MCMCTS PCG 4 SMB: Monte Carlo Tree Search to Guide Platformer Level Generation

Adam Summerville[1], **Shweta Philip, and Michael Mateas**[1]
[1] Expressive Intelligence Studio
University of California, Santa Cruz
asummerv@ucsc.edu, sphilip@ucsc.edu, michaelm@soe.ucsc.edu

Abstract

Markov chains are an enticing option for machine learned generation of platformer levels, but offer poor control for designers and are likely to produce unplayable levels. In this paper we present a method for guiding Markov chain generation using Monte Carlo Tree Search that we call Markov Chain Monte Carlo Tree Search (MCMCTS). We demonstrate an example use for this technique by creating levels trained on a corpus of levels from *Super Mario Bros*. We then present a player modeling study that was run with the hopes of using the data to better inform the generation of levels in future work.

Introduction

Procedural Content Generation (PCG) is a field that has seen a lot of work in the design and generation of levels for platformer games. These have ranged from using modular designer authored pieces (Shaker et al. 2011a) to trying to learn relationships between sprites using techniques from computer vision (Guzdial and Riedl 2015). Markov chains are an appealing approach since they can be quickly trained on a corpus of levels and generate levels with similar properties. However, the main problem with using Markov chains to generate levels is that they offer little to no guarantees about the levels that they generate. This makes it difficult for a designer to rely on them, since the levels are just as likely to be unplayable as playable (Snodgrass and Ontañón 2013). A similar, but less severe problem, is that the designer has very few knobs to tweak the outcome of the generator, with the two major ones being the length of historicity in the chain and the training corpus.

Monte Carlo Tree Search (MCTS) (Browne et al. 2012) is a popular technique in the world of game AI and has seen applications in Poker (Van den Broeck, Driessens, and Ramon 2009), Magic: The Gathering (Ward and Cowling 2009), and perhaps most notably Go (Gelly et al. 2012) where it has pushed the level of play from equaling an advanced novice to mid professional levels. MCTS uses random sampling during the search process to find areas that seem promising, balancing the need to explore the possibility space by sampling broadly and the need to exploit promising avenues of the tree by sampling deeply. It is an appealing technique because it is agnostic of the domain that it is searching, relying only on an evaluation function to determine a good end node from a bad one. Despite its success as an adversarial AI technique, it has seen little use in other roles.

We propose Markov Chain Monte Carlo Tree Search (MCMCTS) as a method for level generation process that uses the information learned from existing levels coupled with playability guarantees and designer control. Using Markov chains trained on a corpus of levels from the original *Super Mario Bros.* we can generate levels that feel like they could have come from the original series. The Markov chain and MCTS work together, with the Markov chain supplying transmission probabilities that guide the search in more probable directions and the MCTS pruning selections that lead to unplayable or otherwise undesirable levels. The properties that make MCTS useful as a game player also make it useful as a level designer. A large component of using PCG for level generation is random variation in the produced artifacts, and the sampling of MCTS helps to ensure that generated levels will have variation. The balance between exploitation and exploration means that it is less likely to explore areas of the search space that are unplayable (e.g. levels that contain gaps too wide for the player to cross), but the exploration might find levels that are close to unplayable (e.g. levels with gaps just wide enough for the player to cross). While we use a designer authored evaluation function to guide the search, the approach can easily be adapted to other generation domains. In future work, the evaluation function could be automatically learned from player data, entirely eliminating the need for human authored design choices. Our contribution is a novel application of Monte Carlo Tree Search to Procedural Content Generation by guiding a Markov chain generator for the creation of platformer levels.

Related Work

Markov chain generation of platformer levels has seen two major approaches, the first looking at tile-to-tile transitions and the other looking at vertical slices of levels. Snodgrass and Ontañón (Snodgrass and Ontañón 2013; 2014) have mainly looked at the tile-to-tile transitions. This approach has a couple of benefits, namely that it can produce very

