSTM when trained with all levels plagiarizes a higher percentage. Notice, both models plagiarize around the same maximum section size in this case. While the LSTM trained with all levels does plagiarize a higher percentage than the original levels, nearly all trained models (except for LSTM 1 level and MdMC Least-to-Most 16 Columns) are roughly comparable or below the plagiarism of the original to themselves, indicating that neither approach suffers greatly from plagiarism when a sufficient amount of data present.

From the plagiarism and volume estimate results, it can be seen that using all of the available training data is not necessary, and in some cases may even hinder the result. Figure 8.3 shows that the MdMC’s expressiveness levels off around 6 levels with the least to most ordering and after only 1 level with the most to least ordering, while the LSTM expressiveness levels off after around 10 training levels in both cases. Additionally, the expressiveness of the LSTM models doesn’t change based on the ordering of the training levels, while it does for MdMCs for small amounts of training data. Finally, training with more information dense levels can reduce the amount the models plagiarize from the training data.

8.1.3 Conclusions

In this section we explored the effects of different quality and quantities of training data on two machine learning-based level generation models. The goal is to better understand how different approaches respond to different types of training data. Gaining a better understanding of how machine learning-based approaches handle differing amounts and quality of training data allows for a more informed application and comparison of the various approaches.

Through the experiments, we found that machine learning-based level generation approaches may require less training data than has been previously used. These results have important implications for the practical application of machine learning-based approaches. Specifically, these results show that these machine learning-based approaches are feasible in domains where there is very little training data, or in domains where training data needs to be created (i.e., games that are currently in development and would like to leverage PCG for the levels). Furthermore, the results show that if a large amount of training data is available, it may be beneficial to selectively choose the training data (this echoes our findings in the domain of Lode Runner in Section 3.3). This line of research could be further pursued in several ways. For instance, additional
training level quality metrics can be devised and tested; additional domains can be investigated, especially more complex domains where accurately learning the structures may be more difficult; and finally, exploring these concepts with other machine learning-based level generation approaches.

Now that we have developed a way of determining how much training data is required for a given domain and model, it is important to explore what we can do if a model requires more training data than is available. In the following section we describe a domain transfer approach that is meant to create supplemental training data by converting levels from another domain into training data for a target domain.
8.2 Domain Transfer

Machine learning-based procedural level generation approaches rely on a set of training data in order to define their models. In domains where sufficient amounts of content are readily available this is not an issue, but sufficient training sets may not always be available. This can be due to the game domain being new or in development, or simply that the game only has a small number of levels. In these cases, in order for machine learning-based PCG methods to be feasible, supplemental training data must be acquired. There are several approaches for supplementing training data in the machine learning literature, such as transforming the training data (e.g., rotations) in image classification tasks or augmenting with additional information. In this chapter we present a domain transfer approach for supplementing a training set.

Domain transfer is the adaptation of knowledge, data, or models from one domain to be of use in or supplement a related target domain. Domain transfer techniques (and domain adaptation, computational analogy, and related techniques) have been explored in the context of cognitive simulation, where concepts are modeled with predicates and objects and analogies between concepts are found via a search over the mapping of predicates and objects. More recently, domain transfer has been used to supplement the training of classifiers and for transferring textures and styles between images via convolutional neural nets. We are interested in developing a domain transfer technique that allows a machine learning-based level generator to supplement its training data using out-of-domain training data.

The remainder of the chapter is organized as follows: first, we present our approach for transferring training levels from one domain to another using a tile mapping approach in Section 8.2.1; next, we discuss our experiments, including the set-up and the results in Section 8.2.2; and lastly, we draw conclusions about our domain transfer approach and its applicability in Section 8.2.3.

8.2.1 Domain Transfer

Machine learning-based level generation approaches are able to sample levels for a target domain when provided with training levels. However, there is no guarantee that the amount of training data available will be enough to train an accurate model. This can result in low quality output levels and a lack of diversity in the
levels sampled with the model. To address a lack of training data, we propose an approach that can convert levels from other domains into training levels for the target domain. At a high level, this domain transfer approach works by finding a correspondences between the tile types used to represent the different domains, then using those correspondences to define a tile type mapping from one domain to another.

**Tile Mappings**

To use levels from one domain to train a model for another domain, the out-of-domain levels need to be translated to the same representation as the in-domain levels. That is, the out-of-domain levels, which use a set of tile types, $T_{out}$, need to use the same set of tile types as the in-domain levels, $T_{in}$. This is achieved via a *tile mapping*. A tile mapping is a function that takes an out-of-domain tile type and returns an in-domain tile type. Formally a tile mapping is defined as a function $m : T_{out}[i] \rightarrow T_{in}[j]$.

Finding an appropriate tile mapping is non-trivial. For example, while empty space tiles are similar in many domains, *Kid Icarus* levels contain door tiles which do not have a direct equivalent in the *Super Mario Bros.* levels we use for training. We propose an approach which automatically defines tile mappings using multi-staged filtering which initially considers the set of all possible mappings that can be defined, and removes undesirable tile mappings with different sieves at each stage.

**Searching for Tile Mappings**

A sketch of the approach for finding tile mappings from a source to target domain can be seen in Figure 8.5. Our approach takes the number of desired mappings, $d$, a threshold, $f$, for the second filter (Jaro-Winkler), a (possibly empty) set of manual constraints on the tile mappings, a set of in-domain levels, $L_{in}$, and a set of out-of-domain levels, $L_{out}$. Our approach returns a set, $M_3$, of $d$ tile mappings, satisfying the provided constraints.

1. **Filter 1: Manual Constraints.** The first filter in this approach allows the user to define a set of constraints (e.g., “empty tiles should map to empty tiles”, “no more than two out-of-domain tiles should be mapped to the same in-domain-tile”) that she believes should be satisfied in all mappings. Mappings that do not satisfy these constraints are removed from the search space. Note that the user may choose
Figure 8.5: The three stage filtering process that we employ to find appropriate tile mappings. We start with the full set of tile mappings, \( M_0 \), and reduce this set to the final set of mappings, \( M_3 \), by applying three consecutive filters.

