Creating Novel Boards in Ticket to Ride using Grammatical Evolution

Ryan Smith
CPE 570
California Polytechnic State University
San Luis Obispo

ABSTRACT

Procedural Content Generation is a popular avenue of research currently employed by many innovative game developers. One large area of focus is that of procedural map generation, or automated level design. Many current researchers employ genetic approaches in this problem domain, but the application of Grammatical Evolution as pertaining to this topic has sparsely been touched upon.

This paper develops a system using Grammatical Evolution to create novel game boards for a computational edition of the board game "Ticket to Ride". The generated boards are evaluated on various metrics, and compared to the same metrics evaluated on an existing standard of a "good" board, the default North American Ticket to Ride board. Using this evaluation criteria, we show that novel board generation using this approach of Grammatical Evolution is both possible and effective.

I. INTRODUCTION

Map generation is an aspect of Procedural Content Generation (PCG) that has been vastly explored. Current strategies range from Answer Set Programming approaches to various evolutionary methods. This paper intends to expand on the evolutionary approach to procedural map generation, and uses Grammatical Evolution to implement PCG for the board game, Ticket to Ride. I intend to show that novel board generation in the game "Ticket to Ride" is achievable using Grammatical Evolution as the main approach.

The remainder of this paper is organized as follows. Section II will give an overview of the game mechanics of Ticket to Ride, and the background of Grammatical Evolution as a PCG technique. Section III will present related work done in this area (procedural map generation), as well as more discussion of previous work in Grammatical Evolution. Sections IV and V give an overview of the system design and sample output, respectively. Section VI gives our analysis of the results, and Section VII concludes the paper with recommendations for future work.

II. BACKGROUND

Ticket to Ride

Ticket to Ride is a largely successful board game released in 2004 by Days of Wonder [5]. The Euro-style board game features a map of a region of the world, with specific locations shown and connected by potential railways. This set up connects well with graph theory, and has even been used to educate students on the basics of graph theory in computing courses [4].

There are currently 13 adaptations of the board game, with the European and North American versions being among the most popular [5]. Days of Wonder regularly releases new boards, with slightly varied rules, in order to appeal to new audiences and to reduce the level of repetition associated with playing the board game.

In recent years, Days of Wonder has released a computer game edition of Ticket to Ride, also available as a phone application. The drawback of the application is that players can get bored of playing the same set of board configurations repeatedly, and recent reviews have said that the digitization of the game draws attention to its lack of original experiences in each play through. This is a main aspect of the motivation behind using Ticket to Ride as the subject of this experiment in PCG.

Gameplay

The board for the North America version of Ticket to Ride is shown in *Figure 1*. Each labeled circle corresponds to a city, or destination, that players create paths through using the colored routes in between.

The goal of the game is to score the most points, where points are awarded by the number and size of routes built on by a player, and by fulfilling any Destination Ticket's that a player might posses. These tickets simply have two destinations and a point value associated with them, which represents how many points a player is awarded if they are able to connect the two destinations shown on the card. In most cases, the number of points is equal to the length of the shortest path between the two destinations (There are a few exceptions where the number of points awarded is 1 or 2 points higher

Figure 1. Ticket to Ride board for North American edition of game.

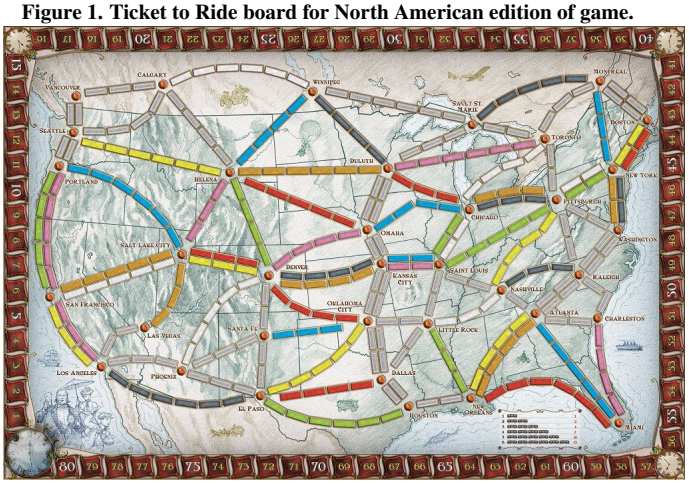


Figure 2. All Train Card types in Ticket to Ride. The Locomotive (rainbow color) serves as a wild.

than the shortest path). Any Destination Tickets held by a player that have not been completed at the game's end subtract their corresponding points from the player's total score.

