Evolving Unique Genre-Agnostic Game Mechanics

Patrick Stephen

December 17, 2017

1 Introduction

In recent years, rapid developments have been made in the automatic generation of creative content based on previous works through the use of deep learning and predictive text modeling. Such developments include the production of art [1], music [2], and scripts for films [3], through deep learning, and in more popular science the production of book [4] and tv show scripts [5] through predictive text. One form of media that has been absent from these developments is the automatic generation of video games.

This project seeks to fill a gap in research for generating complete games: the generation of game mechanics or gameplay primitives. Games can be considered an amalgam of art, music, and text assets along with gameplay, and it's this gameplay that there yet exists a demonstrated solution to automatically geterate. We demonstrate abstract methods for developing mechanics that are distinctly varied or unique, along with basic implementations satisfying these mechanics with experiment results as to the uniqueness of their products.

Whereas previous work in this topic of game generation tends to focus on the complete generation of games with one tool [6] [7] [8], this work is strictly not directed at making entire games with themes, art, stories, and so on, but rather focuses purely on gameplay primitives. The mentioned deep learning and predictive text approaches for making scripts and images could be combined with this work along with themed keywords to develop arbitrary games, but this work is not attempting to demonstrate that or develop an example of that.

In order to obtain the level of uniqueness we want from our mechanics, we utilize evolutionary procedures to randomly search the space of all game mechanics that can be modeled in our schema and save high fitness mechanics produced in a type of graph that rejects adding new nodes insufficiently different from existing nodes in the graph. When evolving new mechanics, if they are too similar to nodes in the graph already, they will have reduced fitness in later generations.

2 Prior Related Work

In 2000, J. Orwant described the EGGG or Extensible Graphical Game Generator to create visual games from short design descriptions [6]. Ultimately this work is more alike to a DSL for designing games than it is automatic generation, but their work does incorporate significant effort in categorizing qualities of games for what could be generated. This focus incorporates multiple players and rules such as how turns advance between players, how games might end, how informed each player is of the game state, whether there exists a referee that judges the game, and so on.

We can take from some of these definitions descriptions of the types of games that our tools and other tools are able to generate.

In 2007, Mark J. Nelson and Michael Mateas proposed a method of generating game mechanics based on word recognition to match to visualizations and verbs to match to mechanics themselves [7]. The idea of picking words for how games should be visualized is similar to how image generation is done in projects unrelated to games, and is exactly how we would extend our methods to functional games. That said, the verb usage to generate mechanics is relatively inflexible. In this sense their tools generate verbs like "dodger" to create a game where the player must avoid something, or "shooter" to create a game where the player must shoot or click on something. Not only are these game type restricted to short, three second games, but these mappings are predefined and therefore don't approach what we're looking for with the creation of unique or new game mechanics from scratch, or from a database of known games.

More recently in 2014, Alexander Zook and Mark O. Riedl developed arbitrary game mechanics for role playing games and platfomer games [8]. The methods used here rely on the genre of game defining how the mechanics will be implemented, with the genres defining restrictions on mechanics in addition to hard restrictions that apply to all mechanics. Mechanics themselves here are a sort of explicit named state manipulation, such as a spell in role playing game removing the player's mana and some corresponding removal to an enemy's health. Mechanics also need to be explicitly restricted by playability– e.g. mechanics can be generated that are unwinnable for a player. Mechanics that fail these restrictions are thrown out and replaced. Ultimately the interesting ideas here for our applications are in the combination of existing mechanics, where they combine the mechnanics in the platformer and the role playing models. Their generation model is similarly inapplicable as Nelson's work, also taking in largely hard-coded elements.

From this brief sampling of previous work, we know that some problems such as classification of game mechanics have been solved, along with the generation of complete games given a set of hard-coded base elements. This leaves us to strive to generate unique base elements instead of having to build these manually.

3 Uniqueness Graphing

To aid in the evaluation and production of unique mechanics, this project utilizes a *Uniqueness Graph*[1] which is a graph wherein all nodes are guaranteed to be sufficiently distant from one another. The implementation of this involves three elements: only adding nodes which can have their distance to other arbitrary nodes measured, refusing to add nodes to the graph which are too close in distance to any existing node, and taking this representation of a node set and converting it into a graphical representation.

In a sense the Uniqueness Graph is not a classical graph in that it does not feature connections between nodes except for implied connections by vicinity.

The interesting algorithmic element to the uniqueness graph is the production of the visual representation. This representation is important for us in order to easily grasp meaning from a set of nodes. In a situation where the nodes can be expected to follow triangle inequality, this isn't difficult, but because we anticipate putting evolutionary components in this uniqueness graph that

[1]This concept probably already exists, but I don't know what it is called it if does. Otherwise I would cite existing work that presents it.

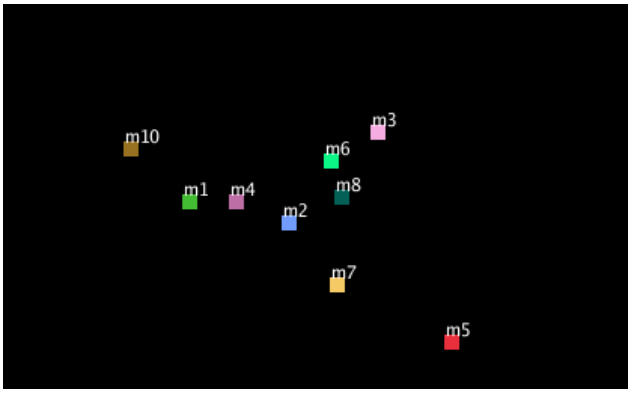

Figure 1: A sample of a uniqueness graph, visual output from the program developed for this paper. Each mechanic (m) can be queried by its number to evaluate its properties.

can have well over two or three[2] variable measures of distance between two nodes, we would need to produce an N-dimensional visual space to accurately represent the distances between nodes. We don't do this because this would defeat the purpose of the visual if it had N dimensions, the purpose being to make it easy to evaluate the data.

We resolve this by evolving a 2D representation of the graph, where all nodes in the graph are placed randomly and have their positions evolved. The fitness of a 2D representation here is the difference between each pair's distance in the 2D representation and their true distance as measured by the nodes themselves. High fitness members of this population will have less of an error between the distances in their nodes and the real distances. For our purposes we evolve this representation with 100 members over 200 generations[3].

