Proceedings of the Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2014)

A Hierarchical Approach to Generating Maps Using Markov Chains

Sam Snodgrass and **Santiago Ontañón**
Drexel University, Department of Computer Science
Philadelphia, PA, USA
sps74@drexel.edu, santi@cs.drexel.edu

Abstract

In this paper we describe a hierarchical method for procedurally generating maps using Markov chains. Our method takes as input a collection of human-authored two-dimensional maps, and splits them into high-level tiles which capture large structures. Markov chains are then learned from those maps to capture the structure of both the high-level tiles, as well as the low-level tiles. Then, the learned Markov chains are used to generate new maps by first generating the high-level structure of the map using high-level tiles, and then generating the low-level layout of the map. We validate our approach using the game *Super Mario Bros.*, by evaluating the quality of maps produced using different configurations for training and generation.

Introduction

Delegating map generation to an algorithmic process can save developers time and money (Togelius et al. 2010), or even allow novel forms of gameplay. Using such algorithmic processes is called procedural content generation (PCG), which generally refers to methods for generating all types of content, such as maps or quests.

In this paper we present a hierarchical approach to generating maps using Markov chains. We chose Markov chains because they can easily represent two-dimensional models, which is how maps are represented in many game genres. Our method learns a statistical model from known high-quality maps, then generates new maps using those models. In our experiments we focus on generating two-dimensional levels for the platformer game *Super Mario Bros.*

This paper focuses on a hierarchical extension of our previous work in map generation (Snodgrass and Ontañón 2014). In this paper we introduce a hierarchical model, which represents maps in two levels: the low-level map is the tile-by-tile representation of the map; the high-level map uses a set of *high-level tiles* which encompass a larger section of the map in order to capture some large structures and long range tile dependencies. Our method breaks a map into high-level tiles then learns the statistical model of both the high-level map and the low-level map. Once we have the models, we generate new high-level maps, then generate low-level maps using both the high-level maps and the statistical models.

The remainder of the paper is organized as follows. First, we give some background on procedural content generation as well as Markov chains. After that, we describe our map representation, our approach for training Markov chains, and our hierarchical Markov chain map generation technique. Then, we present an experimental evaluation. The paper closes with conclusions and directions for future work.

Background

In this section we provide background on procedural content generation, focusing on map generation, and Markov chains.

Procedural Map Generation

Procedural content generation (PCG) refers to methods for creating content algorithmically instead of manually (Togelius et al. 2011). Such methods can be used in games to generate components like maps (Togelius et al. 2010), quests (Bakkes and Dormans 2010), animations (Park, Shin, and Shin 2002), and textures (Lefebvre and Neyret 2003), among others. For a survey of procedural content generation techniques the reader is referred to Hendrikx et. al. (2013).

Most PCG approaches can be classified into three broad categories: Search-based, learning-based, and tiling. Search-based techniques generate content by exploring a defined space using a search technique paired with an evaluation function. Learning-based approaches extract models from player or designer data, or known quality content, then generate novel content using those learned models. Tiling approaches build content algorithmically from smaller parts called "tiles," using different tile sizes and assembly techniques. For more information on search, learning, and tiling techniques, the reader is referred to Togelius et al. (2011), Shaker et al. (2011), and Compton et al. (2006), respectively.

These categories, however, are not mutually exclusive or complete, since there are hybrid methods and methods which do not fit any of these categories. For example, Smith et. al. (2009) developed an approach that generates platformer maps based on the rhythm that the map should achieve and the player's available actions.

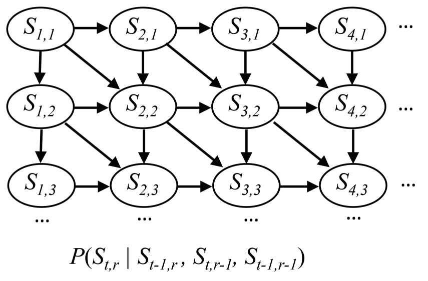

Figure 1: A two-dimensional representation of a higher order Markov chain.

Markov Chains

Markov chains (Markov 1971) are a method for modeling probabilistic transitions between states. Formally, 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 $S_{t-1} \in S$. Notice that Markov chains can be seen as a particular case of Dynamic Bayesian Networks (DBN)(Murphy 2002).

