# Experiments in Map Generation using Markov Chains

Sam Snodgrass
Drexel University
Department of Computer Science
Philadelphia, PA, USA
sps74@drexel.edu

Santiago Ontañón
Drexel University
Department of Computer Science
Philadelphia, PA, USA
santi@cs.drexel.edu

## ABSTRACT

In this paper we describe a method of procedurally generating maps using Markov chains. This method learns statistical patterns from human-authored maps, which are assumed to be of high quality. Our method then uses those learned patterns to generate new maps. We present a collection of strategies both for training the Markov chains, and for generating maps from such Markov chains. We then validate our approach using the game *Super Mario Bros.*, by evaluating the quality of the produced maps based on different configurations for training and generation.

## Categories and Subject Descriptors

I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems—*Games*; G.3 [**Probability and Statistics**]: [Markov processes]

## General Terms

Algorithms, Experimentation

## 1. INTRODUCTION

Manually creating maps for games is expensive and time consuming [18]. Delegating map generation to an algorithmic process can save developers time and money, 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 and experiment with an approach to procedurally generating maps using Markov chains. Specifically, we focus on generating two-dimensional maps for the platformer game, *Super Mario Bros.* We chose to use Markov chains because, after slight alterations, they can easily represent two dimensional models, which is often how maps are represented in many game genres. Our method learns a statistical model from known high quality maps, and then uses this model to generate new maps with similar characteristics. We discuss different strategies to train Markov chains from two-dimensional maps, and also different strategies to use the learnt Markov chain for map generation.

Concerning Markov chain training, we present and explore the idea of learning higher-order Markov chains based on a *dependency matrix*, which captures the dependencies between the tiles in the map. Concerning map generation, we propose a method that combines the trained Markov chain with look-ahead and fallback strategies, in order to maximize map quality. The work presented in this paper builds on our previous work on map generation [14]. The key differences with respect to our previous work include an improved map representation, a new extension to the map generation procedure based on the idea of fallback strategies, and an extended empirical evaluation of our map generation process using the 2009 Mario AI competition software [16].

The remainder of the paper is organized as follows. In Section 2 we give some background and related work on procedural content generation as well as Markov chains. Section 3 describes our strategies for training Markov chains and for using them for map generation. Section 4 presents an experimental evaluation. The paper closes with conclusions and directions for future work.

## 2. BACKGROUND

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

### 2.1 Procedural Map Generation

Procedural content generation (PCG) refers to methods for creating content algorithmically instead of manually [19]. Such methods can be used in games to generate components like maps [18], quests [1], animations [10], and textures [6], among others. For a survey of procedural content generation techniques the reader is referred to Hendrikx et. al. [4] In this paper, we focus on procedurally generating maps.

Most PCG approaches can be classified into three broad categories: Search-based, learning-based, and tiling. 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. [13] developed an approach that generates platformer maps based on the rhythm that the map should achieve and the player's available actions. Let us review existing work on map generation in each of these three categories.

Search-based PCG (SBPCG) techniques rely on defining the space of all potential maps, missions, etc. we want to generate, and then exploring that space using some search technique (for example, a genetic algorithm). SBPCG methods require the use of an evaluation function that can estimate the quality of each element in the search space. Examples of this work include the use of evolutionary algorithms for generating puzzle-game levels [11]. Specially related to the work presented in this paper is the generation of two-dimensional maps for platformer games [15]. A key aspect of this family of PCG methods is to define appropriate evaluation functions, that can guide the search [17, 20]. The reader is referred to Togelius et al. [19], for an in-depth overview of search-based approaches.

Learning-based approaches to PCG take advantage of existing data by using algorithms to extract models from it. Those models, or patterns, are then used to generate new content. The existing data can be information provided by the user or designer, player data, or it can be known high quality models of what the method is trying to generate. Shaker et al. [12] outline different methods that learn a player type by watching that player go through a level.

Tiling is an approach that builds up content from smaller parts, called "tiles." These tiles are then selected and pieced together algorithmically. Techniques in this category use different sized tiles and different methods of assembly. Compton et al. [3] use a tiling approach to build levels for a platform game. This technique is also used in well known games, such as *Spelunky*[1]. The two critical aspects of this family of techniques is to define a good and rich enough set of tiles, and also to define enough constraints on how to combine those tiles, in order to provide interesting levels.

