Generating Maps Using Markov Chains

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

Abstract

In this paper we outline a method of procedurally generating maps using Markov Chains. Our method attempts to learn what makes a "good" map from a set of given human-authored maps, and then uses those learned patterns to generate new maps. We present an empirical evaluation using the game *Super Mario Bros.*, showing encouraging results.

Introduction

Manually creating maps for games is expensive and time consuming (Togelius et al. 2010). Delegating map generation to an algorithmic process can save developers time and money. Using such algorithmic processes is called procedural content generation (PCG), which generally refers to methods for generating all types of content.

In this paper we discuss our approach to procedurally generating maps using Markov chains. We chose to use Markov chains because, after slight alterations, they can easily represent two dimensional models, which is often how maps are represented. Our method learns patterns from known "good" maps, and then uses what it has learned to generate new maps with similar patterns. In addition to Markov chains, our approach implements backtracking during the map generation stage to allow for change if an undesirable state is reached. Our method also allows maps to be split horizontally to isolate patterns in specific portions of the maps.

Background

In this section we provide necessary background 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) or missions (Bakkes and Dormans 2010). In this paper, we focus on procedurally generating maps. PCG approaches can be classified into three broad categories: Search-based, learning-based, and tiling. These categories, however, are not mutually exclusive or complete. There are hybrid methods and methods which do not fit any of these 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. The reader is referred to Togelius et al. (2011), 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, or patterns, 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 "good" models of what the method is trying to generate. Shaker et al. (2011) 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 reassembly. Compton et al. (2006) use a tiling approach to build levels for a platform game. This technique is used in well known games, such as *Spelunky*[1].

Our method uses concepts from each of the above categories. Our method learns from known 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.

Markov Chains

Markov chains (Markov 1971) 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 $P(S_t|S_{t-1})$, representing the probability of transitioning to a state S_t given that the previous state was S_{t-1}.

Standard Markov chains restrict the probabilities to only take into account the previous state. Higher order Markov chains relax this condition by taking into account k previ-

[1]http://spelunkyworld.com

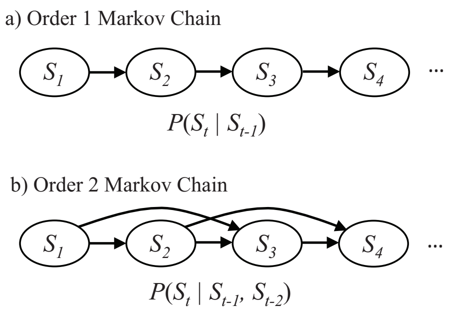


Figure 1: An illustration of: a) a standard Markov Chain, and b) a second order Markov Chain.

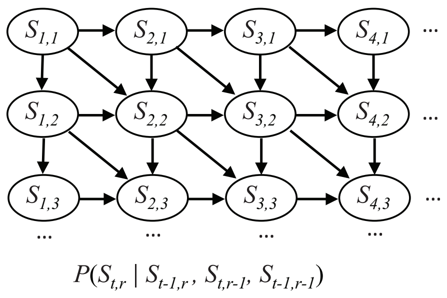


Figure 2: A two dimensional representation of a higher order Markov chain.

ous states, where k is a finite natural number (Ching et al. 2013). This yields $P(S_t|S_{t-1},...,S_{t-k})$. 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.

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

Methods

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 of the positions in the array can take one of a finite set of values S, which represent the different tile types (and which we will make correspond to the different states in a Markov chain, when we learn the model). Figure 3 shows a section of a map we use in our experiments (left) and the representation of the map as an array (right), where each letter represents a different tile type. We only consider the map layout, without taking enemies into account.

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 "good." In order to learn the Markov chain, we need to specify which previous states the current state depends on. 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 previous tile vertically, etc. In order to configure these dependencies, our learning method takes as input a small dependency matrix D. $D_{i,j} = 1$ if the probability of the tile $M_{x,y}$ depends on the tile $M_{x-i,y-j}$. Otherwise, $D_{i,j} = 0$ (by convention $D_{2,2} = 2$, representing the tile to be learned).

