Intelligent Content Generation via Abstraction, Evolution and Reinforcement

Dean M. LeBaron, Logan A. Mitchell and **Dan Ventura**
Computer Science Department
Brigham Young University
Provo, UT 84602
lebarondean@gmail.com, mitchlam711@gmail.com, ventura@cs.byu.edu

Abstract

We present a system for autonomously generating puzzles in the form of a 2D, tile-based world. Puzzle design is entirely dependent on tile characteristics, which are implemented as abstract classes that can be modified by the system. Thus, the system controls not only the base-level puzzle design but also (to some extent) the meta-level component design. The result is a rich space of possible puzzles that the system explores with a combination of evolutionary computation and Q-learning. The system autonomously produces a variety of puzzles of varying difficulty to create a game called *Loki's Castle*. The system is almost completely autonomous, requiring only a minimal description of what a puzzle should include, and the abstraction allows extensibility so that future versions can invent entirely new classes of tiles. Several puzzle examples are presented to demonstrate the system's capability.

Introduction

Arguably the most interesting additions to the field of game development over the last few years have been from the fields of *procedural content generation* (PCG) and *artificial intelligence* (AI). Procedural generation of content for games is increasingly allowing designers to focus on higher-level concerns, while automatic generators produce lower-level content such as textures, landscapes, buildings, layouts, music, simple dialogues, etc. (Togelius et al. 2011). Even so, procedural generation is still mostly focused on a particular type of game content–namely environment and map design (Hendrikx et al. 2013).

Artificial intelligence is making inroads into the gaming industry most obviously in the form of non-player characters and other types of agents that pose challenges to the main players, but it has been used in many other ways as well, including dynamic difficulty balancing (Hunicke and Chapman 2004; Chanel et al. 2011), player experience modeling (Drachen 2008; Yannakakis and Togelius 2011), datamining of user behavior (Bauckhage, Drachen, and Sifa ; Ducheneaut et al. 2006), and even featuring as the "main event" (Horswill 2014). Indeed, Horswill suggests that game design may need to be reconsidered from the ground up to incorporate AI to its best advantage. And, Smith has recently discussed ways in which procedural content generation "can be turned up to eleven" (Smith 2014).

In considering how we might address both of these suggestions, we consider combining AI and PCG in a way that begins to allow the system more autonomy in designing the game itself, in the form of automated puzzle generation. In particular, we present a system which generates 2D puzzles formed in a tile-based world. The puzzles consist of simple-looking 2D environments in which the player must discover how to move from the start tile to the exit tile by navigating the unknown challenges posed by tile characteristics, which change from level to level.

Levels are generated using a two-pass evolutionary process and evaluated with a combination of intelligent heuristics (at the first level) and Q-learning-based playability modeling (at the second level). Similar approaches have been used in various ways in other systems, including the following.

Taylor and Parberry built a system for creating a specific type of puzzle called sokoban using naïve yet effective techniques such as random templates and brute force search (2011). Using a scripting language designed by Stephen Lavelle[1], both Lim and Harrell (2014) and Khalifa and Fayek (2015) have built more general puzzle level generators.

Yoon and Kim presented a system using L-system grammars which created a blueprint of 3D game models. A genetic algorithm (GA) which queries users as part of its fitness function is then used to manipulate the basic shapes produced by the grammars (Yoon and Kim 2012). Ashlock used a GA to produce puzzles at varying levels of difficulty, and the fitness function includes dynamic programming to determine the minimal moves required to solve the puzzle—a measure of playability based on puzzle difficulty (Ashlock 2010).

Williams-King et al. used a two pass GA to generate puzzles inspired by the classic game LodeRunner. Their approach involves using one GA to evaluate the aesthetic quality of a level, and then all members of the population above a certain fitness are passed to a second graph based GA which is intended to discover dynamic elements of the puzzle, i.e., mimic a user-experience (Williams-King et al. 2012). Again this can be seen as an attempt to measure the playability of the generated levels.