Our method employs concepts from search-based, learning-based, and tiling approaches. Our method learns from known high quality maps. Using that information it generates a map from tiles. Additionally, our method has the ability to backtrack while generating the new map, a concept commonly found in search algorithms.

## 2.2 Markov Chains

Markov chains [8] are a method of modeling probabilistic transitions between different 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)[9].

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 [2]. 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 that the Markov chain was in the previous $k$ instants of time. Figure 1 shows a graphical representation of the dependencies in a first and a second order Markov chain.

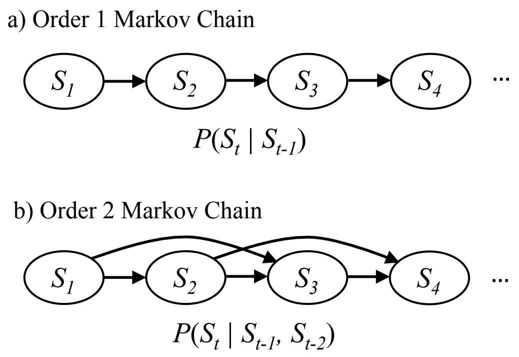


**Figure 1: An illustration of: a) a standard Markov Chain, and b) a Markov Chain of order 2.**

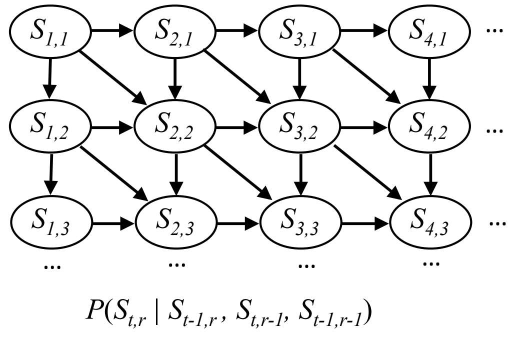


**Figure 2: A two dimensional representation of a higher order Markov chain.**

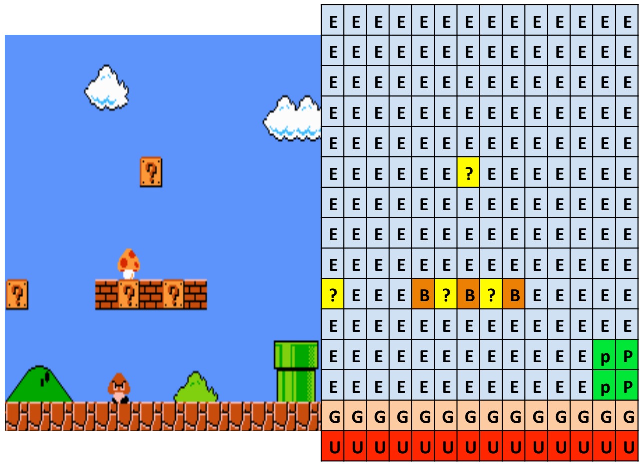
In our application domain, we structure the variables in the Markov chain in a two-dimensional array, in order to suit the map generation application. Figure 2 shows an illustration of this, showing dependencies between the different state variables in an example Markov chain of order 3 (each state variable depends on 3 previous state variables). Notice that Figure 2 is only an example, and we can define Markov chains of different orders with different dependency patterns (for example, where state variables only depend on two variables immediately to the left, etc.).

As elaborated below, in this paper we will use higher order Markov chains to build models of two-dimensional maps, and then use those models for generating new maps that exhibit the same statistical properties of the maps used to train the Markov chain. Moreover, as we will see, the biggest challenge in using higher order Markov chains, is that the amount of data required to estimate the probability distribution grows exponentially with the order $k$.

## 3. METHODS

In this section we discuss how we represent maps, how our model learns from those maps, and finally how our model generates new maps after learning.