In order to build a connection between two destinations, a player must play the number of train cards (shown in *Figure 2*) specified, and corresponding to one of the colors available on that path. Gray paths are wild, meaning any color can be used to build on the route.

The game is started with each player picking 3 Destination Tickets, and deciding to keep at least 2. The player then draws 7 Train Cards, and the first player takes their turn. Each turn is conducted in one of three ways: A player draws 2 Train Cards, draws 3 Destination Tickets (but only has to keep 1 this time), or builds on a route. Each turn is conducted in this manner until a player has 2 or less trains (the object with which the player marks that they have completed a path) remaining. A final round is conducted, and then each player's score is calculated. In addition to the score from the routes built and the Destination Tickets, the player with the longest continuous path is awarded an additional 10 points. The player with the largest resulting score is then crowned the winner.

Procedural Content Generation

Procedural Content Generation is the field concerned with automating the design process for creative content. This applies to games, along with a broad array of other avenues for entertainment. Instances of creative content generated in games include vegetation rendering [2], level layout generation [7] and in-game items/weapons [1], among a wide variety of other game aspects.

The specific application of PCG to map generation has been classified as a search problem. Each possible map can be represented as one combination of state variables, and the objective is to find such a combination that is acceptable to a situation. There are many different approaches to search problems in general, but a favorite among PCG practitioners is that of evolutionary programming.

Evolutionary programming involves, unsurprisingly, evolving a particular generation of states subject to some criteria, or fitness function. Each generation is subject to possible mutations, and optional crossovers are implemented to simulate genetic breeding in the natural selection process. The theory behind this method states that over time, a satisfiable solution will present itself after the weeding out of some number of prior generations. Evolutionary programming in and of itself is a high level process for conquering search problems, and lends itself well to specific implementations of the details involved. One such implementation is Grammatical Evolution.

Grammatical Evolution

Grammatical Evolution is a (relatively) new area of work in evolutionary search algorithms. The approach combines the evolutionary programming process with a grammatical representation of the problem. Specifically, a grammar is created that can map a genotype, a collection of state variables in the search space, to a phenotype, a representation of the genotype that is used in the game and that can be evaluated by some fitness function.

One of the main advantages associated with Grammatical Evolution is its ability to narrow the search space to syntactically correct phenotypes [6]. This is done by allowing the genotype to simply contain either a binary string or string of integers. All evolutionary techniques are performed on the genotypes, and evaluation is then conducted on resulting phenotypes.

The phenotypes are generated by the grammar created in accordance with the problem. The grammar, when represented in Backus-Naur Form, shows the various options that can be taken from a specific stage of the grammar. An example grammar in Backus-Naur Form from [7] is shown below:

$$
\begin{aligned}
&< chunks > ::= < chunk > \mid < chunk >< chunks > \\
&< chunk > ::= gap(< x >, < y >, < w_g >) \\
&\quad \mid platform(< x >, < y >, < w >) \\
&< x > ::= [5,95] \quad < y > ::= [3,5] \\
&< w_g > ::= [2,5] \quad < w > ::= [3,15]
\end{aligned} \tag{1}
$$

This example shows the stop by step process with which a grammar is applied to convert a genotype to a phenotype. The phenotype is initialized with *chunks*, which then breaks into either a terminal *chunk* or one accompanied by more *chunks*. The decision on which option to select is governed by the genotype: each stage of the grammar requiring a decision

uses the next integer or bit of the genotype string to decide the outcome. The *chunk* is then broken down into one of two functions, *platform* or *gap*, which take a set of parameters. The parameters, like the decisions for which grammatical option to select, are determined by the proceeding genotype element. For example, on resolving an x variable, the next genotype element will be converted to a number between 5 and 95.

Functions, like the *platform* and *gap* functions shown in (1) are callable by the grammar. However, the less heavy lifting done by the functions, the more effective Grammatical Evolution will be. As functions become more atomic, the evolutionary aspect of the algorithm can affect a wider range of state variables.

III. RELATED WORK

There has been a large amount of work done in procedurally generating a map or level design. Some of the more relevant research in the area is outlined below.