In the experiments reported in this paper, we used several such dependency matrices. For example, the Markov chain in Figure 2 results from the following dependency matrix:

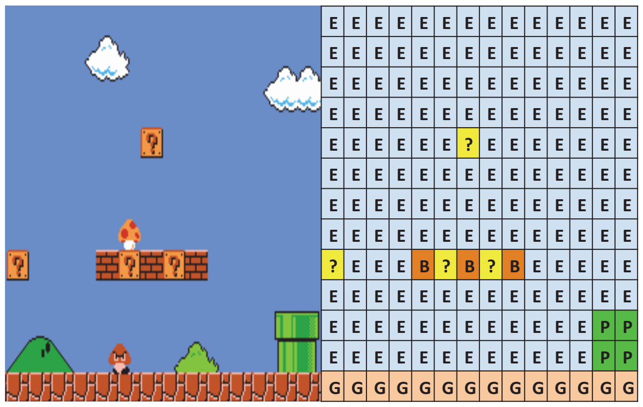
$$D_5 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix}$$

Given a dependency configuration matrix D, and a set of maps represented by the arrays $M_1, ...M_m$, our method learns the Markov chain in two stages:

- Absolute Counts: let p be the number of 1's in the matrix D (i.e. how many past states the model will take into account). If there are n different tile types, there are n^p 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^p previous tile configurations, which we will refer to as $T(s_i|S_{i-1},...,S_{i-p})$.
- 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:

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

However, other approaches, such as a Laplace smoothing (Manning, Raghavan, and Schutze 2009, p. 226) can be used, in case there is not enough input data to have an accurate estimation of the probabilities.



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

Figure 3: A section from a map we use in our experiments (left) and how we represent that map in a array (right). Color has been added to our representation for clarity.

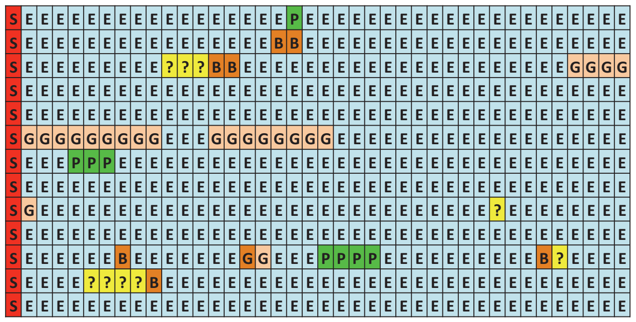


Figure 4: A section from a map generated using D_1 with $R = 1$ and $h = 0$.

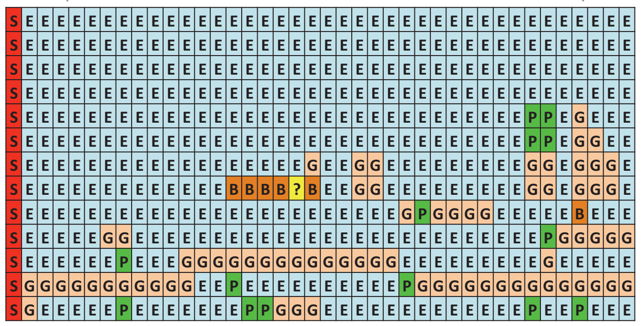


Figure 5: A section from a map generated using D_6 with $R = 3$ and $h = 5$.

Finally, in our experiments, we observed that different parts of the maps have different statistical properties. For that reason, we experimented with learning separate Markov chains 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 splits. For example, if $R = 1$, a single model is learned from the whole map. If $R = 2$, the map is split in two (the upper part, and the lower part), and a model of each part is learned.

Our method generates a new map one tile at a time, starting from the top-left, 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. If a combination of previous tiles has never been seen before (and thus, the probability distribution of tiles is undefined), all the tile types are assumed to have equal probabilities. We call this situation an *unseen state*.

In order to avoid unseen states, we experimented with a map generation method that used look ahead and backtracking. Given a fixed number of tiles to look ahead $d >= 0$, 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. If the search process ends without finding a tile for which we do not reach an unseen state, then our method is forced to generate a tile randomly.

Experiments

We chose to use the classic two dimensional platformer game *Super Mario Bros* for two reasons: 1) maps are readily available, and 2) 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 seven tiles. The first two tiles are special tiles to signify the start and end of a map. We called them **S** and **D**, respectively. The remaining five tiles correspond to components of the maps: **G** is ground, **B** are breakable blocks, **?** are power-up blocks, **P** are pipes. and **E** is empty space.

For our experiments, we had our model learn from 12 maps taken from the original Super Mario Bros. We excluded indoor and underwater maps. We ran our method with different input parameters, as described below.

We used six different dependency matrices in our experiment: D_1 takes into account only the tile immediately to the left of the current one. D_2 takes into account the two tiles to the left of the current one. D_3 takes into account the tile immediately to the left, and the one immediately above. 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.

We also ran experiments with a varying number of splits, $R \in \{1, 3, 6\}$. One split uses the entire map to generate the probabilities. Three row splits break the map into an upper, middle, and lower section. Six splits essentially break the maps into pairs of rows. Lastly, we experimented with multiple depths for the look ahead in the generation stage. We tested with $d \in \{0, 1, 3, 5\}$.