Copyright © 2015, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

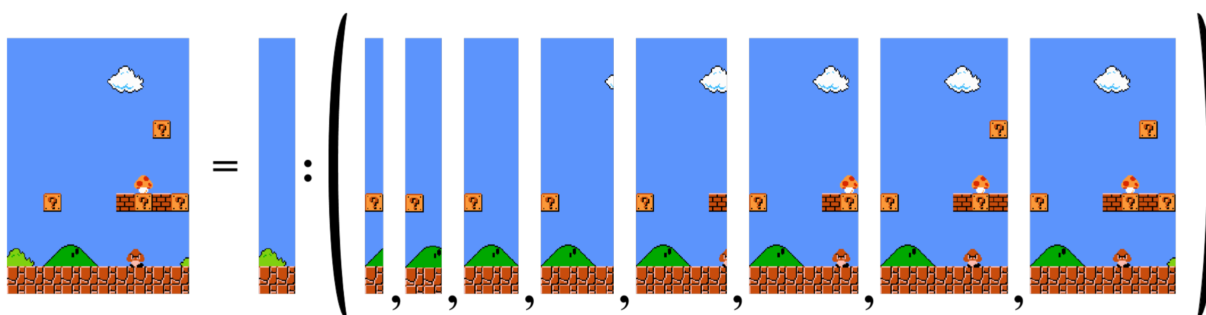


Figure 1: A level block, split into the bigram state and the possible patterns that it can transition to

novel levels. But this comes at a great cost in that a lot of generated levels will be completely unplayable. A key problem of this approach is that empty tiles make up the vast majority of tiles in platformer levels meaning that almost all transitions are either to or from empty tiles. Another problem of this approach is that it entirely washes over a key factor to platformer levels in that transitions are highly dependent on their height in a level, e.g. high tiles are more likely to be empty and low tiles are more likely to contain ground. Snodgrass combated this by learning multiple Markov chains by splitting the levels into multiple height regimes.

The vertical level slices of Dahlskog et al. (Dahlskog, Togelius, and Nelson 2014) are much less likely to have problems with skies full of tiles and large gaps along the bottom due to the fact that the vertical slices inherently capture the height content of the tiles. However, the use of vertical slices loses a large portion of novelty in that it can only ever produce vertical slices that have been seen before, a problem that Snodgrass's work does not have. A key issue with both approaches is that they do not expose many knobs to users. The main way that a user can substantively change the output of the chain is by editing the training corpus. They also come with no guarantee of playability, with over half of the levels produced by Snodgrass's system being unable to be completed.

While MCTS is a popular technique in the games community, it has mainly been used as a technique for game playing. However, Browne (Browne 2013) has proposed using MCTS as a search technique for PCG. Within the field of computational creativity, a system is deemed creative if it can produce novel, high quality artifacts that are recognizable as being representative of the target domain (Ritchie 2007). Browne found that MCTS was able to find high quality solutions with high diversity, diversity being a proxy for novelty.

MCTS has been used for story generation by Kartal et al. (Kartal, Koenig, and Guy 2013). They found that MCTS was well-suited to story generation, particularly in realms with a high branching factor. Other search approaches quickly ran out of memory without being able to search the possibility space in an efficient manner, but MCTS was able to generate stories with a large number of possible player actions (100+).

Level Generation

To generate levels using Markov Chain Monte Carlo Tree Search (MC-MCTS) we first need to learn a representation of the levels suitable for Markov chain generation. We will then use this representation at generation time, guided by Monte Carlo Tree Search to generate learned levels that have desired properties.

