# Literature Review of Procedural Content Generation in Puzzle Games

Ahmed Khalifa Magda Fayek

June 2015

**Abstract**

This is the third chapter from my Master Thesis (Automatic Game Generation). This chapter will provide a review of the past work on Procedural Content Generation. It highlights different efforts towards generating levels and rules for games. These efforts are grouped according to their similarity and sorted chronologically within each group.

## 1 Level Generation

This section presents some of the work related to Level Generation. Most of the previous work focused on generating levels for Platformer and Arcade Games. Most of the work found about Puzzle Games was limited toward specific games such as Sokoban[48, 23], Cut The Rope[37], and Fruit Dating[29]. Even the generic work needs prior human knowledge about the game rules[34, 33].

Generating levels for Puzzle Games seems interesting but it has lots of problems. Some of these problems are the main reason for not gaining much interest in previous work.

- All levels must be solvable (have at least one solution).
- Most of the level objects should be used (adding unused items may be considered as a bad level design).
- Puzzle Games ideas have no restriction. For example some games have continuous space where game actions depend on player's skills and perfect timing (Angry Birds[1], Cut the Rope[8], and Where's my Water?[54]) while other games have discrete space where game actions depend on the best sequence of actions regardless of the time taken (Fruit Dating[14], Sokoban[42], and HueBrix[17]).

### 1.1 Puzzle Level Generation

This section presents previous work that is related to our problem. One of the earliest research in Puzzle Games was by Murase et al.[23]. Murase et al. work focused on generating well designed solvable levels for Sokoban[42]. Sokoban is an old Japanese puzzle game where your goal is to push some crates towards certain locations as shown in Figure 1. Their work consists of the following 3 stages:

- **Generate:** Responsible for generating Level Layouts. Level Layouts are generated by placing predefined templates at random positions. Goal locations and crates are placed afterwards at random positions such that each goal location is reachable from the player location and each crate is reachable from a unique goal location.
- **Check:** Responsible for checking for playability. The system uses Breadth First Search (BFS) to check for solution. All unsolvable levels are removed in this step.
- **Evaluate:** Responsible for selecting best generated levels. The system removes all levels that have a solution that is very short (less than 7 steps), contains less than four direction changes, or does not contain any detours.

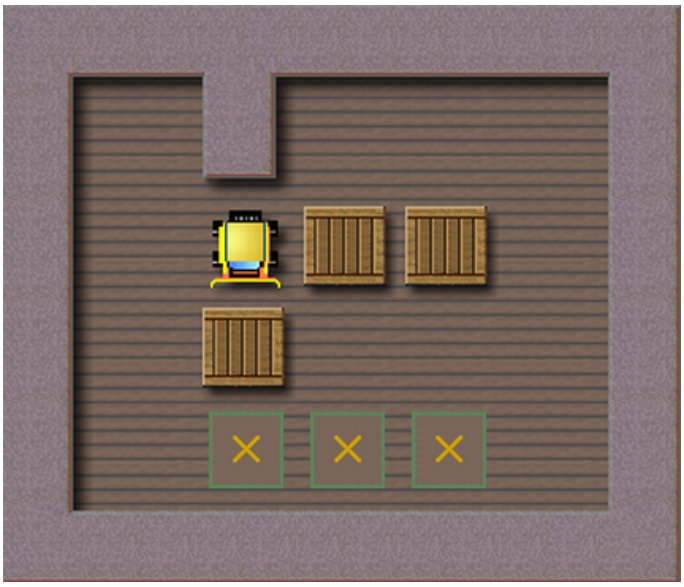

Figure 1: Screenshot from Sokoban

Results show that for every 500 generated levels only 14 are considered as good levels. These good levels are characterized by having a short solution sequence due to the usage of BFS.

Taylor and Parberry[48] followed Yoshio Murase et al.[23] work to improve generated level quality. Their system consists of 4 stages:

- **Generating empty room:** Responsible for generating Level Layouts like Yoshio Murase et al.[23] work. After generating the layouts, the system discards any level that is not completely connected, has huge empty space, or has number of empty floors less than the planned number of boxes and goals.
- **Placing goals:** Responsible for finding the best suitable goal locations. The system uses a brute force algorithm to generate every possible goal location where the ones that give the highest score are chosen during evaluation step.
- **Finding the farthest state:** Responsible for placing crates at farthest location from its goal location. This process is done using a similar algorithm to BFS. The algorithm expands all the reachable locations from a goal location and returns the farthest location. The farthest location is calculated using a box line metric heuristic which calculates the number of unrepeated moves required to reach that location.
- **Evaluating generated levels:** Responsible for evaluating the generated levels. Evaluation is done using some heuristic functions. For example: number of box pushes in the solution, number of levels found at same solution depth, number of box lines, number of boxes, and ...etc.

The generated levels do not suffer from the problem of short level sequences presented in Yoshio Murase et al.[23] work. As the target number of crates increases, the generation process takes more time but delivers much more interesting and difficult levels.