[1] `http://puzzlescript.net`

Other work has been done with the focus of creating and measuring fun while not specifically using GAs. Smith et al. created 2D platformers using a rhythm-based grammar and ideas of flow. They argue that a user who achieves flow, and therefore discovers the rhythms inherent in the level has an enjoyable experience, and they also attempt to measure this directly with anxiety curves (Smith et al. 2009).

Our work fits squarely in with those described above in that we are using GAs to generate content and have attempted to incorporate a measure of playability by using Q-learning as part of the fitness function of the GA. Similar to Holmgård, et al. (2014), we posit that the Q-learner can somewhat mimic the experience of a user, and quantitative notions such as how long the Q-learner takes to solve the puzzle or how much exploration it required before exploitation, can approximate the challenge of a given level.

The goal of our system is to generate unique, challenging levels, presented as a top down, 2D maze-based puzzle game. Each level requires the user to discover tile characteristics and how they interact, so that they can eventually plan for the hero a safe path to the exit tile. As each new level is generated, not only is the level layout changed, but also the tiles' characteristics change as well, so that the player must begin the process all over again. Because this is rather diabolical, we call the game *Loki's Castle*.

Building *Loki's Castle*

The levels of *Loki's Castle* are constructed from a set of tiles, each with unique visual and behavioral properties. While the visual properties are fixed (at least for the moment), the behavioral properties are mutable and change for each level, which means both that the number of possible tiles (and thus the number of possible puzzles) is very large and that the player must discover tile behaviors and interactions anew at each level. Tiles are composed into square levels, 10 tiles on a side. Candidate levels are evaluated by a two-stage evolutionary algorithm that uses an initial fitness function to evaluate (visual) aesthetic qualities of a level and a second fitness function to evaluate playability. Microsoft's XNA framework is used to create the interface.

Tiles

Tiles are the basic building blocks used to create levels. A tile is a 1×1 square with a set of properties and an associated sprite. Every tile possesses three properties—CanEnter, HasEntered and HasExited—and these properties have associated effects implemented by the tile to affect the hero when the player attempts to enter, enters, or exits a tile, respectively. The CanEnter property exhibits four possible effects: TRUE, FALSE, REMOVE (an item), ADD (an item). The HasEntered and HasExited properties can both exhibit seven possible effects: NONE, DEATH, CONDITIONAL DEATH (if no item), MOVE, ADD (an item), REMOVE (an item), and SLIDE. Given that there are 14 unique tile sprites available to the system, there are then $(1 + 3 \times 7 \times 7) \times 14 = 2,072$ general file types [not $(4 \times 7 \times 7) \times 14$ because all tiles with CanEnter→FALSE are functionally equivalent to walls].

In addition, many of the tile effects are parameterized in various ways by an abstract parameter object. For example, any item-based effect can be parameterized by item type (axe, amulet or key) as well as by item number (1, 2 or 3), while the MOVE and SLIDE effects can be parameterized by direction (up, down, right, left) and distance (1, 2, ..., or 9). Taking all the parameterizations into account as well, the number of tiles types rises to $(1+7 \times 95 \times 95) \times 14 = 884,464$ (one could argue that visually different tiles with the same functionality are homomorphic in some sense and that therefore this number should be reduced by a factor of 14, but given that a tile's visual representation can be associated with different functionality at each new level, this homomorphic quality can actually be misleading for the player rather than helpful, suggesting the larger number should be considered more representative of the true size of the tile space).