[1]`http://spelunkyworld.com`



| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E | E | E | E | E | E | E | E | E | E | E | E | E | E |
| E | E | E | E | E | E | E | E | E | E | E | E | E | E |
| E | E | E | E | E | E | E | E | E | E | E | E | E | E |
| E | E | E | E | E | E | E | E | E | E | E | E | E | E |
| E | E | E | E | E | E | E | E | E | E | E | E | E | E |
| E | E | E | E | E | E | ? | E | E | E | E | E | E | E |
| E | E | E | E | E | E | E | E | E | E | E | E | E | E |
| E | E | E | E | E | E | E | E | E | E | E | E | E | E |
| E | E | E | E | E | E | E | E | E | E | E | E | E | E |
| ? | E | E | E | B | ? | B | ? | B | E | E | E | E | E |
| E | E | E | E | E | E | E | E | E | E | E | E | E | E |
| E | E | E | E | E | E | E | E | E | E | E | E | p | P |
| E | E | E | E | E | E | E | E | E | E | E | E | p | P |
| G | G | G | G | G | G | G | G | G | G | G | G | G | G |
| U | U | U | U | U | U | U | U | U | U | U | U | U | U |

**Figure 3: A section from a map we use in our experiments (left) and how we represent that map in an array (right). We added one row above and underneath for Markov chain training purposes. Color has been added to our representation for clarity.**

## 3.1 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 set of values $S$, which represent the different tile types. When modeling maps using Markov chains, the different tile types correspond to the different states of the Markov chain. In general, what constitutes a "tile type" depends on the domain and the discretion of the designer. In the experiments presented in this paper we had nine different tile types (representing the different elements of the maps, such as air, ground, walls, etc.), described in Section 4.

Figure 3 shows a section of a map we use in our experiments (left) from the *Super Mario Bros.* game, and the representation of the map as an array (right), where each letter represents a different tile type. Currently, we only consider the map layout, without taking enemies into account.

## 3.2 Learning

Our method employs higher order Markov chains in order to learn the probabilistic distribution of tiles in a given set of maps. We assume that maps given as input for learning are of high quality. In order to learn the Markov chain, we need to specify which previous states the current state depends on, i.e. we need to specify 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 one defined by the probability of a tile given the previous tile horizontally and the tile immediately above, etc. Automatically learning the structure of a Bayesian network is a well known hard problem [5], thus, in our approach, we configure the dependencies by hand. Our learning method takes as input a $n \times n$ dependency matrix $D$, defined as follows:

- $D(n,n) = 2$, for the tile that we are going to generate.
- $D(i,j) = 1$ if the probability of the tile we are going to generate depends on the tile $(n-i)$ cells to the left, and $(n-j)$ tiles above.
- Otherwise, $D(i,j) = 0$.

In the experiments reported in this paper, we used several dependency matrices. For example, the Markov chain in Figure 2 results from the following dependency matrix (where each tile depends on the tile immediately to the left, the one immediately above, and the one to the left and above):

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

Moreover, notice that in the way we have just described it, the resulting Markov chain learns the probability distribution of a given tile, based on a subset of the "previous" tiles (determined by the dependency matrix), where "previous" is defined as being anywhere to the left and or up from the tile at hand, we call this the *top-down* learning direction. However, we could flip the vertical component and learn maps in a bottom-up manner (learning the dependencies of a tile with the tiles below it, instead of with the tiles above it). We will present experiments with learning Markov chains top-down and bottom-up.

Given a dependency matrix $D$, the learning direction (top-down or bottom-up), and a set of maps represented by the arrays $M_1, ...M_m$, our method learns the Markov chain 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 will take into account, corresponding to the order of the Markov chain). If there are $n$ different tile types, there are $n^k$ different previous tile configurations. Our method counts the total number of times that each tile type $s_i$ appears in all the input maps for each of the $n^k$ previous tile configurations, which we will refer to as $T(s_i|S_{i-1}, ..., S_{i-k})$.

2. Probability Estimation: once these 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:

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

   However, other approaches, such as a Laplace smoothing [7, p. 226] can be used in case there is not enough input data to have an accurate estimation of the probabilities. Moreover, it is important to keep both the probability estimation as well as the absolute counts, in order to be able to determine the confidence with which the probabilities were estimated. If we had very few instances to estimate a given probability, its estimation is not going to be reliable.

Finally, in our experiments, we observed that different parts of the maps have different statistical properties. For example, when looking at *Super Mario Bros.* maps, it is highly unlikely to have a "pipe" towards the top of the map. For that reason, we experimented with learning separate probability distributions for different parts of the map, by splitting the map using horizontal cuts. Specifically, our learning method has an input parameter $R$, that determines the number of horizontal splits. For example, if $R = 1$, a single probability distribution is learned from the whole map. If $R = 2$, the map is split in two (the upper part, and the lower part), and a separate probability distribution is learned for each section.