4 Implementation

All of the described features in this paper, including uniqueness graphing, have been implemented either from scratch or from building upon previous work by Patrick Stephen in other AI papers for courses. The code for all evolutionary mechanics is built into the open source library github.com/200sc/geva. Similarly, all visualization code is built on top of work done by Patrick Stephen and Nate Fudenberg in the open source library github.com/oakmound/oak.

4.1 Mechanic Definition

We define a game mechanic as an initial state of an environment[4], a set of actions possible to use on that environment to change it, and a set of values within the environment which need to match goal ranges for the mechanic to be completed or beaten. In this way we can make mechanics

[2]For 2D or 3D representations respectively.

[3]A complete set of genetic parameters used for these generations: 100 Population Size, 4 elite members held after each generation, randomized crossover pairing, 2-member deterministic tournament selection with half of each new population parents remaining from the previous population, and a mix of multi-point crossover and average crossover. Representations of the graph are floating point vectors, with each pair a new (x,y) pair corresponding to the set of input nodes.

[4]Where an environment is a vector of floating point variables.

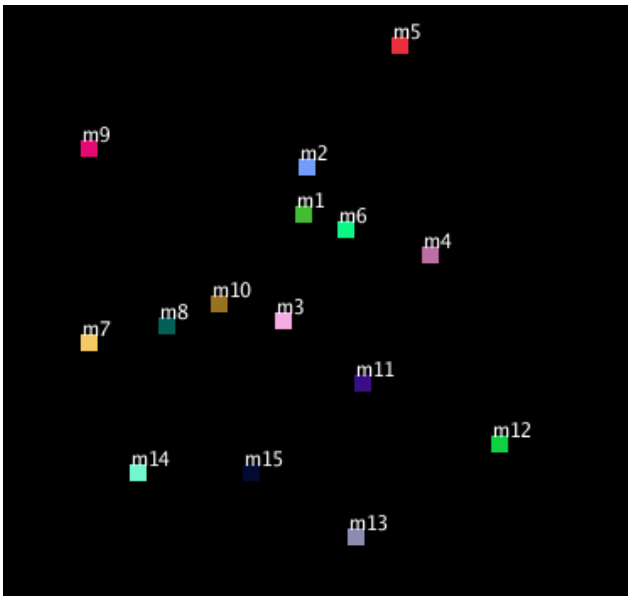

Figure 2: A second sample of a uniqueness graph, the same from Figure 1, but later in the evolutionary process.

that match those generated by Zook and Riedl [8] without hard coding anything. Where Zook and Riedl would declare an RPG mechanic as involving one player's health and mana and two enemies' health and mana, our mechanic representation can define that through six floating points where appropriate rules effect appropriate values.

Failure states in this model are not explicitly generated, but exist in the form of a condition that all goal states require which, after some set of actions, can no longer be satisfied. As an explicit example, one might have a value which needs to stay above zero (such as health or fuel) and no action in the set of actions can raise that value if another reduces it.

4.2 Mechanic Generation

We call the objects responsible for generating mechanics *Developers*. Developers are explicitly anything that can produce a mechanic when queried and be evolved in a population of like typed developers, but how the individual developer performs this is defined separately from the concept of a developer itself.

In the implementation currently, developers have a variable number of actions they can give to the environment, a variable number of initial environment spaces, variable steps to reach the goal of the environment, and so on.[5]

We avoid the problems of unsolvable games by generating mechanics in such a way where the mechanics modify their initial state with their set of actions some number of times in order to establish their goal state(s). Otherwise we would need to ensure that a path existed from the initial state to a goal state, which would be prohibitively computationally expensive.

Developers themselves are generated through developer creators, which have upper and lower bounds for each of the upper and lower bounds that developers themselves use to put mechanics together. This, along with a parallel construction in evaluation, is the only static element of the

[5]A full set of variable ranges used in creating mechanics can be found in the file geva/gg/dev/developer.go

evolutionary loop. The loop will begin with a set developer creator which will start the algorithm off with a set of random developers, and add new random developers to replace low-fitness developers every so often.

Depending on the amount of actions possible and how many actions it requires to reach a goal state from the initial state, the mechanics produced can be of arbitrary complexity and length, however due to this generation method, the mechanics this generates will be on the simpler side. This is because choosing actions randomly will often have actions that counter-act each other, so even when many actions are chosen it is equivalent to having chosen far fewer actions from initial to goal state.

Due to this, the game mechanics generated will tend to be short. This isn't particularly problematic, as most games can be considered a combination of smaller game mechanics. As an example, the role playing games and platforming games from Zook and Riedl's paper can both be structured this way. Role playing games can be seen as a series of battles, each battle their own game, along with side quests which are similarly small games. Platformers, if your overarching goal in a level is to make it to the right side of the screen, for example, can be considered a series of platforming obstacles, each of which is beaten after some amount of x distance has been attained, leading into the next obstacle, each mappable to a mechanic in this system we propose.

4.3 Evaluation

Mechanics are evaluated through two means: first the time it takes to beat a mechanic and how easy it is to beat are determined through playtesting, with each *Player* testing the mechanic deciding for itself whether those values are to its liking. Players test the mechanic developed by whatever Developer they are assigned to, and they report back a level of enjoyment from 0 to 1 from their playtesting. This value is then used to, at random, redistribute unsatisfied players to other developers to give them more chances to enjoy different mechanics. To make sure that this playtesting will terminate in a reasonable timeframe, players are given a limited amount of time to evaluate any mechanic.

The existing player implementation uses a basic Genetic Algorithm (GA) where the individual GA is a list of actions to take in order. The player creates a small set population of these GAs and evolves them briefly until they reach the time limit on playing with the mechanic or they evolve a GA that can reach close enough to the goal state that the player is satisfied.[6]

A new set of players are generated to evaluate developers occasionally throughout the iteration of the evolutionary loop, both so that developers that stick around through multiple generations produce varied games and to keep the developers evolving to new variants.