1. **Filter 1: Manual Constraints.**

To not define any constraints, in which case the entire search space will be explored. This step allows the user some control over the types of mappings explored.

2. **Filter 2: Jaro-Winkler Distance.**

The second filter in this approach automatically reduces the set of tile mappings by comparing the relative frequencies of the tiles in the out-of-domain levels once translated with a mapping, \( m \), to the frequencies of the tiles in the in-domain training levels, and removing those mappings that result in tile frequencies too different from those in the in-domain levels. The idea here is to remove mappings that, intuitively, will lead to poor mappings (e.g., mapping empty tiles to solid tiles. Specifically, for each remaining mapping, \( m \), in the search space, the levels in \( L_{out} \) are converted to in-domain levels using \( m \). Then the tile types are ordered by frequency of appearance in the translated levels, generating a string with each of the tile types in order. This filter then compares this order with the order resulting from sorting the tile types according to their frequency in the in-domain levels using the Jaro-Winkler distance, a distance measure between strings. Each tile mapping whose Jaro-Winkler distance to the in-domain tile ordering is not below a threshold, \( f \), is discarded from the search space.

3. **Filter 3: Unseen Configurations.**

The final filter in this approach, filters all but the top \( d \) mappings from the current set of mappings by considering the number of unseen configurations in the translated
out-of-domain levels. An unseen configuration is a combination of tiles that was not observed in the in-domain levels. In order to determine the number of unseen configurations, a window of tiles that makes up a configuration must be defined (e.g., $2 \times 2$, $3 \times 3$). Next, for each configuration appearing in the translated out-of-domain levels, the filter counts the number of times it appears in the in-domain levels. The $d$ mappings with the lowest number of unseen configurations are kept, and returned as the output of this filter. Intuitively, this step trims the search space by removing mappings that have structures that vary greatly from the in-domain levels.

8.2.2 Experiments

In order to test this domain transfer approach, we consider the following scenario: we assume that a single in-domain level is available for training, but a large number of out-of-domain levels are available. We use Super Mario Bros. as the target domain, and two games as out-of-domain sources: Kid Kool, which is similar to the target domain, and Kid Icarus, which is less similar to the target domain. The technique described in this section is used to translate the out-of-domain levels to in-domain levels. We then use the single in-domain level and the set of translated levels to train an MdMC model that can then be used to sample new levels using the constrained level sampling approach, Violation Location Resampling (VLR) algorithm, described in Section 7.1. By choosing one similar and one different domain, we are able to explore the varying effects of out-of-domain training data on the model. The remainder of this section describes these domains, experimental set-up, and reports the obtained results.

Super Mario Bros.

In these experiments we use a simplified set of 9 tile types to represent the Super Mario Bros. domain (instead of the 36 described in Appendix A.1): $S$ is a sentinel tile, denoting the borders of a level; $G$ represents solid tiles, such as the ground and unbreakable blocks; $B$ represents breakable blocks; $?$ represents power-up and coin blocks; $p$ represents the left section of a pipe; $P$ represents the right section of a pipe; $C$ represents a bullet bill cannon; $X$ represents an enemy; and $E$ represents empty space. We use the first level in the training set that contains all of the above tile types as the only in-domain training level.
Figure 8.6: This figure shows a section of a Kid Kool level. Notice, that levels in this domain are much taller than those in Super Mario Bros., and that they are in fact broken into several height-based sections in game (denoted by the black horizontal line).

Kid Kool

Kid Kool is a platforming game with linear levels, though many levels have multiple sections separated by height (i.e., a sky section that is reachably via platforms, a ground section, and an underground section reachable by some holes in the ground section). Figure 8.6 shows a portion of a Kid Kool level. Notice the black horizontal line in the center of the level which divides it into two sections. The training set contains 12 levels from Kid Kool and the Quest for the Seven Wonder Herbs. These levels are represented using 12 tile types: $S$ is a sentinel tile; $G$ represents solid tiles; $b$ represents collapsing bridges; $B$ represents air cannons that blow the player in various directions; $M$ represents a spring that allows the player to jump higher when jumped on; $H$ represents a tube that can be entered which transports the player to the other end of the same tube; $I$ represents a pole that the player can use to launch themselves forward; $W$ represents water, which the player can slide across if running, but otherwise is fatal; $T$ represents treasure, which can be collected for points; $c$ represents cannons which launch bouncing cannonballs; $X$ represents enemies; and $E$ represents empty space.

Kid Icarus

The levels chosen in this domain are structurally different from the levels in Super Mario Bros. and Kid Kool because of their vertical orientation. Information on the tile types used to represent this domain can be found
Experimental Setup

In order to evaluate this approach, we compare the tile mappings found using this approach against tile mappings found with other methods. The sets of tile mappings compared are defined below and can be found in Table 8.1.

- **$R^1$**: chosen at random from the set of tile mappings that satisfy the manual constraints (i.e., from the set of tile mappings after applying the first filter). We test three such mappings for Kid Icarus and Kid Kool. They are denoted: $R^1_1$, $R^1_2$, and $R^1_3$.

- **$R^2$**: chosen at random from the set of tile mappings remaining after the second filter is applied. We test three such mappings for Kid Icarus and Kid Kool. They are denoted: $R^2_1$, $R^2_2$, and $R^2_3$.

- **$M$**: defined manually based on our knowledge of the domains. We defined three such mappings for Kid Icarus and Kid Kool. They are denoted: $M_1$, $M_2$, and $M_3$.

- **$B$**: found using the proposed domain transfer approach with the manual constraints defined below, $d = 3$, and $f$ set to the lowest computed Jaro-Winkler distance among mappings for each domain. Notice, many mappings in the experiments had the same Jaro-Winkler distance. We assume the existence of a single in-domain training map for this process. These mappings are denoted: $B_1$, $B_2$, and $B_3$.