## 3.3 Map Generation

Our method generates a new map one tile at a time, starting from the top-left (if a top-down learning direction is used), or from the bottom-left (if a bottom-up learning direction is used) and generating one row at a time. In order to generate a tile, the method selects a tile probabilistically, based on the probability distribution learned before.

Moreover, it is possible to encounter a combination of previous tiles that was never seen during training (or that was seen only a very small number of times). We call this an *unseen state*. In general, we can define an *unseen state* as a combination of previous tiles that was observed less than a fixed number of times $U$, in our experiments we used $U = 1$. When encountering an unseen state, the probability estimation of the Markov chain will not be very accurate, and our method must generate a tile randomly. Therefore, we would like to minimize the number of times we encounter unseens states during map generations.

In order to avoid unseen states, we incorporated two different strategies into our map generation procedure:

- **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 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 the one chosen.

- **Fallback Strategies**: When the look-ahead process fails, we allowed our system to fallback to use a second Markov chain, trained with a different dependency matrix $D^{fb}$ that is assumed to be a simpler configuration than the initial dependency matrix $D$. Because $D^{fb}$ is a simpler configuration than $D$, it results in a Markov chain of lower order, and thus has a smaller chance of finding 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. Therefore, in order to have a fallback strategy, we train two Markov chains, one with $D$ and one with $D^{fb}$. If generating a tile using the Markov chain learned with $D$, with the look-ahead process, fails, then we try to generate a tile using the Markov chain learned with $D^{fb}$, with the same look-ahead and backtrack. If that fails, then our method is forced to generate a tile randomly.

## 4. EXPERIMENTS

We chose to use the classic two dimensional platformer game *Super Mario Bros.* as our application domain for three reasons: 1) maps are readily available, 2) a simulator is publicly available that lets us test the maps generated by our method, and 3) popularity (since people are familiar with the game, they know what to expect from a level).

To represent the Super Mario Bros. maps we chose to use nine tile types. The first three types are special tiles to signify the start, end, and underneath of the map. We called them **S**, **D**, and **U** respectively. 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.

### 4.1 Experimental Set-up

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

- *Row Splits (R)*: When $R = 1$, the entire map is used to train the Markov chain. When, $R = 2$, we divide each map into two parts: an upper part and a lower part, and train two independent Markov chains, one for each split. Analogously, when $R = 3$, we divide the maps in three parts (upper, middle and lower sections). We experimented with $R \in \{1, 2, 3, 4, 5\}$. The intuition behind this is that different portions of the maps have inherently different statistical properties (e.g. it's more likely to have ground tiles in the bottom of the maps).

- *Dependency Matrix (D)*: We used six different dependency matrices in our experiments. $D_1$ takes into account only the tile immediately to the left of the current one. $D_2$ takes into account the tile immediately to the left, and the one immediately above. $D_3$ takes into account the two tiles to the left of the current one. $D_4$ takes into account the tile immediately to the left, and the one above that tile. $D_5$ takes into account the tile to the left, the one above, and the one left and above (i.e. this one corresponds to the model shown in Figure 2). Finally, $D_6$ takes into account the two tiles to the left of the current tile, and the tile above the current tile. This is illustrated in Figure 7.

- *Look-ahead (d)*: we experimented with six different values of look-ahead, $d \in \{0, 1, 2, 3, 4, 5\}$

- *Order of Generation*: as described in Section 3.3, our method generates maps one tile at a time, starting from the top-left corner, and generating one row at a time. We experimented with an alternative method, where we flip vertically the dependency matrices, and generate maps in the reverse order, starting from the bottom row. The intuition behind this is that the bottom rows of the map are more likely to influence the top rows, than the other way around.

- *Fallback Strategy*: during map generation, sometimes the trained Markov chain reaches a state that was