After a number of iterations moving players around, the developers with the most players are given better fitnesses than those with less players. At this point developers mechanics' are scrutinized for whether they are unique enough to be added to the uniqueness graph. If they are not, their fitness is significantly reduced. At this point, the fitness value of each developer is the fitness value of their mechanics for that generation of developers, and the top mechanics are added to the uniqueness graph (if they can be).

[6] A full set of variable ranges used in evaluating mechanics can be found in the file geva/gg/player/intEnvPlayer.go

```
Generate N Developers
for i := 0; i < M; i++ {
        Generate Q Players
        Evenly distribute Players to Developers
        for j := 0; j < P; j++ {
                Create a Mechanic for each Developer
                Assign Each Player their Developer's Mechanic
                for k := 0; k < Q; k++ {
                        Enjoyment := player[k].PlayMechanic()
                        if rand() < Enjoyment {
                                Move this player to a new Developer
                        }
                }
        }
        Sort Developers by remaining Players
        Reduce Developer fitness if their mechanics aren't unique
        Add the top Developer's mechanic to the uniqueness graph
        Evolve Developer population (Select, Crossover, Mutate)
}
```

Figure 3: Pseudocode of the evolutionary algorithm loop used to generate mechanics.

4.4 Evolutionary Loop

Figure 3 summarizes the details of this section, describing in pseudocode the evolutionary loop for generation and evaluation of mechanics and developers.[7]

4.5 Example Applications

We'll provide here a few example games whose gameplay can be entirely represented by one game mechanic in our structure.

The easiest example is that of puzzle games. One such game could be a sliding tile game. In this game, there's a grid of square tiles shuffled around, one tile is removed, and the goal is to move tiles into the empty space (thereby moving the empty space) until the original order of tiles is attained.

Figure 4 details this game, where the actions available to the player are to shift the empty space in one of four cardinal directions. The environment of this game has one variable for each tile space, each holding some integer representing which tile is currently occupying that space. Moving the empty space checks one additional environment variable to determine which two spaces' values should be swapped. The goal state is an explicit tile value for each tile space, but other, easier games could be less strict.

A more complex example would be of a platformer, where the goal is to reach a far right point from a starting point on the left. The actions here would be moving left and right and jumping, requiring two variables for the (x, y) position of the player. The goal would just be that the player's end state had a sufficiently high x value. Other values in the environment would need to define positions of gaps and obstacles, and after the player moved checks would need to occur to see if they fell in a gap or if they tried to move into such an obstacle. In the former case, one variable

[7]The real code represented by this figure can be found at geva/gg/instance.go

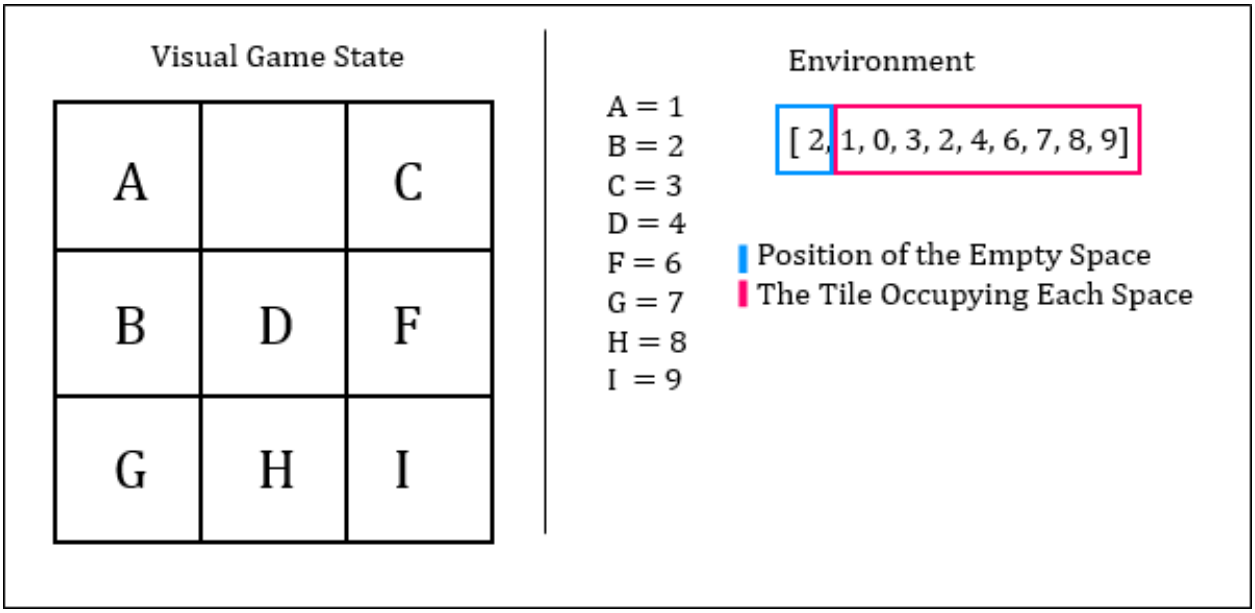

Figure 4: Left, a visual representation of a sliding tiles puzzle game. Right, the literal environment representation given our game mechanic schema.

could be set indicating the game was failed, and in the latter case the action could be reversed. Figure 5 visualizes this example.

5 Results

We ran three tests that we'll compare in this section. Figure 6 shows the parameter settings given to the program for these tests, namely developer count, player count, developer evolution iterations, player movement iterations, and play time for each player-mechanic evaluation.[8]

5.1 Enjoyment

Overall population enjoyment was tracked per generation of developer evolution. As seen in the associated Figures 7, 8, and 9, these values consistently went up seemingly regardless of our input settings, indicating more players were being satisfied by the games presented by most developers. This is not unexpected, but does confirm that the games being built are being adapted to satisfy the population effectively. As players are not maintained between generations but are re-generated from scratch, the mechanics we see can be imagined as the ideal mechanics (or approaching the ideal mechanics) for the given player creation settings instead of accommodating any particular instance of player(s).

Test 1, which gave players more time to play their games and had the fewest number of players, attained the highest enjoyment, the quickest, and either of these parameters could have been responsible. Because players had longer to play games, perhaps they have a bias towards games

[8]In addition to these settings, many mutation, creation, and crossover settings, too many to list here, were defined as the same for each generation. See the file geva/gg/instance_test.go for a complete set of parameters.