As mentioned previously, this approach allows the user to define constraints for the first filtering. Below we describe the sets of constraints used in our experiments:

- **Kid Kool**:
  1. The sentinel tile, $S$, must map to the Super Mario Bros. sentinel tile, $S$.
  2. The empty tile, $E$, must map to the empty tile, $E$.
  3. The solid tile, $G$, must map to the solid tile, $G$.
  4. The enemy tile, $X$, must map to the enemy tile, $X$.

- **Kid Icarus**:
Table 8.1: Tile Mappings: This table shows the different tile mappings found by our various approaches.

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Kid Kool</th>
<th>Kid Icarus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>SEGXB</td>
<td>BMHWTcI</td>
</tr>
<tr>
<td>R₂</td>
<td>SEGXBXGEP?Pp</td>
<td>SEBGppp</td>
</tr>
<tr>
<td>R₃</td>
<td>SEGXpXcG?XP</td>
<td>SEpBc</td>
</tr>
<tr>
<td>R₄</td>
<td>SEGX?cX?GBEE</td>
<td>SE?EBc</td>
</tr>
<tr>
<td>R₅</td>
<td>SEGX??XGBEEE</td>
<td>SE?PEG</td>
</tr>
<tr>
<td>R₆</td>
<td>SEGXBB??GBEE</td>
<td>SEcGB?X</td>
</tr>
<tr>
<td>B₁</td>
<td>SEGXEGEBG?E</td>
<td>SEpBXcG</td>
</tr>
<tr>
<td>B₂</td>
<td>SEGXGGBEBG?E</td>
<td>SEPBXcG</td>
</tr>
<tr>
<td>B₃</td>
<td>SEGXGEBEBG?E</td>
<td>SE?BcG</td>
</tr>
<tr>
<td>M₁</td>
<td>SEGX?BqBBEc</td>
<td>SEGGBc</td>
</tr>
<tr>
<td>M₂</td>
<td>SEGXG?pBE?c</td>
<td>SEBBX?G</td>
</tr>
<tr>
<td>M₃</td>
<td>SEGXGB?PEE?c</td>
<td>SEBBXc</td>
</tr>
</tbody>
</table>

1. The sentinel tile, S, must map to the Super Mario Bros. sentinel tile, S.

2. The empty tile, E, must map to the empty tile, E.

To evaluate this approach, we configure an MdMC with 12 row splits and a look-ahead of 3 and train it using the converted levels and only one training level from Super Mario Bros. For the one in-domain level, we chose the first level in the training set that contained all of the tile types. Once trained, we sampled 100 levels of size 12 × 210 (height × width) using the VLR algorithm with a single constraint to ensure playable levels.

We recorded average linearity and leniency of sampled levels\(^ {83} \), and average log-likelihood of sampled levels (with a Laplacian smoothed MdMC distribution trained on the single in-domain level).

Results

Table 8.2 shows the results of the experiments. The top three rows show the metrics applied to the single training level from the Super Mario Bros. training set used, 16 levels from the Super Mario Bros. training set that can be represented using the simplified set of tile types, and levels sampled using an MdMC (paired with the VLR sampling algorithm ) trained with the single training level. We highlight the evaluation scores most closely matching the set of 16 training levels for both domains.

The table shows manual mappings and those found by the domain transfer approach (M₁ and B₁) produce levels with higher log-likelihoods than levels sampled using random mappings. This shows that choice
of mapping impacts the quality of the levels sampled, and that the proposed domain transfer method is able to choose quality mappings. Additionally, the Kid Kool mappings produce levels closer in likelihood to the in-domain training levels than the Kid Icarus mappings, which is to be expected since Kid Kool is more structurally similar to Super Mario Bros. than Kid Icarus is. Further, some random mappings’ levels have linearity and leniency similar to the training levels, showing that these mappings capture the general layouts of the levels, but not which tiles should be used to create those layouts (as shown by the lower likelihoods).

Furthermore, while the baseline levels closely mirror the single training level, all levels generated by training just with one training level are very similar to each other and do not cover the full spectrum of levels. Supple-

**Table 8.2:** Comparison of Mappings. One corresponds to one map used for training. All corresponds to all the original maps in the Super Mario Bros. data set, and BL is the baseline of training an MdMC with a single training level. This table reports the average log-likelihood, linearity, and leniency for each of the models we train with the different sets of translated training levels.

<table>
<thead>
<tr>
<th></th>
<th>Super Mario Bros, Training Maps</th>
<th>Kid Kool to Super Mario Bros.</th>
<th>Kid Icarus to Super Mario Bros.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Log-Likelihood</td>
<td>Linearity</td>
<td>Leniency</td>
</tr>
<tr>
<td>One</td>
<td>–174.00</td>
<td>1.06</td>
<td>0.13</td>
</tr>
<tr>
<td>All</td>
<td>–352.00</td>
<td>2.35</td>
<td>0.78</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 8.7: 2-D projection of the original Super Mario Bros. levels (green), levels sampled using Kid Kool to supplement the training set (yellow and red), and the levels sampled using Kid Icarus to supplement the training set (blue).

menting the training data with Kid Kool mappings found with the proposed domain transfer method greatly extends the range of levels generated, covering a larger area of the space of levels.

Table 8.2 also shows much higher linearity values (meaning, less linear levels) for the Kid Icarus mappings when compared to the Kid Kool mappings and training levels, showing the effects of the vertical orientation of the Kid Icarus. Conversely, the Kid Kool mappings have linearity values falling between the training level and the Kid Icarus mappings, likely as a result of the mountainous structures and multiple height sections in the Kid Kool levels.

Lastly, the manually defined mappings for Kid Kool and Kid Icarus and the mappings found with the proposed domain transfer approach for Kid Icarus are able to approximate the leniency value of the original
training level, whereas the random mappings for Kid Icarus vary wildly (due to mappings that assign common tiles to enemies), and are often too low in the remaining Kid Kool mappings (due to the vastness of the Kid Kool levels paired with the relative infrequency of enemies).