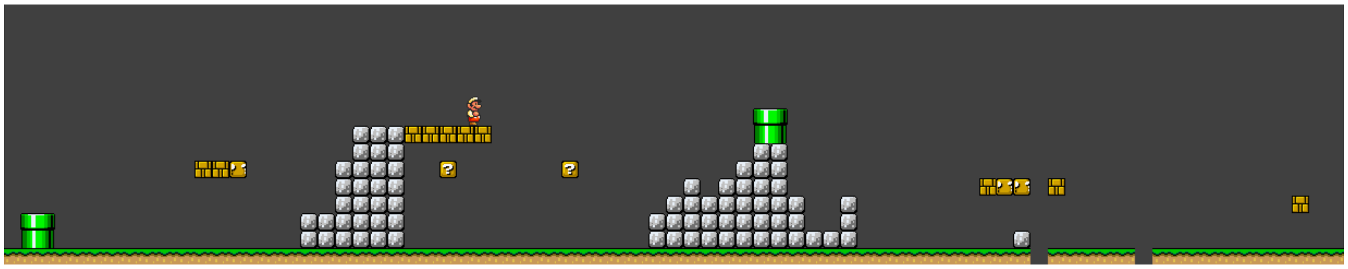


**Figure 4: A section from a map generated using $D_5$ with $R = 4$ and $d = 3$, bottom up and using a fallback to matrix $D_2$. Which is the "baseline" method we used in our experimentation.**

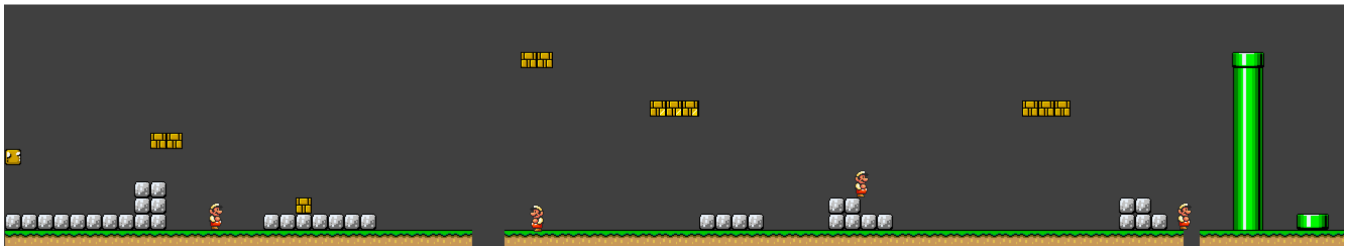


**Figure 5: A section from a map generated using $D_5$ with $R = 1$ and $d = 3$, bottom up and using a fallback to matrix $D_2$. Showing a very tall pipe, which makes the map unplayable.**

never observed during training. In these situations, we experimented with two alternatives. *fallback* to a simpler Markov chain: which trains two Markov chains, one with a complex matrix, and one with a simple matrix. The complex matrix is used, except when an unseen state is reached, at which point, the method defaults to the simpler matrix. *no fallback*: when an unseen state is reached, a tile is generated at random.

To test all of these configurations, we selected a *baseline* configuration that resulted in good results during our experiments. The baseline is configured as: $R = 4$, using matrix $D_5$, $d = 3$, generating maps bottom up, and using a *fallback* to matrix $D_2$. We then varied the value of each of the different variables one by one to evaluate their effect on map generation. Figure 4 shows a section from a map generated using our baseline approach, showing that the generated maps look very much like the original Super Mario Bros.[2] maps. Figure 5 shows a map generated without row splits ($R = 1$), which results in the generator not being able to successfully learn that platforms tend to be lower in the map, and that tall pipes are not common. Finally, Figure 6 shows a section from a map generated from the top down instead of the bottom up. This results in some malformations in the map, such as incorrect pipes on the right hand side of the map, and some strange floor configuration on the left-hand side of the map. This is the result of generating lower rows of the map after generating the higher rows.

In order to evaluate each configuration, we generated 25 maps of width 320 with each one of them and used five different metrics to assess their quality:

- *Backtracks*: average number of backtracks due to the look-ahead. This measures how many times the tile with the highest probability according to the Markov chain could not be selected, since it would create a problem later on (this counts not just the number of backtracks at the top level of the search tree, but also all the internal backtracks).
- *Fallback*: average number of times our method generated a tile with the fallback dependency matrix
- *Random*: average number of times our method had to resort to generating a tile at random.
- *Bad Pipes*: in earlier version of our system, we observed that it struggled in generating properly formed "pipes" in the maps (pipe tiles cannot appear in any configuration, but only in blocks of even width, and they cannot "float" in the air, but need to extend all the way down to the ground). 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, so we did not include any rules of that kind.
- *% Playable*: in order to test whether the generated maps were actually playable. We loaded all the maps generated by our system into the 2009 Mario AI competition software [16], and made Robin Baumgarten's $A^*$ controller play through our maps. Given an upper limit of 200 seconds, we recorded the percentage of maps that this controller was able to complete with each configuration. However, this agent occasionally fails to complete maps that are completable. If a map has an overhang with a dead end underneath, the agent may get stuck under the overhang, without attempting

[2]Notice that "mountains" in these figures are rendered as stacked rocks, since we implemented only a simple script to translate our generated maps into Super Mario Bros.