Rychnovsky[29] work focused on generating levels for his new game Fruit Dating. As shown in Figure 2, Fruit Dating is a puzzle game where the player needs to move all similar fruits beside each other. The game is played by swiping in one of the four direction (up, down, left, right). Swiping in any direction causes all objects to move towards that direction. Rychnovsky developed a level editor that can be used to generate new levels or test playability of certain level. Generating new levels is done using the following steps:

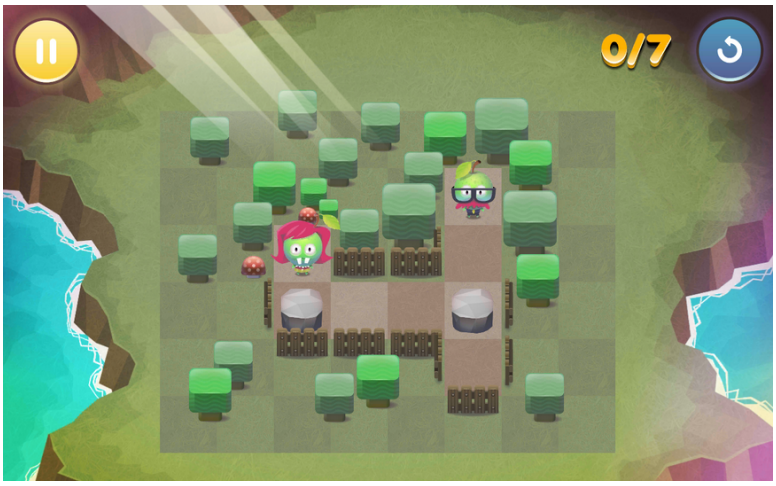


Figure 2: Screenshot from Fruit Dating

- **Generating level structure:** The system generates level structures based on 2 steps. First step is generating external walls. External walls are generated at any random location connected to the border with short random length. Second step is generating inner walls. Inner walls are generated at any free location surrounded by eight free locations.
- **Placing game objects:** The system generates fruits at random locations based on predefined weights. Every empty location is assigned a score using these weights. Locations with highest scores are selected. Other game objects use the same strategy but using different weights.
- **Checking for a solution:** The system uses an algorithm similar to BFS to find the solution of the generated level. If no solution is found, the level is discarded.

The technique doesn't take more than a couple of minutes to generate a level. The main problem is that there is no way to influence difficulty in the generated levels.

Shaker et al.[33] worked on generating levels for physics based puzzle games. They applied their technique on Cut The Rope (CTR). As shown in Figure 3, CTR goal is to cut some ropes to free the attached candy to reach OmNom (a frog monster fixed at a certain position on the screen). As game progresses more game objects are introduced. These objects help in changing the movement direction of the candy to redirect it towards OmNom. Shaker et al. used Grammar Evolution (GE) technique to generate levels for CTR. GE is a result of combining an evolutionary algorithm like GA with a grammar representation. The grammar is designed to ensure that every game object appears at least one time. The fitness function used to rate the generated levels depends on some heuristic measures based on prior knowledge about the game. For example: Candy should be placed higher than the OmNom, OmNom must be placed below the lowest rope, and ...etc. Since using heuristic measures does not ensure playability, each generated level is played 10 times using a random player. The fitness score is reduced by a large number if no solution was found. Shaker et al. generated 100 playable levels and analyzed them according to some metrics such as frequency, density, and ...etc.

Shaker et al.[37] conducted their research on CTR to improve generated level quality. They replaced the random player with an intelligent one. Two types of automated players were developed. The first one takes an action based on the current properties of all objects in the level. While the second one takes an action based on the reachability between every level object and the candy at every time step based on the candy's current position, velocity, and direction. The generated levels are far more diverse because the random player discards some


Figure 3: Screenshot from Cut The Rope

potential levels in the search space.

Later in [34], Shaker et al. introduced a new generation technique named Progressive Approach. Progressive Approach can be used on any kind of games to reduce the generation time. Progressive Approach is divided into two main steps:

- **Time-line Generation:** GE is used to evolve a time-line of game events.
- **Time-line Simulation:** Evolved time-lines are simulated using an intelligent agent. The agent utilize prior knowledge about the game to map the time-line to a possible level layout. Based on the agent result and some desirable heuristics (based on the current generated game), each time-line is assigned a fitness score.

Shaker et al. tested the new technique on CTR and compared its results with their previous work[37]. The results indicates a huge decrease in generation time, as it changed from 82 seconds towards 9.79 seconds. Although Progressive Approach is much faster and better from previous, it is difficult to determine the level difficulty before simulation. Also the quality of the levels depends on the intelligent agent used in the mapping process.

Williams-King et al.[55] work focused on generating levels for their game KGoldRunner. KGoldRunner is just a port for the famous game Lode Runner[22]. As shown in Figure 4, the goal is to collect all gold chunks without getting killed. Player can move left, move right, climb a ladder, move across a bar, or dig the floor. The game contains enemies who changes the position of the gold chunks. After collecting all gold chunks a hidden ladder appears which leads the player to the end of the level. Williams-King et al. used GA to generate random levels. Generated levels are evaluated using fitness function based on some prior knowledge about the game. For example: testing level for connectivity using BFS between the starting point and all the gold coins in the levels. High score levels are simulated using an automated player that