To visualize the space of different levels generated by each approach, Figure 8.7 shows a two-dimensional projection of the sampled levels along with the 16 Super Mario Bros. levels, where each dot represents one level. Levels are projected based on a measure of distance between them. To determine the distance between two levels, We represent them as a histogram of high-level tiles, and compute the Euclidean distance between them. High-level tiles are found by clustering $4 \times 4$ tile sections using $k$-medoids ($k = 40$) with all the training levels and one transformed level from each tile mapping. It is interesting to see how closely grouped all the levels sampled using the Kid Kool levels are (red, yellow, orange). This may be due to the provided mapping constraints locking more tiles, or due to how similar Kid Kool is to Super Mario Bros. Additionally, the original Super Mario Bros. levels and the levels sampled with the baseline (green) are closer to the Kid Kool level clusters than to the Kid Icarus level clusters, which further supports that the Kid Kool levels are more similar to the training levels. Notice, levels sampled using the Kid Icarus (shades of blue) mappings are mostly separated into different clusters corresponding to the different methods (i.e., $M_i$,
Furthermore, levels produced with the manual mappings (dark blue), are close to the levels produced using the mappings found with the proposed domain transfer approach (blue), showing that it finds mappings more similar to human devised mappings than to random mappings. The figure also shows the narrow space covered by the original levels in the training set.

Figure 8.8 shows example levels sampled with each tile mapping. The *Kid Icarus* mapping levels contain a large amount of platforms (made of enemies, pipe pieces, and solid tiles), which mimic the structures in the *Kid Icarus* maps. Also note the mountainous structures in the *Kid Kool* mappings levels, which are present in the *Kid Kool* maps.

### 8.2.3 Conclusions

In this section we present an approach for transferring video game levels from one domain to another. The goal of this approach is to allow for levels in one domain to be used to supplement a training set of levels in another domain in which not much training data exists, allowing for machine learning-based level generation approaches to be used in newer or underdeveloped domains where there are few levels. This domain transfer approach tries to find mappings between the tile types used to represent two different domains. This is done by filtering the space of all possible mappings by investigating the frequency of the tiles, accounting for user provided constraints on the mappings, and exploring how accurately the mappings represent the target domain. We tested this approach by transferring from *Kid Icarus* and *Kid Kool* to *Super Mario Bros.* in order to test far and near transfer, respectively.

The experiments show that in general, our domain transfer approach is able to find tile mappings between the domains that are more useful than random tile mappings, and nearly as useful as the manually defined tile mappings (in terms of evaluations of the generated levels). Specifically, the experiments show that this approach can successfully transfer from a near domain (*Kid Kool*), while preserving the structural properties of the target domain (*Super Mario Bros.*), such as linearity. However, transferring from a far domain (*Kid Icarus*) results in levels that are structurally different from the target domain. For example, the linearity of the sampled levels using this transfer is very different from the target domain training level, because there are many more platforms and floating structures in the transferred levels. This shows that the choice of domains
is an important factor when performing domain transfer.

8.3 Discussion

In this chapter we explored different aspects of training data as it relates to machine learning-based level generation approaches. Specifically, we tried to address the 2 key problems of: (1) how does choice of training data affect the models and output, and (2) how can we supplement a domain that does not have enough training data. For (1) we developed a measure for approximating the quality of training levels (using an entropy measure to determine the amount of novel information in a level), and conducted experiments with two machine learning-based level generation approaches in order to determine how varying amounts of and varying quality training levels affected those models and their outputs. For (2) we developed a domain transfer approach that is able to transform levels from one domain to be used as training levels in another domain.

We found that for the models and domain investigated (multi-dimensional Markov chains and LSTMs, and Super Mario Bros., respectively) that less training data was required for the models than expected if using the high-quality training levels (only around 5 training levels needed). We also found that introducing more low-quality training data increased the size and amount of sections the models plagiarized from the training levels. Requiring little training data in order to train a model able to generate quality levels is an important finding, as it shows that applying these models to new domains (where training data may be very limited) is still possible. However, depending on the domain and chosen model, it may not always be the case that the amount of training data required is small, or that the amount of training data required is available.

We found that by using our domain transfer approach we are able to transform levels from one domain to be used as training levels in another domain. This approach works by finding a mapping between the tile types used to represent the two domains. Our approach is able to find tile mappings that perform almost as well as manually-defined mappings, and allow the models trained on the supplemented data set to create levels with qualities similar to the original training set.

Exploring the limitations of different machine learning-based level generation approaches with regards to training data quality and quantity provides a better understanding of the capabilities of each approach. This
better understanding allows users and researchers to make more informed decisions regarding which models may be applicable in a given domain. Furthermore, by noticing these limitations, we open future avenues of work in pushing those limitations or developing augmentations to the approaches to offset the limitations. Our domain transfer technique is one such augmentation meant to alleviate the problem of not having enough training data for a given model in a given domain. There are many other avenues of future work along these lines. For instance, developing more robust domain transfer techniques can further alleviate the problem of insufficient training data. Additionally, augmentations to machine learning-based approaches that allow for learning with less data can also be explored, such as transforming the available training data in useful ways for additional training data. Lastly, exploring the limitations of more approaches and in more domains can continue to help delineate the current limits of the class of machine learning-based PCG approaches.
Chapter 9: Conclusions

Procedural content generation (PCG) is an area of research that studies the automatic creation of content through the use of artificially intelligent algorithms. PCG has been used to aid human designers in collaborative systems and generate novel content in a variety of domains, such as game levels, trading cards, quests, and tutorials, to name a few. Until recently, many of the PCG approaches explored have relied on domain information from the user or designer in order to understand and create content for the given domain. Recently, machine learning-based PCG approaches have begun to be explored in order to replace user-provided domain knowledge with training data. Replacing domain knowledge with training data can allow for the wider application of PCG approaches across multiple domains, while also lowering the burden on the user of the system.

In this dissertation, we aimed to answer several important questions surrounding procedural level generation. The first such question we address is that of the applicability of machine learning approaches to the task of procedural video game level generation. We did this by exploring Markov models for modeling and generating levels. We found that our Markov model approaches are able to generate usable levels in several domains while requiring only a set of training levels from the user instead of knowledge of the domain. These results, along with results from other machine learning-based approaches developed later, showed that machine learning techniques can be fruitfully applied to level generation.