Sorenson and Pasquier [8] present an evolutionary model for generic level design. In their paper, they present an system making use of the FI-2Pop genetic algorithm approach as a means to solve an optimization problem subject to constraints [8]. They encode genomes as lists of Design Elements, an abstract way to refer to the atomic components of a game [8]. While their approach yields positive results when applied to two different gaming styles (Super Mario Bros. and a Legend of Zelda-like dungeon game), the method is too generic in nature for our use. Our goal is to focus less on constraints, and more on creating similar levels/boards to an original example.

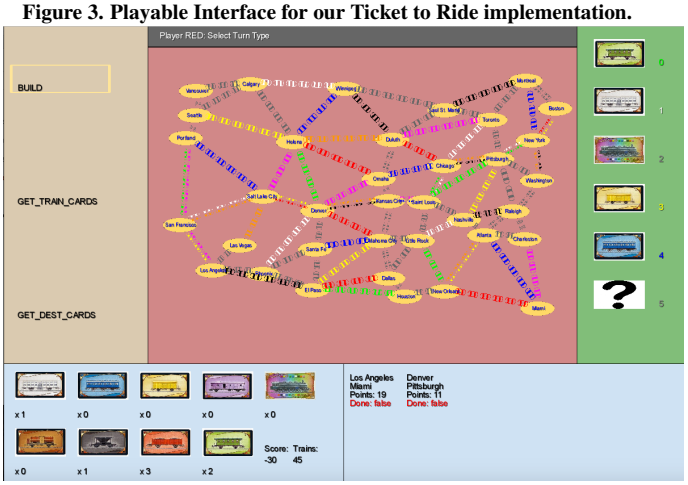
Johnson et. al [3] make use of cellular automata for real time generation of 2D dungeon-like cave levels. This approach, while certainly effective in some situations, is much too unpredictable for use with our problem.

Finally, Shaker et. al [7] employ Grammatical Evolution in level design of platform levels for Infinite Mario Bros. The authors' system for level generation is driven by a context-free grammar, part of which is shown in (1) [7]. The resulting phenotypes are then subjected to a fitness function which merely measures the amount of overlapping chunks and the number of chunks provided [7]. While the results of the paper are positive, with the authors advertising broad distributions in various metrics associated with level complexity, the lack of a good fitness function is cause for concern. In our approach, we base the fitness for the phenotype off of comparison with a baseline enjoyable experience, the original North American Ticket to Ride board.

IV. BOARD GENERATION

The system for automatically generating new boards is composed of 3 components. First, it was necessary to implement the game in a computational capacity (the Ticket to Ride computer game is not open source, unfortunately). Next, opponent AI was developed so as to be able to automate game play a vast number of times for aiding the fitness function. Finally, the Grammatical Evolution approach to designing novel maps was implemented.

Figure 3. Playable Interface for our Ticket to Ride implementation.



Implementing the Game

The mechanics of Ticket to Ride are fairly straightforward, and the implementation of the game was relatively simple. The GUI associated with a human player is shown in *Figure 3*.

The left side of the screen shows the 3 possible turn choices for a player at the start of their turn. The bottom left corner of the screen shows the player's hand of Train Cards, along with their score and number of trains left. The bottom right corner shows the player's collection of Destination Tickets, as well as whether or not the ticket has been completed. The right side of the screen shows the "flop", the 5 cards dealt face up next to the deck which a player can select Train Cards from, as well as the deck, represented by a question mark.

The final component of the GUI is the middle of the screen, where the map is represented. The map shown in *Figure 3* is the default North American map, the same as shown in *Figure 1*. This section of the screen is generated by assignment of each destination to an x, y coordinate pair, with a fixed radius. The routes are then displayed based on their length (number of tracks), and the line segment between the two destinations. The default North American map is created through use of a JSON file to store an array of destinations and their routes.

It is important to note that generating a game board can be done with solely a list of routes, as long as those routes also contain the destination names on either end of the route as well as the coordinates for those destinations.

Opponent AI

The bulk of this paper is focused on novel board generation, and thus opponent AI is not implemented as thoroughly as possible. For our purposes, a simple greedy agent designed to efficiently fill Destination Tickets and maximize points from route building is satisfactory.

Our opponent AI is programmed according to a set of simple if then statements. At the onset of the game, the agent selects Destination Tickets up to those it can fill with the amount of trains remaining. It then creates shortest paths between

each of the starting and ending destinations on the Tickets. These shortest paths are found using a Uniform-Cost search approach, where the cost of edges the agent has already built on is set to 0. This allows the agent to simply add a smaller amount of routes to an existing path in order to connect two distant destinations. These shortest paths are updated at the beginning of each turn, so that if a path becomes blocked the agent can adjust and find a new path between the destinations.