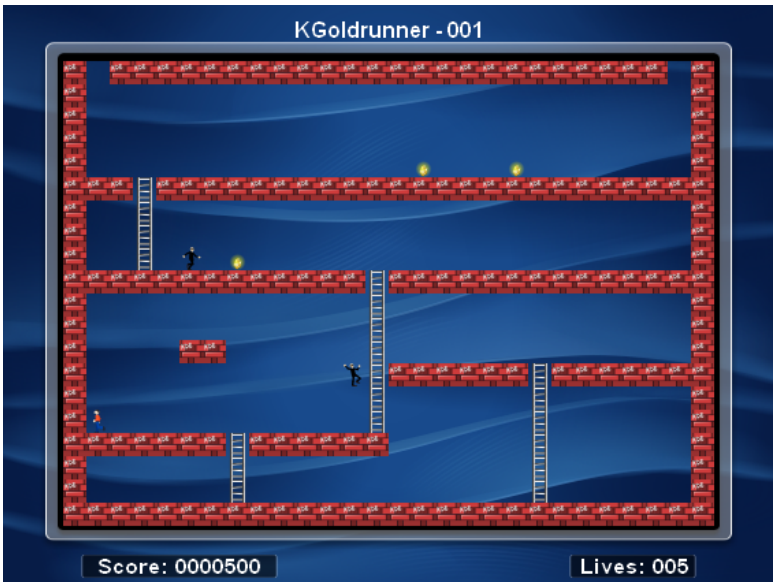


Figure 4: Screenshot from KGoldRunner

tries to collect all gold chunks using 20 different paths. The number of solvable paths indicates the level difficulty. As the number of generations increases more time is required but levels are more likely to be solvable.

Smith et al.[38] worked on generating puzzle levels for Refraction. Refraction is an educational game created by Center for Games Science at the University of Washington. Refraction teaches children concepts in math through a puzzle game. As shown in Figure 5, the goal is to make every alien spaceship fully powered by directing a laser beam towards them. Each ship needs a fraction of the laser to operate. Player should use tools like Benders (change the direction of laser by 90°), Splitters (divide laser power by number of outputs from it), and ...etc to achieve that goal. Smith et al. divided their generator into 3 main components:

- **Mission Generation:** Responsible for generating a general outline showing level solutions.
- **Level Embedder:** Responsible for translating the general outline to a geometric layout.
- **Puzzle Solver:** Responsible for testing generated level for solution.

These components are implemented into two different ways (Algorithmic approach and Answer Set Programming (ASP)). Results shows that ASP is faster than Algorithmic approach specially in Puzzle Solver module, while Algorithmic approach produces more diverse levels than ASP.

## 1.2 Non-Puzzle Level Generation

This section focuses on Level Generation for non Puzzle Games. The previous work in that field focused on different game genres but a lot of work was done in Platformer genre. Showing some of the work will help in understanding the current research direction towards Level Generation.

The constructive approach in PCG has been used in commercial games for a long time. It has been used in generating textures, trees, sound effects, and levels. Ismail and Nijman[30] used an Agent Based PCG to generate levels for their commercial game Nuclear Throne. The system spawns an agent at a random position on the map. This agent starts moving into a random direction with a certain probability to change direction. New agents are spawned based


Figure 5: Screenshot from Refraction

on a certain probability. The agents continue moving till the number of dug floors reaches 110 or reaches a dead end. Coxon[25] used perlin noise and cellular automata algorithm to generate a huge map for his commercial game Lenna's Inception. The game map is divided into small rooms. Perlin noise is used to assign a certain terrain type for each room. Cellular automata algorithm is used to craft the details for each room while ensuring reachability between entrances and exits.

Search algorithms (specially GA) has been used a lot in level generation specially in academia. Sorenson and Pasquier[44] worked on improving GA to adapt with level generation for any game. The new technique divides the population into feasible and infeasible populations. Each population is evolved on its own using crossover and mutation. Chromosomes can transfer from Feasible to Infeasible population and vice versa. The feasible population uses a fitness function that measures the challenge presented in the game. The infeasible population uses a fitness function that measures the distance towards the feasible population. These techniques were tested over two different games The Legend of Zelda and Super Mario Bros. The results show promising levels generated for both games which is an indication of the possibility of using this technique as a generic framework.

Preuss et al.[28] used three different search techniques to generate diverse game maps for Strategy Games. They used Restart Evolution Strategies, Novelty Search, and Niching Evolutionary Algorithm are then compared their results. The fitness function used for evaluation consists of 3 metrics:

- **Tile Based Distance:** measures diversity between each two maps.
- **Objective-based Distance:** measures the quality and playability of the map according to some heuristics for Strategy Games.
- **Visual Impression Distance:** measures how good the map looks using 20 different human designed heuristic measures.

Niching Evolutionary Algorithm generated the best results in diversity and quality measurement, followed by Restart Evolution Strategies which produced same results like Niching Evolutionary Algorithm, while Novelty Search was the worst of them in both quality and diversity.

Liapis et al.[20] improved Novelty Search Algorithm to perform better than previous work[28]. The improved algorithm borrowed an idea from Sorenson and Pasquier work[44] by dividing the population into feasible and infeasible populations. The feasible population uses a fitness function that measures the distance between chromosomes to ensure diversity. The infeasible population was tested using two different fitness functions. The first one is same as that used for the feasible population, while the second one uses the same fitness function used for Sorenson and Pasquier work[44]. The results prove that the second fitness generates more feasible solutions but less diverse compared to the first one.

Baghdadi et al.[2] used GA to evolve levels for the commercial game Spelunky[45]. Spelunky entire map is divided into 4x4 rooms. Levels are represented as a group of integers. These integers represent positions for start room, exit room, room connections, number of enemies, and ...etc. Each level is mapped to a layout before evaluation. Mapping process consists of the following three main steps:

- Generating internal structure for each room using an Agent Based PCG.
- Adding some corridors to ensure connection between rooms.
- Distributing monsters, ladders, and bombs equally across the level.

Levels are evaluated using a fitness function that consists of two heuristics metrics. The first heuristic metric ensures level playability by measuring connectivity between mandatory rooms, placement of starting room, and ...etc. Second heuristic metric measures level difficulty by measuring path length, number of enemies, and ...etc. The results show that the introduced technique is able to generate a good designed levels for Spelunky with a certain difficulty.