After determining that machine learning-based level generation approaches are viable, we wanted to better understand the relationships between the various machine learning approaches. In order to do this, we developed a theoretical framework for the class machine learning-based level generation approaches. Our framework offers a uniform lens with which to view current (and future) approaches. Specifically, the framework provides a more abstract description of the level representations, training procedures, and sampling procedures used by this class of approaches. By doing so, it is easier to compare these approaches to see where they differ, how they are similar, and where there are gaps in the current approaches to allow for future lines of research.
The next question we aimed to address is that of increasing the controllability of machine learning-based level generation approaches. To address this question we proposed several constrained sampling approaches which allowed the user to provide constraints on levels to be generated. We tested these approaches with our multi-dimensional Markov chain approach, but they are applicable with many other machine learning-based level generation approaches. Giving the user the ability to enforce constraints on the generated content provides them with much more control over the qualities of the generated levels than previously possible with these machine learning-based approaches.

In this dissertation we also wanted to address the question of how we can incorporate more than structural information into these machine learning-based approaches. To address this question, we explored two avenues. The first avenue aimed to supplement the structural knowledge learned by machine learning-based approaches with player behavior information in the form of a simple player model learned from gameplay videos. The player model was used to guide the generator towards creating levels that allowed for likely sequences of movements. The second avenue aimed to supplement the structural knowledge learned by machine learning-based approaches by representing levels in multiple layers, where each layer captured a unique aspect of the level design (such as, structural information, player paths, difficulty curves, etc.). With these extensions we saw that we were able to generate levels accounting for player behaviors, and that more accurately capture domain specific interactions (e.g., meaningful spring placement in *Super Mario Bros*). Though we tested both of these approaches with our MdMC approach, they are applicable across the class of machine learning-based level generation approaches.

The final question we looked to address is how the chosen training data can affect these machine learning models and their output. In order to address this question, we first investigated the effects of using varying amounts and varying quality of training levels on our MdMC approach and on Summerville and Matteas’ LSTM approach\textsuperscript{7}. We found that the quality of training data can have a profound effect on the models (i.e., few high-quality levels can be lead to better models than many low-quality levels). Afterwards, we explored an approach for supplementing the training data in a domain where not enough training data exists by translating levels from a similar domain into the target domain’s representation. We found that training a model with these translated levels significantly impacts the quality and structure of the generated levels, but
that by translating from a more similar domain we can mitigate this detriment.

9.1 Contributions

We now review and discuss our contributions in more detail, and in light of the reported results.

1. **Markov Models for Level Generation**: Our first key contribution is the development of several Markov model-based level generation approaches, which all learn the probability of a given tile type or in-game object given a surrounding neighborhood around that tile or object. The goal here is to show that machine learning techniques can be applied profitably to level generation. Our multi-dimensional Markov chain (MdMC) was the first such machine learning-based level generation approach (published shortly prior to another machine learning-based approach that used $n$—grams). It captures the probability of tile-to-tile transitions from a set of training levels and uses those learned transition probabilities to generate new levels. Next, we introduced our hierarchical MdMC approach. The goal of this hierarchical approach was to better capture long-range dependencies and be able to model high-level structures from the training levels. Lastly, we developed a Markov random field (MRF) approach which captures the probability of in-game objects and tiles given a neighborhood surrounding that position. This approach models levels without assuming the sequential spatial dependencies of the previous two approaches. The key result here is that with these approaches we showed that machine learning-based level generation approaches are general enough to be applicable across multiple domains while requiring very little domain knowledge from the user.

2. **A Theoretical Framework for Machine Learning-based PCG**: Our second key contribution is a theoretical framework for the procedural generation of levels via machine learning. This framework provides a way to see the level representation and training and sampling procedures of the existing machine learning-based level generation approaches through the same lens. This allows for uniform comparisons between approaches (i.e., highlight differences and similarities), while also providing insight into avenues of future work or gaps in the current research.

3. **Constrained Sampling Approaches**: Our third key contribution is the set of extensions developed for
machine learning-based PCG approaches which give the user more control over the generated content and allow the models to capture more than just structural information. The first such extension is aimed at improving the controllability of machine learning-based level generation models. Improving the controllability of these techniques can increase their usability by giving the user more input over the type of content created. We developed a constrained sampling approaches that enforces provided constraints in the sampled levels by resampling problematic sections of the level. We showed that our constrained sampling approaches provide the user with more control over the generated levels, and even allowed for the generation of usable levels in a domain where our standard Markov model sampling approaches struggled. This is a promising step towards more controllable machine learning-base level generation approaches.

4. **Tailored Level Generation using Player Modeling**: This extension is meant to increase what information is captured by machine learning-based level generation approaches. Specifically, it is an augmentation to the constrained sampling approach which allows for the inclusion of a probabilistic player behavior model as a constraint. We showed that by using a player model to guide the level generator, we can generate levels that allow for paths through the level that are likely given the player model. This extends the controllability of our approaches, while also allowing for the generation of player tailored content.

5. **Multi-layer Level Representations**: This extension also addresses the issue of capturing more than just structural information in order to develop a more representative model of a given domain. We did this by developing an extension to the level representation used by the machine learning approaches in our framework. We developed a general multi-layer level representation that allows for the representation of level information outside of the standard structural information. This representation builds upon our standard representation by allowing for additional layers of representation. We showed that using this multi-layer representation with an additional player path layer, we were able to generate levels that allowed for paths similar to the provided path. Additionally, though we tested with our MdMC approach, we showed that this level representation can be represented by the level graph level representation of our theoretical framework.
6. **Understanding the Effects of Training Data**: Our next contribution is the exploration of the effects of varying quantities and qualities of training data on our MdMC approach as well as a more complex LSTM approach. The goal here is to gain a better understanding of how these models react to different training sets, and try to generalize to other machine learning-based approaches. Finding and understanding these limitations and requirements for these approaches allows for more informed decisions about which domains to explore, or which extensions to the models can be most beneficial. We developed a measure for evaluating the quality of training data based on the amount of novel information in the level, and then experimented with training the models with varying amounts and orderings of the training data. We found that more training data is not always beneficial, and can in fact be a detriment to the models if the training data is of low quality. Additionally, we found that both models only required around 5 high-quality training levels in order to generate quality output.