Standard Markov chains restrict the probability distribution to only take into account the previous state. Higher order Markov chains relax this condition by taking into account k previous states, where k is a finite natural number (Ching et al. 2013). In certain applications, using higher orders allows Markov chains to model state transitions more accurately. The CPD defining a Markov chain of order k can be written as: $P(S_t|S_{t-1}, ..., S_{t-k})$. That is, P is the conditional probability of transitioning to a state S_t, given the states of the Markov chain in the past k instants of time.

In our application domain, we structure the variables in the Markov chain in a two-dimensional array, in order to suit the map generation application. Figure 1 shows an illustration of this, showing dependencies between the different state variables in an example Markov chain of order 3 (each state variable depends on 3 previous state variables).

Methods

The key idea we explore in this paper is using Markov chains for hierarchical map generation. A low-level Markov model is used to generate the low-level details of the map (the tiles used to render the map, such as *ground*, *blocks*, etc.), and a high-level Markov model is used to generate high-level structures of the map (such as slopes, platforms, etc.). In this section, we discuss how we represent maps, how we categorize sections of those maps into high level tiles, how our model learns from those maps, and finally how our model generates new maps after learning.

Map Representation

We represent a map as an $h \times w$ two-dimensional array M, where h is the height of the map, and w is the width. Each cell $M(i,j)$ corresponds to a tile in the map, and can take one of a finite number of values S, which represent the tile types. When modeling maps using Markov chains, the different tile types correspond to the states of the Markov chain.

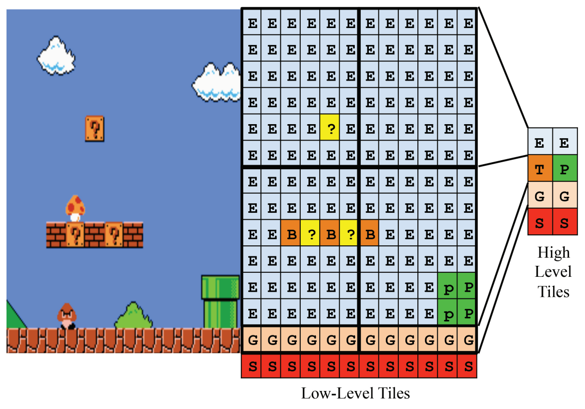

Figure 2: A section from a map used in our experiments (left), and how we represent that map (center) in an array of low-level tiles (we added one row underneath for Markov chain training purposes), and as an array of 6×6 high-level tiles (right). Color has been added for clarity.

Figure 2 shows a section of a map we use in our experiments from the *Super Mario Bros.* game (left), and the representation of the map as an array (center), where each letter represents a different tile type. Currently, we only consider the map layout, without taking enemies into account. We add an additional row of tiles at the bottom of the map for learning purposes (as described later).

Hierarchical Map Conversion

Our method converts a given map into *high-level tiles* that capture larger structures of maps. These high-level tiles are designer-defined. For example, if we use high-level tiles of size 6×6, we would divide the map into 6×6 chunks, and assign a high-level tile type to each chunk. The procedure to determine which high-level tiles corresponds to each 6×6 chunk is domain-specific. If a map is not evenly divisible by the dimensions given, the remaining tiles can either be ignored or categorized as a full section. Thus, the high-level representation of a map is a two dimensional array H of size $(h/T) \times (w/T)$ (where T is the size of the high-level tiles). The set of high-level tiles types used in our experiments is reported in the experimental results section, below.

Figure 2 shows a section of a map used in our experiments (left) from *Super Mario Bros.*, and the high-level tile representation using a 6×6 section for categorization (right).

Learning

Our method learns a high-level model for capturing the distribution of high-level tiles, and a set of low-level models for capturing the distribution of low-level tiles within each high-level tile. Low-level models can only capture relations between tiles that are one or two spaces away. The high-level model is able to capture longer range relations, because of the compression of low-level tiles into fewer high-level tiles.

$$D_0 = \begin{pmatrix} 0 & 0 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad D_1 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \quad D_3 = \begin{pmatrix} 1 & 1 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$D_5 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Figure 3: The dependency matrices used in our experiments. Note that matrix D_4 was used in our previous work (Snodgrass and Ontañón 2014), but was not used in the experiments reported in this paper.