Smith et al.[40] generated levels for Super Mario Bros (SMB) using Rhythms. Rhythms is a way to express the pace of playing a level. Rhythms are a sequence of actions that player needs to finish the current SMB level. The generator is divided into 2 main parts:

- **Rhythm Generation:** Responsible for generating a sequence of actions required to finish the level.
- **Rhythm to Geometry Generation:** Responsible for translating the generated Rhythm into a corresponding level layout.

The generated level is tested against some critic measures provided by the human designer. For example, probability of sloped platforms, frequency of jumps, and ...etc. These critic measures help in enhancing the quality and ensuring the playability of the generated levels. The results show the importance of including the pace of playing in generating platformer levels.

Sorenson and Pasquier[43] used GA to generate SMB levels based on previous studies about GameFlow[47, 5, 46]. GameFlow studies measure the player enjoyment as the amount of challenge facing the player. The studies recommend keeping the challenge level at an optimal rate (too much challenge cause frustration, too little challenge cause boredom). Based on that fact, The study modeled the challenge level as the player's Anxiety. Sorenson and Pasquier used anxiety curve as a fitness function to evaluate the generated levels. The rate of change of anxiety curve is modeled using the following equation:

$$\frac{df}{dt} = m * \frac{da}{dt}$$

Where m can be +1 or -1. The value of m is +ve all the time till a certain threshold. If this threshold is exceeded, m value becomes -ve causing the curve to decrease till another threshold is reached where it starts increasing again. They compared their results with the original SMB levels. This analysis proves that anxiety curves show a promising direction for using GameFlow as a measurement.

Shaker er al.[36] used GE to generate levels for SMB. The level is represented using context free grammar that represents the level as a group of chunks where each chunk can be any game

object. GE tries to increase the number of chunks appearing in the level while minimizing the number of conflicts between them. The results of the generator is compared with two other generators. They introduced a set of metrics to be used in comparison. For example, Linearity is used to measure the difference between platform heights. The results show the efficiency of the introduced metrics in comparing different SMB generators.

Some commercial games generate levels by combining a group of hand crafted game chunks to ensure the high quality of the level. Yu and Hull[50] divided Spelunky map into 4x4 rooms. Each room is chosen from a set of predefined rooms ensuring the existence of a path between start and end points. The generator modifies each selected room by adding some new blocks without blocking any movement path. Enemies and items are added after selecting the layout. Also Mcmillen[49] used a similar technique for his commercial game *The Binding of Isaac*. Due to the huge success of these games, that technique became popular among academia.

Dahlskog and Togelius[9, 10, 12, 11] published several papers on generating levels for SMB game utilizing the patterns found in the original game. They started[9] by analyzing all original levels from SMB and detecting all repeated patterns. Repeated patterns are divided into 5 main categories (Enemies, Gaps, Valleys, Multiple Paths, and Stairs). Each category is divided into multiple configurations called Meso Patterns, for example Enemies pattern can appear as an Enemy (single enemy), 2-Horde (two enemies together), 3-Horde (three enemies together), 4-Horde (four enemies together), and a Roof (Enemies underneath hanging platform). The level generator generates levels by randomly choosing a group of these patterns and generating a geometry corresponding to them.Levels generated by that technique have the same feeling like original SMB levels. They conducted their research[10, 12] to improve the quality of the generated levels by using these analyzed patterns as an evaluation function. Levels are generated from concatenating a group of vertical slices called Micro Patterns. Micro Patterns are generated from original levels in SMB. GA is used to improve the generated levels using a fitness function that rewards levels which contain more Meso Patterns. The generated levels are more similar to the original levels of SMB and more diverse. They conducted their research[11] to improve the quality of the generation by introducing Macro Patterns. A Macro Pattern is a group of Meso Patterns after each other which can be used to identify the level of difficulty. SMB levels were analyzed and Macro Patterns are extracted from it. Macro Patterns from SMB are used as fitness function for GA to reward levels with more Macro Patterns. The results take more time to generate but they have the same flow of SMB levels.

Shaker and Abou-Zleikha[35] used Non-Negative Matrix Factorization (NMF) technique to capture the patterns of level design of SMB. NMF can be used on any game if it has a huge amount of levels to learn the pattern from them. They used five different PCG techniques to generate 5000 levels for SMB (1000 level from each technique). NMF method factorize these levels into two new matrices:

- **Pattern Matrix:** captures the patterns found in each vertical slice of SMB.
- **Weight Matrix:** captures the percentage of each pattern to appear in the current level.

Pattern Matrix can be used to generate same levels like any of the input generators if its original weight vector is used. Shaker and Abou-Zleikha generated levels using free weights and compared them with all the results from the five used techniques. The results show that NMF covers more of the search space than all the five techniques combined, but Shaker and Abou-Zleikha could not know if the whole space provides a good playing experience.

Snodgrass and Ontanon[41] used Markov Chain Model to generate levels for SMB. Markov Chain Model is another way to learn about the design patterns from original SMB levels. The technique can be used on different game genres if there is a huge amount of highly designed levels. They tested the technique with different input parameters and evaluated the generated levels using human players. The results of the best input configuration only generate 44% playable levels.