7. **Domain Transfer**: Our final contribution also addresses the question of how training data affects a model, and specifically, how can we supplement an insufficient training data set. We explored an approach for leveraging training levels from other games as training data for a target domain where training data may be scarce. We accomplished this by developing a procedure for mapping the tile types from one domain representation to another using the likelihood of the tile types in the training levels and a trained conditional probability distributions for each domain. We found that this domain transfer approach is able to find mappings that are more meaningful than random ones, and that when used to transform levels in order to train an MdMC, produce a model that is able to generate usable levels.

Given our contributions, we now discuss potential avenues of future work.

### 9.2 Future Work

In this dissertation we showed that it is possible to define domain agnostic procedural level generation algorithms by leveraging machine learning techniques. Additionally, we have presented a unifying framework for many of these machine learning-based level generation approaches, and have explored extensions to these
approaches. The development of this new class of level generation approaches opens up many potential lines of future work which we detail below.

The most immediate and obvious avenue of future work is exploring the current machine learning-based level generation approaches in more domains. Currently, many of the approaches have only been applied in action platforming games, such as *Super Mario Bros.*, but in order to demonstrate the generality of these approaches and to explore the limits of them as well, we need to begin testing our approaches in a wider variety of domains. For example, there has been some work in adventure game level generation\(^{138}\) and we have explored our approaches in the context of *Lode Runner*, a puzzle platforming game, but little work has been done with first person shooter, puzzle, or strategy games. Each domain has different requirements in terms of what constitutes a usable level, but different genres of games also have their own requirements and quirks. For instance, in strategy game balancing the distribution of resources can be very important, whereas in puzzle games each level should build on the players previous knowledge, or challenge them in some specific way. In order to capture these properties, different methods of representation may be required. Additionally, most of the work in machine learning-based level generation has focused on two-dimensional levels. Exploring three-dimensional domains is an interesting problem that is within reach of the current approaches with minor adjustments. Exploring the limits of the current machine learning-based approaches by exploring additional domains and determining where the various models break down can provide useful insight into where improvements can be made. We previously believed our Markov model approaches were limited to games with grid-based levels. However, after developing our theoretical framework, we see that the class of representable domains is larger, namely, and domains in which the levels can be represented as graphs. This opens our models up to more domains, such as narrative-based games where there are not physical levels that the player traverses but branching stories where each plot point or decision point can be a node in the level graph.

Another line of future work is expanding the models to be able to capture more information about a given domain. We began exploring this line with our multi-layer level representations, but we are interested in continuing to expand this representation to capture more aesthetic and meta information, such as level decorations, level archetypes, and difficulty curves or progressions. However, there are some types of information
which may not easily be represented using a tile layer. In these cases additional extensions to the models or level representation may be required. Along this same line, comes the development of models that are able to more accurately model domains. For example, hidden Markov models are a clear next step for our MdMC approach, which may allow our models to capture more long-range relationships between objects.

As we discussed in Section 6.8 we are interested in extending our theoretical framework in several ways. Currently our framework is limited to domains where the levels are discrete. It would be interesting to attempt to generalize our framework to determine whether the same principles that apply to domains with discrete levels also apply to domains with continuous levels. Additionally, our framework currently only covers algorithms with equal weighting between all neighbors in the level graph (i.e., unweighted edges in the graph). However, there may be instances where it can be beneficial to have weighted edges. For instance, weighted edges can be used to indicate differences in proximity instead of the binary “neighbor” or “not neighbor” covered by the unweighted edges. This can be helpful in domains where there are dependencies across long distances and the strength of those dependencies is decided by the distance.

For our player modeling approach to tailored content described in Section 7.2, we have a few avenues to explore. First, we are interested in exploring more fine-grained and expressive player models. That is, we are interested in developing player models that are able to capture more detailed representations of a player’s behavior and of the action-space itself. Another avenue is testing our approach with multiple different players and player types to explore the effects that different player models have on the generated levels.

To further the line of improving the controllability of these machine learning-based level generation approaches, we are interested in exploring ways of offering control over the models and generated content to the user that are not based on constraints. For example, we have seen that the choice of training data has a large effect on the quality and type of level generated. One way to offer control is by developing an automatic way of determining which training levels should be used in order to create specific types of output or output with certain qualities.

Next, while we explored the effects of training data on two models in one domain, much more work needs to be done. We are interested in expanding this study to include more machine learning-based level generation approaches, and so that it spans multiple domains. Expanding the study in these ways will give a
broader understanding of how the quality and quantity of training levels effects machine learning approaches, and will allow for more informed decisions to be made when choosing which model to use in a given domain.

We are also interested in further exploring whether these machine learning based approaches actually reduce the burden on the user. We believe that this can be accomplished through a user study comparing various approaches (both machine learning and more traditional approaches). This could provide us with important information regarding the empirical benefits of machine learning-based PCG approaches from the perspective of the users of these systems.

More broadly, we are interested in exploring machine learning-based PCG for modeling more general concepts. For example, platforming levels and games often share similar design patterns and mechanics in their levels. We are interested in investigating whether we can develop an approach that is able capture these features of the more general class of platformer levels, instead of just Super Mario Bros. levels or Kid Icarus levels. Being able to learn and generalize across domains will allow for the broader application of machine learning-based PCG.

Lastly, our theoretical framework is a first step towards a deeper understanding of machine learning-based PCG approaches. However, we are interested in exploring machine learning in the broader context of computational creativity. More specifically, after developing this framework for machine learning-based PCG systems, what can we learn about machine learning in this broader context, and how can it fruitfully be applied to the field of creativity?
Bibliography


IEEE Transactions on Computational Intelligence and AI in Games, 2017.