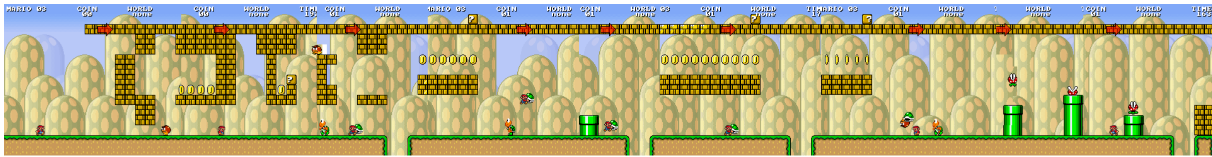
Markov Chain Generation

For this work, we used the levels from *Super Mario Bros.* as the training corpus. We used screencaptured level images produced by a fan on the internet (Albert 2014). To extract the tile information we used the open source computer vision library OpenCV2 (Bradski) to find subimages in the whole level image. While *Super Mario Bros.* contains 32 levels, a number of these are atypical and were excluded from the corpus, namely the underwater levels, the ones that take place on bridges with flying fish, and the boss levels, leaving us with a corpus of 20 levels. While the tile-to-tile and vertical slice methods of platformer level representation both have their benefits and drawbacks, we chose the vertical slice representation for this work due to the higher probability of not producing unplayable levels. During chain generation it is possible to reach what Snodgrass called an *unseen state*, i.e. one in which the history has never been seen before and as such no transmission probabilities are known. To avoid this, if a tile combination would lead to an unseen state we exclude it from our transmission probability table. We only consider 2 slices of history when generating the level chain, but in an attempt to capture larger scale patterns the tables can transition to larger than a single slice, with patterns up to 8 wide being used. See figure 1 for more detail. While these patterns are rare due to their uniqueness, the addition of recognizable large scale patterns helps speed the rollout process and produces familiar, but novel levels.

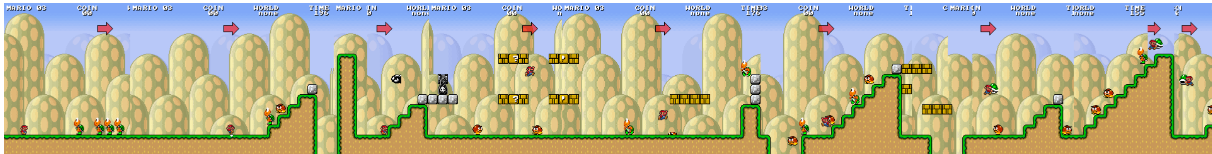
The tile types we considered were:

- *Solid* - Any solid tile, including ground, hard blocks, or the tree tiles
- *Hidden Power Up* - A breakable block that contains a power up instead of just being breakable
- *Breakable* - A breakable block
- *Question Mark Block* - A question mark block, regardless of its contents

(a) A level with moderate difficulty



(b) A level that made heavy use of the "underground" levels



(c) A hard level with a high density of enemies

Figure 2: Screenshots of 3 generated levels being played in *Infinite Mario*

- *Enemy* - An enemy, regardless of its type
- *Bullet Cannon* - The cannon that fires Bullet Bills
- *Pipe* - A pipe, regardless of being a warp pipe, piranha plant, or plain
- *Coin* - A coin

For the purposes of this work we only considered the high level function of a block, as opposed to its most detailed categorization (e.g. we abstract over whether an enemy is a Goomba or a Koopa Troopa). This allows for a more filled out conditional probability table, at the expense of specificity.

Markov Chain Monte Carlo Tree Search

MCTS operates in four steps:

- *Selection* - Starting from the root node, the child with the most potential is chosen successively until a leaf node is found.
- *Expansion* - While that node has untried moves, a move is chosen, at random, and applied
- *Rollout* - Moves are applied randomly until a terminal condition is met
- *Backpropagation* - A score is calculated for the rollout and applied successively to all parent nodes

Standard Markov chain generation starts from a seed state and randomly transitions to successor states. This process repeats until a terminal condition is reached, be it a terminal state or some desired condition of the chain is met (e.g. length $>=$ some threshold). From this description, it is easy to see that standard Markov chain generation is directly analogous to a single rollout in MCTS. While MCTS is typically considered in the style of an adversarial game playing AI, hence the parlance of "moves", for our purposes the "moves" that can be made are the states that can be transitioned into from a parent state. The successor states are chosen with the probabilities learned from the above section for both the *Expansion* and *Rollout* steps.

The most common *Selection* strategy for MCTS is *Upper Confidence bound 1 applied to Trees* (UCT) introduced by Kocsis and Szepesvri (Kocsis and Szepesvári 2006). The UCT formula for node i is:

$$UCT_i = \frac{w_i}{n_i} + c * \sqrt{\frac{\ln t}{n_i}}$$

Where

- w_i - The total score of node i - in game playing applications this is commonly the number of wins
- n_i - The number of simulations considering node i
- c - The exploration parameter. $\sqrt{2}$ theoretically, but used to tune exploration vs. exploitation in practice
- t - The total number of simulations

The main choices for a user of MCMCTS are how the score is calculated, the exploration parameter, and the number of rollouts performed. A higher number of rollouts will better sample the space and produce a better end result, but this comes at the cost of time. In order to keep the level generation time under 30 seconds per level we limited the system to 200 rollouts per "move". We generated 320 vertical slices per level, so we could sample up to 64,000 levels per every generated level. The exploration parameter value matters most in its relative scaling compared to the score value, so we set it to the theoretically correct $\sqrt{2}$ and focused our attentions on the score.