For both levels our method employs higher order Markov chains in order to learn the probabilistic distribution of tiles.

In order to learn a Markov chain, we need to specify which previous states the current state depends on, i.e. the Bayesian network structure. For example, we could learn a Markov chain defined by the probability of one tile in the map given the previous horizontal tile, or we could learn another defined by the probability of a tile given the previous tile horizontally and the tile immediately below, etc. Automatically learning the structure of a Bayesian network is a well known hard problem (Lam and Bacchus 1994), thus, in our approach, we configure the dependencies by hand. Our learning method takes as input an $n \times n$ dependency matrix D, defined as follows: the top right cell is set to 2 and represents the tile we are going to generate. The rest of the tiles are set to 1 or 0 depending on which tiles to consider as dependencies. Figure 3 shows the different dependency matrices used in our experiments. For example, matrix D_2 generates tiles considering the tile immediately to the left, and the one immediately below, while matrix D_3 generates tiles considering the two tiles immediately to the left.

In order to learn both the high-level model and the low-level models, in addition to the dependency matrix, we need to specify the following information:

- When learning the high-level tile distribution: a set of high-level maps $M_1^h, ...M_m^h$, represented as arrays.
- When learning the low-level tile distribution: for a given high-level tile s, the size of the section to be represented by each high-level tile, a set of hierarchical maps represented by the arrays $M_1^h, ...M_m^h$, and a set of low-level maps represented by the arrays $M_1, ...M_m$. The model will be generated using only the low-level tiles from $M_1, ...M_m$ that correspond to high-level tiles of type s.

A Markov chain is learned in two stages:

1. Absolute Counts: let k be the number of 1's in the matrix D (i.e., how many past states the model accounts for, corresponding to the order of the Markov chain), and n be the number of different tile types. There are n^k different previous tile configurations. For each previous tile configuration, our method counts the number of times that each tile type s_i appears in the input maps: $T(s_i|S_{i-1}, ..., S_{i-k})$. When learning a low-level model for a particular high-level tile type s, only the tiles in the low-level maps corresponding to high-level tiles of type s are used.
2. Probability Estimation: once the totals are computed, we can estimate from them the probability distribution that defines the Markov chain. In our experiments, we used a simple frequency count:

$$P(s_i|S_{i-1}, ..., S_{i-k}) = \frac{T(s_i|S_{i-1}, ..., S_{i-k})}{\sum_{j=1...n} T(s_j|S_{j-1}, ..., S_{j-k})}$$

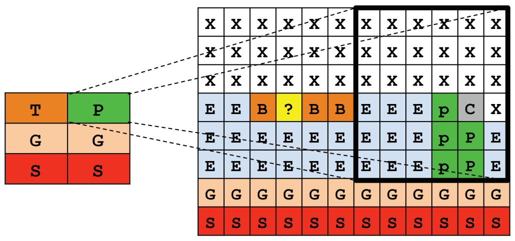

Figure 4: Example of a section of a low-level map (right) being generated using a high-level map (left) as a template. **X** represents tiles not yet generated and **C** represents the tile currently being generated

Other approaches, such as a Laplace smoothing (Manning, Raghavan, and Schutze 2009, p. 226) can be used if there is not enough data to have an accurate estimation.

In our previous work (Snodgrass and Ontañón 2014), we observed that different parts of the maps have different statistical properties. For example, when looking at *Super Mario Bros.* maps, it is impossible to have ground towards the top of the map. For that reason, we allow our method to learn separate probability distributions for different parts of the map, by splitting the map using horizontal *row splits*.

Map Generation

Our method generates a map in two stages: first the high-level map, and then the low-level map. For both stages, our method generates one tile at a time starting in the bottom left and generating row by row. To generate a tile, our method selects a tile probabilistically, based on the probability distribution learned before. When generating the high-level map, we use the probability distribution provided by the high-level model. When generating the low-level map, the low-level Markov chain to use is determined by the high-level tile we are generating within. Figure 4 shows a low-level map being generated from a high-level map.