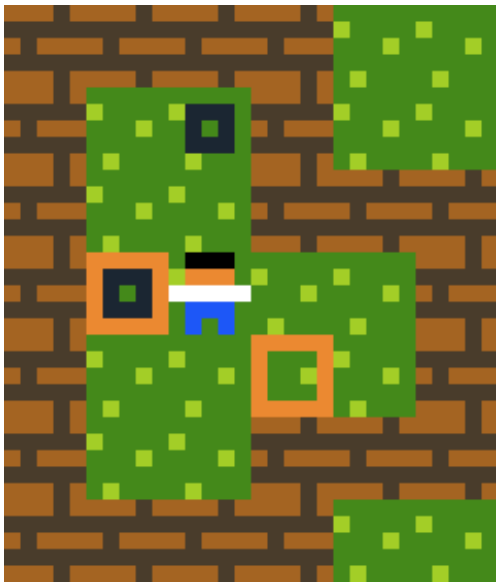

Figure 6: Tested Sokoban level in Lim et al. work

# 2 Rule Generation

This section will show the latest research and the different techniques used to generate rules for different game genres. Most of the work in this section target generating Arcade games[51, 6, 31, 39, 52]. Arcade games are the most popular in research because they are easier to generate (simple rules), they provide a huge diversity in mechanics, and there is a huge research happening in GVGP for Arcade Games[19]. Other work tries to generate Board Games[4], Card Games[13], and Platformer Games[7] but as far as we know no one tried to generate Puzzle Games except for the work by Lim and Harrell[21].

## 2.1 Puzzle Rule Generation

Puzzle rule generation was infamous till 2014., when Lim and Harrel[21] published their work on generating new rules for PuzzleScript games. They use GA to find the possible rules that can solve a certain level for a certain game. For example what are the rules that can solve Sokoban level shown in Figure 6? Lim and Harrel used GA to generate rules. They divided the fitness function into 3 parts:

- **Rule Heuristics:** measures some logical constrains that should be found in rules. For example player should be on the left hand side of at least one rule, all movements in a certain rule should be in the same direction, and ...etc.
- **Validity:** checks for runtime errors in the generated rules.
- **Feasibility:** checks if the level is solvable using an automated player.

After running GA for 50 generations on the level shown in Figure 6, the system discovered new rules such as Crate Pulling, Morphing, and Spawning. Although the new discovered rules are similar to human designed rules, It is still an achievement.

## 2.2 Non-Puzzle Rule Generation

Although Rule Generation is the hardest content to be generated, a computer software written by Browne and Marie invented a board game called Yavalath. Yavalath is listed in the

top 100 abstract board games of all times[56]. Browne and Maire software[4] used standard evolutionary approach to evolve game rules written in Game Description Language (GDL) for Combinatorial Games. Combinatorial Games are games that have finite number of states, deterministic (no randomness), discrete (one step at a time), perfect information (everything is visible), and involve two or more players. Evolved games fitness were measured by automated players for several plays (playouts). Based on the results of the playouts a score is given for each game based on 57 different heuristic metrics. The best scored games were tested by human players where each tester assign a score for these games. Comparing human scores with the system scores show a correlation between both of them.

Jose et al.[13] used Genetic Programming (GP) to generate rules for card games using GDL specified for card games. The generated games are tested using automated players for hundred playouts. Based on the results of these playouts, games are assigned a fitness score. The system removes all games that have never been finished in at least one of the playouts, results in draw for more than 20% of the playouts, or have more than 10 stages for one round. Other games are evaluated based on the difference between the number of wins of the best player to the worst player, the average number of turns required to finish the game, and the average number of turns with no actions. The resulted games are well formed and can be played by humans but at the same time they are boring and not interesting as most of them can be won using simple strategy.

Togelius and Schmidhuber[51] worked on generating rules for Arcade Games. Games are encoded in the form of collision and score matrices and a list of object behaviors. The collision matrix shows the result of colliding any two game objects. For example *red, white, kill, none* means If a red object collides with a white object, the red object will be killed while nothing will happen to the white object. Score matrix is defined in the same way but with values of +1, 0, -1. The list of object behaviors shows how different objects behave in the game (random movement, clockwise movement, still, and ...etc). All generated games terminate when $score \geq score_{max}$ or when $time \geq time_{max}$. Togelius and Schimdhuber used a hillclimbing algorithm to generate game rules. Each iteration new game is generated and evaluated using a fitness function. The current game will be replaced with the new game if the new game scores better. The fitness function is based on the idea of learning progress which is inspired by Koster's Theory of Fun[18] and Schmidhuber's theory of curiosity[32]. A game is considered as a good game according to how much it can be learned. Based on that idea evolutionary strategies algorithm is used to evolve a neural network to play the generated games. The performance of the evolved neural network in playing the generated games is used as a fitness score. That approach needs very long time to run that is why the best generated games are just playable games but not interesting.

Cook and Colton[6] took a new approach in generating Arcade Games. They divided the game into 3 main elements:

- **Map:** is a 2D matrix showing passable and impassable tiles in the levels.
- **Layout:** is a list of position for every object on the map.
- **Rules:** govern how game works and they are represented in the same way in Togelius and Schmidhuber work[51].

Each game element is evolved using GA to maximize its own fitness score. For example map fitness measures the reachability of each tile, layout fitness checks for the distance between different objects, and rules fitness checks for game playability. All game elements communicate with each other through a central game object. After each generation every game element updates the central game object with the best scored output. The system is very easy to add/remove game elements but some game elements can finish evolution early which minimize communications between different elements. Most of the generated games are not interesting although some of them have some similarity with the famous game PacMan.