The score function that we used was:

$$score = S + g + e + r$$

where

- S - Whether the level is solvable or not. If it is solvable

then it was set to 0, otherwise it was set to a large negative number, -100,000 in this study.

- g - The desirability of the level as a function of the number of gaps.
- e - The desirability of the level as a function of the number of enemies.
- r - The desirability of the level as a function of the number of rewards (coins or power-ups).

In theory, these functions could be anything or could be learned via player testing. For the purposes of this work the desirability functions were quadratics that captured our hypotheses about how desired difficulty of the level would affect the desirability of the components. In general as difficulty increased so did the desirability of gaps and enemies, but due to the quadratic nature there was a point at which too many enemies and gaps would be undesirable to anyone. The opposite was true for the rewards, as they would decrease as difficulty increased. This score was then passed through a logistic function to produce a result in the range of 0-1.

$w_d = \frac{1}{1+e^{-score_d}}$

Solvability was determined at a low level of fidelity for the purposes of speeding the MCTS search. Snodgrass used Robin Baumgarten's A* controller (Shaker et al. 2011b) to determine whether a level can be completed or not, but it runs a full simulation of the Mario physics making it too slow for a sampling process that wants to be run thousands of times in a short period of time. We created a simplistic A* Mario controller that operates on the tile level with discretized physics extracted in a preprocessing step from the actual Mario simulation. This simulation captures running and jumping physics at fine enough fidelity to ensure that all jumps that the controller determines are fine will in fact be fine in the full-fledged simulation. However, the low level controller does not consider enemies at all, so it is possible for a section of enemies to be too dense for playability. In practice, this is of not much concern as the learned probabilities make such a section exceedingly unlikely and the score function makes this unlikely as well. A sample of three levels generated by our system can be seen in figure 2. Level 2a and 2b have the same difficulty, but show the variation that can be achieved due to the wide range of levels used during training the Markov chain. Level 2c is a harder difficulty which results in a higher quantity of enemies.

Player Modeling

Our intention is to move away from the hand authored scoring function and move towards a design that incorporates feedback to better tune levels to a player's desired difficulty and skillset (e.g. do they enjoy levels with more/less jumps, gaps, enemies, etc.). Towards this end we first used the generator to generate levels as part of a player modeling experiment based on the work of Pedersen et al. (Pedersen, Togelius, and Yannakakis 2009), with the intention of using these player models as the basis for future work in generating levels for specific players. We have our own intuition about what would make levels more challenging or preferable to different player types, but we used the tunable parameters of our system to model player challenge ratings and preference based on their experience level. We then look at predicting player experience level based on player performance metrics.

Data Collection

We used our MCMCTS system to perform player modeling based on the tunable parameters. We considered each of the three parameters in either a *High* or *Low* setting.The combinations of these parameters result in 2^3 = 8 different variants of the game. The following table displays the detail of all the variants. Each variant is played as the first game marked by ***A*** and as the second game marked by ***B*** in a pair. In total, every game is played 16 times over the course of the experiment. The only fixed feature was that only one power-up was included in the levels.

Controllable Features				*Played as*		
Variants	*Gaps*	*Enemies*	*Rewards*	*A*	*B*	*Total*
1	Low	Low	Low	8	8	16
2	Low	Low	High	8	8	16
3	High	Low	Low	8	8	16
4	High	Low	High	8	8	16
5	High	High	Low	8	8	16
6	High	High	High	8	8	16
7	Low	High	Low	8	8	16
8	Low	High	High	8	8	16

A number of gameplay features were considered for the purpose of modeling player experience. These features are selected based on a previous study done on player experience modeling which defined them to cover most of the behavioral dynamics of *Super Mario Bros.* (Pedersen, Togelius, and Yannakakis 2009). The features fell into a number of different categories:

- **Time**: There were around 10 features related to time. Each player was allowed 3 attempts at a level. Completion time represented the time the player took to complete the level in the last attempt. Total time represented the total time spent by the player on a particular level. There were other features which represented time spent running, running left and also running right. Some features record the time spend in each mode: Small Mode, Large Mode and Fire Mode. The levels in this experiment featured only two modes, Small and Large Mode. Hence only data about these two modes were recorded.
- **Jump**: These features includes the number of times a player pressed jump. Another feature called aimless jumps logs the number of times the player jumped in the absence of an enemy or a gap.
- **Death**: These features record the different ways and the number of times the player died during a game level. There are eight different ways how a player could die in the levels. A player can die by falling into a gap, by coming in contact with a Goomba, a Green Turtle, a Red Turtle, an Armored Turtle, a Jump Flower, a Bullet Bill or a

Chomp Flower. In our experiment, five out of the seven enemies were included in the levels: Goombas, Green Turtles, Red Turtles, Bullet Bills and Chomp Flowers.

- **Item**: The features recorded under this category are related to the items that the player interacts with. Total Coins represent the number of coins contained in each level, as well as a feature which records the total number of coins collected by the player. Similarly, features such as total coin blocks, total power blocks and total empty blocks represent the number of blocks which contain coins, power ups and nothing respectively. There are some features which also capture the percentage of these blocks the player interacts with or destroys during the course of the game.
- **Kill**: As mentioned above there are five types of enemies used in this experiment. These features recorded the number of enemies killed by the player for a given level. These features also recorded the different methods by which the player killed Marios enemies. Mario is capable of killing enemies by stomping on them, by kicking shells at them, and by throwing fire balls at them when in Fire Mode. But as mentioned earlier, Fire mode was not included in the levels, so this feature was always zero. Another feature represents the total number of enemies in a level.
- **Miscellaneous**: These features do not fall into any of the categories mentioned above. They capture the number of times the player switched from one mode to another, and the number of times the player pressed a key to duck or to run.

As part of the experiment, each player played eight variants of the level. After every pair of games the player was asked to answer a series of questions. The experiment was planned such that every level variant was played as both the first in a pair (A) and the second in a pair (B) exactly eight times. Note, this does not mean that they played level type 1 twice consecutively, just that they played a certain level type and then another (randomly chosen) level type. This was done to observe whether the player responses in the questionnaire by the order in which levels were presented. This experiment closely follows the protocol in (Pedersen, Togelius, and Yannakakis 2009), in which the questionnaire focused on challenge and preference. The following pairwise forced choice questions were asked to the players:

- Level A was more challenging/preferred over Level B
- Level B was more challenging/preferred over Level A

After playing four pairs of such levels, the player was asked to describe their experience in playing platformer games in the following categories:

- Beginner
- Intermediate
- Expert

The participants were 16 students at University of California who volunteered for the experiment. Each student played the Java based Infinite Mario game and provided answers to the questionnaire. Each participant played 8 levels, resulting

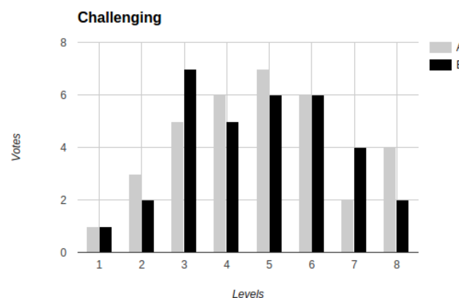


Figure 3: Player rated challenge A vs. B

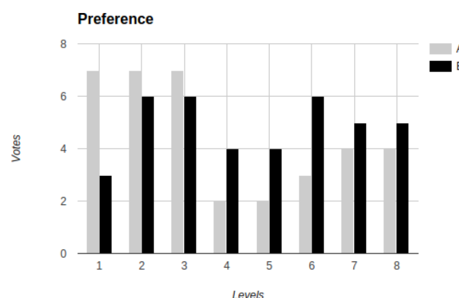


Figure 4: Player rated preference A vs. B

in data being collected for 128 levels. This paper analyzes the data for 64 pairs of games.

Modeling Analysis

We first wanted to verify that order had no effect on how players rated challenge or preference. 3 shows the number of times levels were marked as challenging by the players in different orders.

There is no statistically significant difference between playing a level first or second in terms of challenge rating, but a larger sample size is needed for a definitive answer.