During generation, it is possible to encounter a combination of previous tiles that was never seen during training, and for which we have no probability estimations. We call this an *unseen state*. We assume unseen states correspond to ill-formed structures, and are, thus, undesirable. To avoid unseen states, we build upon our previous work (Snodgrass and Ontañón 2014) and incorporate two strategies:

- **Look-ahead**: Given a fixed number of tiles to look-ahead $d \geq 0$, when our method generates a tile, it tries to generate the following d tiles after that. If during that process, it

Table 1: Best results from our previous work only using a low-level Markov Chain, learned using matrix D_5, and falling back to D_2. The columns $\%D_5$ and $\%D_2$ show the percentage of tiles generated using D_5 and the percentage of tiles generated with the fallback, D_2.

% D_5	*% D_2*	*% Completed*	*Bad Pipes*
98.71%	1.29%	44.00%	0.00

encounters an unseen state, our method backtracks to the previous level and tries with a different tile type. If the d following tiles are generated successfully without reaching an unseen state, then the search process stops, and the tile that was selected at the top level is accepted.

- **Fallback Strategies**: When the look-ahead process fails, our system falls back to a Markov chain trained with a simpler dependency matrix D^{fb}. Because D^{fb} is a simpler configuration than D, it results in a Markov chain of lower order, and has a smaller chance of reaching unseen states. Thus, a Markov chain trained using D^{fb} may be able to generate a tile when a Markov chain trained with D cannot. In our previous work (Snodgrass and Ontañón 2014) we used a single fallback matrix. In this work, we use a sequence of fallback matrices. If we cannot generate a tile with D, we fallback to a simpler matrix. If that matrix still cannot generate a tile, we fall back to an even simpler matrix, and so on, until we reach D_0, which is the raw tile distribution (i.e. a Markov chain or order 0). We use the same look-ahead depth for each matrix.

Note that the goal of these strategies is to guide the map generation. Other techniques exist in the literature for similar purposes, such as (Pachet et al. 2001).

Experiments

We chose to use the classic two-dimensional platformer game *Super Mario Bros.* as our application domain. To represent the Super Mario Bros. maps we chose to use eight low-level tile types. The first two types are special tiles. **S** represents the start and underneath of the map, and **D** signifies the end of the map. The remaining six tiles correspond to components of the maps: **G** are ground tiles, **B** are breakable blocks, **?** are power-up blocks, **p** are left pipe pieces, **P** are right pipe pieces, and **E** is empty space.

We devised ten high-level tiles, which represent larger structures in the maps: We have a special tile that represents the start and underneath of the map called **S**. The remaining nine tiles are as follows: **G** is solid ground, **A** is ground with gaps, **U** is an uphill slope, **D** is a downhill slope, **I** is a stack of low-level ground tiles which is taller than it is wide, **L** is a stack of low-level ground tiles which is wider than it is tall, **P** is a pipe, **T** is a platform of low-level breakable or power-up tiles, and **E** are empty space. We also provided a method that can transform a low-level map into a high-level map.

Experimental Setup

For our experiments, we used 12 maps from Super Mario Bros. to train our models. We excluded indoor and underwater maps, because they are structurally different. We ran our

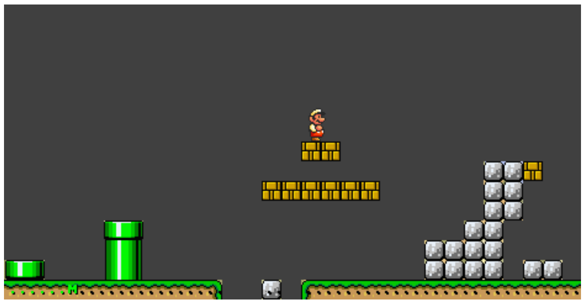

Figure 5: Section of a map generated with $T = 4$, $R = 4$, $D^h = D_2$, $D^l = D_2$, showing a playable and nice looking section of a map.