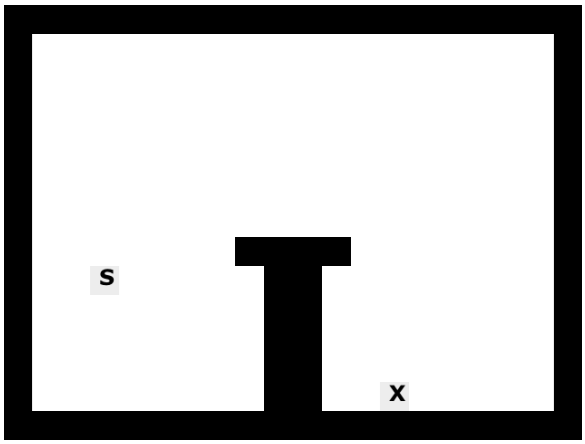


Figure 7: The challenge level for the Toggleable function

All discussed techniques so far are based on having a specific GDL that governs the generation of rules. Cook et al.[7] decided to generate new game rule for his platformer game (A Puzzling Present) by analyzing and generating actual game code. They use code reflection ability to get all data members from the player class. Cook et al. used GP to evolve a new Toggleable function that helps an automated player to overcome a challenge level. A Toggleable function is any function where its effect can be reversed. Toggleable function is applied to one of the player data members. Cook et al. used the challenge level presented in Figure 7 to test the Toggleable function. The challenge level is designed to ensure no possible solution using platformer rules (moving and jumping). After the algorithm finds a Toggleable function that helps in solving the challenge level, the system starts generating levels using GA. Levels are rated based on 3 simulation runs to ensure certain characteristics.

- **First Simulation:** Check for a solution without using the new Toggleable function.
- **Second Simulation:** Check for a solution using the new Toggleable function.
- **Third Simulation:** Check for level difficulty based on the solution length.

Some of the best evolved Toggleable functions are Gravity Inversion, Teleportation, and Bouncing. The mechanics and levels are rated using human users. Human users rated Gravity Inversion and Bouncing higher than Teleportation as they are more understandable.

Farbs released a game called *ROM CHECK FAIL*[31]. ROM CHECK FAIL is a game which changes the rules applied on each object every few seconds. The new rules are selected from a pool of handcrafted rules. Some of rules controls player movements, enemy movements, object collisions, or ...etc. For example the player can be Link from The Legend of Zelda where he can move in all directions or a Spaceship from Space Invaders where he can only move left and right and shoot upwards. Although the game seems unfamiliar and weird, it has made a huge impact on the gaming scene (inspired lots of developers).

Smith and Mateas[39] went for another direction in generating new games. Instead of generating a game then test it against a set of constrains and/or evaluation function. They decided to limit the generative space by not exploring these parts to minimize the processing time. They used ASP to generate games with required aspects and constraining the unplayable parts in the generative space. For example putting a constraint for only winning if all white objects are killed, ensures ASP does not explore games with a different winning condition. Smith and Mateas tested their idea with Arcade Games by creating a game called Variation Forever[53] which is inspired by ROM CHECK FAIL[31]. Games are described using the same way as that of Togelius and Schmidhuber[51]. Results show promising direction in using ASP for limiting the generative space.

# 3 General Video Game Playing

This section presents the latest work in General Video Game Playing (GVGP) either as standalone research or as a part of level or rule generation algorithms. Most of the discussed work with level generation either use BFS algorithm[23] or a tailored AI specially for a certain game[37]. For most of Level Generation work a tailored AI performs better than using a general one as tailored AI is designed with previous knowledge about the game. Utilizing this knowledge helps the automated player to find the solution quickly.

Tailored AI will not work with Rule Generation. Rule Generation searches for new unseen games where automated player is used to test its playability. Most of work done in Rule Generation used some kind of Search Algorithm (variants of BFS) to find the solution except for Togelius and Schmidhuber[51] work. They used an evolutionary Neural Network to measure learning progress through the evolved game. Browne and Maire[4] used Min-Max trees with Alpha Beta Pruning for automating game plays. They utilize the current game description to provide estimates for each board configuration. Jose et al.[13] used two types of automated players. The first one is random player which chooses random actions to model weak players, while the second is based on Monte Carlo Tree Search (MCTS) algorithm to model professional player. Cook et al.[7] used a normal BFS technique to search for a result for the challenge level. Lim and Harrel[21] compared BFS algorithm to solve generated games with Best First Search (BestFS) algorithm. BestFS is another name for A* algorithm where it sorts the explored nodes based on a heuristic function. The system utilizes the knowledge about the goal condition and tries to minimize the distance between goal objects. The system also minimizes the distance between the player object and goal objects based on the fact that the player is the main object affecting the game world. Results show that BestFS finds solution faster than BFS but with slightly longer moves.

Most of GVGP research was done on Arcade Games by ATARI 2600. One of the first work in that direction was the work done by Bellemare et al.[3]. They developed a system called Arcade Learning Enviroment (ALE) which contains lots of different Arcade Games from ATARI 2600. They tested their system against two different AI approaches:

- **Learning techniques:** Reinforcement learning algorithm called $Sarsa(\lambda)$ is tested using different types of features. Features ranged from using screen pixel colors to using the 1024 bit of Atari RAM.
- **Planning techniques:** Normal BFS and BFS with Upper Confidence Bounds Applied to Trees (UCT) are tested.

Results show that both approaches performed better than the average normal player. Although Planning techniques outperformed Learning techniques, it took more time to decide each action.

Hausknecht et al.[16] evolved a NN to play ATARI 2600 games. The system consists of 3 main steps

- **Visual Processing:** The system processes the game screen and detects all moving objects and groups them into classes based on velocity information.
- **Self-Identification:** The system identifies which class is the player. The player object is identified by calculating the entropy over velocities for each class. The class with the highest entropy is considered the player.
- **Perform Action:** The system chooses its next action based on the positions of every object.