This abstract implementation of tiles allows extensibility as well as mutability, allowing the system a non-trivial autonomy in designing puzzles—both effects and parameterizations are extensible and mutable, so that new effects and/or parameterizations can easily be associated with a tile property (note that as of this writing, parameterizations are mutated by the system but the set of effects from which we draw is static, so tile-effect bindings are mutable, but new effect types are not created [yet]). Further, the large tile space means that each generated level is likely to be unique. Consider the number of possible tile combinations that can be used to create a puzzle. We have defined puzzles to include 4 types of tiles not including walls and the start and end tile (via the first stage fitness function discussed below). One of these is always the default tile (CanEnter→TRUE, HasEntered→NONE, HasExited→NONE), but the remaining three are randomly constructed tiles, drawn from the set of $\approx$ 900K possible tiles. To create a level, the system selects 3 tiles with replacement, yielding $\binom{((884464+3)-1)}{3} = 115,316,301,441,393,360$ possible unique choices of effector tile sets. This number still does not account for tile placement in the 10×10 grid. The system discovers which combinations work to create interesting levels while operating in this large space.

Evolving Levels

Levels are created using a two-stage evolutionary approach inspired by (Williams-King et al. 2012). The first stage of the algorithm is concerned with creating visually pleasing mazes, and then the most fit candidates are passed to the Q-learner to be evaluated and further refined.

Stage 1 The first stage of the evolution is concerned with generating levels with some (visual) aesthetic quality. Given a level p, the first stage fitness is computed with a function of the form:

$$fitness(p) = f(NumTypes(p), TypesRatio(p), Symmetry(p))$$

where $NumTypes(p)$ drives the number of different effector file types to some desirable value (in our experiments this value was 3, as mentioned above), $TypesRatio(p)$

drives the ratio of the number of each of those types towards some desirable distribution (in our case, uniform), and $Symmetry(p)$ values (longitudinal or latitudinal) symmetry over asymmetry.

The evolution follows a classical form: first, fitness of an initial population of mazes is computed, and 20% of the population is then selected for reproduction by fitness-weighted sampling. The set of parents are then randomly paired and crossed-over to create new children, which are then mutated and assigned a fitness score. Finally the population is culled back down to its original size, again by fitness-weighted sampling.

Crossover occurs by selecting a random-sized contiguous block of tiles from one parent, and then inserting it into the corresponding location of the other parent to create a new child. The start and end tiles are preserved to ensure puzzle validity.

Mutations takes three different forms: a single tile's type can be (uniformly) randomly changed, 10% of the tiles can randomly exchange locations, or all tiles of one type can have their type changed to another (uniformly at random). The mutation rate is 100%, and the mutation form is chosen (uniformly) randomly.

Stage 2 The second stage of generating puzzles uses Q-learning (Watkins and Dayan 1992) to model how challenging the level will be based on it's complexity. The Q-learner determines whether a level is solvable or not, which is obviously the first criteria for a playable level, and the time the learner requires to find the solution and the total reward gained by the Q-learner gives an indication to the difficulty of the level. Q-learning is a reinforcement learning tool for discovering an optimal policy without modeling the environment, and it learns to solve a puzzle by exploring the level and observing the reward gained by taking each action at each state. Using this experience, the Q-learner computes an expected reward for each state-action pair using the following formula:

$$\hat{Q}_k(s,a) = (1-\alpha_k)\hat{Q}_{k-1}(s,a) + \alpha_k(r(s,a) + \gamma \max_{a' \in A} \hat{Q}_{k-1}(s',a'))$$

where, $s \in S$ and $a \in A$ are the current state and action, respectively, with $S = \{(x, y, w, z, \vec{c}, t_1, ..., t_9)\}$ and $A = \{\texttt{up}, \texttt{down}, \texttt{right}, \texttt{left}\}$. $\hat{Q}_k(s,a)$ is the expected reward for the given state-action pair at iteration k, $r(s,a)$ is the reward gained in state s by taking action a, α_k is the learning rate at iteration k, γ is a discount factor for future (expected) reward, and s' is the new state after taking action a. The components of a state $s \in S$ are

x the x-coordinate of the hero

y the y-coordinate of the hero

w the x-coordinate of the exit tile

z the y-coordinate of the exit tile

$\vec{c}$ a vector of counts for each acquirable item

t_i tile type for each of the nine tiles the hero's neighborhood, with t_1 being the tile above and to the left of the hero, t_5 being the tile on which the hero is standing, and t_9 being the tile below and to the right of the hero.