[22] Kenneth Hullett and Michael Mateas. Scenario generation for emergency rescue training games. In 

[23] Alexander Zook, Stephen Lee-Urban, Mark O Riedl, Heather K Holden, Robert A Sottilare, and 
Keith W Brawner. Automated scenario generation: Toward tailored and optimized military train-
ing in virtual environments. In Proceedings of the International Conference on the Foundations of 

[24] James Michael Niehaus, Boyang Li, and Mark O Riedl. Automated scenario adaptation in support of 
intelligent tutoring systems. In FLAIRS Conference, 2011.

[25] Andrey Markov. Extension of the limit theorems of probability theory to a sum of variables connected 

procedural content generation: A taxonomy and survey. Computational Intelligence and AI in Games, 

[27] Rob A Rutenbar. Simulated annealing algorithms: An overview. Circuits and Devices Magazine, 


[29] David B Fogel. Phenotypes, genotypes, and operators in evolutionary computation. In Evolutionary 


[60] Kate Compton and Michael Mateas. Procedural level design for platform games. In *Proceedings of the
Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE), pages 109–111, 2006.


BIBLIOGRAPHY


Appendix A: Video Game Level Domains

In this appendix we introduce the 4 video game level domains that we use in our experiments. For each domain we first give a high-level description of the domain, and then cover in detail the set of tile types used to represent everything we capture within that domain.

A.1 Super Mario Bros.

*Super Mario Bros.* is platforming game with linear levels (as described by Dahlskog et. al). That is, the levels require paths that typically only require the player to travel from one side of the level to the other, while defeating enemies and avoiding pits. We use a training set of 29 outdoor levels taken from *Super Mario Bros.* and *Super Mario Bros. 2: The Lost Levels*. Figure A.1 shows a few levels from these games.

![Figure A.1: Four example levels from the Super Mario Bros. domain.](image)

A.1.1 Tile Types

We now describe the set of 36 tile types we use to represent levels from *Super Mario Bros.* We present the tile types in semantic groupings for clarity.

**Structural Tile Types**

#: This tile type represents the ground and other “solid” blocks. That is, this represents any block that is unbreakable, cannot be interacted with, and is not part of a pipe structure.
p: This tile type represent the left section of an upright pipe structure.

P: This tile type represent the right section of an upright pipe structure.

[: This tile type represent the top-left section of an upright pipe structure.

]: This tile type represent the top-right section of an upright pipe structure.

d: This tile type represent the left section of a downward facing pipe structure.

D: This tile type represent the right section of a downward facing pipe structure.

{: This tile type represent the top-left section of a downward facing pipe structure.

}: This tile type represent the top-right section of a downward facing pipe structure.

c: This tile type represents a section of bullet-bill cannon structure not containing the cannon.

C: This tile type represents a section of bullet-bill cannon structure containing the cannon.

Y: This tile type represents the bottom half of a springboard. Jumping onto the springboard allows the player to jump much higher or farther than usual.

y: This tile type represents the top half of a springboard.

**Platform Tile Types**

>: This tile type represents a section of a moving platform that moves left and right.

v: This tile type represents a section of a moving platform that moves downward, either falling when you touch it, or at a steady rate.

<: This tile type represents a section of a moving platform that moves upward.

**Block Tile Types**

B: This tile type represents the breakable brick-style blocks in the domain. Notice, this only represents the blocks that do not contain anything; blocks containing power-ups or coins are defined later.
?: This tile type represents the question mark blocks that contain only coins.

M: This tile type represents question mark blocks that contain a power-up (e.g., big mushroom or fire flower).

!: This tile type represents a question mark block with a poison mushroom inside (i.e., an item that damages the player if touched).

+: This tile type represents a block with an extra life item inside.

*: This tile type represents a block with a “star” power-up inside (i.e., an item that give the player temporary invulnerability from enemies).

O: This tile type represents a brick block with coins inside.

H: This tile type represents a brick block with a bean stalk inside. A bean stalk acts as a ladder to a hidden cloud zone with coins.

**Enemy Tile Types**

g: This tile type represents a “goomba” enemy, an enemy that walks slowly in one direction until colliding with something.

k: This tile type represents a “koopa” enemy, a turtle-like enemy that walks more quickly than a goomba in one direction until colliding with something. It turns into a kickable shell when jumped on.

K: This tile type represents a flying “koopa” enemy, a turtle-like enemy that either flies on a path or jumps in one direction until colliding with something.

t: This tile type represents a “buzzy beetle” enemy, a turtle-like enemy that walks in one direction until colliding with something. It is immune to fireballs.

V: This tile type represents a “piranha plant” enemy, a plant that comes out of pipes. This tile represents the plants coming from upright pipes.
**X:** This tile type represents a “piranha plant” enemy, a plant that comes out of pipes. This tile represents the plants coming from downward-facing pipes.

**h:** This tile type represents a “hammer bro” enemy, an enemy that jumps around within a set area and throws hammers in an arc that can damage the player.

**L:** This tile type represents a “lakitu” enemy, an enemy that flies around near the top of the level in a cloud and throws other spiked turtle enemies at the player.

**Other Tile Types**

**S:** This is a sentinel tile that denotes the boundaries of the level. This tile is helpful in providing context for the starting point and boundaries of level during training and sampling.

**o:** This tile type represents a coin.

**—:** This tile type represents empty space in the levels. Notice, we do not capture aesthetic information such as background colors or clouds in our experiments, and thus any background and decorative objects are abstracted into this tile type as well.

**|:** This tile type represents a section of the flagpole that appears at the end of each outdoor level.

### A.2 Kid Icarus

*Kid Icarus* is a platforming game with linear levels (as described by Dahlskog et. al\(^5\)). That is, the levels require paths that typically only require the player to travel from one side of the level to the other, while defeating enemies and avoiding pits. In this domain, we use a set of levels which are vertically oriented (i.e., the player must travel from the bottom of the level to the top in order to complete the level). We use a training set of 6 vertically oriented levels taken from *Kid Icarus*\(^{141}\). Figure A.2 shows a few levels from this game.