Figure 4 shows which level the player would prefer to play.

With no statistically meaningful relationship could be determnined, we are satisfied that there was no strong relationship between the order in which levels were played. However, a larger sample size would help fully settle this. We turn our focus to player experience modeling. We used the data mining tool Weka (Hall et al. 2009) to learn a model for our dependent variables: challenge, preference, and player experience. Since we are trying to learn a binary dependent variable we used logistic regression, and due to our limited sample size we used 10-fold cross-validation.

Challenge

To determine whether the player voted for level A or B we looked at the difference between level A and B for each parameter which were either 1 (A was high, B was low), 0 (A and B were the same), or -1 (A was low, B was high). Our feature space considered the 3 difference parameters, and the 4 interaction terms. Experience level was also con-

sidered as well as its interaction with the tunable parameters. We then performed exhaustive feature selection based on the Akaike information criterion (AIC), which only considered 3 parameters.

Our model was able to successfully classify which level was more challenge 72.66% of the time, with a Kappa statistic of 0.45. A number of different guides for interpretation of the Kappa statistic agree, but 0.45 is generally held in the fair-to-good range, so we are relatively happy with this result.

Variable	*Odds Ratio*	*Direction*
Gaps Difference	7.91	+
Enemies Difference	1.80	+
Experience * Gap Interaction	2.22	-

Not surprisingly, both higher number of gaps and enemies were important determinants for whether players found the level more challenging, with the number of gaps being the most significant determinant. Also of note is that gaps were less important for experienced players.

Preference

Using the same logistic regression methodology as above, we found a model with 4 parameters that was able to determine player preference 64.07% of them time with a kappa statistic of 0.28 which is considered fair, but not strong.

Variable	*Odds Ratio*	*Direction*
Rewards Difference	1.35	+
Experience * Rewards Interaction	3.71	-
Gaps Difference	1.72	-
Experience * Enemies Interaction	1.37	-

Players tended to prefer levels with a higher number of rewards and a lower number of gaps. Interestingly, more experienced players were less likely to prefer higher number of rewards, and while the difference in enemies had no significant effect on player preference, the interaction term with experience did. Most surprisingly, more experience players were more likely to prefer levels with fewer enemies.

Player Experience Level

Since the above models were based on self reported player experience level, we wanted to see how well we could classify player experience based on our models so as to not require players to report their skill level. Originally we wanted learn a model that could predict a player's self-report of their platforming skill as beginner, intermediate or expert based on their gameplay features. Unfortunately none of the participants self-identified as experts. As such we could only try to classify whether they had low or medium experience playing platformer games. Our model was accurate 70.31% of the time with a fair-good Kappa statistic of 0.41. Unlike challenge, the results are somewhat unintuitive. As above we performed feature selection to produce a limited set of 7 features that had enough information gain. In this case being $+$ means that the player is more skilled and $-$ means more likely to be a beginner.

Variable	*Odds Ratio*	*Direction*
Number of Jumps	1.08	+
Time Spent Running Left	1.03	+
Number of Aimless Jumps	1.04	-
Number of Enemies Killed by Shell	3.41	+
Total Time Spent Large	1.04	+
Total Number of Enemies	1.04	+
Number of Green Koopas Killed	1.64	-
Number of Bullet Bills Killed	1.42	+

These are much less intuitive than the other model, but mostly make sense. Beginner players are more likely to make aimless jumps, whereas more advance players make their jumps count. One might expect that running backwards would show beginners' cautiousness, but it seems that running backwards is more a mark of an advanced player making considered actions. Unsurprisingly, killing enemies, specifically by an advanced technique like using a shell or killing difficult enemies like Bullet Bills, is linked to being a more advanced player. The most nonintuitive result is that beginner players are more likely to kill Green Koopas, perhaps because advanced players ignore them because they do not pose a threat. Ultimately, more data needs to be gathered to better predict player performance.

Conclusion and Future Work

We have presented a novel method for guiding Markov chain platformer level generation via Monte Carlo Tree Search. This technique relies on using classic Markov chain state transitions as the "moves" for a Monte Carlo Tree Search and classic Markov chain generation as the "rollouts". This method provides the simplicity of Markov chain generation with a near guarantee of playability along with user and/or designer tunable score function to provide levels that meet the design goals. We then use this level generation framework as a testbed for player modeling, attempting to build on the work of Pedersen et al. (Pedersen, Togelius, and Yannakakis 2009).