The exact reward structure used is not critical as long as it faithfully represents the environment's effect on the player. In this case, that means a (relatively) large positive reward for entering the exit tile, a (relatively) large negative reward for killing the hero, and a (relatively) small negative reward for entering any tile other than the exit tile (to facilitate shortest-path solutions).

Given a level p, the second stage fitness is computed with a function of the form:

$$fitness(p) = f(\sum^{s} QLearn(p))$$

where $QLearn(p)$ applies the Q-learner to p, returning accumulated reward for a solution, and the superscript s on the summation indicates total time allotted in seconds for learning a solution (or solutions). If a solution is found before the allotted time has elapsed, the Q-learner is restarted from scratch, and the accumulated reward from all solutions found in the allotted time is summed (redundant solutions are counted in the summation). So, a puzzle that is easy to solve may have many solutions discovered in the allotted time and thus a high fitness, while a very difficult level may have no solutions discovered in the allotted time and have a fitness of 0—the fitness measure is inversely correlated with level difficulty. For our results, we set $s = 30$, so the Q-learner is run on each level for a total of 30 seconds.

Results

To produce playable levels, six parallel threads of the first stage of the evolution are continuously run for 1500 generations with a population size of 100 puzzles, with each thread

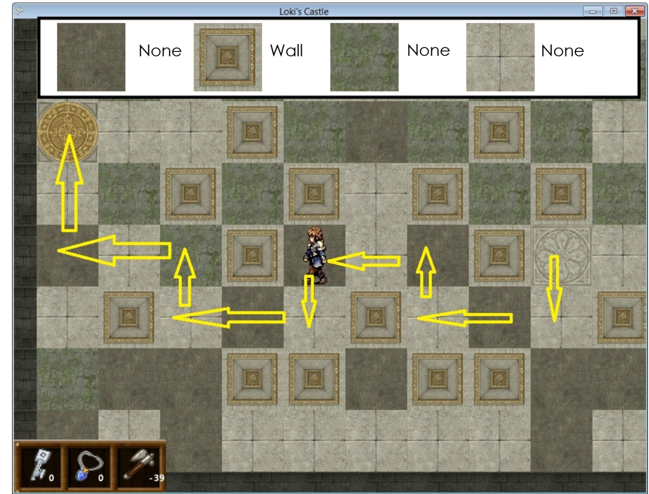

Figure 1: An example generated maze showing a variety of tiles and a visually pleasing layout. Here, and in the following figures, a key labels each tile-type with its associated effect, and a solution path (not necessarily unique) to the exit tile is shown in yellow. In this case, while the solution is not trivial, it is fairly straightforward.

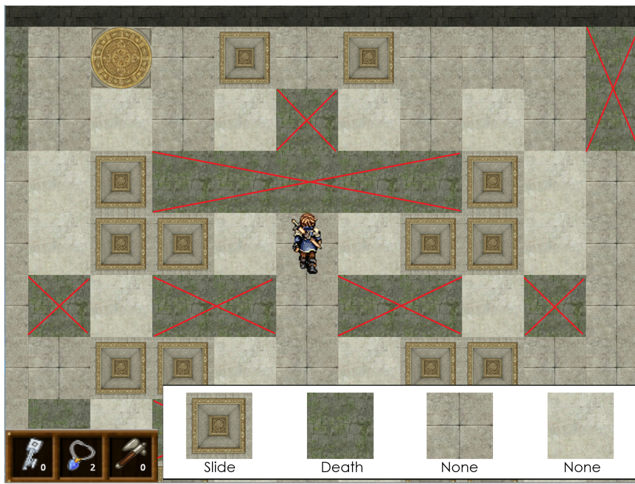

Figure 2: This example highlights a simple kind of challenge: mostly navigating around pitfalls, even though the path is almost trivial. In this puzzle, the green tiles are death (note that the red Xs don't show up in the actual game; they are added here to emphasize the danger of the green tiles). The gold-rimed tiles will slide the player. This can make it difficult to control and can cause a careless player to perish.