#### A.2.1 Tile Types

We now describe the set of 7 tile types we use to represent *Kid Icarus* levels. There are much fewer tile types in this domain, so we do not split the tile types into semantic groupings.
Figure A.2: Four example levels from the Kid Icarus domain.

S: This is a sentinel tile that denotes the boundaries of the level. This tile is helpful in providing context for the starting point and boundaries of level during training and sampling.
G: This tile type represents the solid structures within the levels (i.e., any solid object not otherwise covered with another tile type).

T: This tile type represents a section of a stationary platform. The difference between platforms and solid tiles are that the player can jump up through the bottom of a platform, but not a solid tile.

M: This tile type represents a section on the path of a moving platform. In our set of training levels, moving platforms only move horizontally.

D: This tile type represents a section of a door, which in this domain can lead to rooms with items or enemies. Once the room is completed the player is returned to the position of the door in the level.

H: This tile type represents a stationary hazard that damages the player if touched.

E: This tile type represents empty space in the levels. Notice, we do not capture aesthetic information such as background colors in our experiments, and thus any background and decorative objects are abstracted into this tile type as well.

A.3  Lode Runner

Lode Runner is a puzzle-platforming game which requires the player to collect objects placed throughout the level in order to make the exit ladder appear and complete the level. This game differs from the others in this section, because the player is unable to jump. Instead, the player must make use of ladders and ropes to move around the level, and the player has the ability to temporarily destroy some sections of the level to increase mobility or trap enemies. We use a set of 150 levels taken from Lode Runner\textsuperscript{142}. Figure A.3 shows a few levels from this game.

A.3.1  Tile Types

We now describe the set of 10 tile types we use to represent Lode Runner levels. There are much fewer tile types in this domain than in Super Mario Bros., and as such we do not separate them into semantic groupings.

S: This is a sentinel tile that denotes the boundaries of the level. This tile is helpful in providing context for the starting point and boundaries of level during training and sampling.
Figure A.3: Four example levels from the *Lode Runner* domain.

**B:** This tile type represents a solid section of ground.

**b:** This tile type represents a section of ground, which the player can temporarily destroy using a “dig” action. This can be done to trap enemies, or open up new paths. The section of ground will reappear several seconds after being destroyed.

**T:** This tile type represents a false section of ground (i.e., a sections of ground that appears solid, but that the player will fall through if she walks on it).

**#:** This tile type represents a section of a ladder. The player and enemies can move vertically or horizontally through connected ladder sections.

**-:** This tile type represents a section of rope. The player and enemies can move horizontally across sections of rope.

**X:** This tile type represents an enemy. Contact with an enemy kills the player.

**G:** This tile type represents a piece of treasure. All of the treasure in a level must be collected in order
to complete the level.

**H**: This tile type represents a section of a ladder that will appear once all the treasure is collected. These sections of ladder allow the player to exit the completed level.

**E**: This tile type represents empty space in the levels. Notice, we do not capture aesthetic information such as background colors in our experiments, and thus any background and decorative objects are abstracted into this tile type as well.

### A.4 Kid Kool

*Kid Kool* is platforming game with linear levels (as described by Dahlskog et. al\(^5\)). That is, the levels require paths that typically only require the player to travel from one side of the level to the other, while defeating enemies and avoiding pits. However, unlike *Super Mario Bros.*, these levels have several different height sections that break the level into sections depending on the position of the player. Note that we only use this domain as a source domain during the domain transfer experiments in Section 8.2, and not during our standard level generation experiments. We use a training set of 12 outdoor levels taken from *Kid Kool and the Quest for the Seven Wonder Herbs*\(^{143}\). Figure A.4 shows a few levels from this game.

**Figure A.4**: Four example levels from the *Kid Kool* domain.
A.4.1 Tile Types

We now describe the set of 12 tile types we use to represent Kid Kool levels. There are much fewer tile types in this domain than in Super Mario Bros., and as such we do not separate them into semantic groupings.

S: This is a sentinel tile that denotes the boundaries of the level. This tile is helpful in providing context for the starting point and boundaries of level during training and sampling.

G: This tile type represents the solid structures within the levels (i.e., any solid object not otherwise covered with another tile type).

W: This tile type represents a section of water. The player can skip across the surface of the water if moving horizontally quickly enough, but will fall into the water and lose the level otherwise.

b: This tile type represents a section of collapsable platform (i.e., a section of a platform that will collapse once the player walks on it).

M: This tile type represents a springboard that allows the player to jump much higher than usual.

|: This tile type represents a section of a pole that can launch the player much further than standard jumping allows.

H: This tile type represents a section of a tube which the player can either walk on top of, or enter at one end to be transported to the other end.

B: This tile type represents a wind generator that blows the player in the direction it is facing.

X: This tile type represents the enemies in the level.

c: This tile type represents a cannon that launches bouncing objects that can damage the player.

T: This tile type represents the collectible treasure within the level.

E: This tile type represents empty space in the levels. Notice, we do not capture aesthetic information such as background colors in our experiments, and thus any background and decorative objects are abstracted into this tile type as well.
Appendix B: List of Publications

B.1 Journal Articles


B.2 Invited Talks


B.3 Conference Papers


Snodgrass, Sam, and Santiago Ontaño. A Hierarchical MdMC Approach to 2D Video Game Map Generation. Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference. 2015.


B.4 Workshop Papers


B.5 Other Publications


Sam Philip Snodgrass received a Bachelor of Science in Computer Science from Ursinus College in Collegeville, Pennsylvania in May 2012. He received his Ph.D. in Computer Science with a focus on Artificial Intelligence for Games from Drexel University in December 2017. During his time at Drexel University he published a total of 11 main conference papers and 3 workshop papers at a variety of conferences. He published one journal article, and published one tutorial on the use of machine learning for procedural content generation. Lastly, he gave an invited talk at the first Asynchronous Research on AI and Games (ASYNC). A more detailed list of publications can be found in Appendix B. Additionally, during his time at Drexel University he worked as a contracted researcher at the United States Naval Research Lab for several months (which resulted in a technical report), and as a contracted researcher at the United States Army Research Lab for a Summer.