The agent's actions are governed primarily by each of its paths that it is working towards. If it can build on any of the routes in its list of paths, it does so. If not, it draws Train Cards with preference given to all route colors in its paths. Once the agent has fulfilled all of its Destination Tickets, or if the remainder are unreachable (due to being blocked by another player, or because the agent does not have enough trains left to reach them), it draws more Destination Tickets and chooses them with the same logic as used in the start of the game.

While this is a very basic implementation of an opponent AI agent, they are suitable for running automated games on various generated maps. In a number of occasions while play testing, the agents even defeated a human player actively trying to win (never one on one, however).

Genetic Algorithm

Here we discuss the heart of our system: the approach to generating novel instances of maps for Ticket to Ride.

Grammar

$$
\begin{aligned}
&< map > ::= newOrExistingDest(< x >, < y >)\\
&< newDest > ::= < route >^2 \mid ... \mid < route >^7\\
&< route > ::= newRoute(newDest(< x >, < y >, < name >),\\
&\quad < dest_f >, < routeParams >)\\
&< dest_f > ::= newOrExistingDest(< \theta >, < trackLength >)\\
&< routeParams > ::= < trackColoring >\\
&< trackColoring > ::= < trackColor >\\
&\quad \mid < trackColor >< trackColor >\\
&< trackColor > ::= RED \mid YELLOW \mid BLUE \mid BLACK\\
&\quad \mid GREEN \mid PINK \mid ORANGE \mid WHITE \mid GRAY\\
&< x > ::= [0,1] \;\; < y > ::= [0,1]\\
&< \theta > ::= [0, 2\pi] \;\; < trackLength > ::= int([1,6])\\
&< name > ::= randomName()
\end{aligned}
\tag{2}
$$

The grammar used for converting a genotype to a phenotype is specified in (2).

Each time the newOrExistingDest function is called, a <newDest> destination may be initialized (the first call by map will always initialize a newDest). This is determined by if the given angle and track length (which is mapped to an associated distance) is within range of an existing destination. If so, the route is assigned to that destination. Otherwise, a newDest destination is created. This continues until every newDest being created fails to create any more new destinations. This is made possible by constraining the boundaries of the board, and only allowing valid angles for those destinations around the perimeter.

Genotype

Our genome consists of a 1200 length integer array, ranging from 0 to the maximum int value. In order to determine integer selections in the grammar, a mod function is used on the current genome index. For determining continuous variables in a range, we simply divide the integer by the maximum int size, and use that as the proportion of the interval where the continuous number is selected.

Fitness Function

Our fitness function involves generating the map from the genome, using our grammar. We then conduct a large number of simulations (settled on 100) and average the fitness function performed over each run. This measure consists of a weighted percent error against the metrics shown in *Table 1*. This formula is shown in (3).

$$
fit = \frac{1}{\sum weight_i \cdot \frac{abs(observed_i - default_i)}{default_i}} \tag{3}
$$

Additionally, if a stalemate occurs during the game (if there are no routes to play for an agent and the Train Card deck is empty), the fitness for that particular simulation is assigned a score of 0.

Metric	NA Board Value	Weight
Avg Score	70.86	1.5
Score Var	366.09	0.8
Num Dests	36	0.6
Routes per Dest	4.33	0.8
Avg Turns	195.42	1.0
Perc Train Card Turns	0.64	0.8
Avg Path Changes	11.24	0.4
Clutter	12.44	1.5
Route Intersections	3.64	0.6
Dests Covered	0.01	0.7

Table 1. List of Metrics included in fitness for a single game simulation

The default NA Board Values in *Table 1* are achieved from running 10000 simulations of agents playing on the default North American board, and taking the average of all of the metrics. The weights decided on in *Table 1* resulted from tuning the weights subject to human feedback of the author.

Mutation

Through the evolutionary process, at the onset of each generation each genome is subject to a 5% probability of mutation. Within this mutation, each index of the genome is subject to a 0.2% chance of mutation. This mutation takes the form of a Gaussian random variable, mean 0 and variance inversely proportional to the fitness of the genome. This allows for less drastic perturbations in genomes that have shown promising fitness.