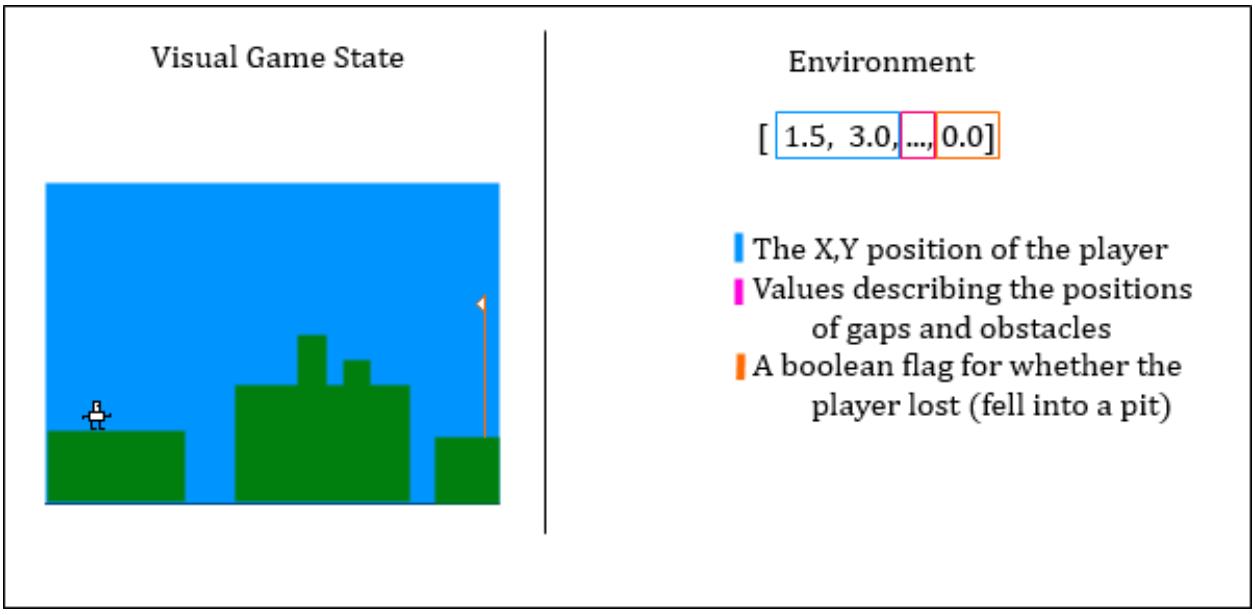

Figure 5: Left, a visual representation of a platformer puzzle game. Right, the literal environment representation given our game mechanic schema.

```
Test 1
DevCt = 80
PlayerCt = 300
DevIterations = 20
PlayIterations = 8
PlayTime = 40
```

```
Test 2
DevCt = 40
PlayerCt = 600
DevIterations = 40
PlayIterations = 5
PlayTime = 20
```

```
Test 3
DevCt = 150
PlayerCt = 700
DevIterations = 5
PlayIterations = 10
PlayTime = 20
```

Figure 6: Test parameters for each run test evaluated in the results section.

that take longer, or because there were less players, perhaps developers were able to specialize to satisfy those fewer players easier.

5.2 Uniqueness

Evaluating the uniqueness of the results is a difficult proposition, in part because of the difficulty in evaluating the concept of uniqueness, but moreso because the program by default produces only members that are sufficiently unique from other members it will produce. In this sense, we'll show the types of members that each test run produced and then evaluate uniqueness by comparing those sets to one another, given their different test parameters. The raw mechanics used for comparison here can be seen in Figure 13. The uniqueness graphs for those same mechanics can bee seen in Figures 10, 11 and 12.

For reading the mechanic data itself, each mechanic is named $t.\mathrm{M}x$, where t is a test number and x is a mechanic number, and where that same name can be found in the appropriate test's graph figure. The number of actions that are possible on the environment is the first field described for each mechanic, followed by the state of its initial environment, followed by the indices checked to match a goal state and the expected values at those indices.

There exist some clear patterns among the data just from these tests. In all three tests there are games which are clearly defined as they are due to some starting creation settings: 1.M1, 2.M1, 2.M2, 2.M3, and 3.M1. These tests share similar environment sizes and all use the number -5 frequently.

Another pattern is in the presence of trivial game mechanics. These are 1.M2, 2.M16, 2.M27, 3.M1, and 3.M4, where either the game starts out in a solved state or there only exists one action to perform which will inevitably bring the game to a goal state. That test 1 evolves out of these but that 2 doesn't and 3 is dominated by these might suggest that this sort of mechanic shows up more often when less play time is given to players to resolve games.

The final pattern which is obvious to see is the presence of games where there is a single variable representing the mechanic. These are 1.M13, 2.M38, and 2.M40. This last pattern probably doesn't show up in test three because of it's limited iterations.

Aside from these examples, no clear patterns stand out from reading the text output describing the mechanics, but the uniqueness graphs suggest that 1.M18, 1.M19, 2.M31, and 3.M5 could be examined for being particularly different than the rest of their population.

6 Future Work

6.1 Game Variety

A number of varieties of games were discounted from what we could generate to aid in the simplicity and quick evaluation of our toolset. Part of our work in the future could involve bringing these elements back in. For one, our game mechanic definition doesn't allow for games where there are ongoing passive effects in the environment such as gravity. In addition we don't consider competitive multi-agent environments. The opponents of the player are explicitly those that exist in the environment in the initial state. Passive effects would also enable more varieties of obstacles that would satisfy pseudo-enemies, such as an enemy that passively moves around, manipulating a pair of (x,y) values in the environment, checking if they hit the player and causing a failure state or reducing health if they do.

6.2 Combining Mechanics

As mentioned in section 4, the mechanics we develop here are self-contained games that can either serve as large games or small sub-games that could be stitched together. These games, however, don't necessarily have any common variables and therefore the problem of combining multiple games mechanics into a series to form a larger, full game is still present.

This could be approached by starting with the uniqueness graphing and looking at extremely similar mechanics, then attempting to combine those similar mechanics into a chain, likely through some tool which would connect some number of variables between mechanics so that the values would pass over from one mechanic to the next.