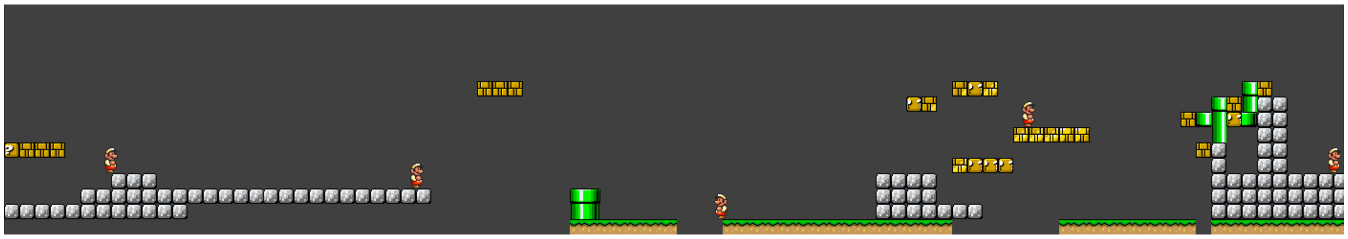


**Figure 6: A section from a map generated using $D_5$ with $R = 4$ and $d = 3$, top down and using a fallback to matrix $D_2$. Showing some bad pipes being generated.**

$$D_1 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 2 \end{pmatrix} \quad D_2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 2 \end{pmatrix} \quad D_3 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 2 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 2 \end{pmatrix} \quad D_5 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix} \quad D_6 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 2 \end{pmatrix}$$

**Figure 7: The different dependency matrices considered in our experiments.**

to navigate over it. This may skew the percentage of playable maps to be lower than the actual value.

- *Play Time*: this represents the average time the $A^*$ player controller required to complete the playable maps.

The following subsections describe the results we obtained by varying each of the variables in our experiments.

## 4.2 Row Splits

**Table 1: Effect of changing the number of row splits.**

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| *Backtracks* | 23.68 | 34.04 | 456.36 | **84.56** | 5986.12 |
| *Fallback* | 0.00 | 1.12 | 5.00 | **1.68** | 344.64 |
| *Random* | 0.00 | 0.00 | 29.56 | **0.00** | 1454.52 |
| *Bad Pipes* | 0.00 | 0.00 | 1.88 | **0.00** | 153.16 |
| *%Playable* | 0.00 | 28.00 | 40.00 | **44.00** | 56.00 |
| *Play Time* | N/A | 39.43 | 42.90 | **41.18** | 41.14 |

Table 1 shows the results we obtained by varying the number of row splits from 1 to 5. The baseline configuration is shown in bold. One clear trend that we can observe is that as the number of row splits grows, the number of backtracks increases. This can be explained by the fact that as the number of row splits grows, there is less data to train each Markov chain, and the likelihood of encountering unseen states increases. There is only one outlier to this trend ($R = 3$), which has a disproportionate number of backtracks. We believe this is due to the rows in which we performed the splits, which might split rows that have strong dependencies between them. The reduction in the amount of data to train each Markov chain, also explains the increase in the number of bad pipes being generated with high number of splits ($R = 5$), and the number of times our system needs to resort to the fallback matrix and to random tile generation.

Finally, we observed that with $R = 1$, no map was playable (due to either walls that were too tall, or gaps that were too wide). However, as soon as we add some splits ($R \geq 2$), playable maps are generated. For example, 44% of the maps generated by our baseline configuration were directly playable. Using $R = 5$ generated more playable maps, but those maps included many mistakes such as bad pipes. Since the currently generated maps do not include enemies, we did not observe much difference in the amount of time it takes to complete a level.

The conclusion is that having some row splits really helps in map generation, since different parts of the map might exhibit different statistical properties, but too many row splits, reduces the amount of data for training too much.

## 4.3 Dependency Matrix

Table 2 shows the effect of using different dependency matrices (shown in Figure 7). The dependency matrix defines the Markov chain to be learned, and thus has a strong impact on the maps being generated. As we can see, the number of backtracks, number of bad pipes, and percentage of playable maps very strongly depends on the matrix being used. For example, simple matrices (like $D_1$) result in Markov chains that do not take into account a sufficient number of tiles, which resulted in unplayable maps. Matrices that take into account more dependencies, such as $D_5$ and $D_6$ can generate a high number of playable maps, and with a very low number of visual mistakes (as can be seen by the low number of bad pipes). Notice that we did not have a fallback strategy for matrix $D_1$, since $D_1$ is the simplest matrix we had.