having a unique pool of tile types from which to compose its population members (as soon as a thread finishes, another is started). For the second stage of the evolution, the Q-learner randomly selects one of the completed threads' populations and attempts to solve its highest rated puzzle. The Q-learner runs for 30 seconds and then selects another completed thread and repeats. Puzzles that pass both stages of the evolution are sorted by second stage fitness, with higher (second stage) fitness values indicating less challenging puzzles (at least for the Q-learner); therefore, puzzles are presented to the player in descending order of (second stage) fitness, so that the levels increase in difficulty over time.

Figures 1–5 show several example levels that have passed both stages of fitness during the evolutionary process, and are thus at least somewhat visually pleasing and solvable. In Figure 1, some tiles cannot be entered so that the solution simply involves finding a path that avoids these tiles. In Figure 2, the stakes are raised slightly—the green tiles will kill the hero. So, again, the solution is one of obstacle avoidance, but in this case, failing to do so results in death. To make things even a bit more tricky, the stylized gold tiles slide the hero, which can cause the player to inadvertently enter the green tiles. The example of Figure 3 adds something slightly more cerebral—fake walls. Some of the black tiles are real walls that cannot be entered, while others are fake and can be. Upon entering the fake wall, the hero will be slid downwards. The solution lies in discovering this (as there is no path to the exit that does not require it), while avoiding the tan tiles which will kill the hero. In Figure 4, both the green tiles and the stylized gray tiles cost one hammer item to enter, while entering water tiles bestows one hammer item. The Q-learner has to learn to enter a water tile multiple times to accumulate hammers in order to get through the "gated" tiles guarding the exit. The example of Figure 5 demonstrates a simple trap that must be avoided—water tiles cost one key to enter, so if the hero gets stuck between two water tiles without any keys, he is trapped. The Q-learner learns to avoid this by finding that the dark grey tiles are the source of a pair of keys.