Currently, our system uses a scoring function that we hand tweaked. We would like to extend this evaluation function to incorporate player modeling. We would also like to use the tool in a mixed initiative style by presenting levels to a designer that they can then rate, from that we could learn a scoring function that is attuned to a designer's desires. Preliminary work in looking at how levels are rated based on the three tuned parameters shows that challenge is most strongly associated with the number of gaps. Using a Pearson chi-square independence test it it is the only parameter that is statistically significant ($p < 0.005$) in determining whether a player finds the level challenging. This leads us to believe that more work needs to be done to determine why the other factors had no effect on player's challenge ratings.

On the Markov chain front we used the vertical slice style, but we would also like to experiment with the tile-to-tile transitions. Our belief is that we would find levels that are more novel without running into the unplayability problems that Snodgrass encountered.

References

Albert, I. 2014. Legend of Zelda Maps. http://ianalbert.com/games/legend_of_zelda_maps/.

Bradski, G. OpenCV, year = 2000. *Dr. Dobb's Journal of Software Tools*.

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

Browne, C. 2013. UCT for PCG. In *Proc. IEEE Conf. Comput. Intell. Games*, 137–144.

Dahlskog, S.; Togelius, J.; and Nelson, M. J. 2014. Linear levels through n-grams. In *Proceedings of the 18th International Academic MindTrek Conference*.

Gelly, S.; Kocsis, L.; Schoenauer, M.; Sebag, M.; Silver, D.; Szepesvári, C.; and Teytaud, O. 2012. The grand challenge of computer go: Monte carlo tree search and extensions. *Commun. ACM* 55(3):106–113.

Guzdial, M., and Riedl, M. O. 2015. Toward game level generation from gameplay videos. In *Proceedings of the FDG workshop on Procedural Content Generation in Games*.

Hall, M.; Frank, E.; Holmes, G.; Pfahringer, B.; Reutemann, P.; and Witten, I. H. 2009. The weka data mining software: An update. *SIGKDD Explor. Newsl.* 11(1):10–18.

Kartal, B.; Koenig, J.; and Guy, S. 2013. Generating believable stories in large domains.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, 282–293. Berlin, Heidelberg: Springer-Verlag.

Pedersen, C.; Togelius, J.; and Yannakakis, G. N. 2009. Modeling player experience in super mario bros. In *Proceedings of the 5th International Conference on Computational Intelligence and Games*, CIG'09, 132–139. Piscataway, NJ, USA: IEEE Press.

Ritchie, G. 2007. Some empirical criteria for attributing creativity to a computer program. *Minds and Machines* 17(1):67–99.

Shaker, N.; Togelius, J.; Yannakakis, G.; Weber, B.; Shimizu, T.; Hashiyama, T.; Sorenson, N.; Pasquier, P.; Mawhorter, P.; Takahashi, G.; Smith, G.; and Baumgarten, R. 2011a. The 2010 mario ai championship: Level generation track. *Computational Intelligence and AI in Games, IEEE Transactions on* 3(4):332–347.

Shaker, N.; Togelius, J.; Yannakakis, G. N.; Weber, B. G.; Shimizu, T.; Hashiyama, T.; Sorenson, N.; Pasquier, P.; Mawhorter, P. A.; Takahashi, G.; Smith, G.; and Baumgarten, R. 2011b. The 2010 mario ai championship: Level generation track. *IEEE Trans. Comput. Intellig. and AI in Games* 3(4):332–347.

Snodgrass, S., and Ontañón, S. 2013. Generating maps using markov chains.

Snodgrass, S., and Ontañón, S. 2014. A hierarchical approach to generating maps using markov chains.

Van den Broeck, G.; Driessens, K.; and Ramon, J. 2009. Monte-Carlo tree search in poker using expected reward distributions. In *Proceedings of the 1st Asian Conference on Machine Learning (ACML), Lecture Notes in Computer Science*, 367–381. Springer.

Ward, C., and Cowling, P. 2009. Monte carlo search applied to card selection in magic: The gathering. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, 9–16.