The conclusion is that simpler dependency matrices are not enough to capture the complexity of Super Mario Bros. maps. Moreover, in our experiments, we were not able to produce any meaningful results with matrices that were more complex than the ones presented in this paper (matrices that resulted in Markov chains of order 4 or higher could not be successfully trained due to the lack of data).

## 4.4 Look-ahead

Table 3 shows the results obtained with different look-ahead values. A clear trend was observed: increasing the length of

**Table 2: Effect of changing the dependency matrix.**

| | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ |
|---|---|---|---|---|---|---|
| *Backtracks* | 99.52 | 99.12 | 1530.32 | 340.88 | **84.56** | 1015.40 |
| *Fallback* | N/A | 1.40 | 47.36 | 0.16 | **1.68** | 44.48 |
| *Random* | 0.00 | 0.00 | 18.76 | 0.00 | **0.00** | 31.20 |
| *Bad Pipes* | 28.00 | 22.36 | 26.92 | 22.20 | **0.00** | 1.90 |
| *%Playable* | 0.00 | 28.00 | 4.00 | 60.00 | **44.00** | 60.00 |
| *Play Time* | N/A | 42.00 | 35.50 | 48.20 | **41.18** | 39.86 |

**Table 3: Effect of changing the length of the look-ahead.**

| | 0 | 1 | 2 | **3** | 4 | 5 |
|---|---|---|---|---|---|---|
| *Backtracks* | N/A | 32.72 | 56.40 | **84.56** | 111.00 | 164.52 |
| *Fallback* | 40.28 | 4.40 | 1.88 | **1.68** | 0.36 | 1.84 |
| *Random* | 18.40 | 0.00 | 0.04 | **0.00** | 0.00 | 0.00 |
| *Bad Pipes* | 12.20 | 0.56 | 0.00 | **0.00** | 0.00 | 0.00 |
| *%Playable* | 36.00 | 36.00 | 40.00 | **44.00** | 40.00 | 36.00 |
| *Play Time* | 41.89 | 41.22 | 41.60 | **41.18** | 42.70 | 41.00 |

look-ahead reduces the number of times we need to resort to the fallback strategy or to random tile generation. Moreover, we also observed that the worst results were obtained using a look-ahead $d = 0$ (only 36% playable maps and an average of 12.20 bad pipes), and that increasing the look-ahead beyond 3 did not accomplish further improvements in results.

The conclusion is that look-ahead is greatly beneficial during sampling (in opposition to just randomly sampling the probabilities in the Markov chain), but that a small amount of look-ahead is enough. Higher values of look-ahead just increase the computational complexity of the method without further yielding benefits.

## 4.5 Order of Generation

**Table 4: Effect of changing from bottom up generation to top down.**

| | **Bottom-Up** | Top-Down |
|---|---|---|
| *Backtracks* | **84.56** | 538.48 |
| *Fallback* | **1.68** | 26.04 |
| *Random* | **0.00** | 23.64 |
| *Bad Pipes* | **0.00** | 3.88 |
| *%Playable* | **44.00** | 0.00 |
| *Play Time* | **41.18** | N/A |

Table 4 shows the results of generating maps bottom-up versus generating them top-down. As we can see, in the case of Super Mario Bros. generating maps bottom-up is clearly superior to generating maps top-down. This can be explained by the fact that it makes more sense to generate the higher rows of the map after generating the lower rows (containing the ground), than the other way around. When maps are generated top-down, our system first generates the sky, then the middle rows, and only at the very end it generates the bottom rows containing the floor, which intuitively seems to be the wrong order of generation.

## 4.6 Fallback Strategy

Finally, Table 5 shows the results both with and without a fallback strategy. The goal of the fallback strategy is to prevent generating tiles at random. As can be seen in Table 5, generating these tiles at random, results both in bad pipes, and in a significantly lower percent of playable maps. We can conclude that having a fallback strategy is extremely useful, and can significantly help in generating higher quality maps. In fact, we could take this concept further, and devise a sequence of matrices of decreasing complexity. If the most complex matrix doesn't work, we can fallback to the next matrix, and if that one doesn't work, then we move on to the next. Thus, minimizing the number of random tiles generated even more, but allowing us to use as complex a matrix as possible.Experimenting with this kind of complex fallback strategies is part of our future work.