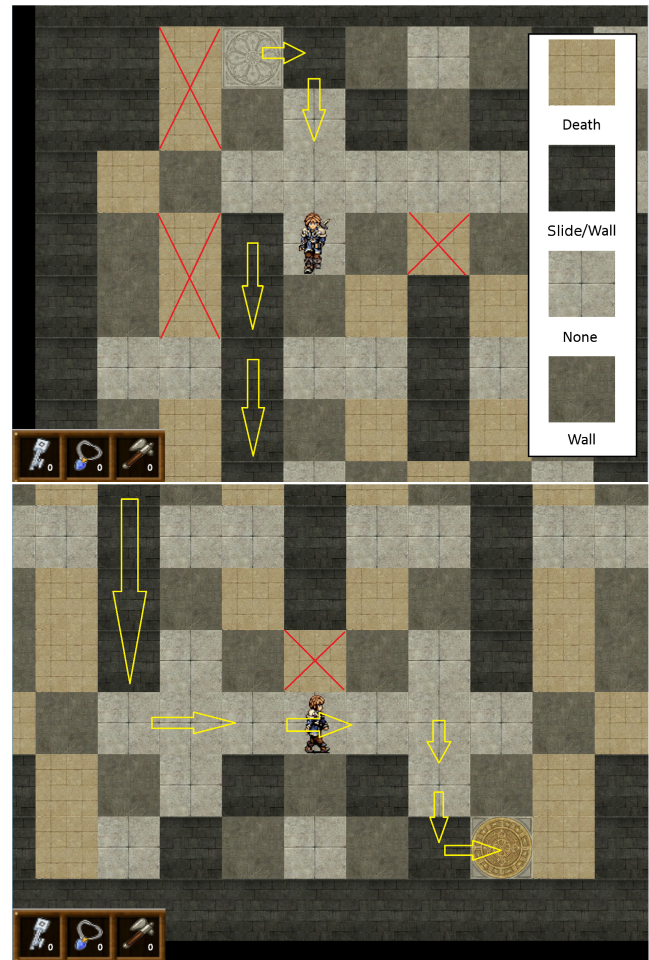

Figure 3: In this puzzle, the tan tiles are death, the dark gray tiles are walls, and only some of the black tiles are walls—some of the black tiles are fake walls. If a player steps off them, they will move the hero downwards. The only way to solve this puzzle is to slide down one of the "wall" sections. This means that the player must realize that he can now walk on this wall. He will learn it quickly, because the only way off the start tile is either to die on the left, or walk on the wall on the right.

Figure 6 shows the number of solvable puzzles found versus time. This grows fairly close to linearly, and therefore approximately one solvable puzzle is found every minute of running the Q-learner: 25 minutes of running the system produced 20 solvable puzzles. The puzzles that are presented to the player are ordered by descending total reward to the Q-learner accumulated within a 30 second learning period (so, "easier" puzzles are seen first). For example,

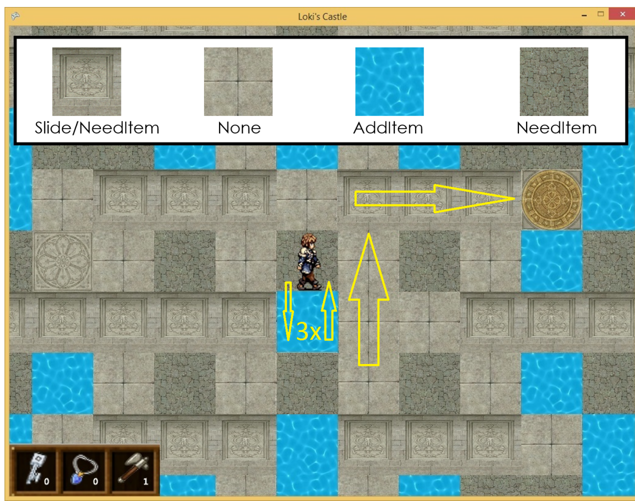

Figure 4: In this example, the greenish tiles and the stylized gray tiles cannot be entered unless an item is received by entering the blue water tiles; therefore, 3x water gives the character the ability to pass through the 3 stylized gray tiles to get to the exit.

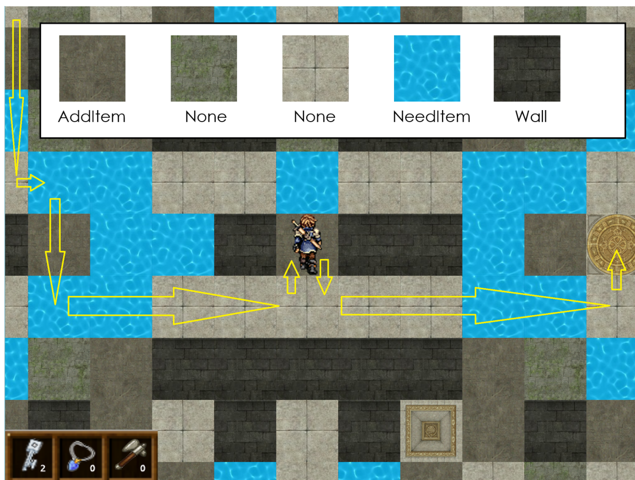

Figure 5: In this puzzle, the dark squares (such as the one that the hero is standing on) set the key count to 2. The water tiles require possession of at least 1 key to enter and doing so costs the player that key. The black tiles are walls. Thus, the player must be careful to avoid getting stuck in between the two water tiles. Furthermore, the player has to take a small detour to pick up more keys to be able to finish.

if it takes the learner 10 seconds to solve the puzzle and it solves it 3 times within the time limit, then the score is the sum of the rewards for each of those runs (different solutions will likely have different rewards, but even the same solution may have slightly different reward each time it is discovered due to the stochastic nature of the Q-learner).