6.3 Complete Game Generation

As mentioned in the introduction, the tools we provide here don't directly produce full games, but in combination with other work, could. Significant work needs to happen to structure what exactly is required by any given game, but once that is defined, wherever art or voice acting or story or so on is needed, keyword queries could be sent to external generation tools to produce those assets.

7 Conclusion

We've developed a library which can produce unique game mechanics and visualize their distinctness from one another for use in aiding game design. As of yet there is no automated application of these mechanics to create real games, but game designers can observe the output of the program (tweaking any of hundreds of parameters to adjust the results) and use the mechanics presented as inspiration to develop new game types or ideas.

With additional computation time and more development work towards organizing the code rapidly developed for this project, many more features to enhance the theoretical benefit of this tool for generating game mechanics could be expanded upon.

References

[1] L. A. Gatys, A. S. Ecker, and M. Bethge, "A neural algorithm of artistic style," *CoRR*, vol. abs/1508.06576, 2015.

[2] M. C. Mozer, "Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multi-scale processing," *Connection Science*, vol. 6, no. 2-3, pp. 247–280, 1994.

[3] A. Newitz, "Movie written by algorithm turns out to be hilarious and intense," 2016.

[4] "Harry potter." `http://botnik.org/content/harry-potter.html`. Accessed: 16-12-2017.

[5] "Scrubs s0xe1." `http://botnik.org/content/scrubs.html`. Accessed: 16-12-2017.

[6] J. Orwant, "Eggg: Automated programming for game generation," *IBM Systems Journal*, vol. 39, no. 3.4, pp. 782–794, 2000.

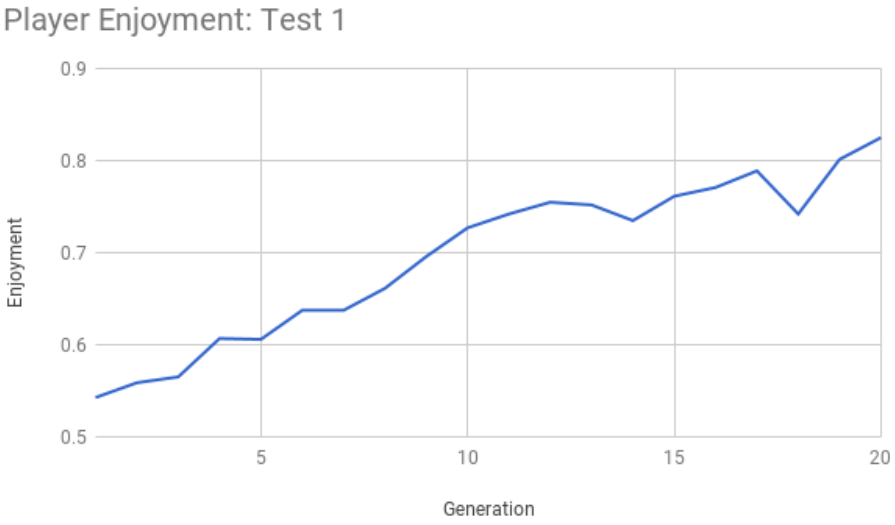

Figure 7: Enjoyment over generation for test 1.

[7] M. J. Nelson and M. Mateas, *Towards Automated Game Design*, pp. 626–637. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

[8] A. Zook and M. O. Riedl, "Automatic game design via mechanic generation.," in *AAAI*, pp. 530–537, 2014.

8 Result Figures

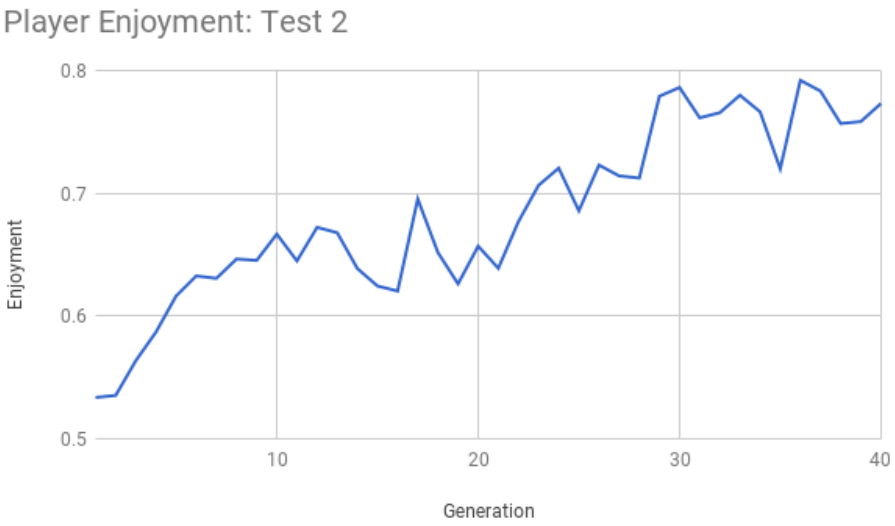

Figure 8: Enjoyment over generation for test 2.

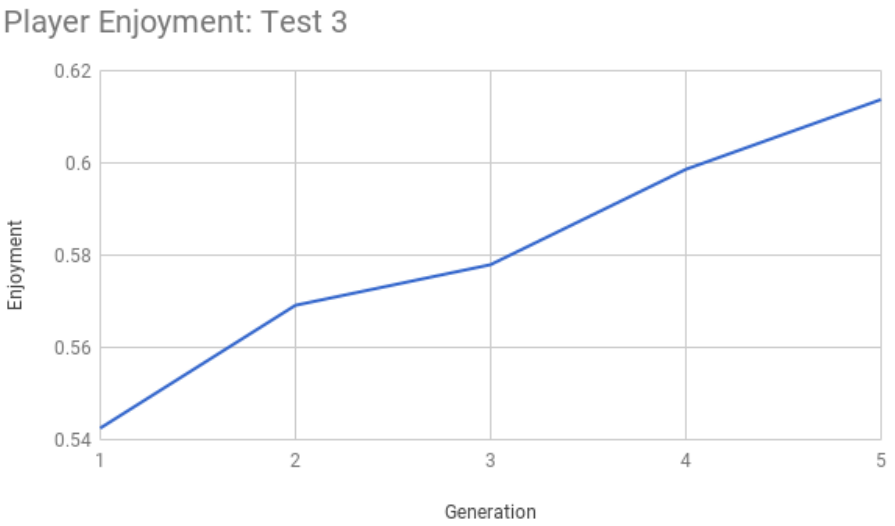

Figure 9: Enjoyment over generation for test 3.