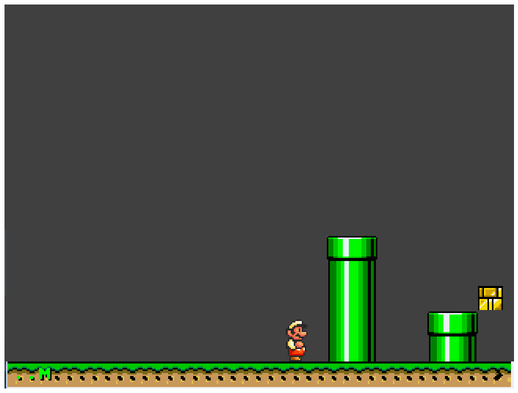

Figure 6: Section of a map generated with $T = 6$, $R = 3$, $D^h = D_2$, $D^l = D_2$, showing a pipe that is too tall to pass.

method using several configurations. Specifically, we experimented with the effect of the following variables:

- *Tile Size (T)*: When $T = k$, our method categorizes each $k \times k$ chunk of the map into a high-level tile type. We experimented with $T \in \{3, 4, 6\}$. We chose these values because the maps we learn from are 13 low-level tiles tall.
- *Row Splits (R)*: When $R = 1$, the entire map is used to train the high-level Markov chain. When $R = h$, where h is the height of the map, we divide each high-level map into individual rows.
- *High-Level Dependency Matrix (D^h)*: We experimented with three different dependency matrices in our experiments. $D^h \in \{D_1, D_2, D_3\}$, as shown in Figure 3.
- *Low-Level Dependency Matrix (D^l)*: We experimented with four different dependency matrices in our experiments. $D^l \in \{D_1, D_2, D_3, D_5\}$, as shown in Figure 3.

As for the *Fallback Strategy*, in our experiments, we used the following fallback matrices. D_5 falls back to D_2. D_3 and D_2 both fall back to D_1. D_1 falls back to D_0, which does not depend on any previous tiles.

To test these configurations, we generated 50 maps of width 320 with each possible configuration, and evaluated them using three metrics:

Table 2: Experimental results using $D^h = D_1$, showing the percentage of tiles generated with the main matrix, and the percentage of tiles generated with each of the fallbacks.

		High Level: $D_1 \rightarrow D_0$		*Low Level: $D_1 \rightarrow D_0$*					
T	*R*	*% D_1*	*% D_0*	*% D_1*	*% D_0*		*% Completed*	*Bad Pipes*	*Misplaced Ground*
3	1	100.00%	0.00%	99.99%	0.01%		52.00%	24.00	46.00%
3	4	100.00%	0.00%	99.99%	0.01%		48.00%	23.24	**0.00%**
4	1	100.00%	0.00%	99.99%	0.01%		28.00%	23.72	48.00%
4	4	100.00%	0.00%	99.99%	0.01%		**56.00%**	**21.34**	**0.00%**
6	1	100.00%	0.00%	99.99%	0.01%		48.00%	24.02	50.00%
6	5	100.00%	0.00%	99.99%	0.01%		22.00%	28.74	**0.00%**
Average							42.33%	24.18	24.00%
		High Level: $D_1 \rightarrow D_0$		*Low Level: $D_2 \rightarrow D_1 \rightarrow D_0$*					
T	*R*	*% D_1*	*% D_0*	*% D_2*	*% D_1*	*% D_0*	*% Completed*	*Bad Pipes*	*Misplaced Ground*
3	1	100.00%	0.00%	80.60%	19.40%	0.00%	**84.00%**	**0.00**	60.00%
3	5	100.00%	0.00%	99.94%	0.06%	0.00%	56.00%	0.10	**0.00%**
4	1	100.00%	0.00%	70.11%	29.89%	0.00%	42.00%	0.08	56.00%
4	4	100.00%	0.00%	99.95%	0.05%	0.00%	66.00%	0.36	**0.00%**
6	1	100.00%	0.00%	72.29%	27.20%	0.51%	64.00%	0.02	52.00%
6	3	100.00%	0.00%	99.98%	0.02%	0.00%	28.00%	0.08	**0.00%**
Average							56.67%	0.11	28.00%
		High Level: $D_1 \rightarrow D_0$		*Low Level: $D_3 \rightarrow D_1 \rightarrow D_0$*					
T	*R*	*% D_1*	*% D_0*	*% D_3*	*% D_1*	*% D_0*	*% Completed*	*Bad Pipes*	*Misplaced Ground*
3	1	100.00%	0.00%	99.98%	0.02%	0.00%	58.00%	22.40	56.00%
3	5	100.00%	0.00%	99.97%	0.02%	0.01%	**62.00%**	21.88	**0.00%**
4	1	100.00%	0.00%	99.99%	0.01%	0.00%	30.00%	21.58	62.00%
4	4	100.00%	0.00%	99.98%	0.01%	0.00%	48.00%	27.86	**0.00%**
6	1	100.00%	0.00%	99.99%	0.01%	0.00%	48.00%	**21.28**	56.00%
6	3	100.00%	0.00%	99.99%	0.01%	0.00%	32.00%	29.10	**0.00%**
Average							46.33%	24.02	29.00%