Discussion

Loki's Castle uses evolutionary computation and Q-learning to generate unique puzzles that exhibit interesting characteristics and varying degrees of challenge. The tile effects are also generated uniquely each time, making use of an abstraction of the tile types that allows significant system autonomy in level creation. Tiles that generate interesting levels can be saved and used in subsequent creations.

While our initial results are promising on the (visual) aesthetic level, the creativity and enjoyability of the puzzles the system produces can still be improved. We have set up an extensible framework to easily include additional types of tiles, and to easily allow the system itself to create and analyze levels created using these new tiles. Currently the most creative aspect of *Loki's Castle* is the fact that tile effects are generated dynamically and then puzzles built using the created tile types are scored by the Q-learning-evaluated stage of the evolutionary computation. This process is akin to learning an autonomous aesthetic by which the program decides that a certain tile *property→effect* leads to levels which are too easy, too difficult, or impossible. Currently, however, a pool of these tiles is being created *a priori* from which to draw, and then the system incorporates (some of) them to create a level with a target number of tile types. The next step would be to incorporate the fact that the tiles can be dynamically created into a mutation function in the evolutionary process itself. Namely, instead of looking at levels with different combinations of pre-made tile types, the evolutionary process would introduce new properties or change current properties of the tiles in a fitness-based feedback loop. This step would not be a difficult one given our abstracted code functionality.

To faciliate generalization and thus speed-up puzzle evaluation, it may be possible to use a variation of the Q-learner that replaces the Q table with a function approximator, such as a multilayer perceptron trained using back-propagation.

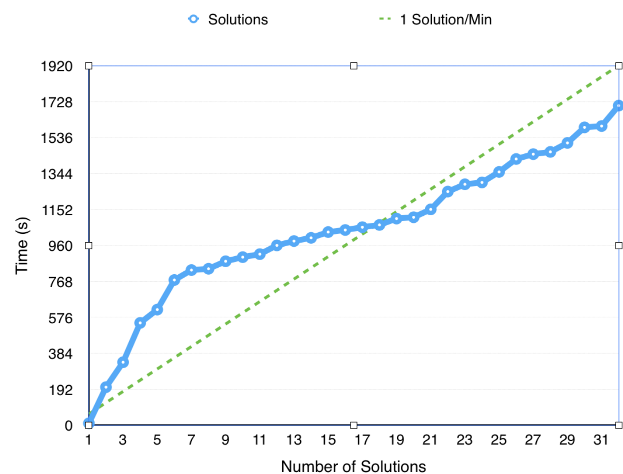

Figure 6: Time vs. number of solvable puzzles generated with fitted $y = x$ line. The system currently generates approximately one solvable puzzle level per minute.

The neural network enables the Q-learner to generalize to new puzzles and states that it has not seen before. To adapt the neural network to handle Q-learning, each action is tied to a single output node, and the expected reward is normalized to be in the interval [0,1] to enable regression (Mnih et al. 2013). (Note: we made some initial attempts to do this, but, unfortunately, we found that the neural network failed to generalize well and we had to resort to a Q-table-based approach that meant resetting the Q-Learner and having it learn each new puzzle from scratch).

Another area for potential improvement would be to improve the Q-learner's ability to say something about the enjoyability of a given puzzle. This may include moving beyond a simple Q-learner and utilising additional algorithms/methods. Specifically, the Q-learner can tell us if a puzzle is solvable, and how long it took to solve it, but does not have a(n explicit) notion of what made the puzzle solvable. Was it necessary to avoid or to visit certain tiles? The answer to this question could be incorporated into a more complex fitness function that would conceivably produce more enjoyable puzzles.