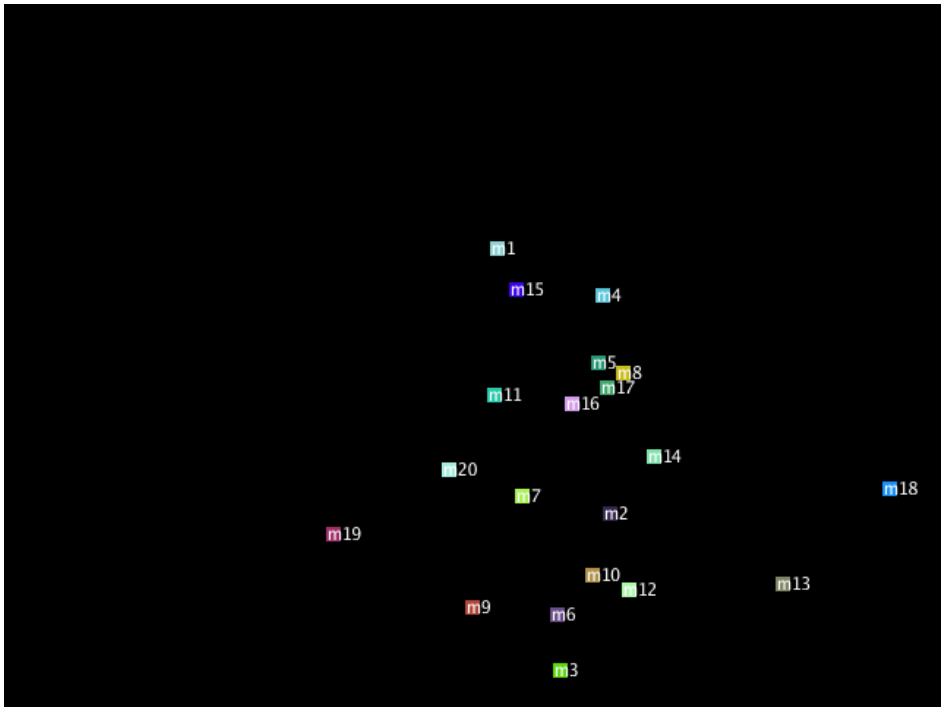

Figure 10: The uniqueness graph for test 1.

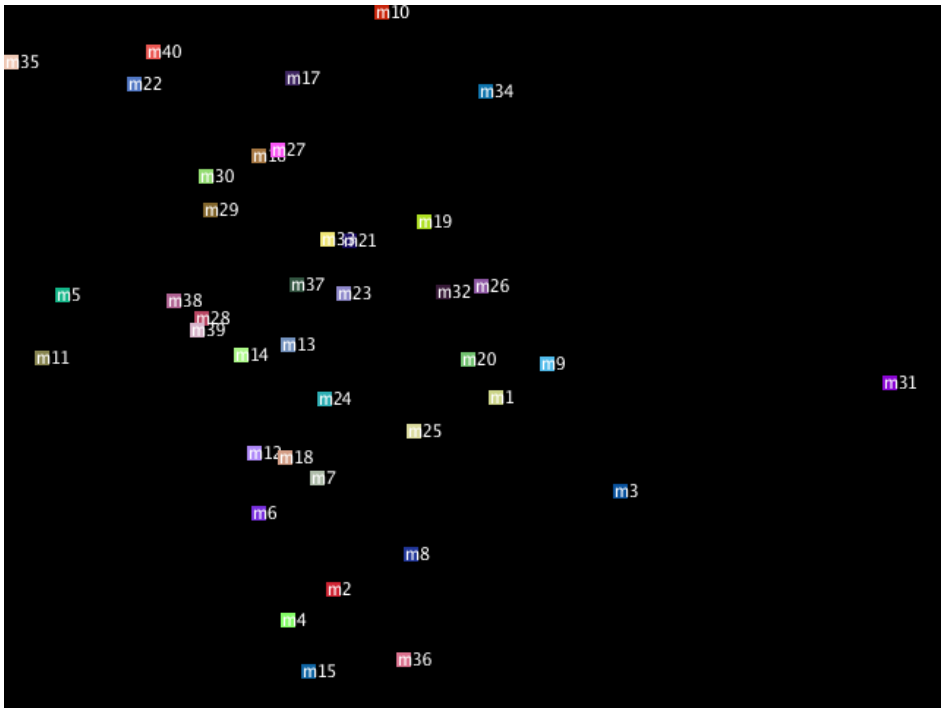

Figure 11: The uniqueness graph for test 2.