- *% Completable*: We loaded all the maps generated by our system into the 2009 Mario AI competition software (Togelius, Karakovskiy, and Baumgarten 2010), and made Robin Baumgarten's A^* controller play through them. Given an upper limit of 70 seconds, we recorded the percentage of maps that this controller was able to complete. However, this agent occasionally fails to complete maps that are completable, skewing the results.
- *Bad Pipes*: We counted how many incorrectly formed pipes appeared in our maps. Notice that it is trivial to write a simple rule to prevent incorrectly formed pipes, but we are interested in methods that can automatically learn how to generate maps with minimal additional human input.
- *Misplaced Ground*: Some configurations generated a high percent of playable maps, but the maps were playable because of a row of high-level ground tiles being placed above where the ground should be placed, allowing the agent to avoid most obstacles by starting on the higher ground. We computed the percentage of maps for each configuration that had a misplaced row of ground tiles.

Results

Table 1 shows the most promising results from our previous work, where we only used low-level Markov chain models to generate maps, and provides a baseline with which to compare our new results. Tables 2, 3, and 4 show the results of the different configurations when the high-level dependency matrix is locked to D_1, D_2, and D_3, respectively.

We can see from the tables that when $D^h = D_1$ or $D^h = D_3$, and $R = 1$ a large percentage (around 50%) of maps with misplaced ground rows results. This is to be expected because D_1 and D_3 do not account for what is below the current tile, therefore they cannot reliably predict where the ground should go without assistance. No high-level maps generated with D_2 had any misplaced ground rows. It is obvious from the tables that having $R = h$ is necessary if the high-level dependency matrix does not take into account tiles below the current one.

Similarly, when $D^l = D_1$ or $D^l = D_3$, we see a very high average number of bad pipes per level, over 20, whereas when $D^l = D_2$ or $D^l = D_5$, the average number of bad pipes per level has a maximum of 0.36 (this means only 0.36 bad pipes generated in $320 \times 13 = 4160$ tiles, which is the size of the maps being generated).

Looking at the effects of T, we see that configurations with $T = 4$ have a higher percentage of playable maps. When $T = 6$, the maps generated tend to have structures that are too tall for the agent to cross. Similarly, when $T = 3$, some structures get split into two separate tiles, which produces stacked structures, having the same effect. Figure 6 shows a section of a map generated with $T = 6$, $R = 3$, $D^h = D_2$, and $D^l = D_2$ with a pipe that is too tall to pass.

Comparing our current results to those of our previous work in Table 1, it can be seen that a large number of configurations are able to generate a higher percentage of playable maps, and still a low number of bad pipes. For example, the configuration with $T = 4$, $R = 4$, $D^h = D_2$, and $D^l = D_2$

Table 3: Experimental results using $D^h = D_2$, showing the percentage of tiles generated with the main matrix, and the percentage of tiles generated with each of the fallbacks.