To test these different combinations, we had our method generate 50 maps for each input combination. While generating the maps, we tracked how many times the method ultimately reached an unseen state (random tiles), and how many times our method needed to backtrack.

Figures 4 and 5 show sections from two maps generated with our approach using different input parameters. Figure 4 shows a map using a very simple Markov chain (only taking into account the tile immediately to the left), with only one split, and without any look ahead when generating the map. This results in an unplayable map. However, Figure 5, generated with a more complex Markov chain shows a playable map (with a few misplaced pipe pieces). This shows that even with very little training data (only 12 maps), Markov chains can actually be used for map generation.

Table 1 shows the numerical results we obtained from our experiments. In the table “Rand” stands for “average number of randomly generated tiles per map” and “BT” stands for “average number of times backtracking occurs per map.” The first trend we see is as the depth of the look ahead, d, increases, so does the average number of backtracks (BT). This can be seen by comparing the average number of backtracks for D_2 with $R = 1$ as d increases. The reason for this

Table 1: Experimental Results

R	d	D_1		D_2		D_3		D_4		D_5		D_6	
-	-	Rand	BT	Rand	BT	Rand	BT	Rand	BT	Rand	BT	Rand	BT
1	0	0	0	0	0	6.04	0	10.22	0	11.88	0	41.2	0
	1	0	27.54	0	27.46	4.68	9.18	0.34	28	6.44	10.71	25.64	48.92
	3	0	84.44	0	81.01	18.66	276.24	0.14	82.7	13.02	110.88	10.94	211.9
	5	0	138.62	0	135.04	12.94	1051.24	0.14	135.48	18.9	1041.68	13.52	774.56
3	0	0	0	0	0	16.62	0	77.66	0	91.08	0	160.28	0
	1	0	16	0	27.62	5.82	12.96	32.54	104.14	11.54	160.84	73.76	183.74
	3	0	49.96	0	82.24	10.28	198.76	57.5	748.06	16.7	567.52	5.78	656.12
	5	0	81.84	0	138.94	11.94	2240.62	150.76	14522.82	28.06	2201.9	0.84	1156.1
6	0	0.06	0	2.16	0	17.6	0	68.16	0	73.6	0	105.72	0
	1	0	14.44	0	28.04	15.32	26.34	26.66	104.44	44.02	64.96	54.78	131.72
	3	0	43.72	0	86.14	34.76	366.82	48.54	781.76	73.38	581.52	12.96	540.94
	5	0	74.5	0	139.48	56.5	3057	88.08	8105.56	124.28	5219.68	13.56	1066.06

trend is apparent: the further the method can look ahead, the more opportunities for backtracking. The second trend is as the number of splits, R, increases, so does the average number of random tiles generated (Rand). This can be seen by looking at any of the columns of average number of random tiles as R increases. This occurs because the more a map is split horizontally, the less data there is for learning, and therefore there will be more unseen states. The number of random tiles also increases as the complexity of the Markov chain increases. This is because as the model takes into account more previous tiles, the number of possible combinations of previous tiles increases, thus requiring more training data to properly estimate the probability distributions of the Markov chain, giving way to more randomly generated tiles. Finally, we observe that using a small look ahead ($d = 1$) can reduce the number of randomly generated tiles, but higher look aheads result in worse results (more experimentation is required to understand this effect).

Although the trends show that as we increase R, d, and the complexity of the Markov model we get more random tiles or more backtracks, these are not proper indicators of the quality of the map generated, as can be seen in the sample maps shown in Figures 4 and 5. As part of our future work, we would like to analyze these results using better metrics, for automatic assessment of quality, building upon some of the fitness functions used in search-based procedural content generation (Togelius et al. 2010).

Conclusions

We developed a method for procedurally generating maps using variations of Markov chains as a tool for learning. Our method learns from established "good" maps in order to generate statistically similar maps. We also incorporate look ahead and backtracking into our map generation method. This helps improve the quality of the generated map. Lastly, our method includes the ability to split the maps used for learning into different horizontal slices. Doing so allows the user to isolate certain qualities of the maps that may be exclusive to specific portions of the given maps.

We obtained encouraging results, being able to generate fully playable maps in some of the tested configurations. Although the maps generated by our model are not directly usable in an actual game, since they require some cleaning (due to encountering unseen states during generation), we believe that our initial study shows that Markov chains could be a useful tool for procedural map generation.

In the future, we plan to find better evaluation criteria for the maps generated by our methods. Exploring better ways of estimating the conditional probabilities given small amounts of map data is another area of future work. Another area of work is to apply our model to a more complex domain to test scalability.

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-On2010)*, 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)*.

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.

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.

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.