Figure 12: The uniqueness graph for test 3.

```
Test 1
M1: Actions: 15 Env: [0 0 -5 0 0 -5] Goal: map[3:4.3948 0:-2.5472 5:-6.0302]
M2: Actions: 7 Env: [0 2.9013 2.2354 0 2.3413 2.3588 0] Goal: map[1:2.9160]
M3: Actions: 27 Env: [0 1.6217 1.9339 0 0 1.7893 0 0 0] Goal: map[2:-10.7437 6:0]
M4: Actions: 60 Env: [0 4 0 4 4 0 4 0 0 4 0 0] Goal: map[1:0.6062]
M5: Actions: 40 Env: [0 4.5 0 0 0 0 0 0 0 0 0 0 4.5 4.5 0 0 0 4.5 4.5 0] Goal: map[12:13.5696 16:0 8:0]
M6: Actions: 60 Env: [0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2] Goal: map[16:0.5039 0:0]
M7: Actions: 6 Env: [1.4395 1.334] Goal: map[0:18.6862]
M8: Actions: 6 Env: [1.3129 1.4907] Goal: map[1:144.9977]
M9: Actions: 18 Env: [2.3009 0 2.3153 0] Goal: map[0:2.3009]
M10: Actions: 40 Env: [21.375 21.717 21.355 20.845] Goal: map[2:15.4609 1:918.0717 3:8.7146]
M11: Actions: 45 Env: [22.407 22.361 22.747 22.206 21.902] Goal: map[2:-8.9616 1:-2.2856 4:6.7407 0:4.5319 3:17.5356]
M12: Actions: 164 Env: [0 0 21 21 21 21 0 0 21 21 21 21 0 21 0 0 21 21 0] Goal: map[4:20.2605 18:-0.7242 1:-1.1067 17:21 10:21 0:0 5:21 6:0
   2:20.6539 14:0 7:0]
M13: Actions: 7 Env: [4.8355] Goal: map[0:5.8198]
M14: Actions: 76 Env: [10.267 10.844 10.28 10.269 10.32 10.016 10.031 10.388 10.272 10.142 10.826 10.884 10.851 10.655 10.878 10.647 10.983]
   Goal: map[0:10.2666 5:0.1671]
M15: Actions: 60 Env: [10.035 10.652 10.197 10.898 10.289 10.526 10.72 10.317 10.111 10.375 10.174 10.968 10.811 10.186 10.105 10.247 10.625
  10.147 10.079 10.684] Goal: map[11:10.9679 6:10.7200]
M16: Actions: 153 Env: [0 0 0 0 0 0 0 0 0 25.985 0 0 0 0 0 0 0 0] Goal: map[1:0 4:0]
M17: Actions: 72 Env: [0 0 5.1874 0 0 0 5.2389 5.1666 0 0 0 0 0 0 0 0 0 0] Goal: map[9:0 3:0 6:2.4804e+54 8:0 13:0 5:0 15:-0]
M18: Actions: 11 Env: [17.502 17.753] Goal: map[1:9.8732e+173 0:-7.3219e+163]
M19: Actions: 12 Env: [8.2791 8.7803 8.4809 8.5208 8.8389 8.6347] Goal: map[2:-14.7368 3:373.5512 5:-171.8703 4:288.5832 0:-18.6940
  1:-42.4946]
M20: Actions: 12 Env: [9.4631 9.9474 9.6141 9.3737 9.9529 9.5408 9.5894 9.2731 9.6091 9.5887 9.9481 9.5427] Goal: map[6:-81.8686 10:186.5505
   2:133.5345 8:-3.7581 5:-81.1749 11:49.0480 7:139.9828 0:242.6188 4:-322.5642 3:117.0011 9:7.5740 1:-4.3797]

Test 2
M1:  Actions: 13 Env: [0 -5 0 0 -5 0] Goal: map[2:-2.1765 3:-2.7243 0:-2.7131]
M2:  Actions: 25 Env: [0 -5 -5 0 -5 -5 -5 0 0] Goal: map[6:-0.0057]
M3:  Actions: 12 Env: [0 0 0 -5 0 -5 -5 0 0] Goal: map[1:0 5:-5]
M4:  Actions: 64 Env: [0 1.8883 1.2895 1.5658 1.9594 0 0 0 0 0 1.7104 0 0 0] Goal: map[3:1.5658 10:1.7103 2:1.2895]
M5:  Actions: 96 Env: [0 1.728 0 1.8174 1.0189 0 0 1.9963 1.0018 1.7465] Goal: map[7:-60.1066 3:1.8173]
M6:  Actions: 42 Env: [0 0 0 0 6.0811 0] Goal: map[4:6.0811]
M7:  Actions: 27 Env: [0 0 6.2263 0 0 0 5.8826 0 0] Goal: map[2:-0.9233 4:-0.3779]
M8:  Actions: 16 Env: [0 6.4732 7.0023 6.3454] Goal: map[1:-1.0450 0:-1.4695 2:-1.7446]
M9:  Actions: 79 Env: [0 0 0 2.4763 0 2.4078 2.4963 0 0 0 0 2.4011 2.4407 0 2.402 2.3838 0 2.4074] Goal: map[3:15.2973 0:0 10:35.5774
  7:-0.6212 12:1.2446 14:13.4048 11:21.8113 15:17.6440]
M10:  Actions: 32 Env: [0 7 0 0 0 0 7 7] Goal: map[5:3.6990 3:0.6346 1:3.2526]
M11:  Actions: 50 Env: [0 0 0 23.595 23.059 24.345 0 23.273 0 0 0 0 0 0 0 24.38 23.782 22.55 0] Goal: map[12:-3.0868 13:-4.6038 15:-3.7093
   0:-4.1742 6:-3.3071]
M12:  Actions: 72 Env: [6.176 0 7.5574 0 6.6439 7.5078 7.6338 6.8942 6.696 0] Goal: map[2:9.5448 3:-9.2565 6:6.4390 5:-1.9463 9:-0.7035
  1:-1.5369 8:8.9439 7:-3.5655]
M13:  Actions: 7 Env: [0 0 0 3.5 0 3.5] Goal: map[5:400.0232 3:3.3849e+13 4:0]
M14:  Actions: 8 Env: [0 0 0 0 3.5 3.5] Goal: map[1:0 0:0]
M15:  Actions: 13 Env: [7 0 0 7 7] Goal: map[2:0]
M16:  Actions: 2 Env: [9 0] Goal: map[0:1.0009 1:0]
M17:  Actions: 87 Env: [1.9448 1.8838 0 0 1.5508 0 0 0 0 1.8037 0 1.9682 1.2927 0 1.8205 0 1.574 1.1209 0 1.3533] Goal: map[15:0]