It would also be desirable to incorporate dynamically generated UI elements (e.g., visual or aural cues) relating to the properties of the tiles. This would require some notion of semantics such as associating certain colors, textures or sounds with the property of a tile (e.g., walls are heavy and dark, dangerous tiles are red or jagged, default tiles are neutral colors, etc.) Further, given some generational semantics, one can imagine giving the game the ability to blend mechanics, tile-types or even entire levels to create new classes of puzzles.

In addition, as levels become more complex, it may be desirable to incorporate some sort of tutorial ability in the game that allows players to learn tile properties and mechanics (or better yet, to learn the same semantics that the game uses to generate content) as they play (the point of the game is that players must [re-]learn at each level how to play the puzzle, but if this is too difficult, the game might provide hints to help the player adapt).

Finally, while we find the puzzles interesting and fun to play, at least at a basic level, it would be desirable to conduct user studies to see a) if the difficulty level ascribed to generated puzzles by the Q-learner-based fitness function accurately correlates with the challenge perceived by a human player, b) whether (some of) the generated puzzles are enjoyable to (some subset of) the general game-playing population, and c) if so, whether we can accurately predict which puzzles will be fun for human players (based on the current fitness measures or variations thereof).

References

Ashlock, D. 2010. Automatic generation of game elements via evolution. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 289–296.

Bauckhage, C.; Drachen, A.; and Sifa, R. Clustering game behavior data. *IEEE Transactions on Computational Intelligence and AI in Games* , to appear.

Chanel, G.; Rebetez, C.; Betrancourt, M.; and Pun, T. 2011. Emotion assessment from physiological signals for adaptation of games difficulty. *IEEE Transactions on Systems Man and Cybernetics, Part A* 41(6):1052 – 1063.

Drachen, A. 2008. Crafting user experience via game metrics analysis. presented at NORDICHI 2008, Lund, Sweden.

Ducheneaut, N.; Yee, N.; Nickell, E.; and Moore, R. J. 2006. Alone together?: exploring the social dynamics of massively multiplayer online games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 407–416.

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

Holmgård, C.; Liapis, A.; Togelius, J.; and Yannakakis, G. N. 2014. Generative agents for player decision modeling in games. In *Foundations of Digital Games*.

Horswill, I. 2014. Game design for classical AI. In *AIIDE Workshop on Experimental AI in Games*.

Hunicke, R., and Chapman, V. 2004. AI for dynamic difficulty adjustment in games. In *AAAI Workshop on Challenges in Game Artificial Intelligence*.

Khalifa, A., and Fayek, M. 2015. Automatic puzzle level generation: A general approach using a description language. In *Computational Creativity and Games Workshop*.

Lim, C.-U., and Harrell, D. F. 2014. An approach to general videogame evaluation and automatic generation using a description language. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, 1–8.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

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

Smith, G. 2014. The future of procedural content generation. In *AIIDE Workshop on Experimental AI in Games*.

Taylor, J., and Parberry, I. 2011. Procedural generation of Sokoban levels. In *Proceedings of the 6th International North American Conference on Intelligent Games and Simulation*, 5–12.

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

Watkins, C. J. C. H., and Dayan, P. 1992. Q-learning. *Machine Learning* 8(3–4):279–292.

Williams-King, D.; Denzinger, J.; Aycock, J.; and Stephenson, B. 2012. The gold standard: Automatically generating puzzle game levels. In *AIIDE*.

Yannakakis, G. N., and Togelius, J. 2011. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing* 2(3):147–161.

Yoon, D., and Kim, K.-J. 2012. 3D game model and texture generation using interactive genetic algorithm. In *Proceedings of the Workshop at SIGGRAPH Asia*, 53–58. ACM.