		High Level: $D_2 \rightarrow D_1 \rightarrow D_0$			*Low Level:* $D_2 \rightarrow D_1 \rightarrow D_0$						
T	*R*	*%* D_2	*%* D_1	*%* D_0	*%* D_2	*%* D_1	*%* D_0		*% Completed*	*Bad Pipes*	*Misplaced Ground*
3	1	100.00%	0.00%	0.00%	99.97%	0.03%	0.00%		46.00%	**0.04**	**0.00%**
3	5	100.00%	0.00%	0.00%	99.97%	0.03%	0.00%		52.00%	0.10	**0.00%**
4	1	100.00%	0.00%	0.00%	99.98%	0.01%	0.00%		52.00%	**0.04**	**0.00%**
4	4	100.00%	0.00%	0.00%	99.97%	0.03%	0.00%		**76.00%**	0.18	**0.00%**
6	1	100.00%	0.00%	0.00%	99.98%	0.02%	0.00%		52.00%	0.12	**0.00%**
6	3	100.00%	0.00%	0.00%	99.98%	0.02%	0.00%		28.00%	0.12	**0.00%**
Average									52.67%	0.10	0.00%
		High Level: $D_2 \rightarrow D_1 \rightarrow D_0$			*Low Level:* $D_5 \rightarrow D_2 \rightarrow D_1 \rightarrow D_0$						
T	*R*	*%* D_2	*%* D_1	*%* D_0	*%* D_5	*%* D_2	*%* D_1	*%* D_0	*% Completed*	*Bad Pipes*	*Misplaced Ground*
3	1	100.00%	0.00%	0.00%	99.89%	0.10%	0.01%	0.00%	38.00%	**0.12**	**0.00%**
3	5	99.99%	0.01%	0.00%	99.88%	0.10%	0.02%	0.00%	42.00%	0.14	**0.00%**
4	1	100.00%	0.00%	0.00%	99.94%	0.03%	0.03%	0.00%	60.00%	0.22	**0.00%**
4	4	100.00%	0.00%	0.00%	99.92%	0.05%	0.03%	0.00%	**62.00%**	0.24	**0.00%**
6	1	100.00%	0.00%	0.00%	99.95%	0.03%	0.03%	0.00%	42.00%	0.20	**0.00%**
6	3	100.00%	0.00%	0.00%	99.96%	0.02%	0.02%	0.00%	30.00%	**0.12**	**0.00%**
Average									45.67%	0.17	0.00%

Table 4: Experimental results using $D^h = D_3$, showing the percentage of tiles generated with the main matrix, and the percentage of tiles generated with each of the fallbacks.

		High Level: $D_3 \rightarrow D_1 \rightarrow D_0$			*Low Level:* $D_3 \rightarrow D_1 \rightarrow D_0$						
T	*R*	*%* D_3	*%* D_1	*%* D_0	*%* D_3	*%* D_1	*%* D_0		*% Completed*	*Bad Pipes*	*Misplaced Ground*
3	1	100.00%	0.00%	0.00%	99.97%	0.02%	0.01%		24.00%	**20.84**	42.00%
3	5	100.00%	0.00%	0.00%	99.98%	0.02%	0.00%		46.00%	22.50	**0.00%**
4	1	100.00%	0.00%	0.00%	99.99%	0.01%	0.00%		26.00%	22.14	58.00%
4	4	100.00%	0.00%	0.00%	99.99%	0.01%	0.00%		**48.00%**	21.18	**0.00%**
6	1	100.00%	0.00%	0.00%	99.98%	0.01%	0.01%		34.00%	25.60	52.00%
6	3	100.00%	0.00%	0.00%	99.99%	0.01%	0.01%		36.00%	28.90	**0.00%**
Average									35.67%	23.53	28.67%
		High Level: $D_3 \rightarrow D_1 \rightarrow D_0$			*Low Level:* $D_5 \rightarrow D_2 \rightarrow D_1 \rightarrow D_0$						
T	*R*	*%* D_3	*%* D_1	*%* D_0	*%* D_5	*%* D_2	*%* D_1	*%* D_0	*% Completed*	*Bad Pipes*	*Misplaced Ground*
3	1	100.00%	0.00%	0.00%	70.94%	0.09%	28.97	0.00%	58.00%	0.14	70.00%
3	5	99.99%	0.01%	0.00%	99.82%	0.13%	0.05%	0.00%	62.00%	0.10	**0.00%**
4	1	100.00%	0.00%	0.00%	82.08%	0.03%	17.89%	0.00%	**80.00%**	0.30	52.00%
4	4	99.99%	0.01%	0.00%	99.90%	0.05%	0.04%	0.01%	68.00%	0.18	**0.00%**
6	1	100.00%	0.00%	0.00%	61.78%	0.04%	38.17%	0.01%	38.00%	**0.08**	56.00%
6	3	99.99%	0.00%	0.01%	99.94%	0.05%	0.00%	0.01%	38.00%	0.22	**0.00%**
Average									57.33%	0.17	29.67%

yields 76% completable maps with only 0.18 bad pipes per level. Figure 5 shows a section of a map generated using this configuration.