**Table 5: Effect of using a simpler dependency matrix if the original fails.**

| | **Fallback** | No Fallback |
|---|---|---|
| *Backtracks* | **84.56** | 90.32 |
| *Fallback* | **1.68** | N/A |
| *Random* | **0.00** | 7.24 |
| *Bad Pipes* | **0.00** | 0.28 |
| *%Playable* | **44.00** | 4.00 |
| *Play Time* | **41.18** | 42.60 |

## 5. CONCLUSIONS

We developed a method for procedurally generating maps using variations of Markov chains as a tool for both learning and generation. Our method learns from established high quality maps in order to generate statistically similar maps.

We incorporate look-ahead, backtracking, and a fallback strategy into our map generation method. This improves the quality of the generated maps, by allowing the use of higher order Markov chains whenever possible, and only defaulting to lower order Markov chains when necessary. Lastly, our method includes the ability to split the maps used for learning into different horizontal slices. Doing so allows our method to isolate certain qualities of the map that may be exclusive to specific portions of the given maps.

Our method gave strong results. Using our baseline configuration, the $A^*$ controller was able to complete 44% of maps generated (which does not necessarily mean that the remaining 56% were not playable). We would like to emphasize, in contrast with search-based generation procedures, map generation using our method (when small values for $d$ like 0, 1, 2 or 3) is almost instantaneous, making it amenable for in-game uses. Furthermore, adding a fallback strategy and a different order of generation has helped ensure that the maps generated do not have any ill-formed structures. Notice that we did not include any sort of additional hard-coded knowledge in our method, and that the generated maps are one-hundred percent generated based on the learned probability distribution in the Markov chain. This shows that

Markov chains are a viable method for procedurally generating playable, well-formed maps. If these methods were to be incorporated into an actual game, additional rules to detect malformed structures and unplayability should be added.

We would like to note that though the generated maps have probability distributions similar to the maps our method learns from, this does not imply the generated maps are structurally the same. By using different configurations, our method is capable of generating maps which are significantly different both from the learning data and maps generated with other configurations. This is due to learning and generating locally, on the tile level, instead of globally, on the map level. Furthermore, though maps may share some similar objects, pipes or platforms for example, the way the objects are combined and placed together in map can yield starkly different results.

Additionally, though we tested our approach with *Super Mario Bros.* it is applicable as is to any game with linear maps. In order to apply our method to *Wonder Boy*, for example, we would simply need to encode the maps into tile types. Then we would run those maps through our system to learn the probability distributions of the tiles and to generate new maps. Furthermore, one area of future work for our method is to expand it to be able to handle non-linear maps, such at *Metroid* or *Megaman*.

In the future, we want to explore better ways to judge whether a map is playable, and to perform user studies to determine whether the maps are actually enjoyable. Additionally, we currently only generate the level layout, without including enemies; which we plan to experiment with in the future, by modeling enemies as just another type of tile. Finally, we would like to experiment with applying naive Bayes approximations to very high-order Markov chains (e.g. order 10 or even 20), to explore whether higher order dependencies compensate for the loss in the approximation of the probability distributions during map generation. We also plan to explore hierarchical models for learning the overall structure of the map as well as the details.

## 6. REFERENCES

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

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

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

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

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

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

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

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

[9] K. P. Murphy. *Dynamic bayesian networks: representation, inference and learning*. PhD thesis, University of California, 2002.

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

[11] M. Shaker, N. Shaker, and J. Togelius. Evolving playable content for cut the rope through a simulation-based approach. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013.

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

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

[14] S. Snodgrass and S. Ontañón. Generating maps using markov chains. In *AIIDE 2013 workshop on AI for Game Aesthetics*, pages 25–28. AAAI, 2013.

[15] N. Sorenson and P. Pasquier. Towards a generic framework for automated video game level creation. In *Applications of Evolutionary Computation*, pages 131–140. Springer, 2010.

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

[17] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelback, and G. N. Yannakakis. Multiobjective exploration of the starcraft map space. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 265–272. IEEE, 2010.

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

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

[20] A. Uriarte and S. Ontanón. Psmage: Balanced map generation for starcraft. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.