Crossover

Each generation (after the initial generation) is created by using roulette selection on the previous generation's population. This is based on each genome's fitness function, resulting in weighted random samples of size 2. These 2 genomes are then subjected to a single point crossover, and the child is included in the generation being initialized.

Additionally, we include the genomes with the top 5 fitness scores from the previous generation in each new generation, without being subject to crossover. However, these 5 are subject to possible mutation, based on the probabilities described above.

Population Size and Number of Generations

Our population size has varied between 100 and 1000, due to speed vs. effectiveness trade offs. The number of generations run initially was cut off at 100, but we have observed emergent fitness scores become evident in 40 or less generations. We have also included a cutoff value for the best fitness in a population, where a genome scoring that level of fitness will terminate the genetic algorithm and be used for the new game board. This value is currently set to 25, based on subjective play testing by the author.

V. ANALYSIS

Output

Figure 4 shows examples of boards generated with random seeds being passed into the grammar. As we can see, there is a high amount of overlap between various elements on the display. The routes are often intersecting other routes and covering destinations, leading to less playability for a user. Additionally, the board on the right hand side of the figure is quite small, and does not support an entire game run through with 5 players.

Simulations on randomly generated boards often end with the game being concluded early because of a lack of routes to build on, or because the distribution of route colors is not suited to a long game. Sometimes, certain Train Cards lose importance when there is only one or two tracks that demand that color, and these cards become the only elements of the deck and flop towards the tail end of the simulation.

Figure 5 shows examples of boards generated using our Grammatical Evolution approach. These are much more visually appealing than those generated in *Figure 4*, and seem to employ qualities evident in the default North American board. The routes rarely intersect, and if so all aspects (track length, colors allowed) are still visible. Additionally, no destinations are covered by routes to the point where identifying a specific route is a challenging process, like in the randomly genersated boards.

In terms of playability, simulations conducted on both boards shown in *Figure 5* resulted in grid lock (0%, 3% of simulations) about the same proportion of the time as that of the default North American board (3.07% of simulations). Additionally, since these boards were achieved by optimizing the fitness function, other aspects of gameplay (number of turns, average score, score variance, proportion of turns selecting train cards, and average number of path changes) are reflected in the high scoring of the fitness function.

Figure 4. Examples of random seeded boards, generated by our grammar

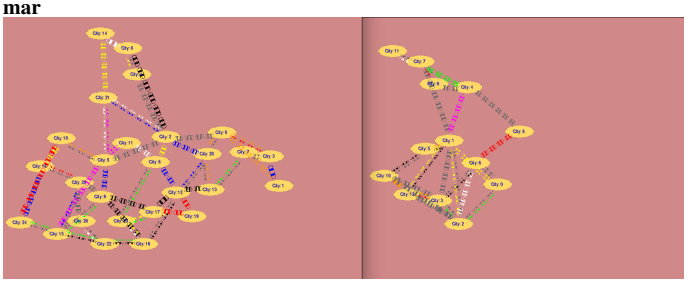
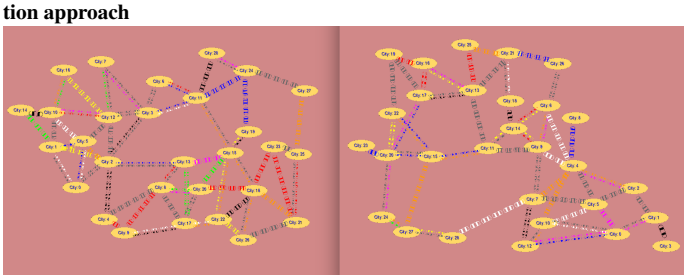


Figure 5. Examples of boards generated with our Grammatical Evolution approach



Generation Time

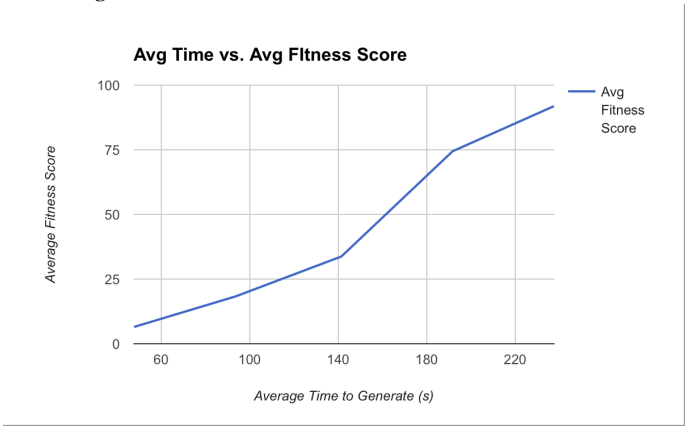
One drawback of evolutionary methods is the amount of time usually required to generate effective solutions to problems. In this case, our goal is to generate a novel game board upon opening our game. Unfortunately, our generation method requires between 30 seconds and 5 minutes to produce satisfiable boards, depending on the definition of satisfiable. Earlier we stated that the system can terminate evaluating the next generation in the algorithm if the fitness of a genome is above 25. This value has been taken to represent the minimum value for a board we are confident is "good".