M18:  Actions: 93 Env: [0 1.2442 1.6597 1.3705 1.9406 1.3966 0 0 1.9986 1.3117 0 1.6445 0 0 0 0 0 0 0 0] Goal: map[7:0]
M19:  Actions: 32 Env: [0 2 2 0 0] Goal: map[3:6.4749]
M20:  Actions: 100 Env: [0 0 0 0 0 0 6.7437 0 0 0 0 0 0 0 0 0 0 0 0 0] Goal: map[16:0 15:0 3:0 1:8.5867 0:0 4:0 8:8.0262 7:0 11:0 14:9.6628
  9:7.5185 2:0 12:0 5:0 13:0 19:0 10:8.4528 6:8.0859]
M21:  Actions: 200 Env: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] Goal: map[2:0 5:7.4218 9:11.4241 18:0 8:7.7622 10:0 19:0 14:10.2174 12:0
  1:0 11:8.8897 3:0 4:0 7:0 16:8.3384 15:0 17:7.5191 6:0 13:0 0:0]
M22:  Actions: 181 Env: [0 0 0 0 0 0 0 0 6.8697 0 0 0 0 0 0 0 0 0 0] Goal: map[0:8.6145 12:9.6844 11:8.5824 13:12.7668 17:0 2:0 4:12.8967
  5:8.4527 8:8.8924 7:0 16:0 3:8.0129 6:0 1:0 9:0 18:11.3831 14:0 10:9.6180 15:9.2123]
M23:  Actions: 20 Env: [0 0 0 7.1832 7.1619 0 0 7.1819 7.2134 0] Goal: map[0:0 4:34.5523 9:0 7:30.0500]
M24:  Actions: 38 Env: [0 9.1746 0 9.1946 0 0 9.2168 0 0 0 0 9.1701 0 0 0 9.1995 0 0 0] Goal: map[15:58.6387]
M25:  Actions: 160 Env: [22.075 22.708 22.552 22.056 22.356 21.778 22.457 22.124 22.709 22.126 21.644 22.155 22.422 21.761 22.323 22.516]
  Goal: map[5:21.7775 6:22.4573 8:22.7090 1:22.7080 11:22.1554 14:4.4533 12:22.4224 0:4.6562 9:4.7317 4:22.3556 3:22.0561 7:22.1241 13:21.7611]
M26:  Actions: 5 Env: [16.006 16.004 16.031 16.028 16.008] Goal: map[4:190.4873 3:184.8911 1:186.0150 2:183.6257]
M27:  Actions: 1 Env: [31.051] Goal: map[0:372.6792]
M28:  Actions: 182 Env: [0 0 0 0 0 0 0 0 0 0 0 0 0 11.532 11.791 0 0 0 0] Goal: map[0:0 11:0 7:0 8:0 18:0 1:0 15:0 10:0 6:0 2:0 3:0 17:0
  9:0 4:0 5:0 16:0]
M29:  Actions: 71 Env: [0 16.291 0 16.272 16.307 0 0 0 0 0 16.27 0 16.295 0 16.301 16.292 0 0 0 0] Goal: map[9:2.3113 8:2.3735 12:2.1458
  10:2.2423 2:2.3221 1:2.2637 3:2.1767 11:2.2024 16:2.2088 18:2.1787 14:2.0718 17:2.3604 6:2.2475 0:2.1521 7:2.3149 15:2.3617 13:2.0787
    5:2.1517 19:2.1279 4:2.2617]
M30:  Actions: 13 Env: [9.1341 9.1314 9.1446 9.1261] Goal: map[1:5.2596 3:5.7424 2:5.1989 0:5.6631]
M31:  Actions: 83 Env: [146.01 147.28 144.19 146.23 145.84 149.25 145.94 145.01 144.49 148.54] Goal: map[4:664985.4567 2:-4.6686e+06
  6:1.6455e+14 0:73079.3861 8:-2.4102e+07 5:4.6863e+10 7:5.6365e+07 1:1.7233e+10 9:4682.8085 3:32769.2270]
M32:  Actions: 24 Env: [144.44 147.76 147 147.61 149.04 145.06 146.65 144.49] Goal: map[6:17826.5283 2:112227.8350 0:-2.7423e+09 3:47429.9567
   5:145.0556 1:2.4030e+07 4:116791.7536 7:-7.5711e+09]
M33:  Actions: 70 Env: [20.794 20.433 20.831 20.841 20.845 20.734 20.544 20.639 20.788 20.668] Goal: map[8:22.6988 0:19.3096 1:22.4002
  7:23.0378 4:26.6333 9:14.8728 5:23.1295 6:25.7584 2:23.1814 3:27.1324]
M34:  Actions: 20 Env: [18.516 18.661] Goal: map[0:19.3727 1:20.3968]
M35:  Actions: 36 Env: [22.325 22.611 22.441 22.54 22.361 22.848 22.558 22.77 22.575] Goal: map[1:22.6108 6:2.8012]
M36:  Actions: 68 Env: [0 0 25.211 0 25.484 0 25.183 25.292 0 0 25.152 25.071 0 0 25.387 0 0 0 0 25.502] Goal: map[8:16.4820 11:18.2209
  2:15.1901 15:18.5423 12:16.7703 3:17.5529 14:16.2691 0:16.1708 1:15.5406 19:15.4889 5:18.3404 17:14.6253 16:18.9822 4:15.9242 10:16.2089]
M37:  Actions: 18 Env: [18.514 18.648] Goal: map[0:30.5118 1:19.6553]
M38:  Actions: 9 Env: [37.818] Goal: map[0:52.9359]
M39:  Actions: 20 Env: [12.687 12.729 12.714 12.712 12.719 12.685 12.684 12.72 12.668 12.672 12.732 12.662 12.654 12.663 12.656 12.654 12.664
  12.667 12.734 12.671] Goal: map[6:3.3117 3:18.9916 18:2.7434 12:4.5548 0:108.8065 2:32.5905 8:8.1374]
M40:  Actions: 9 Env: [36.987] Goal: map[0:2.5454e+76]

Test 3
M1: Actions: 19 Env: [0 -5 -5 0 0 0 -5 0]  Goal: map[3:0]
M2: Actions: 20 Env: [0 0 1.6147 0 0 1.0067 1.0551 0 1.007] Goal: map[5:1.0067 2:1.6146]
M3: Actions: 24 Env: [0 0 0 0 0 0 -5 0 0] Goal: map[4:-0.0549]
M4: Actions: 1 Env: [1.5633] Goal: map[0:1.3989]
M5: Actions: 40 Env: [0 4.8089 0 4.2046 0 0 5.9395 4.4293 0 4.3619 4.4568 4.8499 0 0 4.2991 4.2087 0 0 4.4385 0] Goal: map[7:4.4292]
```


Figure 13