Their system outperformed three variants of $Sarsa(\lambda)$ techniques based on their performance in playing two Atari 2600 games (Freeway and Asterix).

Perez et al.[26, 27] summarized the techniques and the results from General Video Game AI Competition (GVG-AI)[15]. GVG-AI is a competition that takes place every year for creating a General Player that can play some unseen ATARI 2600 games. Lots of techniques and methods used in the entries for the competition. Some of the techniques are based on learning methods while others on general heuristics. Heuristic methods produced better results than most of learning methods in the competition. The best algorithm is an Open Loop Expectimax Tree Search, followed by an algorithm named JinJerry Algorithm which is a variant of MCTS, then some variants of BFS. The first learning algorithm to appear in table of results is at the sixth place. The learning algorithm is a reinforcement learning technique called Q-Learning Algorithm which models the world in a form of Markov Decision Process.

Nielsen et al.[24] tested different AI techniques over ATARI 2600 games. They used different AI technique varies from very intelligent (MCTS) to totally dump (DoNothing). AI techniques are tested against 20 handcrafted games, 200 mutated version, and 80 random games. Based on the results, most intelligent techniques performed better on handcrafted games than other games. On the other hand, the DoNothing algorithm perform very bad on handcrafted games than other games. These results strengthen the idea of the need for an intelligent player to judge procedural generated games.

# References


[1] Angry birds. `http://en.wikipedia.org/wiki/Angry_Birds`. [Accessed: 2015-03-17].

[2] Baghdadi, W., Shams Eddin, F., Al-Omari, R., Alhalawani, Z., Shaker, M., and Shaker, N. A procedural method for automatic generation of spelunky levels. In *Proceedings of EvoGames: Applications of Evolutionary Computation, Lecture Notes on Computer Science* (2015).

[3] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Computing Research Repository* (2012).

[4] Browne, C., and Maire, F. Evolutionary game design. In *IEEE Transactions on Computational Intelligence and AI in Games* (2010), IEEE, pp. 1–16.

[5] Chen, J. Flow in games (and everything else). *Commun. ACM* (2007), 31–34.

[6] Cook, M., and Colton, S. Multi-faceted evolution of simple arcade games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games* (2011), IEEE, pp. 289–296.

[7] Cook, M., Colton, S., Raad, A., and Gow, J. Mechanic miner: Reflection-driven game mechanic discovery and level design. In *Applications of Evolutionary Computation - 16th European Conference* (2013), pp. 284–293.

[8] Cut the rope. `http://en.wikipedia.org/wiki/Cut_the_Rope`. [Accessed: 2015-03-17].

[9] Dahlskog, S., and Togelius, J. Patterns and procedural content generation: Revisiting mario in world 1 level 1. In *Proceedings of the First Workshop on Design Patterns in Games* (2012), ACM, pp. 1:1–1:8.

[10] Dahlskog, S., and Togelius, J. Patterns as objectives for level generation. In *Proceedings of the Workshop on Design Patterns in Games at FDG* (2013), ACM.

[11] Dahlskog, S., and Togelius, J. A multi-level level generator. In *Proceedings of the IEEE Conference on Computational Intelligence and Games* (2014).

[12] Dahlskog, S., and Togelius, J. Procedural content generation using patterns as objectives. In *Proceedings of EvoGames, part of EvoStar* (2014).

[13] Font, J. M., Mahlmann, T., Manrique, D., and Togelius, J. Towards the automatic generation of card games through grammar-guided genetic programming. In *Proceedings of Foundations of Digital Games* (2013).

[14] Fruit dating. `https://www.behance.net/gallery/13640411/Fruit-Dating-game`. [Accessed: 2015-03-17].

[15] Gvg-ai. `http://www.gvgai.net`. [Accessed: 2015-04-04].

[16] Hausknecht, M., Khandelwal, P., Miikkulainen, R., and Stone, P. Hyperneat-ggp: A hyperneat-based atari general game player. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation* (2012), ACM, pp. 217–224.

[17] Huebrix. `http://www.huebrix.com/`. [Accessed: 2015-03-17].

[18] Koster, R., and Wright, W. *A Theory of Fun for Game Design*. Paraglyph Press, 2004.

[19] Levine, J., Congdon, C. B., Ebner, M., Kendall, G., Lucas, S. M., Miikkulainen, R., Schaul, T., and Thompson, T. General Video Game Playing. In *Artificial and Computational Intelligence in Games*, vol. 6 of *Dagstuhl Follow-Ups*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 77–83.

[20] Liapis, A., Yannakakis, G. N., and Togelius, J. Enhancements to constrained novelty search: Two-population novelty search for generating game content. In *GECCO '13 Proceedings of the fifteenth annual conference on Genetic and evolutionary computation conference* (2013), ACM, pp. 343–350.

[21] Lim, C.-U., and Harrell, D. F. An approach to general videogame evaluation and automatic generation using a description language. In *Proceedings of the IEEE Conference on Computational Intelligence and Games* (2014), IEEE, pp. 286–293.

[22] Lode runner. `http://en.wikipedia.org/wiki/Lode_Runner`. [Accessed: 2015-04-04].

[23] Murase, Y., Matsubara, H., and Hiraga, Y. Automatic making of sokoban problems. In *PRICAI'96: Topics in Artificial Intelligence, 4th Pacific Rim International Conference on Artificial Intelligence, Cairns, Australia, August 26-30, 1996, Proceedings* (1996), pp. 592–600.