Figure 6 and *Table 2* show the results of our methods when the population size is varied. The trade off between generation time and fitness score is evident, with the biggest concern being the number of "good" boards generated within 40 generations. Even with a population size of 500, taking about 4 minutes to generate on average, we are only seeing "good" boards generated around 60% of the time. However, when we increase the population size to 1000, we see 91% of the boards generated are "good", leading us to believe with unlimited time we can always be confident to produce a board with a fitness score above 25.

VI. CONCLUSION

The output from our board generation model seems to be effective when the fitness score is high, and given a large amount of time we are able to consistently generate boards

Figure 6. Results of Board Generation Time Evaluation



Pop Size	Avg Time Taken	Avg Fit Score	Perc Above 25
100	47.75	6.50	0.13
200	93.71	18.33	0.25
300	141.44	33.74	0.25
400	191.71	74.43	0.5
500	237.51	91.86	0.63
1000	622.15	162.16	0.91

Table 2. Results of Board Generation Time Evaluation

with high fitness functions. The overall correlation of our fitness function to actual game enjoyment has not been tested, but that is certainly an avenue of future work.

However, we have encountered an issue with time complexity, wherein our system is able to generate "good" (fitness > 25) boards over 90% of the time only when we allow for around 10 minutes of processing time. Despite this, even efforts that only result in "good" boards 13% of the time take almost a minute to generate a board.

Overall, our attempt to generate novel boards for Ticket to Ride was a success, and can be applied to a range of areas, even with the generation time issue. Potential game designers could make use of this software to generate new ideas for boards, and with future developments in city naming and location mapping could place them on real world maps. Additionally, this paper provided support for the approach to Grammatical Evolution by using fitness functions that are based mainly off of attributes of an existing standard (in our case, the default North American board). This is important to consider for future level design strategies that already have a few example levels as a basis to expand upon.

VII. FUTURE WORK

As mentioned in the previous section, one avenue for further exploration is in mapping a given generated board to a real world map. This could be accomplished through the use of map API's, and simply finding popular cities or towns that are within a given radius of the destination coordinates.

Another path to continue upon is to modify the fitness function so as to create boards fit to the number of players entered upon booting up the game. This would probably not involve too much extra work, and simply changes the number of agents passed to the simulator for the fitness function.

Finally, improvements could certainly be made to the opponent AI agents. In addition to improvements in the agents themselves, there could be applications here for better board generation. Potentially, an additional fitness metric could be the proportion of skilled agents that defeat unskilled agents, and attempt to get that metric as close to that found by simulating on the default North American board.

REFERENCES

1. Erin J Hastings, Ratan K Guha, and Kenneth O Stanley. 2009. Evolving content in the galactic arms race video game. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*. IEEE, 241–248.
2. Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. 2013. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 9, 1 (2013), 1.
3. Lawrence Johnson, Georgios N Yannakakis, and Julian Togelius. 2010. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 10.
4. Darren Lim. 2007. Taking students out for a ride: using a board game to teach graph theory. *ACM SIGCSE Bulletin* 39, 1 (2007), 367–371.
5. A Moon. 2004. Ticket to Ride.[Board game]. *Days of Wonder: Los Altos, CA* (2004).
6. Michael O'Neill and Conor Ryan. 2001. Grammatical evolution. *IEEE Transactions on Evolutionary Computation* 5, 4 (2001), 349–358.
7. Noor Shaker, Miguel Nicolau, Georgios N Yannakakis, Julian Togelius, and Michael O'Neill. 2012. Evolving levels for super mario bros using grammatical evolution. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. IEEE, 304–311.
8. Nathan Sorenson and Philippe Pasquier. 2010. Towards a generic framework for automated video game level creation. In *European Conference on the Applications of Evolutionary Computation*. Springer, 131–140.