Conclusions

This paper presented a hierarchical approach to generating two-dimensional maps using Markov Chains. We improved upon our previous results by adding an additional layer of abstraction to the learning and generation processes, as well as employing multiple fallback strategies. By representing the maps as high-level tiles we were able to capture more of the structure of the map, and by using multiple fallback strategies we avoided more unseen states. This helped reduce the number of ill-formed structures and increase the number of playable maps (up to 76% of maps playable, which is a lower bound, because the A^* agent was unable to complete some playable maps).

In the future, we want to automate the detection of high-level tile types. Currently, our approach requires domain knowledge from the designer in order to categorize the high-level tiles. Automating the detection of high-level tile types would remove the need for domain knowledge and possibly create high-level tiles that the designer may not have. We also want to find a more reliable method for testing the playability and quality of our generated maps. Once we can accurately evaluate our maps, our system can generate several maps at run-time and choose the best one to show to the player. Additionally, though we tested our approach with *Super Mario Bros.*, it is applicable to any game with linear maps. One area of future work for our method is to expand it to be able to handle non-linear maps, such as *Metroid* or *Megaman*, where the concept of *paths* is very important.

References

Bakkes, S., and Dormans, J. 2010. Involving player experience in dynamically generated missions and game spaces. In *Eleventh International Conference on Intelligent Games and Simulation (Game-On'2010)*, 72–79.

Ching, W.-K.; Huang, X.; Ng, M. K.; and Siu, T.-K. 2013. Higher-order markov chains. In *Markov Chains*. Springer. 141–176.

Compton, K., and Mateas, M. 2006. Procedural level design for platform games. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE)*.

Hendrikx, M.; Meijer, S.; Van Der Velden, J.; and Iosup, A. 2013. Procedural content generation for games: a survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)* 9(1):1.

Lam, W., and Bacchus, F. 1994. Learning bayesian belief networks: An approach based on the mdl principle. *Computational intelligence* 10(3):269–293.

Lefebvre, S., and Neyret, F. 2003. Pattern based procedural textures. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, 203–212. ACM.

Manning, C.; Raghavan, P.; and Schutze, M. 2009. *Probabilistic information retrieval*. Cambridge University Press.

Markov, A. 1971. Extension of the limit theorems of probability theory to a sum of variables connected in a chain.

Murphy, K. P. 2002. *Dynamic bayesian networks: representation, inference and learning*. Ph.D. Dissertation, University of California.

Pachet, F.; Roy, P.; Barbieri, G.; and Paris, S. C. 2001. Finite-length markov processes with constraints. *transition* 6(1/3).

Park, S. I.; Shin, H. J.; and Shin, S. Y. 2002. On-line locomotion generation based on motion blending. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 105–111. ACM.

Shaker, N.; Togelius, J.; Yannakakis, G. N.; Weber, B.; Shimizu, T.; Hashiyama, T.; Sorenson, N.; Pasquier, P.; Mawhorter, P.; Takahashi, G.; et al. 2011. The 2010 mario AI championship: Level generation track. *TCIAIG, IEEE Transactions on* 3(4):332–347.

Smith, G.; Treanor, M.; Whitehead, J.; and Mateas, M. 2009. Rhythm-based level generation for 2D platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, 175–182. ACM.

Snodgrass, S., and Ontañón, S. 2014. Experiments in map generation using markov chains.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2010. Search-based procedural content generation. In *Applications of Evolutionary Comp.* Springer. 141–150.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2011. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on* 3(3):172–186.

Togelius, J.; Karakovskiy, S.; and Baumgarten, R. 2010. The 2009 mario AI competition. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, 1–8. IEEE.