[24] Nielsen, T. S., Barros, G. A. B., Togelius, J., and Nelson, M. J. General video game evaluation using relative algorithm performance profiles. In *Proceedings of the 18th Conference on Applications of Evolutionary Computation* (2015).

[25] Overworld overview. `http://bytten-studio.com/devlog/2014/09/08/overworld-overview-part-1/`. [Accessed: 2015-03-15].

[26] Perez, D., Samothrakis, S., and Lucas, S. M. Knowledge-based fast evolutionary MCTS for general video game playing. In *IEEE Conference on Computational Intelligence and Games* (2014), pp. 1–8.

[27] Perez, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S., Couetoux, A., Lee, J., Lim, C., and Thompson, T. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games* (2015).

[28] Preuss, M., Liapis, A., and Togelius, J. Searching for good and diverse game levels. In *Proceedings of the IEEE Conference on Computational Intelligence and Games* (2014).

[29] Procedural generation of puzzle game levels. http://www.gamedev.net/page/resources/_/technical/game-programming/procedural-generation-of-puzzle-game-levels-r3862. [Accessed: 2015-02-24].

[30] Random level generation in wasteland kings. http://www.vlambeer.com/2013/04/02/random-level-generation-in-wasteland-kings/. [Accessed: 2015-01-18].

[31] Rom check fail. http://www.farbs.org/romcheckfail.php. [Accessed: 2015-04-03].

[32] Schmidhuber, J. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE Transactions on Autonomous Mental Development 2* (2010), 230–247.

[33] Shaker, M., Sarhan, M. H., Naameh, O. A., Shaker, N., and Togelius, J. Automatic generation and analysis of physics-based puzzle games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games* (2013), IEEE, pp. 1–8.

[34] Shaker, M., Shaker, N., Togelius, J., and Abou Zleikha, M. A progressive approach to content generation. In *EvoGames: Applications of Evolutionary Computation* (2015).

[35] Shaker, N., and Abou-Zleikha, M. Alone we can do so little, together we can do so much: A combinatorial approach for generating game content. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment* (2014).

[36] Shaker, N., Nicolau, M., Yannakakis, G. N., Togelius, J., and O'Neill, M. Evolving levels for super mario bros using grammatical evolution. In *Proceedings of the IEEE Conference on Computational Intelligence and Games* (2012), IEEE, pp. 304–311.

[37] Shaker, N., Shaker, M., and Togelius, J. Evolving playable content for cut the rope through a simulation-based approach. In *Artificial Intelligence and Interactive Digital Entertainment* (2013), G. Sukthankar and I. Horswill, Eds., AAAI.

[38] Smith, A. M., Andersen, E., Mateas, M., and Popovic, Z. A case study of expressively constrainable level design automation tools for a puzzle game. In *Foundations of Digital Games* (2012), ACM, pp. 156–163.

[39] Smith, A. M., and Mateas, M. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games* (2010), IEEE, pp. 273–280.

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

[41] Snodgrass, S., and Ontanon, S. Experiments in map generation using markov chains. In *Proceedings of the International Conference on the Foundations of Digital Games* (2014).

[42] Sokoban. http://en.wikipedia.org/wiki/Sokoban. [Accessed: 2015-01-19].

[43] Sorenson, N., and Pasquier, P. The evolution of fun: Automatic level design through challenge modeling. *Proceedings of the First International Conference on Computational Creativity (ICCCX).* (2010), 258–267.

[44] Sorenson, N., and Pasquier, P. Towards a generic framework for automated video game level creation. In *International Conference on Evolutionary Computation in Games, EvoGame* (2010), Springer, pp. 131–140.

[45] Spelunky. http://en.wikipedia.org/wiki/Spelunky. [Accessed: 2015-03-15].

[46] Sweetser, P., Johnson, D. M., and Wyeth, P. Revisiting the gameflow model with detailed heuristics. *Journal : Creative Technologies* (2012).

[47] Sweetser, P., and Wyeth, P. Gameflow: A model for evaluating player enjoyment in games. *Comput. Entertain.* (2005), 3–3.

[48] Taylor, J., and Parberry, I. Procedural generation of Sokoban levels. In *Proceedings of the International North American Conference on Intelligent Games and Simulation* (2011), EUROSIS, pp. 5–12.

[49] The binding of isaac. `http://edmundmcmillen.blogspot.com/2011/09/binding-of-isaac-gameplay-explained.html`. [Accessed: 2015-03-15].

[50] The full spelunky on spelunky. `http://makegames.tumblr.com/post/4061040007/the-full-spelunky-on-spelunky`. [Accessed: 2015-03-15].

[51] Togelius, J., and Schmidhuber, J. An experiment in automatic game design. In *IEEE Symposium on Computational Intelligence and Games* (2008), IEEE.

[52] Treanor, M., Schweizer, B., Bogost, I., and Mateas, M. The micro-rhetorics of game-o-matic. In *Proceedings of the International Conference on the Foundations of Digital Games* (2012), ACM, pp. 18–25.

[53] Variation forever. `http://eis.ucsc.edu/VariationsForever`. [Accessed: 2015-04-04].

[54] Where's my water? `http://en.wikipedia.org/wiki/Where%27s_My_Water%3F`. [Accessed: 2015-03-17].

[55] Williams-King, D., Denzinger, J., Aycock, J., and Stephenson, B. The gold standard: Automatically generating puzzle game levels. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (2012).

[56] Yavalath. `http://www.boardgamegeek.com/boardgame/33767/yavalath`. [Accessed: 2015-04-03].